Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

В.А. Прытков

Конспект лекций

по дисциплине «Системное программное обеспечение ЭВМ»
для студентов специальности I-40 02 01
«Вычислительные машины, системы и сети»

Минск 2007

Семестр 1. Операционные системы

Введение

Предмет курса, его цели и задачи. Методическое обеспечение. История развития системного ПО. Классификация системного ПО. ОС, системы управления файлами, интерфейсы, системы программирования, утилиты. Понятие ресурса в ОС. Концепция виртуализации ресурса

Учебники

1. Гордеев А.В. Операционные системы. Учебник для ВУЗов. 2-е изд. Спб., Питер, 2005.
2. Таненбаум Э. Современные операционные системы. Спб, Питер, 2004.
3. Карпов В.Е., Коньков К.А. Основы операционных систем. Курс лекций. Учебное пособие. М., Интернет-университет информационных технологий, 2004.
4. Столлингс В. Операционные системы, 4-е изд. М., изд. дом "Вильямс", 2004.

Литература по UNIX

5. Вахалия Ю. UNIX изнутри. СПб, Питер, 2003.
6. Рочкинд М. Программирование для UNIX, 2-е изд. СПб, БХВ-Петербург, 2005.
7. Робачевский А.М., Немнюгин С.А., Стесик О.Л. Операционная система UNIX. СПб, БХВ-Петербург, 2005.

Литература по Windows

8. Руссинович М., Соломон Д. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс. М., изд.-торг. дом "Русская редакция", СПб., Питер, 2005.
9. Харт Д. Системное программирование в среде Windows, 3-е изд. М., изд. дом "Вильямс", 2005.

Литература по другим ОС

10. Минаси М., Камарда Б. И др. OS/2 Warp изнутри. Т.2. СПб., Питер, 1996.
11. Кёртен Р. Введение в QNX Neutrino2. Руководство для разработчиков приложения реального времени. СПб., БХВ-Петербург, 2005.
12. Практика работы с QNX. Алексеев Д., Ведревич Е., Волков А. и др. М., изд. дом "КомБук", 2004.

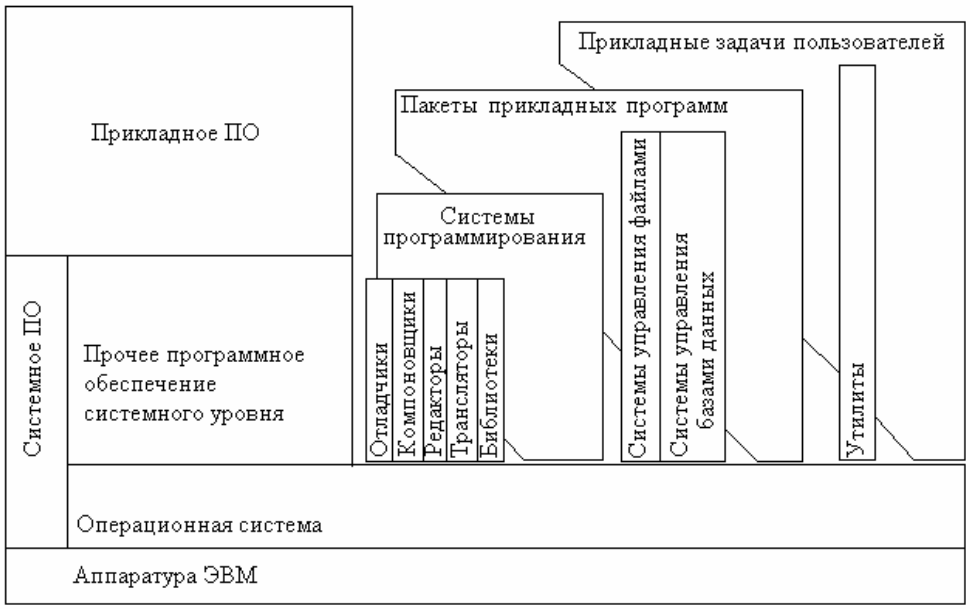
Учебники по всему курсу

13. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. Учебник для ВУЗов. Спб., Питер, 2003.

Учебно-методические пособия

14. Лосич В.А., Радишевский В.А., Отвагин А.В. Основы теории компиляторов. Учебное пособие. Мн., БГУИР, 2000.
15. Глецевич И.И., Лосич В.А., Радишевский В.А. Лабораторный практикум по курсу "Системное программное обеспечение" для студентов специальности Т 10.03.00 "Вычислительные машины, системы и сети". Мн., БГУИР, 2001.
16. Лосич В.А., Радишевский В.А. Основы теории операционных систем. Учебное пособие. Мн., БГУИР, 2001.
17. Прытков В.А., Уваров А.А., Супонев В.А. Типовые механизмы синхронизации процессов : учеб.-метод. пособие по дисц. «Системное программное обеспечение ЭВМ» для студ. спец. I-40 02 01 «Вычислительные машины, системы и сети» – Мн. : БГУИР, 2007

Общая структура программного обеспечения вычислительной системы



К системному ПО относят ПО самого низкого уровня. Таким ПО являются: ОС, системы управления файлами, интерфейсные оболочки для взаимодействия пользователя с ОС, системы программирования, утилиты. В рамках курса изучаются теоретические и практические основы построения, функционирования и проектирования системного ПО.

ОС – это упорядоченная последовательность системных управляющих программ, совместно с необходимыми информационными массивами, предназначенных для планирования и исполнения пользовательских программ, управления всеми ресурсами вычислительной машины (программами, данными, аппаратурой и другими распределяемыми и управляемыми объектами) с целью предоставления возможности пользователям эффективно, в некотором смысле, решать задачи, сформулированные в терминах вычислительной машины. ОС состоит из особых программ и микропрограмм, которые обеспечивают возможность использования аппаратуры. Любой из компонентов прикладного ПО обязательно работает под управлением ОС. Основные функции ОС:

- прием от пользователя заданий и команд в виде директив соответствующего языка или указаний с помощью устройств ввода и их обработка;

- прием и исполнение запросов на запуск, приостановку, остановку программ;
- загрузка в ОЗУ исполняемых программ;
- передача управления программе (инициализация);
- идентификация всех программ и данных;
- обеспечение работы систем управления файлами и иных систем управления низкого уровня, например СУБД;
- обеспечение мультипрограммного режима;
- обеспечение функций по организации и управлению всеми операциями ввода-вывода;
- функционирование в режиме реального времени;
- распределение памяти и организация виртуальной памяти;
- планирование и диспетчеризация задач в соответствии со стратегией и дисциплинами обслуживания;
- организация механизмов обмена сообщениями и данными между программами;
- защита данных программы от воздействия других программ;
- наличие сервисных возможностей для восстановления в случае сбоя;
- обеспечение работ систем программирования.

Следует различать понятие ОС и **операционной среды**. ОС выполняет функции управления вычислительными процессами в вычислительной системе, распределяет ресурсы системы между процессами. Программная среда, в которой выполняется прикладное ПО – операционная среда. Т.о. операционная среда – это набор сервисов и правил обращений к ним, интерфейсы, необходимые для взаимодействия с ОС. ОС в общем случае может поддерживать несколько операционных сред.

Система управления файлами предназначена для организации удобного доступа к данным, структурированным определенным образом. Именно СУФ позволяет заменить низкоуровневый доступ с физической адресацией данных на высокоуровневый с логической адресацией. Современные ОС имеют соответствующие СУФ. Некоторые ОС позволяют работать с несколькими СУФ. Простейшие ОС могут и вовсе не иметь файловой системы.

Назначение **интерфейсных оболочек** – расширение возможностей по взаимодействию с ОС. Примером являются различные варианты графического интерфейса X Windows ОС UNIX, Explorer в ОС Windows. К ПО этого класса относятся и возможности по организации иной операционной среды в рамках виртуальной машины средствами данной ОС. Так, Linux имеет возможности для запуска некоторых приложений ОС Windows. В этот класс входят и эмуляторы ОС, когда одна ОС может быть запущена в рамках другой ОС.

Система программирования предназначена для разработки ПО для конкретной ОС и имеет в качестве составляющих элементов транслятор, редактор, компоновщик, отладчик и библиотеки. Иногда система программирования позволяет получить ПО и для иной ОС. В случае, когда ПО должно функционировать не в иной ОС, а на иной аппаратной базе, используют термин кросс-систем.

Утилиты – это специальное системное ПО, позволяющее выполнять ряд сервисных функций как по обслуживанию самой ОС, так и по подготовке носителей, оптимизации размещения данных и т.д.

История развития ОС и системного ПО тесно связана с историей развития вычислительной техники в целом. Первый цифровой компьютер был изобретен Чарльзом Беббиджем в конце XIX века. Это была чисто механическая машина. В то же время он ясно сознавал, что для аналитической машины требуется программное обеспечение, для чего нанял Аду Лавлейс (Ada Lovelace), дочь знаменитого Байрона. Она стала первым в мире программистом, а язык Ада назван в ее честь. Традиционно историю развития ВТ разделяют на 4 периода.

1. 1945-1955. Первые вычислительные машины на электронных лампах использовали механические реле, длительность такта составляла несколько секунд. Все ПО разрабатывалось непосредственно в машинных кодах. Не существовало ни языков программирования, ни операционных систем. Фактически, на компьютерах занимались прямыми числовыми вычислениями: таблицы тригонометрических функций, логарифмов и т.д. Одна и та же группа людей занимается и проектированием, и и эксплуатацией, и программированием вычислительной машины. Нет специализации.

2. 1955-нач. 60. В конце 50-х – начале 60-х появились компьютеры на транзисторах. Программы набивались уже на перфокартах, для ввода данных в компьютер использовалась промежуточная запись на магнитофонную ленту. Программы обрабатывались в пакетном режиме. Появились языки программирования (Ассемблер, Фортран). ОС в этот период разрабатывались для ускорения и модификации кода перехода с задачи на задачу. Компьютеры использовались главным образом для научных и технических вычислений. Считается, что первую ОС для IBM 701 в начале 50-х создали в лаборатории GM. Следующая ОС была разработана в 1955 для IBM 704. В первых ОС появилась концепция имён системных файлов как средств достижения независимости программ от аппаратуры. Типичными операционными системами были FSM (Fortran Monitor System) и IBSYS (создана IBM). К концу 50-х годов ОС обладали следующими характеристиками:

1. Пакетная обработка одного пакета задач.
2. Наличие стандартных подпрограмм ввода-вывода.
3. Возможность перехода от программы к программе.
4. Наличие средств восстановления после ошибок, обеспечивающих автоматическую чистку машины в случае аварийного завершения задачи, позволяющих запускать следующую задачу при минимальном вмешательстве оператора.
5. Наличие языков управления заданиями, позволяющих описывать задания и ресурсы для их выполнения.

Происходит разделение обслуживающего персонала на программистов, операторов и т.д.

3. Нач. 60-1980. В 60-х годах создаются первые системы коллективного пользования с мультипрограммным обеспечением и первая концепция мультисистемных машин. На этом этапе развиваются методы программирования, обеспечивающие независимость от внешних устройств. Появляются системы с разделением времени и системы реального времени. Наиболее существенные отличия аппаратной базы этого периода, позволившие создать мультипрограммные системы:

1. Реализация защитных механизмов, т.е. наличие привилегированных (используемых только ОС) и непривилегированных команд а также защиты памяти;
2. Наличие прерываний.
3. Развитие параллелизма в архитектуре, т.е. прямой доступ к памяти и каналы ввода-вывода.

ОС этого периода отвечают за :

- организацию интерфейса между прикладной программой и ОС при помощи системных вызовов:
- Организацию очереди заданий и планирование использования процессора:
- сохранение контекста при переключении заданий
- реализация стратегии управления памятью
- поддержка межпрограммных средств коммуникации
- средства синхронизации программ.

В конце 60х начале 70х годов у производителей существовали две совершенно независимые линейки компьютеров: большие компьютеры с пословной обработкой текста для научных и технических вычислений, и коммерческие компьютеры с посимвольной обработкой для банков и страховых компаний для сортировки и печати данных. Ниша между этими линиями была весьма существенной. IBM заполняет ее, выпустив серию машин на ИС IBM/360. Модели в серии различались только ценой и производительностью, являясь совместимыми по структуре и набору команд, и использовали ОС OS/360. Идею совместимости быстро приняли и другие производители. OS/360 на 2-3 порядка превышала по объемам FSM, была написана на ассемблере, имела большое количество ошибок. Важным достижением этой системы явилась многозадачность. Стала обеспечиваться подкачка, т.е. дозагрузка новых задач по мере выполнения предыдущих. Был разработан режим разделения времени, когда у каждого пользователя имелся свой терминал, а машина обслуживала запросы поочередно. Первая серьезная подобного рода система была разработана в Массачусетском технологическом институте – CTSS (Compatible Tme Sharing System – совместимая система разделения времени). Большое внимание стало уделяться разработке систем программирования. Существовала проблема совместимости отдельных машин, для решения которой стали разрабатываться эмуляторы и имитаторы.

К этому времени относится и разработка “компьютерного предприятия общественного пользования” – машины, поддерживающей одновременно сотни пользователей в режиме разделения времени. Система была названа MULTICS (MULTiplex Information and Computing Service - мультиплексная информационная и вычислительная служба). Система была написана на языке PL/1, компилятор же этого языка появился только через несколько лет. Проект с трудом был завершен, но в итоге система была установлена примерно в 80 крупных компаниях и университетах мира. Некоторые их них прекратили использовать ее только через 30 лет, в конце 90х годов.

В этот же период с появлением PDP-1 растет рынок мини-компьютеров. Память PDP-1 составляла всего 4К 18-битовых слов при цене 120 тыс. $ за штуку, однако это составляло порядка 5% от цены IBM7094, хотя на некоторых задачах их производительность была практически равной. Кульминацией их стало появление PDP-11. Впоследствии Кен Томпсон, работавший над проектом MULTICS, нашел PDP-7 и написал усеченную однопользовательскую версию MULTICS. К нему присоединился Деннис Ритчи, система получила название UNIX и была перенесена на PDP-11/20 и ряд других, более совершенных машин. Чтобы не переписывать исходный код каждый раз заново, Томпсон решил переписать его на языке высокого уровня, разработал его, назвал язык B. Попытка оказалась неудачной. Тогда Ритчи разработал следующий язык, назвав его C, и написал к нему хороший компилятор. Вскоре UNIX была переписана на C, разработчики опубликовали статью, получив за нее престижную премию Тьюринга, что принесло UNIX известность. Благодаря широкому распространению в университетских кругах PDP-11 и слабой ОС на них, UNIX быстро приобрела популярность. Для переноса на другие платформы Стивом Джонсоном был написан переносимый компилятор C, позволяющий настраивать его на создание объектного кода для практически любой машины. Вскоре появилась первая переносимая версия UNIX. В конце концов появилось два ведущих клона UNIX – System V и BSD. Для того, чтобы ПО могло функционировать в любой версии UNIX, IEEE разработал стандарт POSIX, определяющий минимальный интерфейс системного вызова.

4. 1980 и далее. Появление БИС дало новый толчок к развитию. В 1974 году Intel выпускает Intel 8080. Для его тестирования была необходима ОС. Гари Килдэлл сконструировал контроллер гибкого диска, подключив его к процессору. Так появился первый микрокомпьютер с диском. Для него была написана система CP/M (Control Programm for Microcomputers). В течении 5 лет CP/M занимала на рынке доминирующее положение. В начале 80х IBM разработала первый персональный компьютер и стала искать для него ОС. Контактируя с Биллом Гейтсом для получения лицензии на его интерпретатор языка BASIC, они поинтересовались и по поводу ОС. Он рекомендовал Килдэлла. Однако тот отказался от личной встречи, более того, его адвокат отказался подписывать соглашение о неразглашении по поводу еще не выпущенного персонального компьютера. В итоге IBM вновь обратилась к Гейтсу. Он нашел у местного производителя компьютеров подходящую ОС (DOS) и выкупил ее. После ряда доработок совместно с Тимом Патерсоном, разработчиком системы, она была переименована в MS-DOS (Microsoft Disc Operation System), и заняла доминирующее положение на рынке ОС для ПК. Важную роль сыграла ориентация Гейтса на продажу системы не конечным пользователям, а, в отличие от Килдэлла, производителям ПК для оснащения ею их машин.

Еще в 60е годы Даг Энгельбарт изобрел графический интерфейс пользователя (GUI). Стив Джобс однажды увидел его в Xerox PERC, и приступил к созданию Apple с графическим интерфейсом. Со второй попытки затея удалась, и ПК приобрел дружественный пользователю интерфейс. Microsoft переняла идею и на основе GUI создала надстройку над MS-DOS (Windows 3.11). В 1995 году вышла первая независимая ОС Windows 95 от Microsoft. На настоящий момент UNIX тожже об-

завелась графическим интерфейсом X Windows. С середины 80х годов с ростом сетей ПК, стали развиваться сетевые и распределенные ОС. Сетевые ОС мало отличаются от однопроцессорных систем, нуждаясь, по большому счету в сетевом интерфейсе, что не изменяет структуру ОС. Распределенная ОС только представляется пользователю традиционной системой, она на деле состоит из множества процессоров, при распределение задач осуществляется средствами самой ОС, освобождая от этого пользователя. Задержки при передаче данных в сетях означают, что распределенная ОС должна уметь работать с неполной, устаревшей или даже неверной информацией, что в корне отличается от однопроцессорных ОС, где система имеет полную информацию относительно состояния системы.

Ресурс, в общем случае - всякий потребляемый объект (независимо от формы его существования), обладающий некоторой практической ценностью для потребителя. **Классификация ресурсов** может быть произведена по широкому ряду свойств.

По реальности существования: физический и виртуальный. Под **физическим** понимают ресурс, который реально существует и при распределении его между пользователями обладает всеми присущими ему физическими характеристиками. **Виртуальный ресурс** - это некоторая модель физического ресурса. Виртуальный ресурс не существует в том виде, в котором он проявляет себя пользователю. Как модель виртуальный ресурс реализуется в некоторой программно-аппаратной форме. В этом смысле виртуальный ресурс существует. Однако виртуальный ресурс может предоставить пользователю при работе с ним не только часть тех свойств, которые присущи объекту моделирования, т.е. физическому ресурсу, но и свойства, которые ему не присущи.

По возможности расширения свойств: эластичный и жесткий. Характеризует ресурс с точки зрения возможности построения на его основе некоторого виртуального ресурса. Физический ресурс, который допускает "виртуализацию", т.е. воспроизведение и (или) расширение своих свойств, называют **эластичным**. **Жестким** называется физический ресурс, который по своим внутренним свойствам не допускает виртуализацию.

По степени активности: активный и пассивный. При использовании **активного ресурса** он способен выполнять действия по отношению к другим ресурсам (или даже в отношении самого себя) или процессам, которые в общем случае приводят к изменению последних. **Пассивный ресурс** не обладает таким свойством. Над таким объектом проводить допустимые для него действия, которые могут привести к изменению его состояния, т.е. к изменению внутренних или внешних характеристик. ЦП - активный ресурс, область памяти, выделяемая по требованию - пассивный ресурс.

По времени существования: постоянный, временный. Если ресурс существует в системе до момента порождения процесса и доступен для использования на всем протяжении интервала существования процесса, то такой ресурс является **постоянным** для данного процесса. **Временный** ресурс может появляться или уничтожаться в системе динамически в течение времени существования рассматриваемого процесса. Причем создание и уничтожение может проводиться как самим процессом, так и другими процессами - системными или пользовательскими. Очевидно, что ресурсы разделяются по определенным правилам системной взаимосвязанных процессов. Поэтому ресурсы, которые являются постоянными для одних процессов, могут быть временными для других, и наоборот.

По степени важности: главный и второстепенный. Ресурс является **главным** по отношению к конкретному процессу, если без его выделения процесс принципиально не может развиваться. К таким ресурсам относятся прежде всего ЦП и ОП. Ресурсы, которые допускают некоторую альтернативу развития процесса, если они не будут выделены, называются **второстепенными**. (Например МЛ, МД).

По функциональной избыточности: дорогие и дешевые. Разделение ресурсов на **дорогие и дешевые** связано с реализацией принципа функциональной избыточности при распределении ресурсов. Перед пользователем стоит задача выбора – получить быстро требуемый ресурс и дорого заплатить за такую услугу, либо подождать выделения требуемого ресурса и после его использования заплатить более дешево. При наличии в системе альтернативных ресурсов вводятся и различные цены за их использование.

По структуре: простой и составной. Структурный признак устанавливает наличие или отсутствие у ресурса некой структуры. Ресурс является **простым**, если не содержит составных элементов и рассматривается при распределении как единое целое. **Составной ресурс** характеризуется некоторой структурой. Он содержит в своем составе ряд однотипных элементов, обладающих с точки зрения пользователей, одинаковыми характеристиками. Процессам-пользователям безразлично, какой или какие из элементов среди прочих из составного ресурса будут выделяться им при удовлетворении их запросов на ресурс. Простой и составной ресурсы отличаются числом состояний. Простой ресурс может быть либо "занят", когда он выделен для пользования какому-либо процессу, либо "свободен". Составной ресурс находится в состоянии "свободен", если ни один из его составных элементов не распределен для использования. Если же все элементы такого ресурса выделены для использования, то он находится в состоянии "занят". Если часть элементов ресурса распределена, а остальные (известно какие) нет, то ресурс "частично занят".

По восстанавливаемости: воспроизводимый, потребляемый. При построении механизмов распределения ресурсов на основе использования той или иной дисциплины особенно важно учитывать характер использования распределяемых ресурсов. По этому признаку учитывается и сущность ресурса, возможность в этой связи восстанавливаемости ресурса в системе после его использования. По возможности восстанавливаемости ресурсы подразделяются на **воспроизводимые и потребляемые**. Предполагаются, что в отношении каждого ресурса процесс-пользователь выполняет три типа действий: ЗАПРОС, ИСПОЛЬЗОВАНИЕ, ОСВОБОЖДЕНИЕ. Если при распределении системой ресурса допускается многократное выполнение действий в последовательности запрос-использование-освобождение, то такой ресурс называют воспроизводимым. После возвращения он доступен для использования его другим процессом. Поэтому, если не учитывать вид изменений ресурса при

каждом разовом использовании, можно считать время жизни ресурса бесконечно большим или достаточно большим, пока он не потеряет своих функциональных свойств. В отношении определенной категории ресурсов правомочно использование действий в следующем порядке: освобождение-запрос-использование, после чего ресурс, который в данном случае называют потребительным, изымается из сферы потребления (например, отношение производитель-потребитель). Срок жизни потребляемого ресурса, определяемый периодом между выполнением действий освобождение и использование, конечен. В отношении процесса производителя и процесса-потребителя потребляемые ресурсы ведут себя как временные.

По характеру использования: параллельно используемый, последовательно используемый. Природа ресурса и (или) используемое правило распределения ресурса обусловлены **параллельной** или **последовательной** схемой использования распределяемого между несколькими процессами ресурса. Последовательная схема предполагает, что в отношении некоторого ресурса, который называют последовательно используемым, допустимо строго последовательное во времени выполнение цепочек действий "запрос-исполнение-освобождение" каждым процессом-потребителем этого ресурса для параллельных процессов такие цепочки действий являются критическими областями и должны выполняться так, чтобы удовлетворять правилу взаимного исключения, определенному ранее. Поэтому последовательно используемый ресурс, разделяемый несколькими параллельными процессами, чаще называют критическим ресурсом. Параллельная схема предполагает параллельное, т.е. одновременное, использование одного ресурса, который поэтому называют параллельно используемым более чем одним процессом. Такое использование не должно вносить каких-либо ошибок в логику развития каждых из процессов (массив в памяти для чтения).

По форме реализации: **твердые и мягкие ресурсы**. Под "твердыми" понимают аппаратные компоненты машины, а также человеческие ресурсы. Все остальные виды ресурсов относятся к разряду "мягких". Существенно разным для твердых и мягких ресурсов помимо сложности и стоимости является их подверженность сбойным или отказываемым ситуациям и последующее восстановление работоспособности. В отличие от "твердых" "мягкие" ресурсы не могут стать неработоспособными из-за усталостного отказа. В классе "мягких" ресурсов выделяют два типа: **программные и информационные**. Если "мягкий" ресурс допускает копирование и эффект от использования ресурса-оригинала и ресурса-копии идентичен, то такой ресурс называют программным мягким ресурсом. В противном случае его следует отнести к информационному типу (это программы, файлы, массивы и т.п.). "Мягкие" информационные ресурсы либо принципиально не допускают копирование, либо допускают копирование, но оно является функцией времени. Это различного вида потребляемые ресурсы: сообщения, сигналы прерывания, запросы к ОС на различного вида услуги, сигналы синхронизации. Такие сообщения и сигналы информационно значимы (доступны и ценны, как ресурс) только в течение некоторого конечного интервала времени. Например, если в некоторую ячейку памяти записывается периодически некоторые сообщения, то возможно копирование конкретного поступившего сообщения от момента записи его в эту ячейку до момента поступления туда нового сообщения. Последующее копирование уже дает другой результат от использования выбранного сообщения.

Ресурсы подразделяются на **выгружаемые** и **невыгружаемые**. Выгружаемый ресурс можно безболезненно забирать у владеющего им процесса, например, память. Невыгружаемый ресурс нельзя забрать от владельца, не уничтожив результаты вычислений. Например, нельзя прервать запись компакт-диска.

В терминах ОС понятие ресурс обычно используется по отношению к повторно используемым, относительно стабильным и зачастую недостающим объектам, которые могут запрашиваться, использоваться и освобождаться.

При разработке первых ОС ресурсами считались процессорное время, память, каналы ВВ и периферийные устройства. Позже понятие ресурса стало более универсальным и общим. К ним стали относиться и разного рода программные и информационные ресурсы, которые с точки зрения системы, также могут являться объектами, которые возможно распределять и управлять доступом. Понятие ресурса превратилось в абстрактную структуру с рядом атрибутов, характеризующих способы доступа к ней и ее физическое представление в системе. Кроме системных ресурсов, в это понятие стали включаться и такие объекты межпроцессного обмена, как сообщения и синхросигналы.

Одним из основных видов ресурсов является процессор. При этом собственно процессор как ресурс выступает лишь для многопроцессорных систем, в однопроцессорных же системах ресурсом является процессорное время. Его разделение производится по параллельной схеме. Методы разделения этого ресурса будут рассмотрены позже.

Следующий вид ресурсов - память. Проблема эффективного разделения оперативной памяти между процессами является одной из самых актуальных. В общем случае, собственно память и доступ к ней являются разными ресурсами. Каждый из них может быть предоставлен независимо друг от друга, но для полной работы с памятью необходимы оба из них.

Внешние устройства являются еще одним видом ресурсов. При наличии механизмов прямого доступа они могут использоваться одновременно. Если же устройство имеет только последовательный доступ, то оно не является разделяемым ресурсом, например, принтер, накопитель на магнитной ленте.

Программные модули так же являются одним из ресурсов. Однократно используемые модули могут быть правильно выполнены только один раз, в процессе работы они могут либо испортить свой код, либо исходные данные. Такие модули являются неделимым ресурсом. Повторно используемые модули могут быть непривилегированными, привилегированными, реентерабельными и повторно входимыми.

Данные выступают в качестве информационных ресурсов. Это либо переменные в ОЗУ, либо файлы. В случае использования данных только для чтения, они легко разделяются. В случае же разрешения процессам изменения этого вида ресурса, то проблема его разделения значительно усложняется.

Понятие операционной системы

Операционная система. Классификация ОС. ОС реального времени. Микроядерные и монолитные ОС. Структура ОС. Ядро, командный процессор, подсистема ввода-вывода, система управления памятью, файловая система. Принципы построения ОС. Понятие виртуальной машины. Безопасность операционных систем. Понятие системных вызовов. Системные вызовы стандарта POSIX. Интерфейс Win32 API.

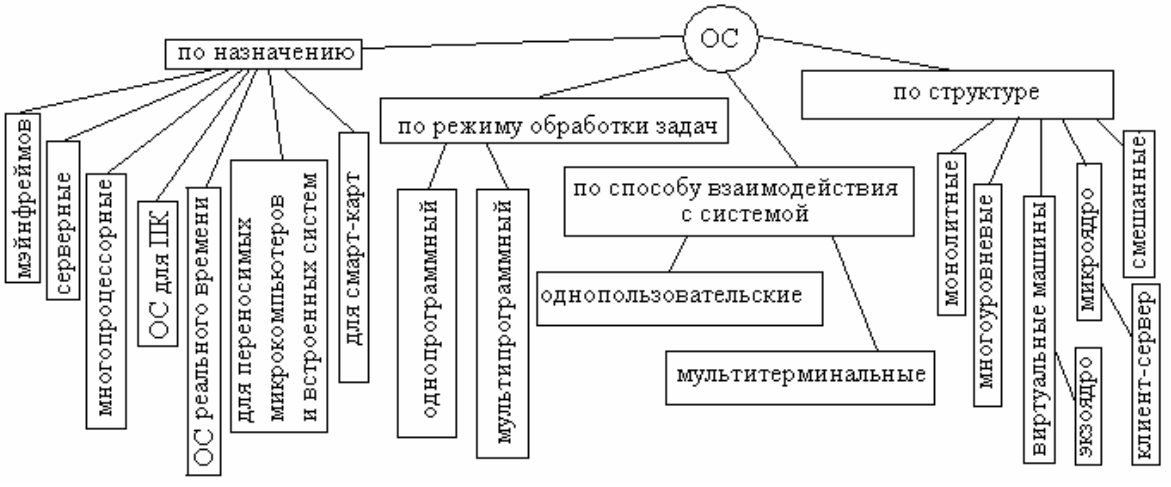
Первые программы разрабатывались непосредственно в машинных кодах. Для этого требовалось владеть в совершенстве архитектурой как самого микропроцессора, так и системы на его основе. Очевидно, что переход к новой системе был связан с большими затратами на обучение. По мере развития вычислительной техники стали выделять наиболее часто встречающиеся операции и создавать для них программные модули, которые затем можно использовать в разрабатываемом ПО. Так, в 50-х гг. при разработке первых систем программирования вначале создавали модули для операций ввода-вывода, после – для вычисления математических операций и функций. Дальнейшее развитие привело к появлению трансляторов высокого уровня, которые могли подставлять вместо операторов необходимые вызовы библиотечных функций. Количество библиотек возрастало. В итоге у разработчиков прикладного ПО отпала необходимость в подробном владении архитектурой системы. Они могли обращаться к программной подсистеме с соответствующими вызовами и получать от нее необходимые функции и сервисы. Эта программная подсистема и является ОС.

Основные составляющие современной ОС – это ядро, подсистема ввода-вывода, командный процессор, система управления памятью, файловая система. **Ядро** обеспечивает основной набор базовых функций по управлению задачами и ресурсами, их синхронизацией и взаимодействием. **Командный процессор** обеспечивает прием и обработку команд, вызов соответствующих сервисов ОС по запросу пользователя.

Подсистема ввода-вывода обеспечивает выполнение задач по вводу-выводу данных с внешними устройствами. Наличие этих библиотек в ОС позволяет не встраивать их средствами систем программирования в каждую из разрабатываемых программ. Системы программирования только генерируют обращения к системному коду ввода-вывода и выполняют подготовку данных. Подсистема ВВ является одной из самых сложных в силу большого числа различных устройств ввода-вывода. При этом недостаточно обеспечить эффективное управление, требуется еще и создать удобный и эффективный виртуальный интерфейс, позволяющий прикладным программистам абстрагироваться от специфики устройств. С другой стороны, требуется обеспечение доступа к устройствам ВВ множества параллельно выполняющихся задач. Некоторые из программ ВВ являются независимыми от устройств, и их можно применять ко многим устройствам ВВ, другое ПО, в т.ч. драйверы устройств, предназначены для конкретных устройств ВВ.

Файл – набор данных, организованных в виде совокупности записей одинаковой структуры. **Файловая система** определяет способ организации данных на диске или ином носите информации и предоставляет пользователю возможность иметь дело с логическим уровнем структур данных и операций. Все современные ОС имеют соответствующие системы управления файлами. Она является основной в большинстве современных ОС. Благодаря СУФ все системные обрабатывающие программы связываются по данным. СУФ решает проблему централизованного распределения дискового пространства и управления данными. СУФ предоставляют пользователям широкие сервисные возможности по работе с файлами и каталогами, в тоже время скрывая от пользователя особенности дисков и других устройств ВВ. В UNIX важным является понятие монтирования дисков. Корневая файловая система и файловая система на диске существуют отдельно и никак не связаны между собой. При этом файлы гибкого диска нельзя использовать, поскольку для них неопределен путь. UNIX не позволяет присоединять к пути название диска или его номер, поскольку это приводит к нежелательной для ОС жесткой зависимости от устройств. Системный вызов mount позволяет монтировать (присоединять) файловую систему гибкого диска к корневой файловой системе в указанное место. Еще одно понятие UNIX – специальные файлы. На самом деле это устройства ввода-вывода, которые выглядят как файлы.

Классификация ОС:



ОС классифицируются по назначению, по режиму обработки задач, по способу взаимодействия с системой, по способу построения. Самый широкий ряд ОС возникает при классификации по назначению, поскольку сколько видов вычислительной техники – столько и видов ОС: ОС мэйнфреймов, серверные ОС, многопроцессорные, для ПК, ОС реального времени, для переносимых компьютеров и встроенных систем, для смарт-карт.

Мэйнфреймы отличаются от ПК по возможностям ввода-вывода, позволяя зачастую обрабатывать терабайты данных. ОС мэйнфреймов ориентированы на обработку множества одновременных заданий с большим количеством операций ввода-

вывода. Как правило, предлагают три вида обслуживания: пакетная обработка, обработка транзакций (групповых операций) и разделение времени. В случае пакетной обработки задачи обрабатываются без участия пользователей. Например, составление разнообразных отчетов может быть выполнено в пакетном режиме. В случае обработки транзакций выполняется большое число маленьких запросов, таких как бронирование билетов, операции с кредитными карточками и т.д. Запросы невелики, но система одновременно обрабатывает сотни и тысячи таких запросов в секунду. В режиме разделения времени система позволяет множеству удаленных пользователей одновременно выполнять задачи на одной машине. Типичным примером является многопользовательская база данных. Пример ОС для мэйнфрейма – OS/390.

Серверы одновременно обслуживают множество пользователей, позволяя им делить между собой программные и аппаратные ресурсы. Серверы предоставляют возможность работы с печатающими устройствами, файлами, Интернетом. На серверах хранятся Web-страницы и обрабатываются входящие запросы. UNIX, Windows 2000, Linux – примеры серверных ОС. Для работы с системами, в которых объединены несколько процессоров, требуются специальные ОС, часто **многопроцессорные ОС** представляют собой серверные ОС со специальными возможностями связи. Основная задача **ОС для ПК** – предоставление удобного интерфейса пользователю. Эти ОС используются для доступа к Интернету, работы с текстом, электронными таблицами и т.д. Все клоны ОС Windows, Linux.

ОС реального времени используются, когда процессы, которыми управляет машина, например, сборочной линией на производстве, должны удовлетворять жестким временным требованиям. Если действия должны происходить строго в указанный диапазон времени – это жесткая СРВ, если же время от времени возможны пропуски сроков выполнения операции, например, цифровое аудио и мультимедийные системы, то это гибкая СРВ. Примерами СРВ являются VxWorks и QNX. **Для карманных компьютеров, а также встроенных систем**, управляющих широким спектром бытовых и прочих устройств (телевизоры, микроволновые печи, мобильные телефоны) используются встроенные ОС. Они могут обладать характеристиками ОС реального времени, но имеют меньший размер, память, ограниченную мощность. Примерами являются PalmOS и Windows CE. **Смарт-карта** – устройство размером с кредитную карту, содержащее центральный процессор. Часто ОС для смарт-карт являются патентованными системами. Часть смарт-карт являются Java-ориентированными, у них ПЗУ содержит интерпретатор виртуальной Java-машины (JVM). Некоторые из смарт-карт позволяют управлять несколькими апплетами одновременно, что приводит к многозадачности и необходимости планирования, требуется управление ресурсами и защитой. Все эти задачи выполняет, как правило, примитивная ОС, находящаяся на смарт-карте.

В **монолитных ОС** все части системы сильно связаны между собой. Поэтому изменение или удаление какой-либо части требует хорошего знания всей архитектуры ОС и может повлечь необходимость изменения остальных модулей. При этом возникает ряд проблем, связанных с тем, что функции макроядра работают в едином адресном пространстве. Это вызывает опасность возникновения конфликтов, а также сложность подключения новых драйверов. Структура как таковая отсутствует. ОС представляет собой набор процедур, каждая из которых может вызывать при необходимости любые другие. Для построения такой системы компилируются все отдельные процедуры, с помощью компоновщика связываются в единый объектный файл. Поскольку для каждой процедуры доступны все другие, практически отсутствует сокрытие деталей реализации. Монолитные системы могут поддерживать механизм прерываний. В этом случае предполагается некоторая структура ОС: На верхнем уровне лежит главная программа, которая вызывает требуемую служебную процедуру. Ниже идет набор служебных процедур, выполняющих системные вызовы. На самом низу лежат утилиты, обслуживающие системные процедуры.

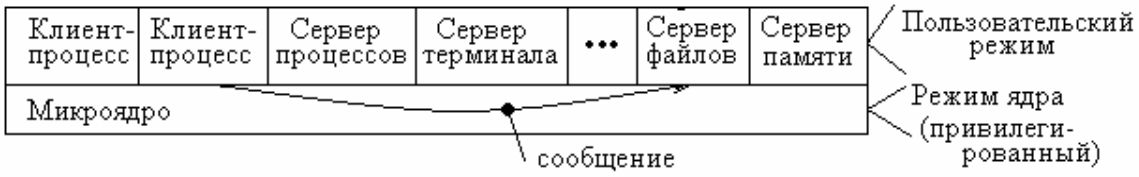
Многоуровневые системы имеют организацию в виде иерархии уровней. Первой подобной системой была система THE, созданная Дейкстрой в 1968 году. Она содержала 6 уровней. 0 – распределение процессора и многозадачность, 1 – управление памятью, 2 – связь оператор-процесс, 3 – управление ВВ, 4 – программы пользователя, 5 – оператор. Уровень 0 занимался распределением времени процессора, переключая процессы при возникновении прерывания или срабатывании таймера. Выше этого уровня система состояла из последовательных процессов, каждый из которых можно было программировать не беспокоясь о том, что на одном процессоре запущено несколько процессов. Т.е. уровень 0 обеспечивал базовую многозадачность процессора. Уровень 1 управлял памятью. Он выделял процессам память в ОЗУ и на магнитном барабане, если ОЗУ не хватало. По мере необходимости страницы с барабана попадали в ОЗУ. Выше этого уровня процессам не было необходимости заботиться о том, где они находятся – в ОЗУ или на барабане. Уровень 2 управлял связью между консолью оператора и процессами. Все процессы выше этого уровня имели свою собственную консоль оператора. Уровень 3 управлял устройствами ВВ и буферизацией данных. Любой процесс выше 3 уровня работал уже не с конкретными устройствами ВВ, а с абстрактными УВВ с удобными для пользователя характеристиками. Дальнейшее обобщение концепции многоуровневых систем было сделано в системе MULTICS. Уровни представляли серию концентрических колец, внутренние кольца являлись более привилегированными, чем внешние. Если внешнее кольцо хотело вызвать процедуру из кольца внутреннего, выполнялся эквивалент системного вызова с тщательной проверкой параметров и возможности доступа. Многоуровневые системы просты в реализации. При разработке каждого из уровней нет необходимости знать устройство более низкого уровня, достаточно знать, как к обратиться к функциям этого уровня. Упрощается тестирование. Поскольку отладка идет снизу вверх послойно, то можно быть уверенным, что возникшая ошибка находится именно в тестируемом слое. Упрощается модификация систем. При необходимости достаточно изменить функциональность одного из уровней, не меняя остальных. Недостаток подобных систем в том, что запрос от пользователя вынужден проходить все слои поочередно, аналогично и результат запроса передается от уровня к уровню. Кроме того весьма непросто разделить систему на нужное количество уровней, определить их иерархию и разграничить возможности каждого слоя.

Виртуальные машины развились на основе проработки двух принципов: 1. система с разделением времени обеспечивает многозадачность, 2. расширенную машину с более удобным интерфейсом, чем предоставляемый непосредственно оборудованием. Первая ОС такого рода VM/370. Монитор виртуальной машины работает с оборудованием и обеспечивает многозадачность, предоставляя верхнему слою не одну, а несколько виртуальных машин. В отличие от других ОС, эти ВМ не явля-

ются расширенными, а представляют собой точную копию аппаратуры, включая режимы ядра и пользователя, ВВ, прерывания и т.д. В итоге на каждой из таких ВМ может быть запущена любая ОС. Когда программа выполняет системный вызов, он прерывает ОС на виртуальной машине, а не на VM/370. В случае ВМ многозадачность реализуется на уровне ядра, и она отделена от ОС пользователя. Недостаток в том, что снижается эффективность, кроме того, подобные системы очень громоздки. Однако имеется возможность использования на одной машине программ, написанных для разных ОС. Сейчас ВМ используются несколько в ином контексте. Например, для организации нескольких операционных сред. Примером этого является VDM-машина (Virtual DOS machine) – защищенная подсистема, предоставляющая полную среду MS-DOS и консоль для выполнения ее приложений. Одновременно может выполняться практически произвольное число VDM-сессий. Однако здесь пользователю предоставляется виртуальный процессор 8086, не обладающий функциональностью реальной системы на уровне Pentium. Понятие виртуальной машины используется и при построении Java-апплетов. Компилятор Java строит код для JVM. Этот код может быть выполнен на любой платформе, для которой существует интерпретатор JVM.

Развитие концепции ВМ привело к появлению систем, обеспечивающих пользователя абсолютной копией реального компьютера, но с подмножеством ресурсов. На нижнем уровне в режиме ядра работает программа, называемая **экзоядром**, распределяющая ресурсы для ВМ и защиту их использования. Каждая ВМ на уровне пользователя может работать со своей собственной ОС, с тем отличием, что она ограничена предоставленным набором ресурсов. Преимущество схемы в том, что не требуется таблицы преобразования адресов ВМ в реальные адреса диска, так как каждой ВМ выделяется свой блок адресов.

В современных ОС наблюдается тенденция в сторону переноса кода в верхние уровни, оставляя в режиме ядра минимально необходимые функции, т.н. **микроядро**. Микроядро функционирует в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использование процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью. Остальные функции, характерные для ОС, проектируются как модульные дополнения-процессы, взаимодействующие между собой путем передачи сообщений через микроядро. Реализуется клиент-серверная схема взаимодействия. Получая запрос на операцию, пользовательский процесс **(клиент)** посылает запрос обслуживающему процессу **(серверу**), который выполняет обработку и возвращает ответ.



Благодаря разделению ОС на части, каждая из которых управляет одним элементом системы, все части становятся маленькими и управляемыми. Серверные процессы ничем принципиально не отличаются от клиентских. В таких системах можно без перезагрузки добавлять и удалять из системы драйверы устройств, файловые системы и т.д. Поскольку все серверы работают как процессы в режиме пользователя, они не имеют прямого доступа к оборудованию, что повышает устойчивость системы к сбоям. Некоторые функции ОС, например, загрузка команд в регистры устройств ВВ, практически невозможно выполнить из программ в пространстве пользователя. Одно из решений заключается в том, что критические серверные процессы (например, драйверы устройств), запускаются в режиме ядра, но общаются с другими процессами по традиционной схеме путем передачи сообщений. Преимущество модели клиент-сервер еще и в том, что она легко адаптируется к распределенным системам. Действительно, поскольку части независимы, любая из них легко может быть выполнена на удаленной машине, при этом с точки зрения клиента, происходит то же самое: посылается запрос и возвращается ответ. Однако затраты на передачу сообщений существенно влияют на производительность, поэтому требуется очень грамотное разбиение системы на компоненты.

Большинство структур имеют как достоинства, так и недостатки. Как правило, современные ОС комбинируют несколько подходов. Например, ядро Linux является монолитным с элементами микроядерной архитектуры. При компиляции ядра можно разрешить динамическую загрузку/выгрузку многих модулей ядра. Монолитное ядро с микроядерными элементами имеет и Windows NT. Компоненты системы располагаются в вытесняемой памяти и взаимодействуют путем передачи сообщений, что характерно для микроядерных ОС, однако они работают в едином адресном пространстве и используют общие структуры данных, что является признаком монолитных ОС.

Еще один пример смешанной архитектуры – возможность запуска ОС с монолитным ядром под управлением микроядра. Например, 4.4BSD и MkLinux, основанные на микроядре Mach. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов, а все остальные функции обеспечиваются монолитным ядром.

Основные принципы построения ОС:
Частотный принцип. Основан на выделении в алгоритмах программ, а в обрабатываемых массивах действий и данных по частоте использования. Действия и данные, которые часто используются, располагаются в операционной памяти, для обеспечения наиболее быстрого доступа. Основным средством такого доступа является организация многоуровневого планирования. На уровень долгосрочного планирования выносятся редкие и длинные операции управления деятельностью системы. К краткосрочному планированию подвергаются часто используемые и короткие операции. Система инициирует или прерывает исполнение программ, предоставляет или забирает динамически требуемые ресурсы, и прежде всего центральный процессор и память.

Принцип модульности. Модуль – это функционально законченный элемент системы, выполненный в соответствии с принятыми межмодульными интерфейсами. Модуль по определению предполагает возможность замены его на любой другой при наличии соответствующих интерфейсов. Чаще всего при построении ОС разделение на модули происходит по функцио-

нальному признаку. Важное значение при построении ОС имеют привилегированные, повторно входимые и реентерабельные модули. Привилегированные модули функционируют в привилегированном режиме, при котором отключается система прерываний, и никакие внешние события не могут нарушить последовательность вычислений. Реентерабельные модули допускают повторное многократное прерывание исполнения и повторный запуск из других задач. Для этого обеспечивается сохранение промежуточных вычислений и возврат к ним с прерванной точки. Повторно входимые модули допускают многократное параллельное использование, однако не допускают прерываний. Они состоят из привилегированных блоков и повторное обращение к ним возможно после завершения какого-либо из этих блоков. Принцип модульности отражает технологические и эксплуатационные свойства системы. Максимальный эффект от использования достигается, если принцип распространяется и на ОС, и на прикладные программы, и на аппаратуру.

Принцип функциональной избирательности. Этот принцип подразумевает выделение некоторых модулей, которые должны постоянно находиться в оперативной памяти для повышения производительности вычислений. Эту часть ОС называют ядром. С одной стороны, чем больше модулей в ОЗУ, тем выше скорость выполнения операций. С другой стороны, объем памяти, занимаемой ядром, не должен быть слишком большим, поскольку в противном случае обработка прикладных задач будет низкоэффективной. В состав ядра включают модули по управлению прерываниями, модули для обеспечения мультизадачности и передачи управления между процессами, модули по распределению памяти и т.д.

Принцип генерируемости ОС. Этот принцип определяет такой способ организации архитектуры ядра ОС, который позволял бы настраивать его, исходя из конкретной конфигурации вычислительного комплекса и круга решаемых задач. Эта процедура выполняется редко, перед достаточно протяженным периодом эксплуатации ОС. Процесс генерации осуществляется с помощью специальной программы-генератора и соответствующего входного языка. В результате генерации получается полная версия ОС, представляющая собой совокупность системных наборов модулей и данных. Принцип модульности существенно упрощает генерацию. Наиболее ярко этот принцип используется в ОС Linux, которая позволяет не только генерировать ядро ОС, но указывать состав подгружаемых, т.н. транзитных модулей. В остальных ОС конфигурирование выполняется в процессе инсталляции.

Принцип функциональной избыточности. Принцип учитывает возможность проведения одной и той же операции различными средствами. В состав ОС могут входить несколько разных мониторов, управляющих тем или иным видом ресурса, несколько систем управления файлами и т.д. Это позволяет быстро и достаточно адекватно адаптировать ОС к определенной конфигурации вычислительной системы, обеспечить максимально эффективную загрузку технических средств при решении конкретного класса задач и получить при этом максимальную производительность.

Принцип умолчания. Применяется для облегчения организации связи с системами, как на стадии генерации, так и при работе с системой. Принцип основан на хранении в системе некоторых базовых описаний, структур процесса, модулей, конфигураций оборудования и данных, определяющих прогнозируемые объемы требуемой памяти, времени счета программы, потребности во внешних устройствах, которые характеризуют пользовательские программы и условия их выполнения. Эту информацию пользовательская система использует в качестве заданной, если она не будет заданна или сознательно не конкретизирована. В целом применение этого принципа позволяет сократить число параметров устанавливаемых пользователем, когда он работает с системой.

Принцип перемещаемости. Предусматривает построение модулей, исполнение которых не зависит от места расположения в операционной памяти. Настройка текста модуля в соответствии с его расположением в памяти осуществляется либо специальными механизмами, либо по мере ее выполнения. Настройка заключается в определении фактических адресов, используемых в адресных частях команды, и определяется применяемым способом адресации и алгоритмом распределения оперативной памяти, принятой для данной ОС. Она может быть распределена и на пользовательские программы.

Принцип виртуализации. Принцип позволяет представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов (мониторов), используя единую централизованную схему. Концепция виртуальности выражается в понятии виртуальной машины. Любая ОС фактически скрывает от пользователя реальные аппаратные и иные ресурсы, заменяя их некоторой абстракцией. В результате пользователи видят и используют виртуальную машину как достаточно абстрактное устройство, способное воспринимать их программы, выполнять их и выдавать результат. Пользователю совершенно не интересна реальная конфигурация вычислительной системы и способы эффективного использования ее компонентов. Он работает в терминах используемого им языка и представленных ему виртуальной машиной ресурсов. Для нескольких параллельных процессов создается иллюзия одновременного использования того, что одновременно в реальной системе существовать не может. Виртуальная машина может воспроизводить и реальную архитектуру, однако элементы архитектуры выступают с новыми, либо улучшенными, характеристиками, зачастую упрощающими работу с системой. Идеальная, с точки зрения пользователя, машина должна иметь:

- единообразную по логике работы виртуальную память практически неограниченного объема;
- произвольное количество виртуальных процессоров, способных функционировать параллельно и взаимодействовать во время работы;
- произвольное количество виртуальных внешних устройств, способных получать доступ к памяти виртуальной машины последовательно или параллельно, синхронно или асинхронно. Объемы информации не ограничиваются.

Чем больше виртуальная машина, реализуемая ОС, приближена к идеальной, т.е. чем больше ее архитектурно-логические характеристики отличны от реальных, тем большая степень виртуальности достигнута. ОС строится как иерархия вложенных друг в друга виртуальных машин. Нижним уровнем программ является аппаратные средства машин. Следующим уровнем уже является программным, который совместно с нижним уровнем обеспечивает достижение машиной новых свойств. Каждый новый уровень дает возможность расширять функции возможности по обработке данных и позволяет достаточно просто производить доступ к низшим уровням. Применение метода иерархического упорядочивания виртуальных машин

наряду с достоинствами: систематичность проекта, возрастание надежности программных систем, уменьшение сроков разработки имеет проблемы. Основная из них: определение свойств и количества уровней виртуализации, определения правил внесения на каждый уровень необходимых частей ОС. Свойства отдельных уровней абстракции (виртуализации):

1. На каждом уровне ничего не известно о свойствах и о существовании более высоких уровней.
2. На каждом уровне ничего не известно о внутреннем строении других уровней. Связь между ними осуществляется только через жесткие, заранее определенные сопряжения.
3. Каждый уровень представляет собой группу модулей, некоторые из них являются внутренними для данного и доступны для других уровней. Имена остальных модулей известны на следующим, более высоком уровне, и представляют собой сопряжение с этим уровнем.
4. Каждый уровень располагает определенными ресурсами и либо скрывает от других уровней, либо представляет другим уровням их абстракции (виртуальные ресурсы).
5. Каждый уровень может обеспечивать некоторую абстракцию данных в системе.
6. Предположения, что на каждом уровне делается относительно других уровней, должны быть минимальными.
7. Связь между уровнями ограничена явными аргументами, передаваемыми с одного уровня на другой.
8. Недопустимо совместное использование несколькими уровнями глобальных данных.
9. Каждый уровень должен иметь более прочное и слабое сцепление с другими уровнями.
10. Всякая функция, выполняемая уровнем абстракции должна иметь единственный вход.

Принцип независимости ПО от внешних устройств. Принцип заключается в том, что связь программы с конкретными устройствами производится не на уровне трансляции программы, а в период планирования ее использования. При работе программы с новым устройством, перекомпиляция не требуется. Принцип реализуется в подавляющем большинстве ОС.

Принцип совместимости. Этот принцип определяет возможность выполнения ПО, написанного для другой ОС или для более ранних версий данной ОС. Различают совместимость на уровне исполняемых файлов и на уровне исходных текстов программ. В первом случае готовую программу можно запустить на другой ОС. Для этого требуется совместимость на уровне команд микропроцессора, на уровне системных и библиотечных вызовов. Как правило, используются специально разработанные эмуляторы, позволяющие декодировать машинный код и заменить его эквивалентной последовательностью команд в терминах другого процессора. Совместимость на уровне исходных текстов требует наличия соответствующего транслятора и также совместимости на уровне системных вызовов и библиотек.

Принцип открытости и наращиваемости. Открытость подразумевает возможность доступа для анализа как системным специалистам, так и пользователям. Наращиваемость подразумевает возможность введения в состав ОС новых модулей и модификации существующих. Построение ОС по принципу клиент-сервер с использованием микроядерной структуры обеспечивает широкие возможности по наращиваемости. В этом случае ОС строится как совокупность привилегированной управляющей программы и непривилегированных услуг-серверов. Основная часть остается неизменной, тогда как серверы могут быть легко заменены или добавлены.

Принцип мобильности (переносимости). Подразумевает возможность перенесения ОС с аппаратной платформы одного типа на платформу другого типа. При разработке переносимой ОС следуют следующим правилам: большая часть ОС пишется на языке, который имеет трансляторы на всех платформах, предназначенных для использования. Это язык высокого уровня, как правило, С. Программа на ассемблере в общем случае не является переносимой. Далее, минимизируют или исключают те фрагменты кода, которые непосредственно взаимодействуют с аппаратными ресурсами. Аппаратно-зависимый код изолируется в нескольких хорошо локализуемых модулях.

Принцип безопасности. Подразумевает защиту ресурсов одного пользователя от другого, а также предотвращения захвата всех системных ресурсов одним пользователем, включая и защиту от несанкционированного доступа. Безопасная система должна обладать конфиденциальностью, доступностью и целостностью. **Конфиденциальность** – это возможность доступа к данным только тем пользователям, которым этот доступ разрешен. **Доступность** – это гарантия того, что авторизованные пользователи всегда получат информацию, которая им необходима. **Целостность** – это невозможность модификации данных неавторизованными пользователями. При защите использую механизмы идентификации, аутентификации и авторизации. **Идентификация** – это сообщение пользователем своего идентификатора. Для проверки, что пользователь именно тот, за кого себя выдает (т.е. он предоставил действительно свой идентификатор), используется **аутентификация**. В простейшем случае для аутентификации используется пароль. Предоставление пользователю прав на доступ к объекту – это **авторизация**. Различают **избирательный** (дискреционный) и **полномочный** (мандатный) способ управления доступом. В первом случае определенные операции над конкретным ресурсом запрещаются или разрешаются пользователям либо группам пользователей. Во втором случае все объекты имеют уровень секретности, а пользователи делятся на группы в соответствии с уровнем допуска к информации. При этом обеспечиваются правила:
- простое правило секретности. Пользователь может читать информацию только из объекта, уровень секретности которого не выше уровня доступа пользователя. Т.е. генерал читает документы лейтенанта.
*-свойство. Пользователь может записывать информацию только в объекты, уровень секретности которых не ниже уровня доступа пользователя. Т.е. генерал может отправить секретную информацию маршалу или другому генералу, но не майору.

Наиболее известна Оранжевая книга безопасности (стандарт Министерства обороны США). Все системы делятся на 4 уровня безопасности: A, B, C, D. Уровень С делится на классы С1 и С2, уровень В – на классы В1, В2, В3. А является уровнем с максимальной защитой. Большинство современных ОС отвечают требованиям уровня С2. Он обеспечивает:

- средства секретного входа, позволяющие идентифицировать пользователя путем ввода уникального имени и пароля при входе в систему;

- избирательный контроль доступа, позволяющий владельцу ресурса определить, кто имеет доступ к ресурсу и его права;
- средства учета и наблюдения (аудита), обеспечивающие возможность обнаружения и фиксации событий, связанных с безопасностью системы и доступом к системным ресурсам;
- защита памяти, подразумевающая инициализацию перед повторным использованием.

На этом уровне система не защищена от ошибок пользователя, но его действия легко отслеживаются по журналу. Системы уровня B распределяют пользователей по категориям, присваивая определенный рейтинг защиты, и предоставляя доступ к данным только в соответствии с этим рейтингом. Уровень A требует выполнения формального, математически обоснованного доказательства соответствия системы определенным критериям безопасности. На уровне A управляющие безопасностью механизмы занимают до 90% процессорного времени. Сейчас используется и новый стандарт Common Criteria, а набор критериев Controlled Access Protection Profile примерно соответствует классу C2.

В ОС реализуется несколько подходов для обеспечения защиты. Одним из них является двухконтекстность работы процессора, т.е. в каждый момент времени процессор может выполнить либо программу из состава ОС, либо прикладную или служебную программу, не входящую в состав ОС. Для того, чтобы гарантировать невозможность непосредственного доступа к любому разделяемому ресурсу со стороны пользовательских и служебных программ, в состав машинных команд вводятся специальные привилегированные команды, управляющие распределением и использованием ресурсов. Эти команды разрешается выполнять только ОС. Контроль за их выполнением производится аппаратно. При попытке выполнить такую команду возникает прерывание, и процессор переводится в привилегированный режим. Для реализации принципа защиты используется механизм защиты данных и текста программ, находящихся в ОЗУ. Самым распространенным подходом при этом является контекстная защита. Для программ и пользователей выделяется определенный участок памяти, и выход за его пределы приводит к прерыванию по защите. Механизм контроля реализуется аппаратным способом на основе ограниченных регистров или ключей памяти. Применяются различные способы защиты хранения данных в файлах. Самый простой способ защиты – парольный.

Интерфейс между ОС и программами пользователя определяется набором **системных вызовов**, предоставляемых ОС. Если процесс выполняет программу в пользовательском режиме, и ему необходимо произвести при этом системный вызов, то происходит прерывание либо вызывается команда системного вызова для передачи управления ОС. Далее в зависимости от параметров, ОС определяет, что требуется вызывающему процессу. Далее обрабатывается системный вызов, и возвращает управление команде, следующей за системным вызовом. Системные вызовы выполняются в привилегированном режиме. В общем случае системный вызов может и блокировать вызвавшую его процедуру, например при чтении символа с клавиатуры. В UNIX системах библиотека libc обеспечивает C-интерфейс каждому системному вызову. В результате системный вызов для прикладного программиста ничем не отличается от вызова обычной библиотечной процедуры.

Интерфейс прикладного программирования (API - Application Program Interface) является более широким понятием, чем системные вызовы. Он представляет собой средства для использования прикладными программами системных ресурсов ОС и реализуемых ею функций. Соответственно, API может реализовываться на уровне ОС, представляя собой системные вызовы, на уровне систем программирования, как правило в виде библиотеки RTL. Кроме того, API могут реализовываться в виде внешних библиотек, например, MFC, VCL.

Платформенно независимый системный интерфейс для компьютерных сред, описываемый стандартом **POSIX** (Portable Operating System Interface for Computer Environments) – определяет минимальный набор системных вызовов для открытых ОС, базируясь на UNIX системах. Как правило, каждая UNIX система имеет и дополнительные, присущие только ей вызовы. POSIX описывает более 100 вызовов.

В отличие от UNIX, в Windows системные вызовы и запускающиеся для их выполнения библиотечные вызовы полностью разделены. Для вызова служб ОС используется набор процедур **Win32 API**. Количество вызовов в WinAPI составляет несколько тысяч, причем многие из них целиком работают в пространстве пользователя, а не ядра. Далее, в UNIX графический интерфейс пользователя GUI запускается в пространстве пользователя, тогда как WinAPI имеет огромное количество процедур для работы с графической оболочкой – управление окнами, меню, шрифтами и т.д. В большинстве версий Windows графическая система запускается в режиме ядра, и в этом случае соответствующие вызовы являются системными, в противном случае только библиотечными. В Win32API не существует понятия связанных файлов, монтирования файловой системы, сигналов. WinAPI для Windows систем на основе ядра NT 5.x (Windows 2000, Windows XP, Windows Server 2003) поддерживает вызовы POSIX, определенные стандартом POSIX.1, и имеет возможность монтирования файловой системы.

Основные системные вызовы:

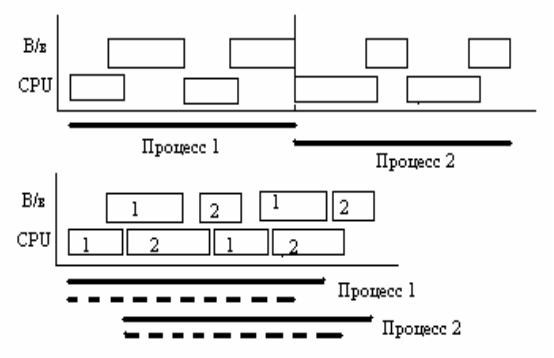
POSIX	**назначение**	**Win32 API**
fork	создать дочерний процесс, идентичный родительскому	CreateProcess (fork+execve)
waitpid	ожидать завершение дочернего процесса	WaitForSingleObject
execve	переместить образ памяти процесса	-
exit	завершить выполнение процесса	ExitProcess
open	открыть файл	CreateFile
close	закрыть файл	CloseHandle
read	чтение данных из файла в буфер	ReadFile
write	записьданных из буфера в файл	WriteFile
lseek	переместить указатель файла	SetFilePointer
stat	информация о состоянии файла	GetFileAttributesEx
mkdir	создать каталог	CreateDirectory

rmdir	удалить каталог	RemoveDirectory
link	создать новый элемент каталога, ссылающийся на другой	-
unlink	удалить элемент каталога	DeleteFile
mount	монтирование файловой системы	-
umount	демонтирование файловой системы	-
chdir	изменить рабочий каталог	SetCurrentDirectory
chmod	изменить биты защиты файла	-
kill	послать сигнал процессу	-
time	получить системное время	GetLocalTime

Понятие процесса и потока

Концепция процесса. Диаграмма состояний процесса. Операции над процессами. Создание и завершение процесса. Иерархия процессов. Структуры управления процессами. Процессы-зомби. Системные вызовы для управления процессами. Процессы в Windows и UNIX. Процессы и потоки. Понятия мультизадачности и многопоточности. Потоки в пространстве пользователя. Потоки в ядре. Облегченные потоки. Потоки в Windows и UNIX. Всплывающие потоки. Понятие о прерываниях. Параллельные процессы. Независимые и взаимодействующие процессы. Сигналы UNIX. Сообщения Windows

В первых системах программа могла выполняться только после завершения предыдущей. В полном соответствии с принципами, разработанными фон Нейманом, все подсистемы и устройства, в т.ч. управление памятью и в/в с внешними устройствами управлялись центральным процессором. Введение в состав системы специальных контроллеров позволило распараллелить эти процессы и непосредственно обработку данных. Был предложен **мультипрограммный** режим работы: пока одна программа (процесс) ожидает завершения операций ввода-вывода, другой может быть поставлен на обработку данных.



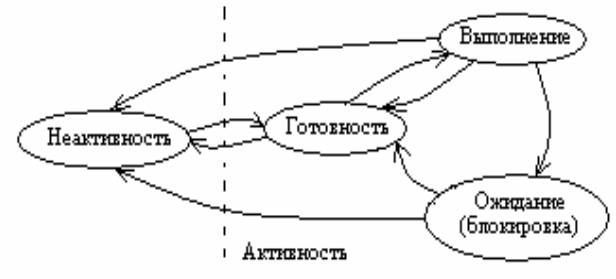
Общее время выполнения двух задач меньше, чем при последовательной обработке, однако время выполнения отдельной задачи несколько увеличивается. Т.е. при мультипрограммировании повышается пропускная способность системы, но отдельная задача не может быть выполнен быстрее, чем в однопрограммном режиме в силу затрат времени на ожидание освобождения ресурса. В такой многозадачной системе процессор переключается между задачами, предоставляя каждой из них от десятков до сотен миллисекунд, при этом в каждый конкретный момент времени процессор занят только одной программой, но за секунду он успевает поработать с несколькими, создавая у пользователя иллюзию параллельной работы.

Процесс – это выполняемая программа, включая текущие значения счетчика команд, регистров и переменных. С этой точки зрения, у каждого процесса есть собственный виртуальный процессор. При необходимости использования ресурса, процесс обращается с запросом на ресурс к супервизору ОС (центральному управляющему модулю), который может состоять из нескольких, например, супервизор в/в, супервизор прерываний и т.д. При этом указывается вид ресурса и при необходимости, параметры (объем памяти, например, или необходимость в монопольном использовании ресурса). Управление передается ОС, переводя процессор в привилегированный режим работы. Ресурс может быть выделен процессу, если:

- он свободен и в системе нет запросов на этот ресурс от задач с более высоким приоритетом
- текущий запрос и ранее выданные запросы допускают совместное использование ресурсов
- ресурс используется процессом с более низким приоритетом и может быть временно отобран (разделяемый ресурс)

Получив запрос, ОС либо удовлетворяет его и возвращает управление процессу, либо, если ресурс занят, ставит задачу в очередь, переводя ее в состояние ожидания. После использования ресурса процесс с помощью специального вызова супервизора сообщает об этом ОС, либо ОС сама забирает ресурс у процесса, если управление возвращается супервизору после вызова какой-либо системной функции. Супервизор освобождает ресурс и проверяет очередь. В зависимости от принятой дисциплины обслуживания и приоритета запросов, он выводит из состояния ожидания соответствующую задачу и переводит ее в состояние готовности. Управление передается либо этой задаче, либо той, которая только что освободила ресурс.

Диаграмма состояний процесса.



Процесс может находиться в **активном** и **пассивном** состоянии. В активном он может участвовать в конкуренции за использование ресурсов системы, а в пассивном – он только известен системе, а в конкуренции не участвует, хотя его существование в системе и связано с предоставлением ему памяти. Активный процесс может быть в одном из 3 состояний:

- **выполнение** – все затребованные процессом ресурсы выделены. В этом состоянии в однопроцессорной системе в каждый момент времени может находиться только один процесс
- **готовность** – ресурсы могут быть предоставлены, тогда процесс перейдет в состояние выполнения
- **ожидание** – ресурсы не могут быть предоставлены, либо не завершена операция ввода-вывода.

В большинстве ОС процесс возникает при запуске программы на выполнение. ОС выделяет для него соответствующий дескриптор, и процесс начинает выполняться. Состояние неактивности отсутствует. В ОС реального времени зачастую состав процессов известен заранее, в т.ч. и многие их параметры, поэтому с целью экономии времени, дескрипторы выделены заранее, а многие процессы находятся в пассивном состоянии.

За время существования процесс может неоднократно переходить из одного состояния в другое. Из состояния пассивности в состояние готовности процесс может перейти в следующих случаях:

- по команде оператора. Происходит в диалоговых ОС, где программа имеет статус задачи и может являться пассивной, а не просто быть исполняемым файлом
- при выборе из очереди планировщиком (характерно для пакетного режима)

- по вызову из другой задачи (с помощью обращения к супервизору один процесс может создать, инициировать, остановить, уничтожить другой процесс)
- по прерыванию от внешнего устройства
- при наступлении запланированного времени запуска задачи.

Процесс, который может исполняться, как только ему будет предоставлен процессор (в некоторых системах и память), находится в состоянии готовности. Ему выделены все необходимые ресурсы, кроме процессора. Из состояния выполнения процесс может выйти по следующим причинам:
- процесс завершается, с помощью супервизора передает управление ОС. Супервизор либо переводит его в список пассивных процессов, либо уничтожает. В пассивное состояние процесс может быть переведен и принудительно по команде оператора, либо при обращении к супервизору иного процесса с запросом на остановку данного процесса
- переводится супервизором в состояние готовности в связи с появлением более приоритетной задачи или по окончанию выделенного кванта времени
- процесс блокируется либо вследствие запроса операции ввода-вывода, либо из-за занятости ресурса, а также по команде оператора или по требованию супервизора от другой задачи. При наступлении соответствующего события (завершилась операция ввода-вывода, освободился ресурс и т.д. процесс деблокируется и переводится в состояние готовности.

Понятие последовательного процесса в ОС.
Все функционирующее на компьютере ПО, иногда включая ОС, организовано в виде набора последовательных процессов. Обычно при загрузке ОС создается несколько процессов. Одни из них являются высокоприоритетными, другие – фоновыми. Фоновые процессы, связанные с электронной почтой, web-страницами, новостями, выводом на печать, называют **демонами**. Если процесс может создавать другие процессы, и т.д., то образуется соответствующее дерево процессов. Процессы связаны, если они объединены для решения какой-либо задачи и им необходимо передавать данные от одного к другому и синхронизировать свои действия. Эта связь называется **межпроцессным взаимодействием**.

На каждый процесс выделяется специальная информационная структура – **дескриптор процесса**. В общем случае он содержит:
- идентификатор процесса PID – process identificator
- тип/класс процесса, который определяет для супервизора некоторые правила предоставления ресурсов
- приоритет процесса. В рамках класса в первую очередь обслуживаются процессы с более высоким приоритетом
- переменная состояния, определяющую, в каком состоянии находится процесс
- защищенную область памяти (ее адрес), в которой сохраняются текущие значения регистров процессора при прерывании выполнения процесса. Эта область называется контекстом задачи.
- Информацию о ресурсах, которыми владеет процесс или имеет право пользоваться (указатели на файлы, информация о незавершенных операциях в/в)
- Область памяти (адрес) для организации взаимодействия с другими процессами
- Параметры времени запуска (момент активации и периодичность этого)
- При отсутствии системы управления файлами – адрес задачи на диске в ее исходном состоянии и адрес, куда она выгружается, если ее вытесняет другая.

Дескрипторы, как правило, постоянно находятся в оперативной памяти с целью ускорения работы супервизора. Для каждого состояния ОС ведет соответствующий список процессов, находящихся в этом состоянии. При изменении состояния процесса, супервизор перемещает дескриптор из одного списка в другой. Для состояния ожидания может быть не один список, а несколько – по количеству соответствующих видов ресурсов. Часто в ОС заранее определяется максимальное количество дескрипторов задач. Существует и аппаратная поддержка дескрипторов задач. Так в процессорах i80x86 начиная с 80286 имеется регистр TR (task register), указывающий местонахождение сегмента состояния задачи TSS, где при переключении задач сохраняется содержание регистров.

В UNIX используются 2 структуры: u и proc. Ядро имеет массив структур proc, который называется таблицей процессов. Поскольку таблица находится в системном пространстве, она всегда доступна ядру. Область u является часть пространства процесса, т.е. видна только в момент выполнения процесса, и содержит данные, необходимые только в период выполнения процесса.

В UNIX существует только один системный запрос для создания процесса: **fork**. Этот запрос создает дубликат существующего, т.е. дочерний процесс полностью повторяет родительский процесс. Адресное пространство полностью повторяет адресное пространство родительского процесса, аналогичны и строки окружения, эти процессы имеют одни и те же открытые файлы. Все переменные имеют одинаковые величины во время вызова fork как у родительского, так и у дочернего процесса, но как только данные скопированы для дочернего процесса, дальнейшие изменения в одном из них уже не влияют на другой. Чтобы можно было отличить родительский процесс от дочернего, fork возвращает 0 для дочернего процесса и PID дочернего процесса для родительского, –1 в случае ошибки. Вызов fork выполняет следующие действия:
- резервирует пространство своппинга для данных и стека процесса-потомка
- назначает новый идентификатор PID и структуру proc потомка
- инициализирует структуру proc потомка. Часть полей копируются от родителя, часть устанавливается в 0, часть устанавливается в специфические для потомка значения
- размещает карты трансляции адресов для потомка
- выделяет область u потомка и копирует в нее содержимое области u родителя
- изменяет ссылки области u на новые карты адресации и своппигна
- добавляет потомка в набор процессов, разделяющих между собой область кода программы, выполняемой родителем.

- Постранично дублирует области данных и стека родителя и модифицирует карты адресации потомка в соответствии с этими новыми страницами
- Получает ссылки на разделяемые ресурсы, наследуемые потомком, такие как открытые файлы и рабочий каталог
- Инициализирует аппаратный контекст потомка копируя текущие состояния регистров родителя
- Процесс потомок становится выполняемым и помещается в очередь планировщика
- Для потомка установить возвращаемое значение в 0
- Для родителя возвращается PID потомка

Очевидно, что если после fork сразу использовать exec, то в копировании образа памяти родительского процесса нет необходимости. Один из подходов – метод копирования при записи. Страницы данных и стека родителя временно получают атрибут только для чтения и помечаются как копируемые при записи. Потомок получает собственную копию карт трансляции адресов, но использует те же самые страницы, что и родитель. Если любой из этих двух процессов попытается изменить страницу, произойдет исключительная ситуация, ядро запустит соответствующий обработчик, который, увидев пометку копирования при записи, создаст копию страницы, которую уже можно изменять. Таким образом создаются копии только тех страниц памяти, которые изменяются в одном из процессов. Второй подход – использование системного вызова vfork, который не копирует страницы памяти. Процесс родитель предоставляет свое адресное пространство потомку и блокируется до тех пор, пока потомок не выполнит exec или exit, после чего ядро вернет родителю его адресное пространство и переведет его из состояния ожидания.

Для изменения образа памяти и запуска новой программы дочерний процесс выполняет системный вызов **execve**. Вызов заменяет весь образ памяти процесса файлом, имя которого указано в первом параметре. Как правило, у вызова имеется еще два параметра: указатель на массив аргументов и указатель на массив переменных окружения. Например, при выполнении команды cp file1 file2 произойдет следующее: оболочка с помощью fork создаст дочерний процесс, далее с помощью exec будет помещен в память файл cp (код программы копирования), а вторым аргументом вызова будет указатель на массив параметров, содержащих имена файлов. Вызов выполняет следующие действия:

- Разбирает путь к исполняемому файлу и осуществляет доступ к нему
- Проверяет, есть у процесса полномочия на выполнение файла
- Читает заголовок, чтобы убедиться что файл исполняемый
- Идентификаторы пользователя и группы UID и GID изменяются на соответствующие владельцу файла
- Копируются передаваемые аргументы и переменные среды в пространство ядра, подготавливая текущее пользовательское пространство к уничтожению
- Выделяется пространство свопинга для областей данных и стека
- Освобождается старое адресное пространство и связанное с ним пространство свопинга
- Выделяются карты трансляции адресов для нового текста, данных и стека
- Устанавливается новое адресное пространство
- Аргументы и переменные среды копируются обратно в новый стек приложения
- Все обработчики сигналов сбрасываются в действия по умолчанию
- Инициализируется аппаратный контекст

Родительский процесс может ожидать окончания дочернего с помощью системного вызова **waitpid**. В качестве первого параметра выступает pid дочернего процесса, либо –1, если достаточно дождаться завершения любого из них. Вторым параметром передается указатель, который будет установлен на статус завершения дочернего процесса. Вызов **wait** тоже позволяет ожидать завершения работы потомка, однако без указания какого именно. Wait возвращает PID завершившегося процесса, освобождает его структуру proc, сохраняет его статус выхода в соответствующей переменной. Если работу завершили несколько дочерних процессов, то обработан будет только один из них. Пример кода создания дочернего процесса:

```
if (fork()!=0)
  {waitpid(-1,&status,0); }          // код родительского процесса
else execve (command, parameters,0);        // а это дочерний процесс
```

Вызов **exit** завершает процесс. При этом выполняется:

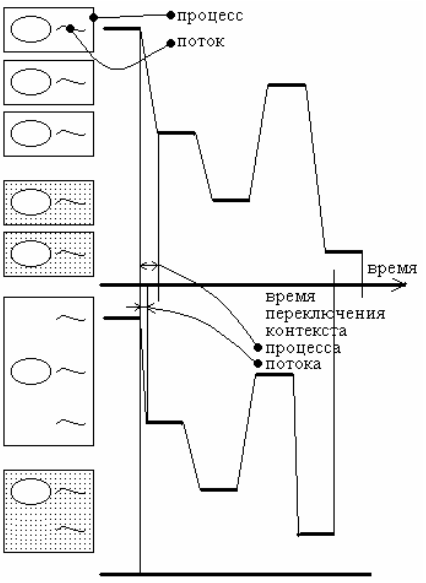
- Отключаются все сигналы
- Закрываются открытые файлы
- Освобождается файл программы и другие ресурсы
- Делается запись в журнал
- Сохраняются данные об использованных ресурсах и статус выхода в структуре proc
- Состояние процесса изменяется на SZOMB, его структура proc помещается в список процессов-зомби
- Все потомки процесса получают в качестве нового родителя системный процесс init
- Освобождается адресное пространство, область u, карты трансляции адресов и пространство свопинга
- Родителю процесса посылается сигнал SIGCHLD
- Будится родительский процесс
- Вызывается swtch для перехода к следующему процессу

После выполнения вызова процесс находится в состоянии **зомби**. В этом состоянии не освобождена структура proc, данные из которой могут понадобиться родителю. Соответственно родитель и отвечает за ее освобождение. Здесь существует проблема. Если родительский процесс завершен раньше дочернего, то дочерний усыновляется процессом init. Все в порядке. Если дочерний завершается раньше, а родитель не вызывает wait, который освобождает proc, то процесс-зомби так и останется. Эти процессы видны при вызове ps, однако завершить их невозможно, поскольку они уже завершены. Кроме того, они продолжают занимать структуру proc, число которых в таблице процессов как правило, ограничено.

В Windows и созданием процесса и запуском в нем нужной программы управляет функция **CreateProcess**. Родительский и дочерний процессы имеют собственные адресные пространства, различные изначально. Процессы в Windows идентифицируются как дескрипторами, так и идентификаторами процессов. **GetCurrentProcess** и **GetCurrentProcessId** возвращают дескриптор и идентификатор соответственно. Завершается процесс функцией **ExitProcess**. Ожидание завершения процесса или группы процессов выполняется функциями **WaitForSingleObject** и **WaitForMultipleObject**. Во втором случае происходит ожидание либо одного из указанных объектов либо всех. Выполнение дочернего процесса не зависит от родительского.
Функция CreateProcess выполняет следующие действия:

- открывается исполняемый файл
- если файл не является Windows-приложением, ищется образ поддержки (программа) для запуска этого приложения
- создается и инициализируется объект процесса исполнительной системы ОС
- создается первичный поток – стек, контекст и объект потока исполнительной системы
- подсистема Windows получает сообщение о создании нового процесса и потока
- начинается выполнение первичного потока
- в контексте нового процесса и потока инициализируется адресное пространство и начинается выполнение программы

Процессы и потоки.
Процесс можно рассматривать как способ объединения используемых ресурсов в одну группу. Процесс имеет свое виртуальное адресное пространство, ему назначаются ресурсы – файлы, окна, семафоры и т.д. Это позволяет защитить процессы друг от друга. ОС считает процессы совершенно несвязанными между собой. С другой стороны процесс можно рассматривать как поток исполняемых команд. У потока или нити (thread) есть счетчик команд, отслеживающий последовательность операций, регистры, хранящие текущие значения, стек, содержащий протокол выполнения. Концепция потоков добавляет к модели процесса возможность одновременного выполнения в одной и той же среде процесса нескольких в достаточной степени независимых программ. Для них ОС не требуется организовывать полноценную виртуальную машину. Они не имеют собственных ресурсов, пользуясь общими для процесса ресурсами. Единственный ресурс, который им необходим – это процессор. В однопроцессорной системе потоки разделяют между собой процессорное время точно так же, как и процессы, в мультипроцессорной могут выполняться одновременно.



Многопоточность обеспечивает параллельное выполнение нескольких видов операций в одной программе. Особо эффективно выполнение многопоточных приложений на распределенных системах. Т.о., процесс предполагает, что при диспетчеризации требуется учитывать все ресурсы, закрепленные за ним. При переключении между потоками достаточно изменять только контекст задачи, не затрагивая всех остальных ресурсов. Каждый процесс всегда имеет как минимум один поток. Каждый поток выполняется строго последовательно. Потоки, как и процессы могут порождать потоки-потомки. Аналогично процессам, поток может находиться в одном из активных состояний. Пока один поток заблокирован, другой может выполняться. Поскольку потоки одного процесса выполняются в одном и том же виртуальном адресном пространстве, между ними легче организовать тесное взаимодействие, в отличие от процессов, которым необходимы специальные механизмы обмена сообщениями и данными. Для потоков не существует прерываний по таймеру, позволяющему установить режим разделения времени, поэтому существует запрос, позволяющий потоку самостоятельно передать управление другому.

Возможны два основных варианта реализации потоков – на уровне пользователя и на уровне ядра. В первом случае у каждого процесса имеется своя собственная таблица потоков, в которой хранится информация, необходимая для переключения потока в состояние выполнения. Когда поток на уровне пользователя завершает на время свою работу, процедуре передачи управления нет необходимости использовать системные вызовы на уровне ядра, поскольку вся информация о потоках находится внутри процесса-хозяина. Соответственно, процедура может сама сохранить информацию в таблице потоков, более того, даже вызвать планировщик потоков для выбора следующего. Соответственно не требуется прерывание, переключение контекста, сохранение кэша и т.д., что дает значительное ускорение. Потоки на уровне пользователя позволяют каждому процессу иметь собственный алгоритм планирования потоков. Однако в общем случае, при блокировке одного потока блокируется весь процесс. Ядру ничего не известно о том, что приложение многопоточное, поэтому вся синхронизация при доступе к общим переменным из разных потоков должна быть выполнена на уровне пользователя. В Windows такие потоки называются **облегченными**.

При реализации на уровне ядра, таблица потоков единая для всех процессов. Ядро в общем случае может при блокировании потока выбрать новый, не принадлежащий текущему процессу. В терминологии UNIX такие потоки часто называют “легковесными процессами”. (LWP, lightweight process). В отличие от пользовательских потоков, при блокировке одного LWP остальные продолжают работать. Поскольку все потоки LWP планируются на выполнение ядром независимо друг от друга, но в отличие от полновесных процессов, разделяют общее адресное пространство, при доступе к переменным, используемым несколькими LWP, требуется применение специальных механизмов синхронизации на уровне ядра. Все запросы, которые могут блокировать поток, реализуются как системные, что увеличивает временные издержки. Чтобы их снизить некоторые системы после завершения потока не уничтожают его структуры, только помечая, как неработающий. При запросе на создание нового потока используются уже готовые структуры. В Windows именно потоки, а не процессы являются объектами диспетчеризации. Т.е. планировщик выбирает из очереди готовый поток, а не процесс.

Концепция потока не дает увеличения производительности в однопроцессорной системе, если все они ограничены возможностями процессора. Но если имеется потребность в выполнении большого объема вычислений и операций ввода-вывода, то потоки позволяют совместить эти действия во времени, увеличивая общую скорость работы. Для обработки входящих сообщений можно использовать концепцию **всплывающих потоков**. В обычном случае в процессе существует поток, который в обычном состоянии заблокирован, и активизируется с приходом сообщения. В случае всплывающего потока он создается с нуля с приходом сообщения. При этом нет необходимости в восстановлении контекста потока. Однако при этом необходимо предварительное планирование. Например, в каком процессе должен возникнуть новый поток? Он должен быть на уровне ядра или пользователя ? и т.д.

Потоки в Windows создаются функцией **CreateThread**. Завершить поток можно функцией **ExitThread**, либо возврат из функции потока с использованием кода завершения в качестве возвращаемого значения. Функции **GetCurrentThread** и **GetCurrentThreadId** позволяют получить дескриптор и соответственно идентификатор вызывающего потока, **OpenThread** позволяет получить дескриптор потока по известному идентификатору. Функции **SuspendThread** и **ResumeThread** позволяют приостановить поток и возобновить его выполнение. Ожидание завершения потока выполняется функциями **WaitForSingleObjects** и **WaitForMultipleObjects**. **CreateRemoteThread** позволяет создать поток в другом процессе.

POSIX описывает стандарт библиотеки **pthreads** для поддержки потоков. Поддержка включена в ряд реализаций UNIX и Linux. Соответственно создание и завершение потока соответствуют системные вызовы **pthread_create** и **pthread_exit**, для организации ожидания завершения потока **pthread_join**.

Стандартные библиотеки С исторически были рассчитаны на однопоточные системы. Поэтому многие библиотечные функции используют глобальные данные для хранения переменных. В результате потоки будут работать не каждый со своими данными отдельно, а с общими глобальными, что потенциально является источником проблем.

В архитектуру современных процессоров включена возможность работать с задачами (Task), объединяющими в себе понятие потока и процесса. Это при разработке ОС позволяет построить соответствующие дескрипторы как для процесса, так и для потока.

Прерывания. Основная причина изменения состояний процесса – события. Одним из видов событий являются прерывания. Прерывания представляют собой механизм, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие при работе процессора. Это принудительная передача управления от выполняемой программы к системе, происходящая при возникновении определенного события. Основная цель введения прерываний – реализация асинхронного режима работы и распараллеливание. Механизм прерываний реализуется программно-аппаратными средствами. Механизм обработки прерываний независимо от архитектуры системы включает следующие элементы:

- прием запроса на прерывание и его идентификация
- запоминание состояния прерванного процесса. Сохраняется счетчик команд, регистры процессора, а также другая информация.
- Передача управления подпрограмме обработки прерываний. В простейшем случае в счетчик команд заносится начальный адрес обработчика, а в регистры – информация из слова состояния.
- Сохранение информации о прерванной программе, не сохраненную ранее аппаратными средствами
- Обработка прерывания
- Восстановление информации прерванного процесса
- Возврат управления прерванной программе

Первые три шага реализуются аппаратно, остальные – программно.

Переход от прерываемой программы к обработчику должен быть максимально быстрым. Один из используемых подходов – использование таблиц, содержащих перечень всех допустимых прерываний и адреса соответствующих обработчиков. Содержимое регистров процессора запоминается либо в памяти с прямым доступом либо в системном стеке. Прерывания, возникающие в системе, можно разделить на внутренние и внешние.

Внешние прерывания вызываются асинхронными событиями, происходящими вне прерываемого процесса, например:

- прерывание от таймера
- прерывание от ВУ на ввод-вывод
- прерывание по нарушению питания
- прерывание от оператора
- прерывание от другого процесса или системы

Внутренние прерывания вызываются событиями, связанными с работой процессора и синхронны относительно его операций:

- нарушение адресации (используется запрещенный или несуществующий адрес, обращение к отстутствующей странице виртуальной памяти)
- ошибочный код операции (неизвестная процессору команда)
- деление на ноль
- переполнение
- обнаружение ошибок четности и пр.

Существуют программные прерывания. Происходят по соответствующей команде прерывания. Были введены для того, чтобы переключение на системные модули происходило не как переход к подпрограмме, а тем же самым образом, как и при асинхронных прерываниях. Этим самым обеспечивается переключение процессора в привилегированный режим с возможностью исполнения любых команд.

Сигналы, вызывающие прерывания, могут возникать одновременно. Выбор одного из них происходит в зависимости от приоритета. Прерывания от схем контроля процессора имеют наивысший приоритет, программные прерывания – самый низкий. Учет приоритетов может быть как аппаратным, так программным на уровне ОС.

В общем случае процессор обладает системой защиты от прерываний: отключение механизма, маскирование отдельных сигналов и т.п., что позволяет ОС регулировать их обработку. Обычно операция прерывания начинается только после выполнения текущей команды процессора, даже если оно возникло асинхронно в процессе ее выполнения. Программное управление позволяет реализовать разные дисциплины обслуживания прерываний:

- с относительными приоритетами. Обслуживание не прерывается, даже если пришел запрос на прерывание с более высоким приоритетом. После окончания обслуживания текущего запроса обслуживается запрос с наивысшим приоритетом. Для организации этого либо отключается система прерываний, либо накладываются маски на все сигналы.
- С абсолютными приоритетами. Всегда обслуживается прерывание с наивысшим приоритетом. Для реализации маскируются все запросы с низшим приоритетом. При этом возможно многоуровневое прерывание.
- По принципу стека. Любой приходящий запрос прерывает текущее обслуживание. В этом случае маски не накладываются.

Во время сохранения контекста задачи и затем последующего его восстановления система прерываний должна быть отключена. В процессе выполнения собственно обработчика прерывания, система функционирует в соответствии с организуемой дисциплиной обслуживания. Очень часто сохранение контекста задачи возлагается на супервизор прерываний. В его функции входит: сохранение в дескрипторе текущей задачи рабочие регистры процессора (контекст задачи), определение адреса необходимого обработчика, установление необходимого режима обслуживания прерываний и передача управления обработчику. После выполнения, обработчик вновь передает управление супервизору, на этот раз на модуль, управляющий диспетчеризацией задач. И уже диспетчер задач, в соответствии с принятым режимом распределения процессорного времени, восстановит контекст той задачи, которой будет предоставлен процессор. Т.е. возврата в прерванную программу может и не произойти.

Особенностью мультипрограммных ОС является то, что в их среде параллельно развивается несколько последовательных процессов. **Параллельные процессы** – это последовательные вычислительные процессы, которые одновременно находятся в каком-либо активном состоянии. Они могут быть независимыми либо взаимодействующими. Независимыми являются процессы, множество данных (переменных и файлов) которых не пересекается. Независимые процессы не влияют на работу друг друга. Взаимодействующие процессы совместно используют некоторые общие переменные и выполнение одного процесса может повлиять на выполнение другого.

В UNIX для оповещения процесса о возникновении системных событий используются **сигналы**. Они же могут использоваться и как средство синхронизации и обмена данными. Сигналы позволяют вызвать какую-либо процедуру при возникновении события из их определенного набора. Процесс состоит из двух этапов – генерирования и доставки. Число поддерживаемых сигналов в различных системах различно. POSIX определяет стандартные символические имена для наиболее часто используемых сигналов. Каждый сигнал обладает некоторым действием, которое производится ядром системы, если процесс не имеет альтернативного обработчика. Их всего пять:

- аварийное завершение abort. Завершает процесс, создавая дамп состояния процесса, который может использоваться в дальнейшем при отладке и т.д.
- выход exit. Завершает процесс без создания дампа
- игнорирование ignore. Игнорировать сигнал
- Остановка stop. Приостанавливает процесс
- Продолжение continue. Возобновляет работу приостановленного процесса

Процесс может переопределить действия по умолчания для любого сигнала, в т.ч. и запуск определенной в приложении фйнкции – обработчика сигнала. Процесс может временно блокировать сигнал. Сигналы SIGKILL и SIGSTOP являются специальными, и процесс не может их блокировать, игнорировать или определять собственные обработчики. Основными источниками сигналов являются:

- исключительные состояния,
- другие процессы
- прерывания от терминала, например при нажатии определенной комбинации клавиш
- управление заданиями, это извещение родителя о завершении или приостановке дочернего процесса, управление фоновыми и текущим процессами;
- квоты, при превышении процессом временных квот или размера файла
- уведомления, например, о готовности устройства в/в
- будильники.

Для поддержки механизма сигналов в структуре u содержатся следующие поля:

u_signal[] вектор обработчиков для каждого сигнала
u_sigmask[] маски сигналов, ассоциированных с каждым обработчиком
u_sigaltstack указатель на альтернативный стек сигнала
u_sigonstack маска сигналов, обрабатываемых с альтернативным стеком
u_oldsig набор обработчиков для имитации базового механизма сигналов

Структура proc также содержит несколько полей :

p_cursig текущий сигнал, обрабатываемый в данный момент
p_sig маска ожидающих сигналов
p_hold маска блокируемых сигналов
p_ignore маска игнорируемых сигналов

Перечень некоторых типовых сигналов:

Сигнал	**описание**	**действие по умолчанию**
SIGABRT	процесс аварийно завершен	abort
SIGALRM	сигнал тревоги реального времени	exit
SIGCHLD	потомок завершил работу или приостановлен	ignore
SIGCONT	возобновить приостановленный процесс	ignore
SIGFPE	арифметическая ошибка	abort
SIGILL	недопустимая инструкция	abort
SIGINT	прерывание терминала	exit
SIGKILL	завершить процесс	exit
SIGPIPE	запись в канал при отсутствии считывающих процессов	exit
SIGQUIT	выход из терминала	abort
SIGSEGV	ошибка сегментации	abort
SIGSTOP	остановить процесс	stop
SIGSYS	неверный системный вызов	exit
SIGTERM	завершить процесс	exit
SIGUSR1	определяется процессом	exit
SIGUSR2	определяется процессом	exit

В Windows для оповещения процессов о возникновении каких-либо событий используется концепция **сообщений**. Существует около 900 различных типов сообщений. Каждый тип сообщений имеет свой числовой идентификатор. В случаях, когда пользователь выбирает пункт меню, щелкает на кнопке, или выбирает из списка и т.д. формируется сообщения специального типа WM_COMMAND или WM_NOTIFY. Сообщения этих типов принято называть **командами**. Процесс может передавать сообщения с помощью функции Win32 API **PostMessage** или **PostThreadMessage**, в первом случае указывается, какому окну передается сообщение, во втором – какому потоку. Остальные параметры указывают собственно сообщение, и дополнительную информацию. Сообщение помещается в специальную очередь сообщений, связанную с потоком, которому они посланы. Получить сообщение из этой очереди можно функцией **GetMessage**. Идентификаторы от 0 до WM_USER – 1 используются для сообщеинй Windows, от WM_USER до 0x7FFF, от 0x8000 до 0xBFFF зарезервированы для последующего использования Windows, от 0xC000 до 0xFFFF строковые сообщения пользователя, от 0xFFFF и выше – зарезервированы Windows.

Диспетчеризация процессов

Стратегии планирования. Дисциплины диспетчеризации. Вытесняющие и невытесняющие алгоритмы. Алгоритмы планирования без переключений. Циклическое и приоритетное планирование. Динамические приоритеты. Планирование в системах реального времени. Планирование потоков. Гарантии обслуживания процесса

Существует задача такой организации работы параллельных процессов, при которой они как можно реже конфликтуют из-за имеющихся в системе ресурсов. Эта задача называется **планированием**. Сейчас наиболее актуальны задачи динамического (краткосрочного) планирования. Эти задачи называются **диспетчеризацией**. Планирование осуществляется значительно реже, чем задачи текущего распределения ресурсов между уже выполняющимися процессами. При долгосрочном планировании планировщик решает, какой из процессов, находящихся во входной очереди, должен быть переведен в очередь готовых процессов в случае освобождения ресурсов памяти. При этом он пытается спланировать активные процессы таким образом, чтобы в списке готовых процессов находились как процессы, занятые преимущественно вводом-выводом, так и процессы, занятые преимущественно вычислениями. Краткосрочный планировщик решает, какая из задач в очереди готовых должна быть передана на исполнение. В современных ОС долгосрочный планировщик зачастую отсутствует. Для обычных персональных компьютеров планирование не играет важной роли, поскольку большую часть времени активен только один процесс, а процессорное время перестало быть дефицитным ресурсом. Для рабочих станций и серверов планирование играет важную роль, поскольку доступ к процессору пытаются получить одновременно несколько процессов. Помимо выбора задачи для выполнения планировщик должен решать и задачу эффективного использования процессора, поскольку переключение контекста требует затрат: переключение из режима пользователя в режим ядра, сохранение состояния текущего процесса, карту памяти (признаки обращения к страницам памяти), запуск следующего процесса, а также в большинстве случаев перезагрузка кэша.

Все процессы можно разделить на 2 большие группы: процессы, занятые в основном вычислениями, и изредка требующими операций ввода-вывода (ограниченные возможностями процессора), и процессы, большую часть времени ожидающими ввода-вывода (ограниченные возможностями устройств в/в). С увеличением быстродействия процессоров, процессы все более смещаются в сторону задач, ограниченных возможностями устройств вв.

Планирование необходимо в следующих случаях:
1. Создание нового процесса. Требуется принятие решения, какой процесс – родительский или дочерний запускать.
2. Завершение процесса. Необходимо выбрать из очереди готовых процессов.
3. При блокировке процесса. Требуется выбор, какой процесс следующий. Иногда причина блокировки может влиять на выбор. Если А – высокоприоритетный процесс, и он блокируется в ожидании выхода процесса В из КС, то можно запустить процесс В, чтобы быстрее продолжил работу А.
4. При прерывании в/в. Если прерывание пришло от устройства, завершившего операцию, можно запустить процесс, ожидающий именно этого устройства.

Стратегия планирования определяет каким образом следует выбирать процессы в очереди, чтобы достичь оптимальной эффективности при выполнении параллельных процессов. Три наиболее известных стратегии следующие:
- по возможности процессы завершаются в том же самом порядке, в каком они были начаты.
- отдавать предпочтение коротким процессам
- всем процесса предоставлять одинаковое время ожидания.

Дисциплина диспетчеризации – совокупность правил, в соответствии с которыми формируется очередь готовых к выполнению задач. Дисциплины диспетчеризации могут быть разбиты на 2 класса: **вытесняющие и невытесняющие**. Если после выбора процесса он работает вплоть до блокировки или пока сам не отдаст управление другому процессу, это невытесняющая многозадачность. Другими словами, используются алгоритмы планирования без переключений. В случае, когда принудительно средствами ОС выполняется переключение между задачами, речь идет о вытесняющей многозадачности. При этом используются алгоритмы с переключениями. В современных ОС реализуется как правило вытесняющая многозадачность. Процесс выбирается и ему предоставляется какое-то количество времени. Если за этот отрезок процесс не завершил работы, он приостанавливается, и запускается другой процесс. Как правило, для организации переключений используется прерывание от таймера. В случае вытесняющей многозадачности механизм диспетчеризации целиком сосредоточен в ОС, что снимает с программиста обязанность о передаче управления другим процессам системы, т.е. он может программировать задачу так, как если бы задача запускалась в однопроцессной системе. При невытесняющей многозадачности управление системой может теряться на некоторый (в общем случае на достаточно большой) период времени, величина которого в значительной мере зависит от степени эффективной передачи управления иным процессам. Примером такой системы может служить Windows 3.x.

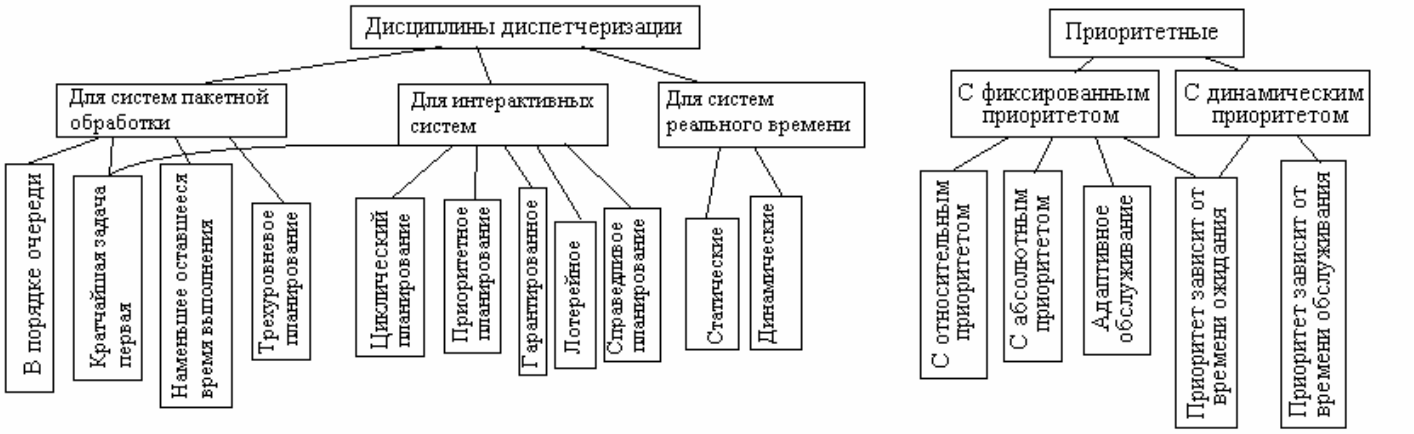
Для сравнения алгоритмов диспетчеризации используют следующие **критерии**:
- Загрузка центрального процессора. В большинстве персональных систем средняя загрузка процессора не превышает 2-3%, на серверах – до 20-40%
- Пропускная способность. Измеряется количеством выполненных процессов за час
- Среднее время оборота. Время от момента появления процесса во входной очереди до его завершения. Включает в себя время ожидания во входной очереди, время ожидания в очереди готовых процессов, время ожидания устройств вв, время выполнения.
- Время ожидания. Суммарное время нахождения процесса в очереди готовых процессов

- Время отклика. Для интерактивных программ является важным показателем. Время, прошедшее от момента поступления команды до получения результата.

Выбор конкретного алгоритма определяется классом решаемых задач и **целями**, которых стараются достичь. К таким целям относятся:
- справедливость. Гарантия того, что каждому процессу будет предоставлена определенная часть времени процессора, причем без возникновения ситуации голодовки.
- Эффективность. Процессор должен быть максимально нагружен. В идеальном варианте на 100%.
- Сокращение полного времени выполнения
- Сокращение времени ожидания
- Сокращение времени отклика.

Кроме того, желательно, чтобы алгоритмы обладали следующими **свойствами**:
- предсказуемость. Одно и то же задание должно выполняться примерно за одинаковое время
- минимизация накладных расходов, т.е. время на выбор очередного процесса, переключение контекста и т.д.
- равномерная загрузка ресурсов системы
- масштабируемость, т.е. при увеличении нагрузки не должна теряться работоспособность алгоритма.

Классификация дисциплин диспетчеризации:
Дисциплины диспетчеризации удобно классифицировать по типу оптимизации под конкретный класс задач: для систем пакетной обработки данных, для интерактивных систем, для систем реального времени.



Диспетчеризация FCFS (First Come First Served) – в порядке очереди. Самый простой вариант, использует невытесняющие алгоритмы (без переключений). Задачи обслуживаются в том порядке, в котором они возникли. Как правило, формируется общая очередь задач. Задачи, которые были заблокированы в процессе выполнения, после окончания ожидания (блокировки) ставятся в очередь перед теми процессами, которые еще не выполнялись либо в конец очереди. В итоге реализуется стратегия завершения процессов в том порядке, в котором они были начаты. Дисциплина не требует внешнего вмешательства в процесс вычислений, не происходит перераспределение процессорного времени. Эта дисциплина достаточно просто реализуются, требует малых затрат на реализацию очереди задач. Однако при увеличении нагрузки на систему, среднее время ожидания обслуживания возрастает. При этом короткие задачи ожидают столько же, сколько и длинные. Например, запускаются три задачи, время выполнения которых составляет 13, 4, 1 квант машинного времени. Тогда время ожидания процессов составит: 0, 13, 13+4=17 квантов, а полное время выполнения 0+13=13, 13+4=17, 17+1=18 квантов, соответственно среднее время ожидания 0+13+17=30/3=10 квантов, среднее время выполнения 13+17+18=48/3=16 квантов. Т.о. среднее время ожидания и среднее полное время выполнения зависят от порядка расположения процессов, поэтому алгоритм практически неприменим в системах разделения времени – т.к. среднее время отклика велико.

Дисциплина SJN (shortest job next) – следующим выполняется кратчайшее задание. Для задач должна быть известна оценка потребностей в машинном времени. Необходимость сообщать ОС о потребности задач в машинном времени привела и к появлению соответствующих языков, например, JCL (job control language). Пользователи указывали предполагаемое время, а чтобы не было явно заниженных оценок, использовался подсчет реальных потребностей. Диспетчер сравнивал заявленное время и расчетное, и если оценка явно занижалась, задача попадала не в начало, а в конец очереди. Заблокированные в процессе выполнения задачи попадают в конец очереди готовых к выполнению задач. Невытесняющая. Для того же примера: время ожидания составит 0, 1, 1+4=5, полное время выполнения 0+1=1, 1+4=5, 5+13=18, среднее время ожидания 0+1+5=6/3=2, среднее время выполнения 1+5+18=24/3=8.

Дисциплина SRT (shortest remaining time) – следующее задание требует наименьшего времени для завершения. В отличие от предыдущего случая, после блокировки задача может попасть в начало очереди, если для завершения требует минимум времени. При поступлении новой задачи ее время выполнения сравнивается с оставшимся временем до завершения текущей задачи, и если пришедшая задача короче, текущий процесс останавливается, а управление переходит новой задаче. Невытесняющая.

Трехуровневое планирование. По мере поступления новые задачи сначала помещаются в очередь на диске. Планировщик выбирает задание и передает его системе, остальные остаются в очереди. При этом может быть использован любой алгоритм

выбора задачи из очереди. Попав в систему, для задачи запускается соответствующий процесс, и он конкурирует за доступ к процессору. При большом количестве процессов часть из них сбрасывается на диск. Это второй уровень планирования – какие из процессов сбросить, а какие оставить. Этим занят планировщик памяти. Третий уровень планирования отвечает собственно за доступ процессов к процессору.

Дисциплина RR (Round Robin). Циклическая. Предполагает, что каждый процесс получает время порциями (квантами). После окончания выделенного кванта времени (или при блокировке процесса), задача снимается с выполнения и запускается следующая. Снятая задача ставится в конец очереди готовых задач. Рассмотрим тот же пример, при условии, что квант времени постоянен и равен 4.

0(13)	+	+	+	+						+	+	+	+	+	+	+	+
1(4)					+	+	+	+									
2(1)									+								

Время ожидания 5, 4, 8. Полное время исполнения – 5+13=18, 4+4=8, 8+1=9. Среднее время ожидания 5+4+8=17/3=5.67. Среднее полное время выполнения 18+8+9=35/3=11.67. Величина кванта влияет на производительность. Пусть квант равен 1:

0(13)	+			+		+		+		+	+	+	+	+	+	+	+
1(4)		+			+		+		+								
2(1)			+														

Время ожидания 5, 5, 2. Полное время исполнения – 5+13=18, 5+4=9, 2+1=3. Среднее время ожидания 5+5+2=12/3=4. Среднее полное время выполнения 18+9+3=30/3=10. Для оптимальной работы системы требуется правильно выбрать закон, по которому распределяются кванты времени. Величина кванта выбирается как компромисс между приемлемой реакцией системы на действия пользователя и накладными расходами на смену контекста задачи. В случае, если квант достаточно мал, реакция системы будет высокой, однако частая смена контекста снизит производительность системы. Если же квант велик, то при высокой производительности значительно снижается реакция системы. В некоторых ОС величина кванта указывается явно. Например, OS/2 имела переменную в системном файле конфигурации TIMESLICE, которая позволяла указывать минимальную и максимальную величину кванта. Если процесс прервался из-за окончания кванта времени, то новый выделяемый ему квант будет увеличен на время одно периода таймера, и так до тех пор, пока она не сравняется с максимальной. Это позволяет эффективнее распределять время для длительных задач. Циклическая дисциплина одна из самых распространенных.

Приоритетное планирование. В случае бесприоритетного обслуживания выбор задачи производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания. В случае с приоритетом у каждой задачи есть приоритет, в зависимости от которого она с большей или меньшей частотой (вероятностью) попадает в выполнения. Если приоритет задач не изменяется со временем, то это диспетчеризация с фиксированными приоритетами, если же он изменяется в процессе выполнения задачи, то это диспетчеризация с динамическим приоритетом. Этот вариант требует дополнительных временных затрат на расчет приоритетов, зато позволяет обеспечить гарантированное обслуживание процесса. Фиксированные приоритеты часто используются в ОС реального времени.

Использование динамических приоритетов.
Часто используется две составляющих приоритета – первая, заданная жестко при создании процесса, и вторая, формируемая диспетчером задач, и изменяемая в зависимости от текущей ситуации. Чтобы высокоприоритеные процессы не работали постоянно, не предоставляя время низкоприоритетным, с течением времени, по прошествии каждого кванта, приоритет процесса снижается. В то же время, периодически ядро пересчитывает текущие приоритеты процессов, готовых к запуску, увеличивая их. В итоге приоритет текущего процесса оказывается меньше, чем у одного из очереди, и происходит переключение. При динамическом назначении приоритетов для процессов, активно использующих устройства вв, т.е. требующих мало процессорного времени, приоритет может выбираться как 1/f, где f – часть использованного времени кванта, т.е. если процесс использовал 1/10 часть кванта, то приоритет назначается 10, если 1/25, то 25. Часто удобно группировать приоритеты по классам, используя приоритетное планирование между классами, но циклическое внутри класса. Пока в классе с высшими приоритетами есть задачи, они запускаются согласно циклическому планированию. Если их нет, обрабатывается очередь процессов более низкого приоритета. Во всех ОС имеются средства для изменения приоритета процесса.

Многоуровневые очереди с обратной связью. Развитие систем с динамическим приоритетом, в которых задачи разбиваются на классы приоритетов. В данном методе задачи могут перемещаться между классами. Например, класс с наивысшим приоритетом имеет квант, равный 8. Следующий по приоритетности класс – 16, следующий – 32, и последний класс обслуживается не циклически, а в порядке очереди. Задача изначально поступает в класс с максимальным приоритетом. Если процесс отработал эти кванты и требует еще времени, он переводится в менее приоритетный класс, если и там ему не хватило времени, то в еще менее приоритетный класс. В итоге слишком длинные задачи попадут в последний класс. В результате, чем длиннее процесс, тем в более низкий по приоритету класс он попадет, в результате время ожидания увеличивается, но зато количество предоставляемых квантов увеличивается. При завершении ожидания (например, от устройств в/в) процессы могут помещаться в более приоритетный класс задач.

Гарантированное планирование. При этом варианте гарантируется предоставление процессу определенной доли процессорного времени, тогда как в системе с приоритетами нет гарантии обслуживания.

Лотерейное. Процессам предоставляются “лотерейные билеты” на доступ к различным ресурсам, в т.ч. и к процессору. Когда планировщику необходимо принять решение, случайным образом выбирается лотерейный билет, и его обладатель получает доступ. Для важных процессов раздается больше лотерейных билетов, повышая шанс на выигрыш. В итоге каждый про-

цесс в среднем получает такую долю ресурса, какая доля билетов у него находится. Взаимодействующие процессы могут обмениваться лотерейными билетами. Например, если клиент посылает запрос серверу и блокируется в ожидании, он может предварительно передать все билеты серверу, чтобы ускорить получение ответа. Сервер, выполнив запрос, возвращает лотерейные билеты.

Справедливое. Все предыдущие стратегии не учитывают, что многие процессы могут быть созданы одним и тем же пользователем. Т.о., если один пользователь создал несколько процессов для решения одной задачи, а другой –только один, то первый получит и большую часть процессорного времени, что не справедливо. Поэтому перед планированием можно обращать внимание на хозяина процесса.

В системах реального времени время является самым важным критерием. Задача разделяется на несколько процессов, каждый из которых предсказуем, как правило это очень короткие процессы. При этом могут возникать внешние сигналы, как периодические (с определенной частотой), так и непериодические. Может случиться так, что система не в состоянии обработать все возникшие события за необходимое время. Если в систему поступает m периодических событий, событие i поступает с периодом Pi, и обрабатывается за Ci секунд, то все потоки могут быть обработаны только при следующем условии:
Системы, удовлетворяющие этому условию, называются **планируемыми**. В случае статических алгоритмов планирования все решения планирования приняты заранее, до запуска системы. Во втором случае решения принимаются по мере функционирования системы. Статическое возможно только при наличии достоверной информации о работе и временных ограничениях. Динамическое планирование не требует этого.

$$\sum_{i=1}^{m} Ci / Pi \leq 1$$

Основная проблема при диспетчеризации – это гарантия обслуживания, поскольку, например, при приоритетном планировании низкоприоритетные процессы могут бесконечно долго ожидать своей очереди. Более жестким требование является не просто гарантия обслуживания, а обслуживание за указанный период времени или к указанному моменту.

Гарантировать обслуживание можно следующими способами:
- Выделять минимальную долю процессорного времени некоторому классу процессов, если по крайней мере один из них готов к выполнению
- Выделять минимальную долю процессорного времени некоторому конкретному процессу, если он готов к выполнению
- Выделять столько процессорного времени готовому процессу, чтобы он мог завершить работу к указанному моменту времени

Производительность системы уменьшается в основном из-за затрат на переключение контекста и непродуманной системы назначения квантов времени. Особенно велики затраты на переключение контекста в том случае, если прерывается процесс, находящийся в КС, а другие при этом находятся в состоянии активного ожидания. Для повышения производительности применяются следующие методы:

- совместное планирование, при котором все потоки одного приложения одновременно выбираются для выполнения процессорами и одновременно снимаются с них:
- планирование. при котором находящиеся в КС процессы не прерываются, а активно ожидающие задачи не запускаются пока КС не освободится
- планирование с учетом подсказок ОС, когда ОС сообщает пользователю о необходимости снятия или запуска какого-либо процесса.

Рассмотрим потоки на уровне пользователя. Пусть есть 2 процесса, у каждого из них по три потока. Ядро не знает о существовании потоков пользователя, выполняет обычное планирование, выбирает процесс и предоставляет ему время. Планировщик потоков внутри процесса выбирает поток и запускает его. Если потоку требуется время, меньшее, чем выделенный квант, он отработает и вернет управление планировщику потоков. Если нет – будет работать до конца выделенного кванта. Т.о. в течение кванта времени могут быть запущены потоки по следующей схеме: A1, A2, A3, A1, A2, A3… В случае потоков на уровне ядра, ядро выбирает не процесс, а поток, не беспокоясь о том, какому процессу этот поток принадлежит. В результате цепочка запуска потоков может выглядеть так: A1, B1, A2, B2, A3, B3…, что невозможно на уровне пользователя. Однако на уровне пользователя для переключения потоков требуется несколько машинных команд, тогда как для переключения потоков на уровне ядра требуется полное переключение контекста с заменой карты памяти и перегрузкой кэша. Поскольку ядро знает, что при переключение от потока процесса А к потоку процесса В будет затрачено больше времени за счет карты памяти и кэша, это может учитываться. Например, при наличии двух одинаково важных потоков, один из которых принадлежит текущему процессу, а другой – нет, логичнее запустить поток текущего процесса.

ОС UNIX. Ядро в традиционной системе UNIX является невытесняющим. Если процесс выполняется в режиме ядра, то он не может быть прерван. Имеется два поля дескриптора процесса – p_nice и p_cpu. Первое назначается пользователем явно или формируется системой программирования. Второе – это текущий приоритет, изменяемый диспетчером задач. Процессам, выполняющимся в режиме задачи назначается приоритет ниже, чем для задач режима ядра. Например, для режима задач диапазон приоритетов может быть 0-65, для режима ядра 66-95, а диапазон 96-127 используется для процессов с фиксированным приоритетом для поддержки приложений реального времени. Процессу, ожидающему доступ к ресурсу, система определяет приоритет сна, выбираемый ядром из диапазона системных приоритетов и связанный с событием вызвавшим состояние ожидания. Поскольку этот приоритет из системного диапазона, то после активизации процесса вероятность запуска его на выполнение высока. После отработки с таким приоритетом, восстанавливается сохраненный ранее приоритет задачи, что снижает приоритет процесса. С каждым тиком таймера текущий приоритет активной задачи снижается на 1. Ежесекундно ядро пересчитывает приоритет процессов режима задачи, готовых к запуску. Планировщик содержит несколько очередей, каждая из которых соответствует 4 соседним приоритетам. Процесс с наивысшим приоритетом запускается всегда, если только текущий процесс не выполняется в режиме ядра.

WIN NT 5.x. Здесь каждый поток имеет базовый приоритет, который может быть в пределах +/-2 уровня относительно родительского процесса. Диспетчер направляет на выполнение потоки с высоким приоритетом раньше потоков с низким приоритетом. Текущий поток прерывается, если появляется готовый к выполнение поток с более высоким приоритетом. Для каждого уровня приоритета существует своя очередь. Приоритеты от 16 до 31 – это потоки реального времени. Диспетчер задач просматривает очереди, начиная с самой приоритетной. Если она пуста, просматривается следующая. Для системных модулей, работающих в режиме задач, зарезервирована очередь с номером 0. Большинство пользовательских потоков имеют уровень приоритета 1-15. Для этих очередей, диспетчер по окончании кванта времени снижает приоритет на 1 и перемещает в другую очередь. В тоже время при выходе из состояния ожидания диспетчер увеличивает приоритет потока. По умолчанию величина кванта – 2 тика, для систем Server – 12. Ключ реестра Hkey_Local_Machine\System\CurrentControlSet\Control\Priority Control\Win32PrioritySeparation позволяет настраивать величину кванта. Два старших бита определяют, короткие или длинные кванты (2 или 12 тиков – значения 1 и 2 соответственно), 0 и 3 интерпретируется как по умолчанию. Два средних бита указывают, переменные или фиксированные кванты у активного процесса. 1 – переменные, 2 – фиксированные, 0 или 3 – по умолчанию (переменные, Server - фиксированные). Младшие два бита указывают величину приращения кванта активного процесса – 0,1,2. Значение 3 интерпретируется как 2. Величина (в тиках) определяется по таблице:

	Короткие кванты			**Длинные кванты**		
	0	**1**	**2**	**0**	**1**	**2**
Фиксированные	6	6	6	12	12	12
Переменные	2	4	6	4	8	12

OS/2.Имеет 4 класса задач. У каждого класса своя группа приоритетов с номерами 0-31. Задачи наивысшего приоритета называются критическими. Это задачи реального времени. Типичный пример – обслуживание каналов связи, например, последовательный порт, сеть и т.д. Второй класс называется приоритетный или серверный. К этому классу относятся задачи, которые по отношению к другим играют роль сервера. Их приоритет должен быть выше, чтобы запросы к серверу выполнялись достаточно быстро. Третий класс –регулярный или стандартный. Это обычные пользовательские задачи. ; класс – остаточный, это фоновые задачи. Они получают время только тогда когда нет задач из других классов. Внутри каждого класса обслуживание происходит по циклической дисциплине. Система сама меняет приоритет задачи 1. увеличение приоритета, когда задача становится активной, 2. По завершении операции вв задача получает самый высокий приоритет своего класса 3. увеличение приоритета забытых задач. Если задача не получает управление в течение времени большем чем указано в MAXWAIT, задача временно получает приоритет вплоть до уровня задач 1 класса, по истечении кванта времени, приоритет задачи восстанавливается. Такая схема гарантирует, что за время MAXWAIT задача запустится хотя бы 1 раз.

Механизмы взаимного исключения

Состояние состязания. Задача об обедающих философах. Детерминированный набор. Условия Бернстайна. Понятие критического ресурса. Критическая область. Взаимное исключение. Механизмы взаимного исключения. Алгоритм Деккера. Алгоритм Петерсона

Взаимодействовать могут либо конкурирующие, либо сотрудничающие процессы. Существует три основных типа взаимодействий:

- передача информации от одного процесса другому;
- исключение того, что процессы пересекутся в конфликтных ситуациях;
- согласование обмена данными: если процесс А поставляет данные, а процесс В использует их, то он должен ждать, пока процесс А не даст данные.

Вторая и третья ситуации относятся и к потокам. Передача информации для потоков не является проблемой, поскольку у них общее адресное пространство. Рассмотрим ситуацию, когда два процесса используют один и тот же ресурс, при этом работают асинхронно. Пусть каждый из процессов хочет вывести файл на печать, для этого он читает индекс свободной ячейки в очереди заданий на печать, помещает туда имя файла и увеличивает индекс. Демон печати обрабатывает эту очередь. Например, в очереди 2 задания уже есть, номер свободной ячейки в очереди заданий соответственно 3. Процесс А читает индекс, это число 3. Тут время, выделенное ему диспетчером задач, заканчивается, и управление переходит к процессу В. Он также читает индекс, это по прежнему 3. Он помещает туда имя файла, увеличивает значение индекса на 1. Тут управление возвращается к процессу А. Он знает, что индекс равен 3, хотя на самом деле уже 4, и заносит в него имя файла, увеличивает индекс. В итоге имя, занесенное процессом В, затирается, и этот файл никогда не будет напечатан. С точки зрения демона, ошибок не произошло – число заданий в очереди находится в полном соответствии с индексом. Ситуация, когда несколько процессов считывают или записывают данные одновременно, и результат зависит от того, кто из них был первым, называется **состоянием состязания**.

Проблема обедающих философов. Сформулирована в 1965 году Дейкстрой. Пять философов сидят за круглым столом и у каждого есть тарелка со спагетти. Чтобы пообедать, необходимо две вилки. Между каждыми тарелками лежит по одной вилке. Философы в течение дня размышляют и обедают. Когда философ голоден, он берет правую вилку, затем левую, съедает спагетти, кладет вилки и какое-то время размышляет. Однако, если философы подойдут к столу одновременно, и одновременно возьмут правые вилки, никто из них не сможет начать кушать: нет ни одной свободной вилки. Произошла взаимоблокировка. Фактически каждый из процессов (философов) захватил часть критического ресурса (одну из двух вилок). Требуется организовать взаимное исключение. Здесь же имеется и еще одна проблема. Например, можно решать задачу так. Если философ взял одну вилку, но вторую взять не может, он должен положить вилку обратно, подождать и повторить все сначала. Но если один философ очень медлителен, а его сосед, напротив, очень быстр, то медленный философ умрет от голода: он подходит к столу, берет вилку, вторая захвачена быстрым соседом, кладет вилку, немного ждет. За это время быстрый сосед успевает покушать, положить вилки, поразмышлять, вновь подойти к столу, и взять вилки для обеда. Медленный философ берет вилку, а вторая занята соседом… И так до бесконечности. Это проблема называется **голоданием**.

Бернстайн сформулировал условия, при соблюдении которых при псевдопараллельном выполнении нескольких процессов для одного и того же набора входных данных они дают одинаковые выходные данные (т.е. набор процессов д**етерминирован**). Обобщение условий на N процессов выглядит следующим образом:
Пусть Ri – набор входных переменных процесса i, Wi – выходных. Тогда выполнение N процессов детерминировано, если:

- $Wi \cap Wj = \varnothing$, i,j=1,2,…,N; i≠j
- $Wi \cap Rj = \varnothing$, i,j=1,2,…,N; i≠j

Пример. Пусть процесс А вычисляет: x=u+v; y=x*w; а процесс В вычисляет: a=b*c; z=x+a;
Wa = {x,y}, Ra = {u,v,x,w}; Wb = {a,z}; Rb = {b,c,x,a}.
$Wa \cap Wb = \varnothing$, $Wa \cap Rb = \{x\}$, $Wb \cap Ra = \varnothing$. Условия не выполняются.

Если условия не соблюдаются, то параллельное выполнение набора процессов может быть детерминированным, а может быть недетерминированным. Условия по сути требуют практически невзаимодействующих процессов. Недетерминированный набор и приводит к состояниям состязания.

Ресурсы, не допускающие одновременного использования, называются **критическими**. Для предотвращения состязания, необходим механизм **взаимного исключения**, не позволяющий процессам обращаться к критическому ресурсу одновременно. В рассмотренном случае с принтером проблема возникла из-за того, что процесс В начал работу с ресурсом до того, как процесс А закончил с ним работать. Кроме реализации в ОС средств для организации взаимного исключения, в ней должны быть средства для синхронизации процессов с целью обмена данными. Типичным примером является взаимодействие типа поставщик-потребитель. Фрагмент кода, в котором происходит обращение к критическим ресурсам, называется **критической областью** или критической секцией. Решение задачи взаимного исключения в том, чтобы организовать такой доступ к критическому ресурсу, когда только одному процессу разрешается находиться в

критической области. Если один из процессов владеет критическим ресурсом, остальные процессы должны получить отказ и ждать освобождения ресурса. При этом, если процессы выполняют операции, не приводящие к конфликтам, т.е. вне критической области, они должны иметь возможность параллельной работы. Если процесс, имеющий доступ к критическому ресурсу, выходит из своей критической области, доступ должен быть передан другому процессу, ожидающему доступа. Ситуация взаимного исключения поясняется на рисунке:

Для корректной организации взаимодействия параллельных процессов необходимо выполнение следующих 4 условий:

- в любой момент времени только один процесс должен находиться в своей критической области (условие взаимного исключения);
- никакой процесс, находящийся вне своей критической секции, не должен влиять на выполнение других процессов, ожидающих входа в критические область, т.е. он не должен блокировать критическую область другого процесса; если несколько процессов одновременно хотят войти в критическую область, то принятие решения, кому предоставить доступ, не должно откладываться бесконечно долго (условие прогресса)
- ни один процесс не должен ждать бесконечно долго вхождения в критическую область и, следовательно, ни один процесс не должен находиться в критической секции бесконечно долго; (условие ограниченного ожидания)
- не должно быть предположений о скорости или количестве процессов.

Все системы имеют такое средство для организации взаимного исключения, как блокировка памяти, запрещающее одновременное исполнение двух или более команд, которые обращаются к одной и той же ячейке памяти, однако для полноценной поддержки взаимного исключения его недостаточно.

Решения организации взаимного исключения. Самый простой способ состоит в **запрещении прерываний** при входе в критическую область. Однако при этом запрещаются и прерывания от таймера. Поскольку переключение задач происходит по прерыванию, то отключение прерываний исключает передачу процессора другому процессу, что и требовалось. Однако существуют довольно серьезные доводы против такого подхода. Например, результате сбоя процесс может не вернуть систему прерываний в разблокированное состояние. Далее, в многопроцессорной системе команда блокирования прерываний повлияет только на один процессор, а ведь другие процессы могут выполняться и на другом процессоре.

Другое решение состоит в использовании специальных **переменных для блокировки.** Выделяется переменная, изначально равная 0. Если процесс желает попасть в критическую область, он считывает эту переменную, и если она равна 0, то устанавливает ее в 1 и входит в критическую область. Если же переменная равна 1, то процесс ждет, пока она не установится в 0. Однако такое решение позволяет нескольким процессам одновременно войти в критические области.

Еще одним вариантом организации взаимного исключения является **чередование доступа** к критическому ресурсу. Для этого имеется общая переменная, указывающая, чья очередь входить в критическую область:

```
int turn;                          // переменная, указывающая, чья очередь доступа

void process1 (void)
    {while (TRUE)
        {while(turn!=1);           // ожидание, если очередь другого процесса
         critical_section1();      // выполнение критической секции
         turn=2;                   // передача очереди другому процессу
         non_critical_section1();}}  // выполнение оставшейся, некритической части кода

void process2 (void)
    {while (TRUE)
        {while(turn!=2);
         critical_section2();
         turn=1;
         non_critical_section2();}}
```

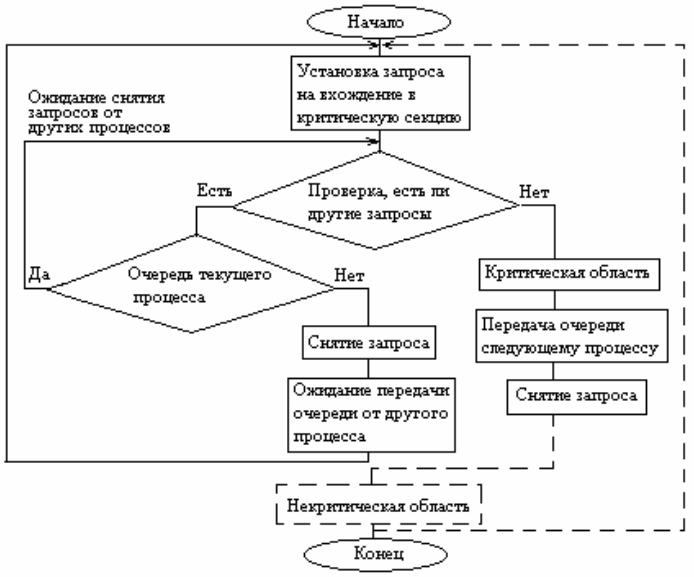
Постоянная проверка значения переменной в ожидании некоторого значения называется **активным ожиданием**. При этом нерационально тратится процессорное время. Блокировка, использующая активное ожидание, называется **спин-блокировкой**. Здесь возможны и другие проблемы. Например, некритическая секция процесса 2 значительно длиннее, чем у процесса 1. Пусть очередь доступа у 1 процесса. Он входит в критическую область. Второй процесс в это время ожидает вхождения в цикле. Первый процесс выполняет критическую область, передает очередь. Второй процесс входит в критическую область. Первый процесс выполняет некритическую область. Второй процесс выполняет критическую область, передает очередь, и тоже приступает к выполнению некритической области. Первый процесс вновь входит в критическую область, выполняет ее, передает очередь, входит в некритическую область. Второй процесс по прежнему находится вне критической области. Первый процесс выполняет некритический фрагмент, и переходит к ожиданию в цикле, пока второй процесс не передаст очередь. Но при этом второй процесс находится вне критической секции, и выполнит ее еще очень не скоро. Т.о., нарушается одно из условий, рассмотренных ранее: процесс, находящийся вне критической области влияет на функционирование процесса, ожидающего доступа. Фактически требуется, чтобы процессы попадали в критические секции строго поочередно, ни один из процессов не может попасть в критическую секцию дважды подряд.

Алгоритм Деккера. Датским математиком Деккером впервые был предложен алгоритм взаимного исключения, не требующий строгого чередования. По сути, он объединяет два предыдущих подхода. Он основан на наличии 3 переменных:

switch[0], switch[1], turn, отвечающих за требования процессов на вхождение в критическую область, и чья очередь на вхождение при условии, если оба процесса требуют ресурс. Алгоритм работы процесса представлен на граф-схеме.

```
// глобальные переменные, изначально все в значении 0
int turn;                                  // переменная, указывающая, чья очередь доступа
int switch[2];                             // запросы на вхождение в критическую область
#define PROC_NUM 0                         // 0 – для 1 процесса, 1 – для второго

void process (void)
  {while (1)
     {switch[PROC_NUM]=1;                  // запрос на вхождение в критическую область
L:    if (switch[1-PROC_NUM]==1)           // если есть другие запросы, то
        {if(turn==PROC_NUM) {goto L;}      // если очередь данного процесса, то ждем снятия других запросов
         else
           {switch[PROC_NUM]=0;            // если очередь другого процесса, снять запрос
            while (turn==1-PROC_NUM);}}    // ожидание, пока другой запрос не передаст очередь
      else
        {critical_section();               // если нет других запросов, то выполнение критической секции
         turn=1-PROC_NUM;                  // передача очереди другому процессу
         switch[PROC_NUM]=0;}              // снятие запроса
      non_critical_section();}}            // выполнение оставшейся, некритической части кода
```



Если установлен флаг запроса от процесса 0, и нет флага запроса от процесса 1, то выполняется критическая секция процесса 0 независимо от того, чья очередь, и очередь передается другому процессу. Если же установлены оба флага, то выполняется критическая секция того процесса, чья очередь. Второй процесс при этом ожидает передачи очереди. Алгоритм Деккера позволяет гарантированно решить проблему критических интервалов. Флаги запроса обеспечивают невозможность одновременного вхождения в критическую область, переменная очереди избавляет от взаимной блокировки. Однако в случае обобщения на N процессов алгоритм усложняется.

Алгоритм Петерсона. В 1981 году был разработан более простой алгоритм взаимного исключения. Алгоритм основан на двух процедурах. Одна из них вызывается перед вхождением в критическую область, вторая - по окончании ее.

```
// глобальные переменные, изначально все в значении 0
int turn;                  // переменная, указывающая, чья очередь доступа
int switch[2];             // запросы на вхождение в критическую область

#define PROC_NUM 0         // 0 – для 1 процесса, 1 – для второго

void process(void)
  {while (1)
     {switch[PROC_NUM]=1;      // запрос на вхождение в КС
      turn=PROC_NUM;           // установка очереди на текущий процесс
      while (turn==PROC_NUM && switch[1-PROC_NUM]==1);}// ожид., если
                               // очередь тек. проц., но есть другие запросы
       critical_section();     // выполнение критической секции
       switch[PROC_NUM]=0;     // снятие запроса
       non_critical_section();}}  // выполнение оставшейся, некритической
                               // части кода
```

Изначально оба процесса вне критических областей. Процесс 0 вызывает функцию вхождения в критическую область, устанавливает там флаг запроса, и переменную очереди в 0 (процесс 0). Поскольку процесс 1 не устанавливал флаг запроса, то процесс 0 входит в критическую область, затем, после ее выполнения, в функции выхода снимает запрос. Если оба процесса вызывают функцию вхождения практически одновременно, происходит следующее. Процесс 0 устанавливает флаг запроса, и устанавливает переменную очереди на свой номер (0). В это время процесс 1 устанавливает свой флаг запроса и изменяет переменную очереди на свой номер (1). Проверяется условие цикла: от процесса 0 есть запрос и очередь установлена на текущий процесс. Процесс 1 переходит к активному ожиданию. Тем временем процесс 0 приступает к определению истинности условия цикла: запрос от процесса 1 есть, но очередь также установлена на процесс 1. Условие ложно, ожидания не происходит, и процесс входит в критическую область. После ее прохождения он снимает флаг запроса. В этот момент у процесса 1 условие также становится ложным, поскольку уже нет запросов от других процессов, и он входит в критическую секцию.

Типовые механизмы синхронизации

Операции Test & Set. Поддержка механизма TS в современных процессорах. Семафоры Дейкстры. Базовые операции над семафорами. Мьютексы. Задача "поставщик-потребитель". Поддержка механизмов синхронизации в Windows и UNIX. Мониторы Хоара. Поддержка мониторов в языках программирования. Задача о парикмахерской. Задача "читатели-писатели"

Операция "Проверка и установка" является аппаратным механизмом организации взаимного исключения. В IBM360 эта команда называлась TS (Test & Set). Команда имеет два операнда и выполняется следующим образом. Значение второго операнда присваивается первому, после этого второй операнд устанавливается в единицу. Особенность команды в том, что она является неделимой, т.е. оба эти действия выполняются неразрывно. Для возможности использования этой операции требуется одна общая переменная, доступная всем процессам. Переменная должна принимать значение 1, если какой-либо процесс находится в своей критической области. Кроме того, с каждым из процессов связана еще и своя персональная переменная, устанавливаемая в 1, если процесс хочет войти в критическую секцию. TS будет использоваться так: значение общей переменной будет считываться в локальную и устанавливаться в 1.

```
int common=0;                          // глобальная переменная

#define PROC_NUM 0                     // 0 – для 1 процесса, 1 – для второго

void process(void)
{int proc;                             // персональная переменная запроса
 while(1)
      {proc=1;                         //запрос на вхождение в критическую секцию
       while(proc==1) TS(proc, common); //пока запрос не обслужен, выполняется операция TS
       critical_section();             //выполнение критического интервала
       common=0;                       //флаг, что в критической секции нет процесса
       non_critical_section();}}       //дальнейшие действия
```

Происходит следующее. Допустим, что в критической секции процесса нет. В этом случае все переменные равны 0. Первый процесс хочет получить доступ к критическому ресурсу. Он выставляет флаг. Операция TS выполняет proc=common, common=1, т.е. поскольку common была 0, то фактически запрос снимается, флаг входа в КС устанавливается, процесс выходит из цикла и входит в КС. Допустим, что в этот момент процесс 2 устанавливает свой флаг. Он входит в цикл, TS выполняет proc=common, common=1. Поскольку Common и так уже в 1, т.е. в КС находится другой процесс, то запрос не снимается, так же как и сам флаг не изменяет значения. Процесс2 ожидает в цикле. Теперь процесс1 выходит из КС и снимает флаг. Процесс2 в очередной раз выполняет TS, поскольку common=0, то запрос снимается, флаг входа в КС устанавливается, процесс выходит из цикла и входит в КС.

В современных микропроцессорах есть специальные команды, являющиеся разновидностью TS: BTC, BTS, BTR. BTS (bit test & set) также имеет два операнда: **BTS Op, B**. Процессор сохраняет бит с номером B переменной Op во флаге CF (carry - перенос), и устанавливает бит B переменной Op в 1. В качестве номера бита может быть указан регистр процессора, в котором этот номер хранится. Этот индекс берется по модулю 32, т.е. использует только 5 младших бит числа, соответственно индекс находится в диапазоне 0 – 31, что позволяет выбрать любой бит в пределах 4-байтной переменной или регистра.

```
L:  BTS   m, 1        ; бит 1 глобальной переменной m – флаг нахождения в КС какого либо процесса
    JC L              ; пока в ней есть процесс, ожидание; флаг переноса выступает в качестве локальной переменной
    CALL critical_section
    AND m, 0fffffffeh ; сброс флага нахождения в критической секции
```

Однако и у этой операции имеется тот недостаток, что ожидающий процесс находится в состоянии активного ожидания, т.е. фактически в холостую простаивает в цикле, потребляя процессорное время. По сути, процессы, обнаружив, что доступ к критическому ресурсу закрыт, должны перейти в состояние блокировки, а как только ресурс освободится, иметь возможность выйти из нее для использования ресурса. Очевидно, что если процесс при ожидании ресурса не должен использовать процессорное время, то сам он не сможет выйти из этого состояния, и требуются соответствующие механизмы ОС.

Команда BTR Op, B выполняет полностью аналогичные действия, но бит B сбрасывается в 0. Команда BTC Op, B инвертирует значение бита B. Обе эти команды сохраняют прежнее значение бита во флаге переноса.

Существует и еще одна проблема при использовании TS или алгоритма Петерсона. Это **проблема инверсии приоритетов**. Пусть имеется 2 процесса: H с высоким приоритетом и L с низким, которым требуется один и тот же ресурс. Планировщик в этом случае немедленно запускает процесс H, если тот оказался в состоянии ожидания. Допустим, процесс L находится в критической области. В этот момент H попадает в состояние ожидания. Планировщик запускает его, но поскольку ресурс занят процессом L, то процесс H остается в состоянии активного ожидания. Однако процесс L не может завершить критическую секцию, т.к. ему не будет предоставлено время, соответственно H останется в активном ожидании.

Семафоры Дейкстры. Понятие семафоров было введено Дейкстрой. Семафор S – это переменная специального типа, доступная параллельным процессам для проведения над ней только двух неделимых операций: закрытия P(S) и открытия V(S).

Поскольку эти операции неделимы, то они исключают друг друга. Семафорный механизм работает по следующей схеме: вначале исследуется состояние критического ресурса, определяемое значением семафора. В зависимости от результата происходит или предоставление ресурса или ожидание доступа в очереди в режиме "пассивного ожидания". В состав механизма включаются специальные средства формирования и обслуживания очереди ожидающих процессов ОС. В силу неделимости операций, даже если некоторые процессы одновременно захотят использовать критический ресурс, доступ получит только один, а второй будет помещен в очередь. Процессам из очереди не предоставляется процессорное время, пока ресурс занят. Допустимыми значениями семафоров являются целые числа. Семафор называется двоичным, если максимальное значение, которое он может принять, это 1. Если больше, то семафор – N-ричный. Рассмотрим пример реализации. Допустим, семафор S инициализируется ОС в значение 1. Тогда операции P и V имеют вид:

```
void P (int *S)
{(*S)--;                        // закрытие семафора и доступа
 if (*S<0) block_process();}    // если семафор уже был закрыт, поместить в очередь блокировки

void V (int *S)
{if (*S<0) activate_process();  // если есть очередь блокировки, поставить процесс на готовность
 (*S)++;}                       // открыть семафор
```

Сами процессы будут иметь вид:

```
void process(void)
{while(1)
    {P(&S);                     //установка семафора
     critical_section();        //или операция успешна или процесс взят из очереди
     V(&S);                     //восстановление семафора
    non_critical_section();}}
```

Пусть оба процесса пытаются выполнить P(S), и это успешно удается второму процессу. Он устанавливает семафор в 0 и переходит к выполнению КС. Тем временем процесс 1 пытается выполнить P(S). Он устанавливает семафор в –1, и помещается ОС в очередь ожидания. Процесс 2 выполняет КС, и вызывает V(S). Поскольку семафор<0, т.е. есть очередь, то процесс 1 выводится из очереди и переходит к выполнению КС. Тем временем процесс 2 восстанавливает значение семафора до 0. В итоге противоречий нет: семафор закрыт, в очереди нет процессов.

Возможны и другие реализации семафоров. Неделимость примитивов P и V обеспечивается в однопроцессорной системе простым запретом прерываний на время их выполнения. В многопроцессорных системах это проблему не решит, поскольку доступ к семафору по-прежнему будет иметь несколько процессов одновременно. Для разграничения доступа требуется использовать переменные блокировки и операцию TS.

Одним из вариантов организации работы с семафором являются **мьютексы** mutex (mutual exclusion – взаимное исключение). Реализованы во многих ОС, представляют собой простейшие двоичные семафоры, которые могут находиться только в одном из двух состояний – открыт или закрыт. Соответственно для реализации требуется всего 1 бит, хотя обычно используют переменную типа int, у которой 0 – открытое состояние, 1 – закрытое. Значение мьютекса устанавливается 2 процедурами. Если процесс хочет войти в КС, он вызывает процедуру закрытия мьютекса, например mutex_lock. Если мьютекс открыт, то запрос выполняется и вызывающий процесс может попасть в КС. Если же мьютекс закрыт, то процесс блокируется, пока другой процесс, находящийся в КО, не выйдет из нее, открыв мьютекс соответствующей процедурой, например, mutex_unlock. Как только какой-либо процесс становится владельцем объекта mutex, он закрывается, а когда задача его освобождает, он открывается. Пример реализации мьютекса:

```
mutex_lock:
    bts   mutex, 1          ; закрыть мьютекс
    jnc OK                  ; если мьютекс был открыт, то конец процедуры
    call thread_yield       ;иначе поток блокируется, передается управление другому
    jmp mutex_lock          ;новая попытка
OK: ret;

mutex_unlock:
    move mutex, 0           ;открыть мьютекс
    ret;
```

От прямого использования операции TS имеется одно существенное отличие: не используется активного ожидания. Существует еще одна проблема, связанная с синхронизацией, про которую не было сказано. В рассмотренных ранее случаях (алгоритмы Деккера и Петерсона, семафоры) предполагалось, что имеется общая глобальная переменная. Однако процессы имеют каждый свое адресное пространство. Как организовать доступ к общему участку памяти? Существует 2 варианта решения. 1: совместно используемые переменные, например, семафоры, хранить в ядре с доступом через системные запросы. 2: позволить процессам использовать некоторую общую область памяти. Как правило, в современных ОС реализованы оба варианта.

Задача производитель-потребитель.
Два процесса совместно используют буфер ограниченного размера. Процесс-производитель помещает туда данные, а процесс потребитель считывает. Проблема возникает, когда производитель пытается поместить данные, и обнаруживает, что буфер полон. Решение в том, чтобы ждать, пока буфер частично или полностью не освободится. Аналогично, если потребитель обращается за данными, а буфер пуст, он также переводится в состояние ожидания. Решение "в лоб" приводит к состоянию состязания, как в случае с печатью файлов. Рассмотрим тривиальное, но неверное решение. Нужна переменная для отслеживания количества записей в буфере count. Пусть N - максимальная длина буфера:

```
#define N 100                          //длина буфера
int count=0;                           //число заполненных элементов – глобальная переменная

void producer(void)
{int item;
    while (TRUE)
        {item=produce_item();          //сформировать новый элемент
        if (count==N) sleep();         //если буфер полон, ожидание
        insert_item(item);             //поместить элемент в буфер
        count++;                       //изменить значение кол-ва элементов
        if (count==1) wakeup(consumer);}}  //если до этого буфер был пуст, вывести из ожидания потребителя

void consumer(void)
{int item;
    while (TRUE)
        {if (count==0) sleep();        //если буфер пуст, ожидание
        item=remove_item();            //взять элемент из буфера
        count--;                       //изменить кол-во
        if(count==N-1) wakeup(producer);   //если буфер был заполнен, вывести из ожидания производителя
        consume_item(item);}}          //использовать элемент
```

count	производитель	потребитель
N	*produce_item()*	
N	*if (count==N) //true*	
N		*if (count==0) //false*
N		*remove_item()*
N-1		*count--*
N-1		*if(count==N-1) // true*
N-1		*wakeup(producer)*
N-1		*consume_item(item)*
N-1	*sleep() //###*	
		…
0		*if (count==0) //true*
0		*sleep() // ###*

Рассмотрим эту ситуацию. Производитель подготовил объект и пытается поместить его в буфер, который заполнен до отказа (count=N). Он читает значение счетчика, сравнивает его с максимальным, но перейти в состояние ожидания не успевает, поскольку планировщик переключает процессы, передавая управление потребителю. Тот читает счетчик, берет элемент из буфера, уменьшает счетчик и, поскольку буфер был полон, посылает сигнал производителю на выход из состояния ожидания. Однако производитель не находится в состоянии ожидания и сигнал пропадает. Когда, наконец, планировщик вновь переключит процессы, производитель благополучно перейдет к ожиданию, несмотря на то, что состояние буфера уже поменялось. В результате потребитель выберет из буфера все элементы и также перейдет в состояние ожидания. Оба процесса будут ожидать – один освобождения буфера, другой – его заполнения. Аналогичные проблемы возникают при пустом буфере Проблема в том, что производитель не успел перейти в состояние ожидания и сигнал активации пропал впустую, создавая типичную ситуацию гонок. Одно из решений проблемы в использовании бита активации. Если сигнал активации посылается процессу, который не находился в состоянии ожидания, то бит устанавливается. Если процесс пытается уйти в состояние ожидания, проверяется этот бит. Если он установлен, то процесс не переводится в состояние ожидания, а всего лишь сбрасывает этот бит, продолжая обработку. Однако задача может быть обобщена на случай N производителей и M потребителей, и одного бита становится недостаточно.

Рассмотрим решение с помощью семафоров.

```
#define N 100
typedef int semaphore;
semaphore mutex=1;          // контроль доступа в КО
semaphore empty=N;          // число пустых ячеек
semaphore full=0;           // число заполненных ячеек

void producer(void)
{int item;
```

```
    while (1)
        {item=produce_item();      // создание данных
        P(&empty);                 // уменьшить счетчик пустых сегментов
        P(&mutex);                 // вход в критическую область
        insert_item(item);         // поместить элемент в буфер
        V(&mutex);                 // выход из КС
        V(&full);}}                // увеличить счетчик полных ячеек

void consumer (void)
{int item;
    while (1)
        {P(&full);                 // уменьшить число полных ячеек
        P(&mutex);                 // вход в КС
        item=remove_item();        // взять данные из буфера
        V(&mutex);                 // выход из КС
        V(&empty);                 // увеличить число пустых ячеек
        consume_item(item);}}      // обработать элемент
```

Используется 3 семафора: для пустых сегментов, для полных, и для исключения одновременного доступа к буферу. Первые два используются не для разграничения доступа, а для синхронизации процессов.

empty	full	mutex	производитель	потребитель
N	*0*	*1*		*P(&full) // ###*
N	*-1*	*1*	*P(&empty)*	
N-1	*-1*	*1*	*P(&mutex)*	
N-1	*-1*	*0*	*КС*	
N-1	*-1*	*0*	*V(&mutex)*	
N-1	*-1*	*1*	*V(&full)*	=>
N-1	*0*	*1*		*P(&mutex)*
N-1	*0*	*0*		*КС*
N-1	*0*	*0*	*P(&empty)*	
N-2	*0*	*0*	*P(&mutex) // ###*	
N-2	*0*	*-1*	=>	*V(&mutex)*
N-2	*0*	*0*	*КС*	
N-2	*0*	*0*		*V(&empty)*
N-1	*0*	*0*	*...*	

Поддержка механизмов синхронизации в ОС Windows. Поддерживаются следующие объекты синхронизации: критические секции, мьютексы, семафоры, события. Объект **критической секции** создается и удаляется функциями

void **InitializeCriticalSection** (LPCRITICAL_SECTION lpCriticalSection)
void **DeleteCriticalSection** (LPCRITICAL_SECTION lpCriticalSection)

В качестве параметра передается указатель на переменную типа CRITICAL_SECTION. Эти объекты не имеют дескрипторов и соответственно не могут использоваться совместно разными процессами, только потоками. При входе и выходе из критического участка кода поток вызывает соответственно функции:

void **EnterCriticalSection** (LPCRITICAL_SECTION lpCriticalSection)
void **LeaveCriticalSection** (LPCRITICAL_SECTION lpCriticalSection)

Внутри области кода, защищенной одним и тем же объектом КС может находиться только один поток. Если другой в это время вызывает EnterCriticalSection, он блокируется. Объекты КС не являются объектами ядра и поддерживаются в пространстве пользователя, что позволяет увеличить производительность. Это простейший механизм синхронизации потоков одного и того же процесса.

В отличие от объектов КС, мьютексы могут иметь имена и дескрипторы, реализуются в пространстве ядра и их можно использовать для синхронизации потоков разных процессов. Дополнительно мьютексы позволяют задавать конечный период ожидания в состоянии блокировки. Мьютекс создается функцией

HANDLE **CreateMutex** (
LPSECURITY_ATTRIBUTES lpsa,
BOOL bInitialOwner,
LPCTSTR lpName)

Здесь lpsa – указатель на структуру атрибутов защиты, определяет возможность наследования указателя на объект мьютекса дочерними процессами. Если NULL – не наследуется. BInitialOwner определяет, требуется ли захват мьютекса сразу после создания. TRUE – да, FALSE – нет. LpName – указатель на нуль-строку, содержащую имя создаваемого объекта. Функция возвращает указатель типа HANDLE на созданный объект. Если создание не удалось, возвращается NULL. Созданный мью-

текс имеет два состояния – сигнальное и нет. В сигнальном состоянии он может быть захвачен, в противном случае при попытке захвата поток блокируется. Для получения указателя на уже имеющийся объект используется функция

HANDLE **OpenMutex** (
DWORD dwDesiredAccess,
BOOL bInheritHandle,
LPCTSTR lpName)

Первый параметр определяет режим использования. MUTEX_ALL_ACCESS определяет разрешение всех возможных вариантов работы с объектом. Второй параметр определяет возможность наследования указателя на объект мьютекса. Третий – указатель на имя. Для удаления ссылки на мьютекс используется

BOOL **CloseHandle** (HANDLE hObject)

Ссылка удаляется и автоматически при закрытии потока. Мьютекс уничтожается системой автоматически, если с ним не связано ни одного указателя. Для захвата мьютекса используется одна из функций ожидания сигнального состояния. Если объект находится в этом состоянии, происходит захват, и он переводится в несигнальное состояние. Другой поток буде вынужден ждать. Рассмотрим две функции из этого множества:

DWORD **WaitForSingleObject** (
HANDLE hHandle,
DWORD dwMilliseconds)

Здесь hHandle –указатель на объект, сигнальное состояние которого ожидается, второй параметр – максимально допустимое время ожидания в милисекундах. Если интервал задан 0 – тестируется текущее состояние и происходит немедленный возврат. Если указано INFINITE интервал тайм-аута не используется. В случае ошибки возвращается значение WAIT_FAILED. Иначе WAIT_ABANDONED, если поток, который создал мьютекс, уже завершился без освобождения объекта с помощью Release. Вызвавший функцию процесс становится владельцем, а мьютекс переходит в несигнальное состояние. WAIT_OBJECT_0, если состояние объекта сигнальное, WAIT_TIMEOUT, если завершился тайм-аут. Функция позволяет работать не только с мьютексами, но и с другими объектами, в т.ч. семафорами, события, процессы, потоки, таймеры и др.

DWORD **WaitForMultipleObjects** (
DWORD nCount, число указателей на объекты
CONST HANDLE *lpHandles, массив указателей
BOOL bWaitAll, если TRUE, то ждет всех объектов в сигнальном состоянии, FALSE – любого из них
DWORD dwMilliseconds) тайм-аут

Возвращаемые значения аналогичны, однако, если bWaitAll=FALSE, то возвращаемое значение минус WAIT_OBJECT_0 (либо WAIT_ABANDONED) указывает номер указателя в массиве на тот объект, который находится в сигнальном состоянии. Если bWaitAll=TRUE функция не изменяет значения объектов синхронизации до тех пор, пока все они не будут в сигнальном состоянии, т.е. либо атомарно выполняются все операции сразу, либо не выполняются вообще.

DWORD **MsgWaitForMultipleObjects**(
DWORD nCount,
LPHANDLE pHandles,
BOOL fWaitAll,
DWORD dwMilliseconds,
DWORD dwWakeMask) тип сообщений в очереди, по которым ожидание так же должно закончиться

Аналогичная функция, позволяющая дополнительно обрабатывать сообщения из очереди, например, для обработки клавиатуры или мыши. Параметры идентичны. WakeMask: QS_ALLINPUT – любое сообщение в очереди, QS_HOTKEY - сообщение WM_HOTKEY, QS_INPUT – сообщение ввода, QS_KEY – сообщения WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, WM_SYSKEYDOWN, QS_MOUSE - сообщения WM_MOUSEMOVE или нажатия кнопок мыши (WM_LBUTTONUP, WM_RBUTTONDOWN, и др.), QS_MOUSEBUTTON – нажатие кнопок мыши (WM_LBUTTONUP, WM_RBUTTONDOWN, и др.), QS_MOUSEMOVE - сообщение WM_MOUSEMOVE, QS_PAINT - сообщение WM_PAINT, QS_POSTMESSAGE – сообщение иное, чем в данном списке, QS_SENDMESSAGE – сообщение присланное другим потоком или приложением, QS_TIMER - сообщение WM_TIMER. Возвращаемые значения аналогичны, дополнительно WAIT_OBJECT_0 + nCount указывает, что появилось сообщение в очереди.

DWORD **SignalObjectAndWait**(
HANDLE hObjectToSignal, объект, устанавливаемый в сигнальное состояние
HANDLE hObjectToWaitOn, объект, сигнальное состояние которого ожидается
DWORD dwMilliseconds, тайм-аут
BOOL bAlertaible)

Эта функция позволяет атомарным образом один объект установить в сигнальное состояние, после чего дождаться сигнального состояния второго объекта. Windows поддерживает асинхронные вызовы процедур. При создании потока с ним связывается очередь асинхронных вызовов процедур (APC queue). ОС либо пользователь могут помещать в нее запросы на выполнение функций в контексте данного потока. Эти функции не могут быть выполнены немедленно, поскольку поток может быть занят. Параметр bAlertaible указывает, оканчивать ли ожидание, если пришел запрос на асинхронный вызов процедуры. Возвращаемые значения аналогичны WaitForSingleObject.

BOOL **ReleaseMutex** (HANDLE hMutex)

Освобождает захваченный мьютекс. Передается указатель на объект, возвращает TRUE, если успешно, FALSE, если ложно.

Механизм семафоров поддерживает счетчики, если значение счетчика больше 0, он в сигнальном состоянии. Отрицательные значения не допускаются. Создание:

HANDLE **CreateSemaphore** (
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, - права наследования
LONG lInitialCount, - начальное значение
LONG lMaximumCount, - максимально допустимое значение

LPCTSTR lpName) - указатель на имя

Удаление ссылки производится уже рассмотренной функцией CloseHandle или автоматически при завершении потока. Объект автоматически удаляется, если нет связанных с ним указателей. Получение указателя на существующий объект –

HANDLE **OpenSemaphore** (
DWORD dwDesiredAccess,
BOOL bInheritHandle,
LPCTSTR lpName)

Параметры полностью идентичны функции OpenMutex. Ожидание сигнального состояния производится с помощью рассмотренных WaitForSingleObject, WaitForMultipleObjects и др. Снятие захвата (операция V) производится функцией

BOOL **ReleaseSemaphore** (
HANDLE hSemaphore, указатель на объект
LONG lReleaseCount, величина, на которую надо увеличить значение семафора, больше 0
LPLONG lpPreviousCount) указатель на переменную, куда будет возвращено предыдущее значение семафора

Объект события позволяет сигнализировать другим процессам о наступлении какого-либо события. Существуют сбрасываемые программно (вручную) события, когда сигнал может быть передан одновременно всем потокам, и сбрасываемые автоматически после освобождения одного из потоков. Создание объекта:

HANDLE **CreateEvent** (
LPSECURITY_ATTRIBUTES lpEventAttributes, атрибут наследования
BOOL bManualReset, TRUE для ручного сброса
BOOL bInitialState, начальное состояние, TRUE - сигнальное
LPCTSTR lpName) указатель на имя.

Ручной сброс выполняется функцией BOOL **ResetEvent** (HANDLE hEvent). **OpenEvent** аналогично уже рассмотренным вариантам возвращает указатель на существующий объект события. BOOL **SetEvent** (HANDLE hEvent) устанавливает событие в сигнальное состояние. BOOL **PulseEvent** (HANDLE hEvent) освобождает все потоки, ожидающие сбрасываемого вручную события, и событие сразу же сбрасывается.

Для работы с разделяемой памятью используются функции **OpenFileMapping, MapViewOfFile, UnmapViewOfFile, CloseHandle**. Разделяемая память организуется либо через отображение на память файла, либо непосредственно в памяти. Далее этот объект отображается на адресное пространство потока, в результате с файлом можно работать как с обычным массивом данных в памяти.

Все рассмотренные функции создания объектов синхронизации и разделяемой памяти CreateXXX возвращают дескриптор и в том случае, если объект с таким именем уже существует. Однако в этом случае функция GetLastError возвращает значение ERROR_ALREADY_EXISTS, в случае же первичного создания она возвращает 0.

Поддержка механизмов синхронизации в UNIX. POSIX библиотека pthreads поддерживает мьютексы следующими функциями для создания, удаления, захвата и освобождения мьютекса: **pthread_mutex_init**, **pthread_mutex_destroy**, **pthread_mutex_lock**, **pthread_mutex_unlock**. В системах на основе System V, в т.ч. и ряде дистрибутивов Linux реализована наиболее общая версия семафоров. Создание массива семафоров:

int **semget** (key_t key, int nsems, int semflg);

key – 32 разрядная переменная-ключ, идентифицирующая набор семафоров, nsems – число семафоров в наборе, semflg – режим доступа Значение IPC_CREAT для создания нового набора, если он уже существует, проверяются права доступа. Если необходимо, чтобы возвращалась ошибка, если набор уже существует, должна использоваться совместно еще одна константа IPC_EXCL. Для подключения к уже существующему набору эти флаги не нужны. Младшие 9 бит флага отвечают за права доступа для root'а, владельца, группы. Возвращает идентификатор массива семафоров или –1 в случае ошибки. При создании набора создается одна общая для набора структура:

struct **semid_ds**{
struct ipc_rerm sem_perm; поля этой структуры содержат информацию о владельце, группе и правах доступа
struct sem* sem_base; указатель на массив объектов-семафоров
ushort sem_nsems; число семафоров в наборе
time_t sem_otime; время последней операции (P или V)
time_t sem_ctime; время последнего изменения
…}

При создании инициализируются поля структуры sem_perm, sem_otime устанавливается в 0, sem_ctime в текущее время. Для каждого семафора создается структура следующего вида:

struct **sem**{
ushort semval; текущее значение семафора
pid_t sempid; идентификатор процесса, вызвавшего последнюю операцию
ushort semncnt; кол-во процессов ожидающих увеличения значения семафора
ushort semzcnt;} кол-во процессов ожидающих нулевого значения семафора

Ключ key можно задавать явно, однако это не гарантирует уникальности ключа – такой уже может существовать в системе. Большую гарантию (однако не 100%) дает использование функции

key_t **ftok**(char* pathname, int proj_id);

Первый параметр задает путь доступа и имя существующего файла, в качестве которого может выступать и корневой каталог ("/"), и текущий каталог("./"). Второй параметр – заданный пользователем ненулевой идентификатор. Хотя это переменная типа int, для генерации реально используется только 8 младших бит. Гарантируется уникальность ключа для отличающихся пар и идентичность для одинаковых. В случае ошибки возвращает -1, и ключ-идентификатор набора в случае успеха.

Операции над семафорами в массиве выполняются вызовом

int **semop** (int semid, struct sembuf *sops, unsigned nsops)

semid – идентификатор набора, sosp – указатель на массив структур sembuf, определяющих, что выполняется с набором, nsosp – число элементов в этом массиве. В результате один вызов обрабатывает несколько семафоров. Ядро гарантирует атомарность этих операций, т.е. никакой другой процесс не начнет обработку над тем же набором, пока текущий процесс не завершит ее. Структура **sembuf** имеет следующий вид:

```
struct sembuf {
unsigned sort sem_num;
short sem_op;
shot sem_flg}
```

Здесь sem_mun – номер семафора в наборе. Если sem_op = 0, то происходит блокировка процесса до тех пор, пока семафор не станет равным нулю, если sem_op>0, то sem_op добавляется к текущему значению семафора, в результате может быть разбужен процесс, ожидающий увеличения семафора. Если sem_op<0, то происходит блокировка процесса до тех пор, пока значение семафора не станет равным или больше по абсолютному значению, чем sem_op, после чего от текущего значения семафора вычитается sem_op. Если флаг sem_flg = IPC_NOWAIT, вместо блокировки происходит возврат с возвращением ошибки. Если процесс завершает работу, не освободив захваченный семафор, то ожидающий процесс будет заблокирован бесконечно. Чтобы этого не происходило, указывается флаг SEM_UNDO. В этом случае ядро запоминает произведенные процессом операции над семафором и автоматически прокручивает их в обратном порядке при завершении работы процесса. Из системы семафоры удаляются принудительно. В противном случае они будут существовать в системе, даже если ни один процесс с ними не связан. С другой стороны это позволяет поддерживать семафоры, не зависящие от жизненного цикла процессов. Для управления объектами набора используется вызова

int **semctl** (int semid, int semnum, int cmd, …)

У вызова 3 или 4 параметра. Первый – идентификатор набора, второй – номер семафора в наборе, если операция выполняется над одним семафором набора, третий – команда, четвертый параметр arg – объединение следующего вида:

```
union semun {
int val;
struct semid_ds *buf;
unsigned short *array;
struct seminfo *__buf;}
```

Возможны следующие команды:

IPC_STAT – копирует данные о наборе в структуру, определяемую arg.buf. Параметр semnum игнорируется.

IPC_SET – устанавливает поля структуры semid_ds о пользователе, группе и режиме доступа из arg.buf. При этом модифицируется поле sem_ctime. Параметр semnum игнорируется.

IPC_RMID – немедленно удаляет набор семафоров, активируя все заблокированные на нем процессы. Параметр semnum игнорируется.

GETALL возвращает semval для всех семафоров набора в arg.array. Параметр semnum игнорируется.

GETVAL возвращает semval для семафора semnum набора semid.

GETNCNT возвращает значение semncnt (число процессов, ожидающих увеличения) для семафора semnum.

GETZCNT возвращает значение semzcnt (число процессов, ожидающих 0) для семафора semnum.

GETPID возвращает sempid (т.е. pid процесса, последним выполнявшим операцию) для семафора semnum

SETALL устанавливает semval для всех семафоров набора из arg.array. Модифицирует sem_ctime набора. Изменение влияет на пробуждение ожидающих процессов. Параметр semnum игнорируется. Отрицательные значения не допускаются.

SETVAL устанавливает semval для семафора semnum. Модифицирует sem_ctime набора. Изменение влияет на пробуждение ожидающих процессов. Отрицательные значения не допускаются.

Для любой команды у процесса должны быть необходимые привилегии. Для команд, изменяющих значения, идентификатор вызвавшего их процесса должен иметь права суперпользователя или же процесс должен быть владельцем либо создателем набора семафоров. В случае ошибки возвращает –1, 0 в случае успеха (за исключением команд GETNCNT, GETZCNT, GETVAL, GETPID).

UNIX также поддерживает разделяемые страницы памяти. Создание разделяемой области выполняется вызовом

int **shmget** (key_t key, int size, int shmflg)

Здесь size – размер задаваемой области, остальные параметры аналогичны рассмотренным для семафоров. Подключение области к виртуальному адресному пространству процесса производится вызовом

void * **shmat** (int shmid, const void *shmaddr, int shmflg)

Здесь shmid – идентификатор области, shmaddr – адрес виртуального пространства, куда необходимо произвести отображение, если NULL , то система вправе подключить область по любому адресу. Если флаг shmflg –SHM_RDONLY – то область подключается как только для чтения. Возвращает адрес, на который происходит отображение. В случае ошибки возвращает –1. Отключение отображения в виртуальное адресное пространство процесса выполняет вызов

int **shmdt** (const void *shmaddr)

Здесь shmaddr – адрес отображения, возвращенный ранее shmat. Вызов

int **shmctl** (int shmid, int cmd, struct shmid_ds *buf)

позволяет управлять соответствующими структурами, отвечающими за разделяемую память и отображение. Shmid – идентификатор области, cmd – команда, buf – указатель на структуру с информацией об области. Команда IPC_RMID позволяет освободить занятые структуры, при этом область удаляется лишь в том случае, если все отображения будут отключены процессами. При ошибке возвращает –1, при успешном завершении 0.

Сравнение возможностей механизмов синхронизации:

ОС	Windows	UNIX
Разнообразие механизмов синхронизации	4 различающихся типа	Один универсальный механизм, позволяющий реализовать все варианты
Создание объектов синхронизации	Отдельно для каждого объекта	Для набора объектов
Удаление объектов синхронизации	Автоматическое при отсутствии дескрипторов	Ручное
Операция P(S)	Только –1	На любую величину
Операция V(S)	На любую величину	На любую величину
Атомарное выполнение нескольких операций	Только для P(S)	Любое сочетание из набора
Получение значения семафора	Только при V(S)	Отдельная команда
Инициализация объектов	При создании	Отдельная команда
Управление набором	Отсутствует	Поддерживается
Отрицательные значения семафоров	Не поддерживается	Не поддерживается

Однако использование семафоров тоже имеет свои ограничения. Если семафоры в листинге поменять местами, то может произойти взаимоблокировка, т.е. программы не застрахованы от случайных ошибок пользователя. Для организации взаимодействия в 1974 г. Хоар и Хансен предложили примитив синхронизации более высокого уровня – **монитор**. Монитор – это набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет. Например, какой-либо ресурс должен разделяться между процессами. Соответственно для получения ресурса процесс должен обратиться к некоторому планировщику, который с помощью внутренних переменных отслеживает, занят ресурс или свободен и выполняет последующие необходимые действия. Процедуры этого планировщика разделяются всеми процессами, однако планировщик может обслуживать только один процесс. Такой планировщик и является монитором.

Процессы могут вызывать процедуры монитора, однако у внешних процедур нет прямого доступа к внутренним структурам монитора. Важным свойством монитора является следующее: при обращении к монитору в любой момент времени активным может быть только один процесс. Мониторы являются структурным компонентом языка программирования, и компилятор знает, что процедуры монитора надо обрабатывать иначе, чем вызовы других процедур. Как правило, первые несколько команд процедуры монитора проверяют, нет ли уже в мониторе активного процесса. Если есть, то вызывающий процесс ожидает. Для реализации взаимного исключения обычно используется мьютекс или бинарный семафор. Этого еще недостаточно. Необходим способ блокировки процессов, которые не могут продолжаться дальше. Решение в том, чтобы ввести переменные состояния и две операции wait и signal. Если процедура монитора обнаруживает, что она не может продолжать работу, то она выполняет операцию wait на какой-либо переменной состояния. Это приводит к блокировке вызывающего процесса и позволяет другому процессу войти в монитор. Другой процесс может активизировать его, выполнив операцию signal на той переменной состояния, на которой тот был заблокирован. Еще надо сделать так, чтобы в мониторе не было двух процессов одновременно. Хоар предложил запуск разбуженного процесса и остановку второго. Хансен предложил, что процесс, выполнивший signal, должен немедленно покинуть монитор. Т.е. операция signal должна выполняться в самом конце монитора. Если signal выполнена на переменной, с которой связаны несколько заблокированных процессов, планировщик выбирает только один из них. Существует и третье решение: позволить процессу, выполнившему signal, продолжать работу и запустить ожидающий процесс только после того, как первый процесс покинет монитор. Переменные состояния не являются счетчиками. Они не накапливают значения. Это значит, что в случае выполнения операции signal на переменной состояния, с которой не связано ни одного блокированного процесса, сигнал будет утерян. Т.е. операция wait должна выполняться раньше, чем signal.

Механизм мониторов поддерживается Java. Ключевое слово **synchronized** гарантирует, что если хотя бы один поток начал выполнение этого метода, ни один другой поток не сможет выполнять другой синхронизированный метод данного класса. Т.е. идет поддержка механизма мониторов на уровне языка, что отсутствует в C. Решение задачи производители-потребители с помощью мониторов на Java:

```
public class ProducerConsumer
    {static final int N=100;                    // размер буфера
     static producer p=new producer();          // экземпляр потока производителя
     static consumer c=new consumer();          // экземпляр потока потребителя
     static pc_monitor mon=new pc_monitor();    // экземпляр монитора

     public static void main (String args[])
        {p.start();                             // запуск потоков производителя и потребителя
         c.start();}

     static class producer extends Thread       // класс производителя
        {public void run()                      // процедура потока
            {int item;
             while (true)
                {item=produce_item();           // произвести элемент
                 mon.insert(item);}}            // добавить его в буфер с помощью монитора
```

```
        private int produce_item()              // процедура получения элемента
          {...}}

    static class consumer extends Thread        // класс потребителя
        {public void run()                      // процедура потока
            int item;
            while (true)
                {item = mon.remove();           // взять элемент из буфера с помощью монитора
                consume_item(item);}}           // потребить элемент

         private void comsume_item(int item)    // процедура использования элемента
           {...}}

    static class pc_monitor                     // класс монитора
        {private int buffer[]=new int[N];       // содержит буфер
         private int count=0, lo=0, hi=0;       // счетчик и индексы

         public synchronized void insert (int val)  // процедура добавления элемента в буфер
              {if (count==N) go_to_sleep();     // если буфер полон, ожидание
               buffer[hi]=val;                  // поместить элемент
               hi=(hi+1)%N;                     // увеличить индекс для записи
               count=count+1;                   // число элементов в буфере
               if (count==1) notify();}         // если буфер был пуст, разбудить процесс

         public synchronized int remove()       // процедура взятия объекта из буфера
              {int val;
               if (count==0) go_to_sleep();             // если буфер пуст, ожидание
               val=buffer[lo];                  // взятие элемента
               lo=(lo+1)%N;                     // увеличение индекса для взятия
               count=count-1;                   // число элементов в буфере
              if (count==N-1) notify();         // если буфер был полон, разбудить процесс
              return val;}

        private void go_to_sleep()              // процедура ожидания
              {try
                  {wait();}                     // ожидание с обработкой исключения
               catch (InterruptedException exc) {};}}}
```

Внешний класс ProducerConsumer создает и запускает два потока. Класс монитора содержит два синхронизированных потока, используемых для текущего помещения элементов в буфер и извлечения их оттуда. Поскольку эти методы синхронизированы, то состояние состязания исключается автоматически средствами языка программирования. Переменная Count отслеживает количество элементов в буфере, lo – содержит индекс, откуда надо извлекать элемент, hi – куда помещать. Если lo=hi, то в буфере или нет элементов, или он заполнен полностью, что можно отследить с помощью count. Синхронизированные методы в Java несколько отличаются от классических мониторов тем, что у них отсутствует переменная состояния. Взамен предлагаются аналогичные процедуры wait и notify.

Доступ к разделяемым переменным всегда ограничен телом монитора, что автоматически исключает критические интервалы. Монитор является пассивным объектом в том смысле, что это не процесс, его процедуры выполняются только по требованию процессов. Мониторы обладают следующими свойствами:

- Высокая гибкость при реализации синхронизирующих операций
- Локализация разделяемых переменных в теле монитора позволяет вынести специфические синхронизирующие конструкции из параллельных процессов
- Процессы имеют возможность совместно использовать модули, имеющие критические секции
- Если несколько процессов разделяют некоторый ресурс, и работают с ним совершенно одинаково,то в мониторе для этого требуется только одна процедура. При других решениях соответствующий код должен быть в теле всех процессов

Volatile. Даже если синхронизация успешно решена без помощи специальных механизмов, этого может быть недостаточно, если используются оптимизирующая компиляция программы. В этом случае измененное значение переменной может оставляться в регистре процессора, а не заноситься сразу в заданную ячейку памяти, что опять приводит к гонкам. В стандарте ANSI C описан спецификатор volatile позволяющий гарантировать, что переменная после изменения будет возвращена в память.

Проблема обедающих философов. Решение:

```
#define N 5                 // число философов
#define LEFT (i+N)%N        // левый сосед философа i
#define RIGHT (I+1)%N       // правый сосед философа i
```

```
#define THINKING 0              // думает
#define HUNGRY 1                // голодает
#define EATING 2                // ест

typedef int semaphore;
int state[N];                   // состояния философов
semaphore mutex=1;              // взаимное исключение для критич. обл.
semaphore s[N];                 // семафоры для каждого философа

void philosopher(int i)         // процессы для философов
{while(1)
    {think();                   // размышляет
     take_forks(i);             // берет вилки
     eat();                     // ест
     put_forks();}}             // кладет вилки

void take_forks(int i)          // взятие вилки
{P(&mutex);                     // закрыть мьютекс
 state[i]=HUNGRY;               // состояние философа – нужен ресурс
 test(i);                       // попытка получить вилки
 V(&mutex);                     // открытие мьютекса, выход из КС
 P(&s[i]);}                     // философ в ожидании или использовании вилок

void put_forks(int i)           // отпустить вилки
{P(&mutex);                     // вход в КС
 state[i]=THINKING;             // перестал есть, размышляет
 test(LEFT);                    // может ли есть сосед справа
 test(RIGHT);                   // может ли есть сосед слева
 V(&mutex);}                    // выход из КС

void test(int i)                // если хочет есть и соседи не едят
{if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)
  {state[i]=EATING;             // есть
   V(&s[i]);}}                  // философ не использует и не требует вилок
```

Задача читатели-писатели. Ситуация типична при доступе к БД. Например, бронирование билетов. В системе есть процессы-читатели, которые только читают какую-то общую для всех информацию, и писатели, которые могут ее изменять. Чтение может быть разрешено одновременно, однако при записи доступ для всех процессов, в. т.ч. и на чтение, должен быть закрыт. Решение:

```
typedef int semaphore;
semaphore mutex = 1;            // контроль доступа к переменной rc
semaphore db=1;                 // контроль доступа к базе данных
int rc=0;                       // кол-во процессов читающих или желающих читать

void reader(void)               // процесс читатель
{while(1)
  {P(&mutex);           // закрыть доступ к счетчику читателей
   rc++;                        // увеличить его
   if (rc==1) P(&db);           // если процесс первый, закрыть доступ к БД
   V(&mutex);                   // открыть доступ к счетчику
   read_database();             // чтение БД
   P(&mutex);                   // закрыть доступ к счетчику
   rc=rc-1;                     // уменьшить его
   if (rc==0) V(&db);           // если нет других читателей (процесс последний) открыть доступ к БД
   V(&mutex);                   // открыть доступ к счетчику
   use_read_data();}}           // использование данных

void writer(void)               // процессы писатели
{while(1)
  {make_data();                 // подготовка данных к записи
   P(&db);                      // закрыть доступ
   write_database();            // записать данные
   V(&db); }}                   // открыть доступ
```

Первый читатель закрывает доступ к БД. Остальные всего лишь увеличивают счетчик читателей. Когда базу покинет последний читатель, он открывает к ней доступ. Если один читатель уже работает с базой, второй тоже получит к ней доступ и

т.д. Если в этот момент доступ запросит писатель, он перейдет к ожиданию, поскольку доступ закрыт. Доступ писатель получит только когда базу покинет последний читатель.

Пусть одновременно начали работу 2 читателя и 1 писатель. Ч2 успевает первым получить доступ к счетчику rc, наращивает его. Ч1 ожидает доступа к rc. Ч2 получает доступ к БД, открывает доступ к счетчику. Ч1 разблокируется. В этот момент диспетчер включает писателя. Тот блокируется на доступе к БД. Ч1 получает доступ к счетчику, увеличивает его. Поскольку rc=2, блокировки на БД не происходит, хотя доступ к БД закрыт. Ч1 заканчивает чтение, уменьшает счетчик, однако БД не открывает, поскольку rc=1. Ч2 читает данные, уменьшает счетчик, открывает доступ к БД. Писатель продолжает работу. Если бы П активировался когда оба читателя не дошли до блокировки базы, он бы начал работу, захватив db. Ч2 был бы заблокирован на db, Ч1 соответственно на mutex. После освобождения db писателем, Ч2 продолжил бы работу, захватив доступ к БД, далее разблокировал бы mutex, и Ч1 также смог бы продолжить работу. Однако такое решение позволяет получить доступ вновь прибывающим читателям, даже если в системе есть ожидающий писатель. В результате он может ожидать бесконечно.

Mutex	Db	Rc	Читатель 1	Читатель 2	Писатель
1	*1*	*0*		*P(&mutex)*	
0	*1*	*0*	*P(&mutex) // ###*		
-1	*1*	*0*		*rc++*	
-1	*1*	*1*		*if (rc==1) P(&db) // (true)*	
-1	*0*	*1*			*P(&db) // ###*
-1	*-1*	*1*	*=>*	*V(&mutex)*	
0	*-1*	*1*	*rc++*		
0	*-1*	*2*	*if (rc==1) // (false)*		
0	*-1*	*2*	*V(&mutex)*		
1	*-1*	*2*	*KC*		
1	*-1*	*2*	*P(&mutex)*		
0	*-1*	*2*	*rc--*		
0	*-1*	*1*	*if(rc==0) // (false)*		
0	*-1*	*1*	*V(&mutex)*		
1	*-1*	*1*		*KC*	
				...	
0	*-1*	*0*		*if(rc==0) V(&db) // (true)*	*=>*
0	*0*	*0*			*KC*
0	*0*	*0*		*V(&mutex)*	
1	*0*	*0*			*V(&db)*
1	*1*	*0*	*...*		

Задача о парикмахерской. Есть парикмахерская, в ней один парикмахер, его кресло и n стульев для посетителей. Если его услугами никто не пользуется, он спит, сидя в своем кресле. Когда приходит клиент, он должен разбудить парикмахера. Если клиент приходит, а мастер занят, то клиент либо садится на стул, если есть место, либо уходит, когда места нет. Требуется избежать состояния состязания. Решение:

```
#define CHAIRS 5                // кол-во стульев
typedef int semaphore;
semaphore customers=0;  // Кол-во ожидающих посетителей
semaphore barbers=0;            // Кол-во брадобреев, готовых к работе
semaphore mutex=1;              // Для взаимного исключения
int waiting=0;                  // ожидающие посетители

void barber(void)           // брабобрей
{while(1)
      {P(&customers);           // попытка взять клиента
       P(&mutex);               // получение доступа к счетчику посетителей
       waiting--;               // уменьшение его
       V(&barbers);             // брадобрей готов к работе
       V(&mutex);               // открыть доступ к счетчику
       cut_hair();}}            // обслуживание

void customer(void)             // посетитель
{P(&mutex);                     // доступ к счетчику
 if(waiting<CHAIRS)             // есть свободное место
      {waiting++;               // увеличить число ждущих посетителей
       V(&customers);           // пришел клиент, разбудить брадобрея
       V(&mutex);               // открыть доступ
```

```
    P(barbers);              // доступ к брадобрею
    get_haircut();}          // получить обслуживание
else {V(&mutex);}}           // открыть доступ и уйти
```

Для подсчета ожидающих посетителей используется семафор customers, при этом обслуживаемый клиент не учитывается. Barbers, кол-во готовых к работе брадобреев. Waiting фактически дублирует семафор customers, поскольку прочитать значение семафора невозможно, а приходящий посетитель должен знать сколько их в очереди, и если стульев больше, то остаться для ожидания.

Начал работу процесс парикмахер. Он пытается получить доступ к посетителю, и уходит в ожидание, поскольку семафор =0. Приходит первый посетитель, запрашивает доступ к счетчику, увеличивает его, открывает семафор, открывает доступ к счетчику, пытается получить доступ к парикмахеру. Поскольку семафор customer уже открыт, мастер просыпается, получает доступ к счетчику, уменьшает его, открывает семафор burber, открывает доступ к счетчику. Теперь посетитель получает доступ к услугам. Приходит 2 клиент. Получает доступ к счетчику, увеличивает его, увеличивает семафор customers, освобождает доступ к счетчику. Мастер обслуживает первого клиента, пытается обслужить второго. Поскольку посетитель есть, то запрашивается доступ к счетчику, счетчик уменьшается, открывается семафор парикмахера, освобождает счетчик. Второй посетитель пытается получить доступ в кресло парикмахера и получает его. Посетитель уходит. Парикмахер вновь пытается пригласить клиента. Их нет, и он переходит к ожиданию.

Механизмы межпроцессного взаимодействия

Механизмы для взаимодействия процессов. Сигналы. Каналы. Именованные и анонимные каналы. Сообщения. Очереди сообщений. Порты. Буферы сообщений. Сообщения как средство синхронизации. Рандеву. Барьеры. Поддержка механизмов взаимодействия в ОС UNIX и Windows

Почтовые ящики (буферы сообщений). Процессы в ОС могут обмениваться сообщениями. Для хранения посланного, но еще не полученного сообщения, необходимо место, называемое почтовым ящиком или буфером сообщения. Если процесс хочет общаться с другим процессом, то он просит систему выделить ему почтовый ящик, который свяжет эти два процесса. Для отправления сообщения процесс просто помещает его в почтовый ящик, откуда второй процесс может взять его в любое время. Второй процесс должен знать о существовании ящика, и для получения сообщения выполнить к нему обращение. Если объем данных велик, целесообразно не помещать их в ящик, а оставлять всего лишь информацию, где их можно найти. Если почтовый ящик не связан жестко с конкретными процессами, то сообщение должно содержать идентификаторы и процесса отправителя и процесса получателя. Почтовый ящик состоит из заголовка, где содержится информация о ящике и из нескольких буферов (ячеек) для сообщений. В простейшем случае сообщения передаются только в одном направлении. Процесс может посылать сообщения, пока есть свободные ячейки. Если они все заполнены, то процесс может или ожидать или выполнять другие операции. Аналогично и процесс получатель может получать сообщения, пока есть заполненные ячейки. Можно организовать более сложные ящики. Например, двунаправленные. Такой ящик позволяет подтверждать прием сообщений. Чтобы гарантировать доставку подтверждения в случае, когда все ячейки заняты, подтверждение помещается туда же, где лежало исходное сообщение. В эту ячейку не может быть помещено новое сообщение до тех пор, пока не будет получено подтверждение.

Как правило, используется буфер из определенного количества элементов, тип которых задается при создании ящика. Для реализации механизма достаточно двух примитивов: **send** – отправить, и **receive** – принять. Примитивы имеют два параметра, один из которых – собственно сообщение или его адрес, второй параметр указывает или идентификатор взаимодействующего процесса, или идентификатор почтового ящика. Еще один вариант организации сообщений – не использовать буферизацию. В этом случае, если send выполняется раньше receive, посылающий процесс блокируется до выполнения receive, когда сообщение может быть напрямую скопировано от производителя к потребителю без промежуточной буферизации и наоборот. Этот метод называется **рандеву**, он легче реализуется, чем схема с буферизацией, однако процессы должны быть жестко синхронизированы. Примитивы send и receive имеют скрытый механизм взаимоисключения, а в большинстве систем и блокировки при чтении из пустого ящика или записи в заполненный. Однако, несмотря на простоту использования, это решение менее производительно.

Очереди сообщений. Очереди сообщений позволяют обрабатывать сообщения в соответствии с разными дисциплинами обслуживания:

- FIFO – первым пришел первым ушел (очередь).
- FILO – первым пришел последним вышел (стек)
- приоритетный – в зависимости от приоритета
- произвольный

В отличие от рассмотренных вариантов, можно организовать такой режим, при котором прочитанное сообщение не удаляется из очереди, и может быть прочитано повторно другими процессами. В очередях присутствуют не сами сообщения, а их адреса и размер. При этом используется общая для всех процессов память, и процесс должен вначале получить разрешение на доступ к ней с помощью системных запросов. Дополнительной информацией при работе с очередями служит флаг, указывающий требуется ли ожидание, если очередь пуста (полностью заполнена). Очередь сообщений может быть реализована в виде **порта**. Право на отправку сообщений в конкретный порт могут иметь несколько разных процессов, но право получать сообщения из порта имеет только одна задача.

Решение производитель-потребитель с помощью сообщений:

```
#define N 100                                   // ячеек в буфере
void producer (void)                            // производитель
{int item;
 message m;
 while (1)
      {item=produce_item();                     // создать элемент
       receive(consumer, &m);                   // получить пустое сообщение
       build_message(&m, item);                 // сформировать сообщение для отправки
       send(consumer, &m);}}                    // послать сообщение

void consumer (void)                            // потребитель
{int item, i;
 message m;
 for (i=0; i<N; i++) send(producer, &m);  // послать серию пустых сообщений
 while (1)
      {receive(producer, &m);                   // получить сообщение
       item=extract_item(&m);                   // извлечь элемент из сообщения
       send(producer,&m);                       // послать пустое сообщение
       consume_item(item);}}                    // использовать элемент
```

В UNIX для поддержки механизма сообщений используются следующие вызовы:

int **msgget**(key_t key, int msgflg); - для создания очереди сообщений, связанных с данным ключом
int **msgsnd** (int msqid, struct msgbuf* msgp, size_t msgsz, int msgflg) – для отправки сообщения
ssize_t **msqrcv** (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg) – для получения сообщения
int **msqctl** (int msqid, int cmd, struct msqid_ds *buf); - для управления структурами

В Windows также поддерживается модель буферов сообщений (почтовых ящиков). Они являются однонаправленными. Сервер (считывающий процесс) создает дескриптор почтового ящика **CreateMailSlot**, далее он ожидает сообщения с помощью **ReadFile**. Клиент (отправляющий процесс) открывает ящик с помощью **CreateFile** и передает сообщение с помощью **WriteFile**. Если ни один сервер не ожидает сообщения, открытие ящика завершится ошибкой. Сообщение может быть прочитано всеми серверами.

Проблемы использования сообщений: сообщение может потеряться (особенно если процессы происходят на различных машинах, объединенных сетью). Подтверждение призвано гарантировать, что сообщение не потерялось. Если в течение некоторого времени подтверждение не пришло, сообщение посылается повторно. Однако само подтверждение также может потеряться. В итоге сообщение будет отправлено повторно, хотя оно уже было получено. Требуется отличать копии от оригинала. Обычно в тело сообщения помещают его порядковый номер. Если номер вновь полученного сообщения совпадает с одним из номеров ранее принятых сообщений, то оно считается копией и игнорируется. Кроме этого, необходимо однозначно определять процесс (его имя). Встает и вопрос аутентификации (действительно ли идет взаимодействие с файловым сервером, а не с процессом, пытающимся нелегально получить информацию).

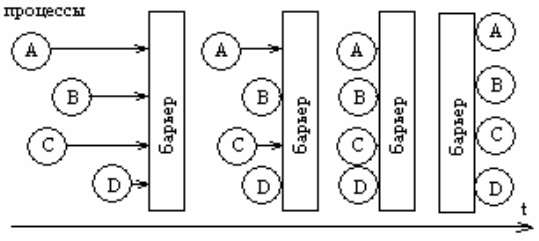
Программные каналы (pipe). Принцип работы основан на механизме в/в используемый при работе с файлами в UNIX. Задача, передающая информацию, действует так, словно она записывает данные в файл, а задача, принимающая – словно читает файл. Это упрощает программирование и не требует каких-либо новых механизмов. На самом деле каналы не являются файлами, а представляют собой особые буферы, работающие по принципу очереди (FIFO). Имеется два указателя, в начальный момент равные нулю. При записи увеличивается один из них, при чтении второй. При достижении максимального элемента, указатель опять изменяется на 0, т.е. наращивание происходит циклически. Читать из канала может только тот процесс, который знает его идентификатор. Каналы являются однонаправленными. После чтения данных они становятся недоступными. При чтении из пустого канала процесс блокируется до прихода данных, блокируется и процесс, записывающий в заполненный канал.

В UNIX канал создается вызовом **pipe**. Вызов возвращает 2 дескриптора – для чтения и для записи, которые могут наследоваться потомками. Чтение и запись выполняются вызовами **read** и **write** с помощью дескриптора канала. Типичное применение механизма программных каналов – это перенаправление вывода одного процесса на ввод другого (Оператор | командной строки). Ограничения, присущие каналам:

- канал не может использоваться для передачи данных нескольким процессам;
- данные интерпретируются как поток байт с заранее неизвестной длиной;
- если данные читает несколько процессов, отправитель не может указать какие данные какому из них предназначены.

Помимо неименованных каналов (анонимных), существуют именованные – named pipe. Они отличаются способами создания и доступа. Они могут быть доступны любым процессам, не только потомкам, имеют имя в пространстве имен файловой системы и являются постоянными, т.е. могут содержать данные уже завершившего работу процесса. После использования их надо удалять принудительно.

Современные Windows также поддерживает каналы. Сервер создает именованный канал с помощью **CreateNamedPipe**. При этом сервер может создавать несколько дополнительных экземпляров именованного канала. Для возможности соединения с клиентами сервером используется функция **ConnectNamedPipe**. Для подключения к серверу клиенты используют **CreateFile**. Чтение и запись данных выполняется функциями **ReadFile** и **WriteFile** и является двусторонней. Определить, есть ли данные в канале без их уничтожения позволяет функция **PeekNamedPipe**. Анонимные каналы однонаправлены и создаются функцией **CreatePipe**.



Барьеры. Предназначены для группы процессов. Некоторые приложения делятся на фазы, и существует правило, что процесс не может перейти к следующей, пока к этому не готовы все остальные процессы. Для этого в конце каждой фазы располагается барьер. Процесс, доходя до барьера, блокируется, пока все процессы не дойдут до него. Через какое-то время процесс D выполняет оставшиеся вычисления, и запускает примитив barrier, являющийся обычно библиотечной функцией. Поскольку только один процесс у барьера, он переходит в ожидание. Затем аналогично процесс B, потом C, и наконец A. Как только последний из процессов выполнил вызов, блокировка снимается для всех ожидающих процессов, и они переходят за барьер.

Обработка тупиковых ситуаций

Понятие тупика. Условия Коффмана. Модель Холта. Модель пространства состояний. Надежное, опасное и безопасное состояния. Обнаружение тупика при наличии одного ресурса каждого вида. Обнаружение тупика при наличии нескольких ресурсов каждого вида. Выход из взаимоблокировки. Обход и предотвращение тупиковой ситуации. Двухфазовое блокирование. Проблема "голодания"

Ситуация, когда каждый из параллельных процессов ожидает освобождения ресурса, занятого другим процессом из данного множества, называется **взаимоблокировкой**, или тупиком. Для использования ресурса выполняется следующая последовательность действий: запрос, использование, освобождение. Один из способов разграничения доступа – использование семафоров, присоединенных к каждому из ресурсов. Для запроса используется вызов down(), а для освобождения up(). Рассмотрим случай, когда процессу необходимо 2 или больше ресурсов:

```
typedef int semaphore
semaphore rc1, rc2;
```

```
void process_A(void)
{down(&rc1);
down(&rc2);
use_2_resources();
up(&rc2);
up(&rc1);}
```

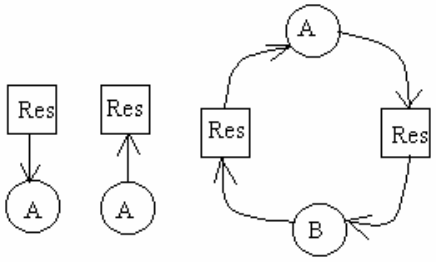
```
void process_B(void)
{down(&rc1);
down(&rc2);
use_2_resources();
up(&rc2);
up(&rc1);}
```

```
void process_C(void)
{down(&rc2);
down(&rc1);
use_2_resources();
up(&rc1);
up(&rc2);}
```

Пусть пока в системе только 2 процесса A и B. В этом случае никакая взаимоблокировка невозможна. Действительно, если процесс A получит доступ к ресурсу 1, процесс B будет заблокирован на получении доступа. Процесс 1 блокирует второй ресурс, использует ресурсы, и освобождает их, после чего доступ получает второй процесс. В случае, если это процессы A и C возможна следующая ситуация: процесс A захватил ресурс 1, диспетчер переключил процессы, процесс C получил управление и захватил ресурс 2. Теперь процесс A ожидает освобождение ресурса 2, а процесс C – ресурса 1. Произошла взаимоблокировка. Т.о., на взаимоблокировки существенное влияние может оказывать код программы. Однако в системе, в которой существует несколько самых разных видов ресурсов, и за доступ к которым конкурируют несколько процессов предусмотреть корректное написание последовательности запросов не представляется возможным.

Коффман доказал, что для возникновения взаимоблокировки необходимо выполнение следующих 4 условий:

- условие взаимного исключения. Каждый ресурс в данный момент времени или отдан ровно одному процессу, или доступен
- Условие удержания и ожидания. Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы
- - условие отсутствия принудительной выгрузки ресурса. У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс-владелец должен сам освободить их.
- условие циклического ожидания. Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.



Для моделирования взаимоблокировок используют графы:
Ребро, направленное от ресурса к процессу, означает, что ресурс ранее был запрошен процессом, получен и в данный момент используется процессом. Ребро, направленное от процесса к ресурсу. означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к этому ресурсу. Далее показана ситуация взаимоблокировки: Процесс A ожидает ресурс, который захвачен процессом B, а процесс B ожидает ресурс, который захвачен процессом A. В обозначении ресурса может быть указано число доступных единиц ресурса. Число ребер, исходящих из ресурса, не может превышать числа единиц этого ресурса. Модель, представленную таким графом, называют моделью повторно используемых ресурсов Холта.

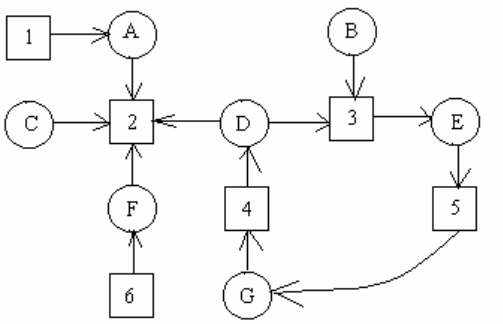
Возможны 4 стратегии при решении проблемы тупиковых ситуаций:

- пренебрежение проблемой.
- Обнаружение и восстановление: позволить взаимоблокировке произойти, обнаружить ее и предпринять какие-либо действия.
- предотвращение тупиковой ситуации с помощью структурного опровержения одного из четырех условий, необходимых для взаимоблокировки
- избегание тупиковых ситуаций с помощью аккуратного распределения ресурсов

Фактически тупиковые ситуации возможны не только при борьбе за ресурсы устройств ввода-вывода. Например, в системе всегда ограничивается максимальное число открытых файлов. Пусть это 100. В системе существует 10 процессов, каждый из которых требует открытия 12 файлов. Возможна ситуация, когда каждый из этих процессов откроет ровно 10 файлов, полностью исчерпав ресурс таблицы. Ни один из процессов не сможет продолжить работу. Большинство ОС игнорируют такую проблему.

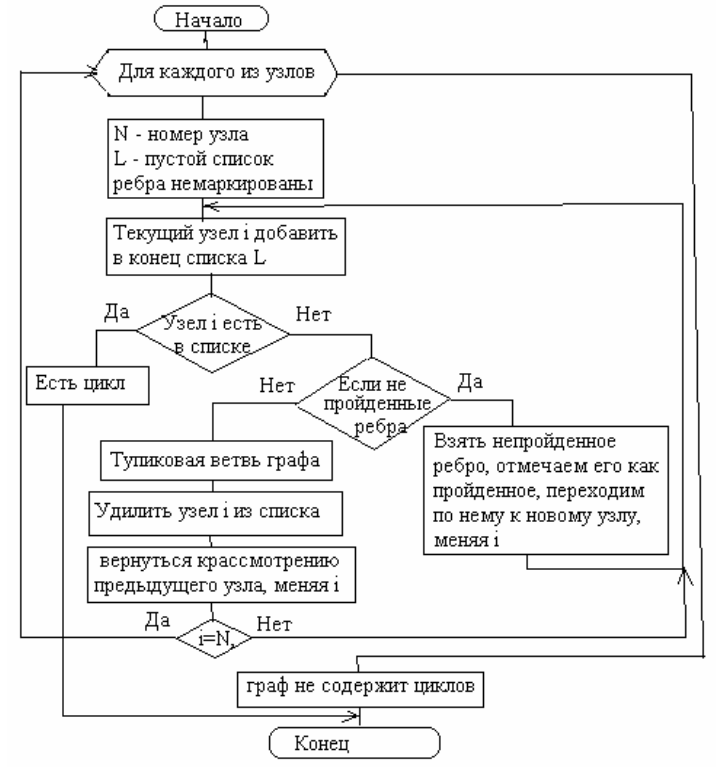
Обнаружение взаимоблокировок. При наличии в системе одного ресурса каждого вида.

Для системы подобного рода удобно построить граф и проанализировать его. Если на графе нет замкнутых циклов, то взаимоблокировка отсутствует. В примере процессы ACF не попали в тупик, любому из них система предоставит ресурс 2, процесс его получит, использует и вернет, после чего ресурс сможет быть предоставлен другому процессу. Вместе с тем в графе содержится цикл, соответствующие процессы которого находятся во взаимоблокированном состоянии.



Рассмотрим алгоритм поиска в графе замкнутых циклов. Алгоритм использует одну единственную структуру данных – список узлов L.

Алгоритм по очереди берет каждый узел в качестве корня того, что, как он надеется, окажется деревом, и выполняет в дереве поиск в глубину. Если при этом происходит возврат к уже пройденному узлу, то обнаружен цикл. Если алгоритм прошел уже все ребра какого-либо узла, то он возвращается к предыдущему узлу. Если при этом он вернулся к корню, то подграф текущего узла не содержит циклов и проверяется следующая вершина.



Еще один вариант - это редукция графа в модели Холта. В этом случае на графе выполняются последовательные сокращения (редукции) всех ребер, как входящих в вершину определенного процесса, так и исходящие из него, если процесс не является заблокированным. Такое редуцирование эквивалентно ситуации, когда процесс получит запрашиваемые ресурсы, а потом все их освободит. Признаком блокированного процесса является то, что ресурс, который он запрашивает, на текущем шаге редуцирования, уже полностью выделен другим процессам. Редуцирование выполняется итерационно до тех пор, пока на графе возможны изменения. Если в результате граф не является полностью сокращаемым, то рассмотренное состояние является состоянием тупика. Например, процесс D не может быть редуцирован, поскольку он запрашивает ресурс 3, а тот уже предоставлен процессу E.

Обнаружение взаимоблокировок при наличии нескольких ресурсов каждого типа.

В этом случае используется другой подход. Пусть в системе n процессов $P_1 \ldots P_n$, m – число классов ресурсов, E_i – число ресурсов класса i, $1<=i<=m$, $E\{E_1,\ldots E_m\}$ – вектор существующих ресурсов. Он содержит общее количество имеющихся в наличии экземпляров каждого вида ресурса. Аналогично $A\{A_1 \ldots A_m\}$ – вектор доступных ресурсов, где A_i – это количество экземпляров ресурса i, не использующихся, т.е. доступных в текущий момент. Пусть C –матрица текущего распределения, где C_{ij} – количество экземпляров ресурса j, которое занимает процесс i. Таким образом, i строка матрицы показывает, какое количество ресурсов каждого класса использует процесс Pi. Аналогично R – матрица запросов, где Rij – количество экземпляров ресурса j, которое хочет получить процесс Pi.

$$C = \begin{bmatrix} C_{11}C_{12}C_{13}\ldots C_{1m} \\ C_{21}C_{22}C_{23}\ldots C_{2m} \\ C_{n1}C_{n2}C_{n3}\ldots C_{nm} \end{bmatrix} \quad R = \begin{bmatrix} R_{11}R_{12}R_{13}\ldots R_{1m} \\ R_{21}R_{22}R_{23}\ldots R_{2m} \\ R_{n1}R_{n2}R_{n3}\ldots R_{nm} \end{bmatrix}$$

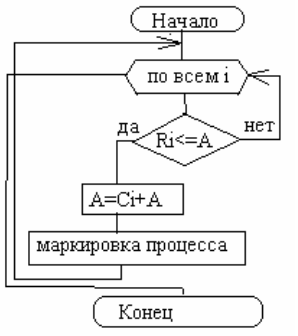
Если сложить все экземпляры ресурса j, предоставленные процессам и доступные в данный момент, то в результате получим существующее в системе количество экземпляров данного класса ресурсов, т.е. справедливо равенство

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Алгоритм обнаружения взаимоблокировок основан на сравнении векторов. Определим, что A<=B тогда и только тогда когда Ai<=Bi, $1<=i<=m$.

В начальном состоянии все процессы немаркированы. По мере выполнения алгоритма на процессы ставится отметка, служащая признаком того, что они могут завершить свою работу, т.е. для этого достаточно ресурсов и процесс не находится в тупике. После окончания алгоритма любой немаркированный процесс будет находиться в тупике.

Алгоритм ищет процесс, который может быть завершен. Для такого процесса все требуемые ресурсы должны находиться среди доступных в данный момент. Тогда найденный процесс сможет завершить свою работу и вернуть ресурсы в общий фонд доступных ресурсов. Процесс



отмечается как завершивший работу. Если в результате остались немаркированные процессы, то значит они не могут завершить работу, и следовательно блокированы.

Пример

	НМД	плоттеры	сканеры	CD	
E={	4	2	3	1}	- существующие ресурсы
A={	2	1	0	0}	- доступные ресурсы

М-ца текущего распределения					Матрица запросов				
	0	0	1	0		2	0	0	1
C=	2	0	0	1	R=	1	0	1	0
	0	1	2	0		2	1	0	0

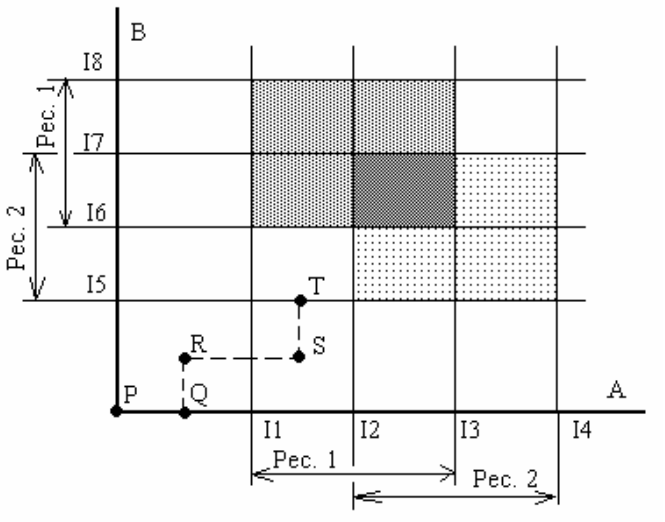
Имеется 3 процесса и 4 класса ресурсов: НМД, плоттеры, сканеры, CD. Первый процесс использует сканер и требует доступа к двум НМД и CD. Поскольку свободного CD нет, процесс не может быть завершен, R_1>A. Аналогично второй процесс. Однако третий процесс может завершиться, R_3=A. После завершения, A изменяется на {2 2 2 0}. Теперь может завершиться процесс 2, а после его завершения и освобождения распределенных ему ресурсов выполнится и процесс 1. Однако если бы строка R_3 выглядела не как 2 1 0 0 , а 2 1 0 1, то вначале ни один из процессов не смог бы функционировать, поскольку и 1 и 3 процессы нуждались бы в CD, который был у процесса 2. Этот процесс в свою очередь требует сканера, все из которых используются 1 и 3 процессами.

Возникновение взаимоблокировок можно проверять каждый раз, когда запрашивается очередной ресурс. Однако такой подход требует существенных временных издержек. Альтернатива: проверять периодически с интервалом в несколько минут.

Восстановление при помощи принудительной выгрузки ресурса. Иногда есть возможность отобрать ресурс у его владельца и временно отдать другому процессу. В этом случае часто требуется ручное вмешательство. Способность забирать ресурс у процесса, отдавать его другому, а потом возвращать назад так, что исходный процесс этого не замечает, в значительной мере зависит от свойств ресурса. Выйти из тупика таким образом зачастую трудно или невозможно. Выбор приостанавливаемого процесса зависит главным образом от того, какой процесс владеет ресурсами, которые могут быть легко отняты.

Восстановление через откат. Работа организуется таким образом, что процессы периодически **создают контрольные точки**. Состояние процесса записывается в файл, впоследствии процесс может быть восстановлен из этого файла. Контрольные точки содержат не только образ памяти, но и состояние ресурсов, т.е. какие из них в данный момент предоставлены процессу. Для повышения эффективности новая контрольная точка должна записываться не поверх старой, а в новый файл. При обнаружении взаимоблокировки требуется определить, каких ресурсов процессам не хватает. Чтобы выйти из тупика, процесс-владелец этого ресурса выполняет откат к той контрольной точке, когда ресурс был свободен. Вся работа, выполненная после этой КТ, теряется. Фактически процесс просто запускается с более раннего момента, повторяя часть уже выполненной работы. Однако, поскольку ресурс займет уже другой процесс, то выполнившему откакт придется ожидать освобождения ресурса.

Восстановление путем уничтожения процессов. Грубый, но наиболее эффективный способ. Уничтожается процесс, находящийся в цикле взаимоблокировки. Если после этого взаимоблокировка не исчезла, удаляют еще один процесс из цикла. Можно в качестве жертвы выбрать процесс не находящийся в цикле, чтобы освободились требуемые ресурсы. По возможности, следует уничтожать те процессы, которые можно запустить с самого начала безо всяких излишеств. Например, процесс, выполняющий компиляцию, можно уничтожить, а потом запустить заново – это ни на чем не отразится. А процесс, выполняющий добавление данных в таблицу, уничтожить нельзя – в этом случае часть данных будет добавлена дважды.



Избежание взаимоблокировок. При рассмотрении тупиков, предполагалось, что процесс запрашивает все требуемые ресурсы одновременно. Однако на самом деле они запрашиваются поочередно. Система должна уметь решать, является предоставление ресурса безопасным, и только если это так, предоставлять ресурс. Алгоритмы, позволяющие избегать тупиков, используют концепцию **безопасного состояния**. Рассмотрим модель с 2 процессами и 2 ресурсами разных типов. По горизонтали отрабатывает процесс A. По вертикали – B. В момент I1 процесс A запрашивает ресурс 1, в момент I2 – второй.. Он освобождает эти ресурсы в моменты I3 и I4 соответственно. Процесс B требует ресурса 2 в момент I5 до I7, а в момент I6 до I8 – ресурса 1. Точки PQRST – состояния системы. Изначально система находится в состоянии P. Планировщик запускает процесс A, который отрабатывает до точки Q, далее планировщик запускает процесс B, система попадает в точку R, и вновь процесс A. В однопроцессорной системе все линии будут либо вертикальными либо горизонтальными. Кроме этого линии могут идти только вверх или направо. Процесс A пересекает I1, запрашивая и получая ресурс 1. Точка S. Переключение на процесс B. При пересечении I5, B

запрашивает ресурс 2. Заштрихованные области представляют собой состояния, когда оба процесса должны использовать один и тот же ресурс. Учитывая, что взаимное исключение делает это невозможным, то попадания в эти области не происходит. Если система войдет в прямоугольник I1I2I5I6, то из него ей не будет выхода. Это взаимоблокировка. В точке T ситуацию еще можно спасти – дать работать процессу A, пока он не пересечет I4. В точке T процесс B запрашивает ресурс. Система должна принять решение – предоставлять его или нет.

Пусть вектора E и A – существующие в системе ресурсы и доступные соответственно, а матрицы C и R -матрица текущего распределения и запроса соответственно. Состояние **безопасно**, если система не находится в тупике и существует такой порядок планирования, при котором каждый процесс может работать до завершения, даже если все процессы захотят немедленно получить свое максимальное количество ресурсов.

	Макс.	**Имеет**									
		Безопасно					**Небезопасно**				
A	9	3	3	3	3	3	3	4	4	4	
B	4	2	4	0	0	0	2	2	4	0	
C	7	2	2	2	7	0	2	2	2	2	
Свободно	10	3	1	5	0	7	3	2	0	4	

Например, в системе 3 процесса, и свободно 10 экземпляров ресурса одного типа. Процессу A требуется максимум 9 экземпляров, B – 4, C – 7. Пусть в какой-то момент они имеют соответственно 3,2,2 экземпляра. Если процесс B запросит оставшиеся 2 ресурса, он их получит, отработает и освободит, в системе станет 5 свободных ресурсов. Если теперь процесс C запросит недостающие ему 5 ресурсов, он их сможет получить, отработает и освободит. Теперь 7 ресурсов предостаточно, чтобы завершить процесс A. Рассмотрим небезопасную ситуацию. Процесс A запрашивает еще один ресурс и получает его. В системе остается 2 свободных экземпляра. Процесс B запрашивает 2 ресурса, освобождает. Но 4 свободных экземпляров недостаточно ни процессу A, ни процессу C для завершения. Значит процессу A нельзя было предоставлять еще один ресурс, поскольку при этом система перешла из безопасного в небезопасное состояние. Небезопасное состояние еще не означает тупика. Действительно, процесс A может успеть частично освободить ресурсы, позволяя процессу C завершиться, после чего уже потребовать максимально возможное число ресурсов. Т.е. небезопасное состояние приводит к тупику лишь потенциально. Безопасное же состояние гарантирует, что тупика не произойдет, т.е. все процессы завершат свою работу. Однако эта модель требует выполнения следующих условий:

- число пользователей и ресурсов фиксировано;
- число работающих пользователей остается постоянным;
- клиенты должны гарантированно возвращать ресурсы;
- максимальные требования процесса к ресурсам должны быть известны заранее.

Алгоритм банкира. Алгоритм планирования, позволяющий избегать взаимоблокировок был разработан Дейкстрой в 1965 году, и называется алгоритмом банкира. Он представляет собой расширение алгоритма обнаружения тупиков, рассмотренный ранее. Модель основана на примере банкира в небольшом городке, который имеет несколько клиентов (процессов), которым выдает по несколько кредитов (ресурсов). При заявке клиента на получение еще одного кредита, проверяется, хватит ли у банкира сумм, чтобы в итоге все клиенты смогли завершить свои операции. Если нет, то запрос отклоняется. Фактически алгоритм идентичен алгоритму поиска взаимоблокировок. При запросе очередного ресурса, система проверяет, что произойдет, если ресурс будет выделен. Фактически моделируется ситуация, при которой процесс получил требуемый ресурс и выполняется проверка на взаимоблокировку. Если ее нет, состояние системы останется безопасным и ресурс процессу предоставляется. Если же обнаруживается взаимоблокировка, то состояние становится небезопасным, и запрос отклоняется. Однако подобный алгоритм заранее требует полных сведений о том, сколько ресурсов может потребоваться процессу. Кроме того, количество процессов динамически изменяется в процессе функционирования системы. Более того, ресурсы, считающиеся доступными, могут перестать такими быть в результате поломки или отказа. В результате на практике рассмотренный алгоритм практически не используется.

Предотвращение взаимоблокировок в реальных системах основано на использовании сформулированных выше условий Коффмана. Рассмотрим условие взаимного исключения. Если в системе нет ресурсов, отдаваемых в единоличное владение одному процессу, взаимоблокировка невозможна. Однако на практике очень часто невозможно предоставить одновременный доступ нескольких процессов к ресурсу. Например, невозможно позволить разным процессам одновременно печатать на принтере. Правда, в этом случае можно избежать взаимоблокировки, используя подкачку. В этой модели только один процесс реально запрашивает физический принтер, являясь демоном печати. Процессы ставят задания в очередь, а демон организует их поочередную печать. Но и такая модель доступна не для всех видов ресурсов. Кроме того, конкуренция за дисковое пространство для подкачки сама может привести к тупику.

Условие удержания и ожидания. Можно попытаться уберечь процессы, занимающие некоторые ресурсы от ожидания остальных ресурсов, взаимоблокировок не будет. Один из возможных путей решения в том, чтобы процесс запрашивал все необходимые ресурсы до начала работы. В результате или процесс получает все необходимое для работы, что исключает взаимоблокировки или ожидает ресурсы, не блокируя другие процессы. Проблема в том, что предварительно не всегда известно, какие ресурсы будут требоваться. Кроме этого, ресурсы используются не оптимально.

Условие отсутствия принудительной выгрузки ресурса. Уже рассматривали, что далеко не всегда есть возможность принудительно отобрать ресурс у процесса.

Условие циклического ожидания. Циклическое ожидание можно устранить несколькими способами. 1. Процесс имеет право только на один ресурс в конкретный момент времени. Если нужен второй ресурс, процесс должен освободить первый. Это также не всегда возможно. 2. Общая нумерация всех ресурсов. Процессы могут запрашивать столько ресурсов сколько захо-

тят, но запросы должны быть сделаны в соответствии с нумерацией ресурсов. Из двух требуемых ресурсов сначала должен запрашиваться ресурс с меньшим номером. В этом случае граф распределения ресурсов никогда не будет иметь циклов. Однако не всегда возможно выполнить требуемую нумерацию ресурсов. В реальных системах ресурсов столь много (области таблицы процессов, дисковое пространство подкачки, записи баз данных, прочие абстрактные ресурсы), что систематизировать их и предусмотреть все возможные варианты не представляется возможным.

Двухфазовое блокирование. Если процессу требуется несколько ресурсов, он пытается их заблокировать (получить) по одному. Если получить все необходимые ресурсы не удалось, выполняется освобождение занятых ресурсов и попытка повторяется. На этой первой фазе не выполняется никаких реальных действий. Получив требуемые ресурсы, выполняется обработка и освобождение ресурсов -–это вторая фаза. Фактически подход требует грамотного программирования, кроме того, он не обобщается на системы реального времени. Используется при построении БД. Подход выражается в понятии транзакции -–неделимой последовательности действий с БД в т.ч. и над несколькими записями. Действия или выполняются все, или не выполняется ни одно.

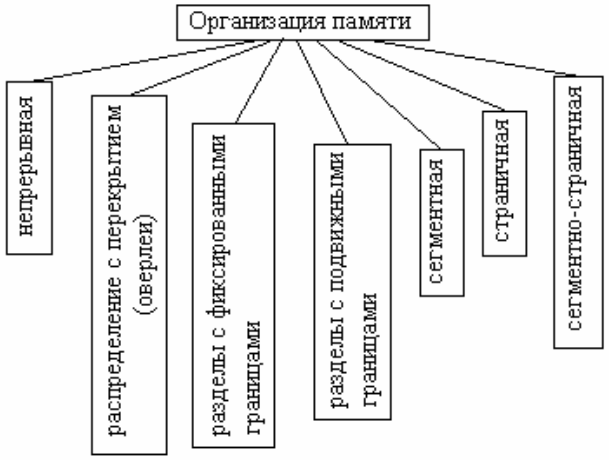
Тупиковые ситуации возникают и **без использования ресурсов**. Например. каждый из процессов просто ожидает, когда второй выполнит какое-либо действие.

Существует понятие **голодания**. При использовании некоторых алгоритмов планирования может возникать ситуация, когда некоторые процессы вообще не получат требуемый ресурс. Происходит голодание. Ситуация предотвращается, если используется стратегия ”первым пришел – первым обслужен”.

Управление памятью

Управление памятью. Задачи управления памятью. Однозадачная система. Сегментная организация памяти. Настройка адресов и защита памяти. Подкачка. Битовые массивы и связные списки. Алгоритмы выбора сегмента для размещения процесса. Страничная организации памяти. Виртуальная память. Таблицы страниц. Многоуровневые таблицы страниц. Буферы быстрого преобразования TLB. Инвертированные таблицы страниц. Обработка страничного прерывания. Поддержка сегментно-страничной организации памяти в процессорах семейства Pentium. Структура элемента таблицы страниц. Таблица страничных блоков. Отдельные пространства команд и данных. Совместно используемые страницы. Проблемы, возникающие при вызове страничного прерывания. Распределение оперативной памяти в Windows NT, UNIX

Память в современных ПК представляет собой иерархическую структуру. Небольшая часть –быстрая, дорогая, энергозависимая кэш-память. Среднее кол-во среднескоростной, средней по цене, энергозависимой памяти ОЗУ, и большое кол-во медленной, дешевой, энергонезависимой памяти на жестком диске. Одна из задач ОС – координация использования всех этих составляющих. Часть ОС, отвечающая за управление памятью называется **менеджером памяти**. Он отслеживает, какая часть памяти свободна, какая занята, при необходимости выделяет память ресурсам и освобождает их по завершении процесса, управляет обменом между ОЗУ и диском. В общем случае адреса в программе могут быть абсолютными, т.е. компилятор генерирует код с непосредственными физическими адресами. Могут быть относительными. В этом случае процесс загружается в память целиком, однако адрес его загрузки заранее неизвестен. Компилятор генерирует специальную таблицу настройки адресов, которая используется при загрузке процесса. Термин **виртуальная память** используется, когда физический адрес страницы может изменяться в процессе выполнения программы в результате выгрузки и загрузки в другую область. В этом случае настройка адресов происходит аппаратно-программным способом. Рассмотрим методы организации памяти.

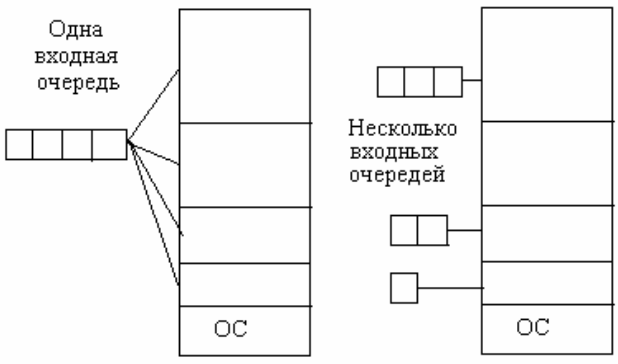


Непрерывная организация. В этом случае фактически вся память разделена на три области: память, занимаемая ОС, память, занимаемая программой, свободная память. ОС может находиться в нижней части памяти, начиная с нулевых адресов ОЗУ. Эта модель ранее применялась на майнфреймах и миникомпьютерах, в настоящее время малоупотребима. ОС может располагаться в верхних адресах памяти, в ПЗУ. Этот вариант используется в некоторых карманных компьютерах и встроенных системам. Третий вариант – драйверы устройств находятся вверху, в ПЗУ, а остальная часть ОС в ОЗУ. Этот вариант устанавливался на ранних моделях ПК, например, MS-DOS. В случае непрерывной организации в каждый конкретный момент времени может работать только один процесс. Как только с терминала вводится команда. ОС копирует заданную программу с диска в память, запускает ее. После окончания ждет новой команды. Новая программа в памяти грузится поверх старой.

Использование **оверлеев** предполагает, что вся программа может быть разбита на части – сегменты. Оверлейная программа имеет одну главную часть и несколько сегментов, причем в памяти одновременно могут находиться только часть сегментов, включая главный. Структура использовалась, когда логическое адресное пространство программы было больше, чем свободная память .Пока в ОЗУ находятся выполняющиеся сегменты, остальные находятся во внешней памяти. После завершения текущего сегмента либо он сам обращается к ОС с указанием, какой сегмент теперь необходимо загрузить, либо возвращает управление главному сегменту, который и взаимодействует с ОС, какой сегмент выгрузить, а какой загрузить. Вначале обязанность выполнить соответствующие системные вызовы ложилась на программистов, потом системы программирования стали выполнять это автоматически при указании соответствующего режима.

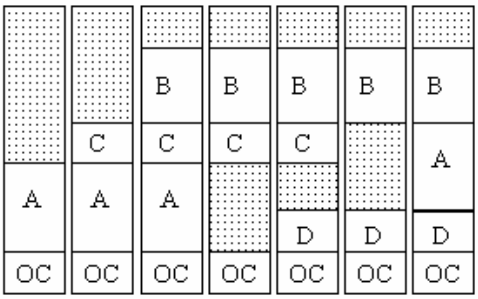
Фиксированные границы раздела. Весь объем ОЗУ разбивается на несколько, возможно, различных по объему, разделов. В каждом разделе в один момент времени может располагаться только один процесс. При этом можно использовать и оверлейные структуры. Первые мультипрограммные ОС строились по такому принципу.

В случае отдельных очередей к каждому из сегментов возникает проблема: к некоторым большим сегментам нет очереди, в то же время небольшие задачи могут организовать значительную очередь, ожидая запуска, хотя память в принципе свободна. В случае общей очереди имеется несколько стратегий выбора раздела для размещения задачи. 1 вариант. Задача, находящаяся ближе всего к началу очереди и подходящую для выполнения в освободившемся разделе, загружается и начинается обработка. 2 вариант. Небольшие задачи нежелательно загружать в большие сегменты. В очереди происходит поиск наибольшей из помещающихся в разделе задач, которая и запускается. Чтобы избежать дискриминации маленьких задач, в системе существует хотя бы один раздел достаточно маленького размера. Второй вариант – задача не может быть пропущена более заданного числа раз.



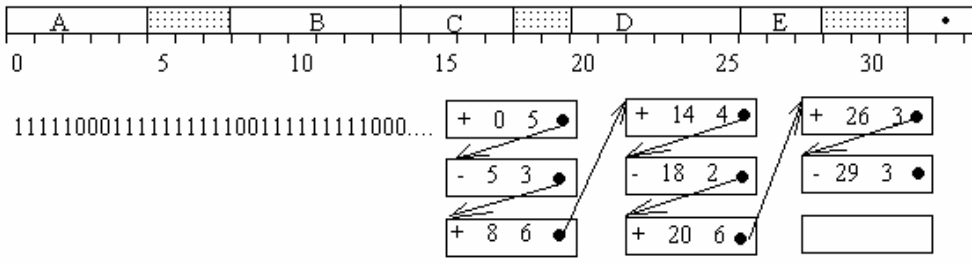
В любом случае существует проблема – часть памяти в каждом разделе не используется, что не рационально. Использование двух принципов решает эту проблему: выделять ровно такой памяти, какой необходим процессу, размещать процесс не в одной непрерывной области, а в нескольких, в т.ч. и не смежных.

Разделы с подвижными границами. В этих системах использованы указанные выше принципы. При этом используется **своппинг или подкачка**. Каждый процесс полностью переносится в память, работает, затем целиком выгружается на диск. Основное отличие в том, что количество, размещение и размер разделов изменяются динамически по мере поступления и завершения процессов. Это улучшает использование памяти, но усложняет операции размещения процессов и освобождения памяти.



Как видно из рисунка, возникает фрагментация памяти. Из-за сильной фрагментации может сложиться ситуация, когда диспетчер не сможет образовать новый раздел, хотя общий объем свободной памяти и достаточен для запуска процесса. В этом случае может использоваться **уплотнение памяти** – когда все разделы сдвигаются, устраняя **фрагментацию**. На это может потребоваться достаточно длинное время. Помимо фрагментации имеется и еще одна проблема – процесс может потребовать дополнительного распределения памяти уже во время выполнения. Какой объем памяти предоставлять процессу при образовании сегмента? Как правило, им предоставляют несколько больше памяти, чем необходимо. Если же и эта память закончится, то либо процесс переносится в больший сегмент, либо выгружается на диск до момента, когда станет возможным организовать сегмент необходимого размера. При этом возможна следующая схема: область данных и стек, которые по мере необходимости растут по направлению друг к другу. Свободная область используется как куча для распределения памяти под данные.

Многозадачность требует решения двух важных проблем – **настройка адресов и защита**. Поскольку до запуска программы неизвестно, по каким адресам она будет располагаться, при выделении сегмента памяти требуется настройка адресов в коде программы именно на выделенный блок. Один из вариантов решения – при запуске изменять адреса. Для этого в код программы добавляется компоновщиком таблица настройки адресов, используя которую при запуске и происходит перезапись. Однако это не решает проблемы защиты – любая программа может использовать любой адрес и соответственно, поменять код или данные в ином сегменте. Второй вариант решения, который одновременно решает и проблему защиты – использование базового и предельного регистров. В первых из них заносится адрес начала раздела памяти, а во второй помещается длина раздела. К каждому адресу перед его использованием в команде автоматически аппаратно добавляется значение базового регистра. Дополнительно проверяется, что адрес не вышел за границу, дозволенную предельным регистром. Пользовательские программы не могут поменять значения этих регистров. Схема использовалась на первом суперкомпьютере CDC6600. Сейчас схема используется редко.



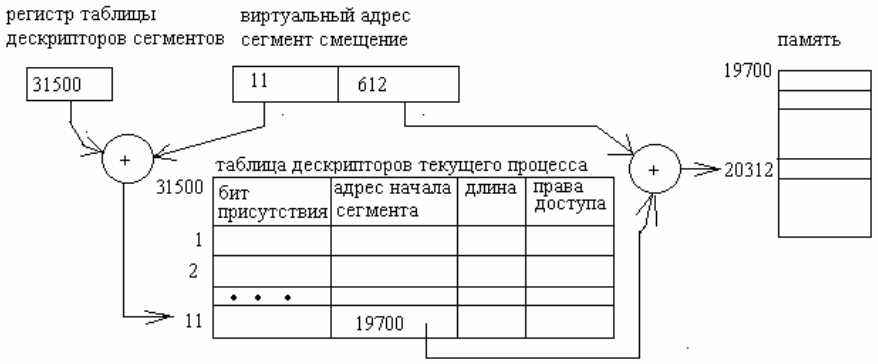
При динамическом выделении памяти этим процессом должна управлять ОС. Существует два варианта управления: **использование битовых массивов** и **связных списков**.

В случае битовой карты каждому блоку соответствует один бит – 1, если блок занят и 0, если свободен. При размещении нового процесса длиной N блоков требуется найти последовательную серию из N нулей. Это медленная операция. В случае связных списков для каждого блока формируется элемент, содержащий флаг, занят блок или свободен, его начальный адрес, длину, указатель на следующий элемент списка. Список может быть и двусвязным. Закончившийся процесс имеет два соседних блока. Если оба они заняты, требуется только лишь изменить флаг занятости. Если занят один из соседей, то требуется еще и удалить один из элементов списка, а в оставшемся скорректировать начало и длину блока. В случае, если оба соседних блока свободны, из трех блоков остается только один с соответствующей корректировкой.

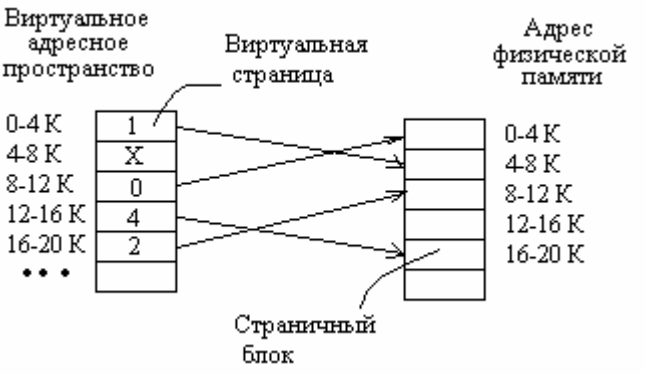
Существует несколько алгоритмов выбора свободных участков для размещения процесса. **Первый подходящий**. Менеджер последовательно просматривает список и как только находит подходящий, распределяет его. В итоге участок делится на два – занятый под процесс и оставшийся свободным остаток. Поиск максимально уменьшен, если списки отсортированы по адресам, и алгоритм довольно быстр. **Следующий подходящий**. Похож на первый. Отличие в том, что после поиска запоминается положение указателя поиска в списке, и в следующий раз поиск продолжается с остановленного места. Производительность у этого алгоритма несколько ниже. **Самый подходящий**. Полный поиск по всему списку, выбирается подходящий фрагмент минимального размера. Производительность его еще ниже, поскольку он заполняет память большим количеством небольших областей, т.е. фрагментирует. **Самый неподходящий**. Выбирается самый большой подходящий участок. После размещения процесса остается свободным достаточно большой фрагмент для размещения нового процесса. Средняя производительность. Любой из алгоритмов ускоряется, если списки занятых и свободных областей ведутся отдельно. Однако тогда увеличивается сложность и происходит замедление при освобождении областей, поскольку требуется еще и переместить элемент из одного списка в другой**. Быстрый подходящий**. Дополнительно поддерживаются списки блоков наиболее употребительных размеров.

Оставшиеся три метода называют разрывными – при их использовании задача помещается уже не в один непрерывный блок, а в несколько. Для этого дополнительно требуется относительная адресация.

Сегментный. Программа разбивается на части и уже каждой из частей предоставляется память. В результате программа представляет собой множество сегментов. Обращение к элементам программы выглядит в этом случае как указание имени сегмента и смещение относительно его начала. В итоге виртуальный адрес состоит из двух полей – номер сегмента и смещение. Сегментная организация требует поддержки на уровне системы программирования, которая преобразует имена сегментов в их номера и определяет их объем. Для каждого процесса ОС строит таблицу **дескрипторов сегментов**, в которой отмечает местоположение сегмента в оперативной или внешней памяти. В ней имеются следующие поля: бит присутствия, указывающий в каком типе памяти находится сегмент, адрес начала сегмента и его длина, тип (код или данные), права доступа, информация об обращениях к этому сегменту, на основании которой ОС принимает решение о замещении сегмента при необходимости загрузки нового. При передаче управления новому процессу ОС заносит в соответствующий регистр адрес таблицы дескрипторов сегментов данного процесса. В оперативной памяти могут размещаться не все сегменты, а только те, с которыми идет реальная работа. Общий объем виртуальной памяти. предоставляемой задаче может быть значительно больше, чем объем предоставленных ей сегментов. Однако нельзя загрузить в память слишком большое количество задач – тогда потребуется частое переключение и соответственно загрузка-выгрузка сегментов, что резко снизит производительность. Если требуемого сегмента в памяти нет, то происходит прерывание и диспетчер памяти передает управление программе загрузки сегмента. Пока происходит поиск сегмента во внешней памяти, диспетчер определяет подходящее место. Если свободного места нет, принимается решение о том, какой из сегментов будет выгружен (своппинг сегментов). Интересной возникающей возможностью следует считать разделяемые программные модули. Сегмент с разделяемым кодом располагается в памяти в единственном экземпляре, а ссылки на его использование будут содержаться в нескольких таблицах дескрипторов сегментов разных процессов. Для снижения потерь времени на вычисление необходимых адресов используется кэширование таблицы дескрипторов – она располагается в быстродействующем по сравнению с остальной памятью кэше. Сегментную организацию использовала OS/2 v.1.

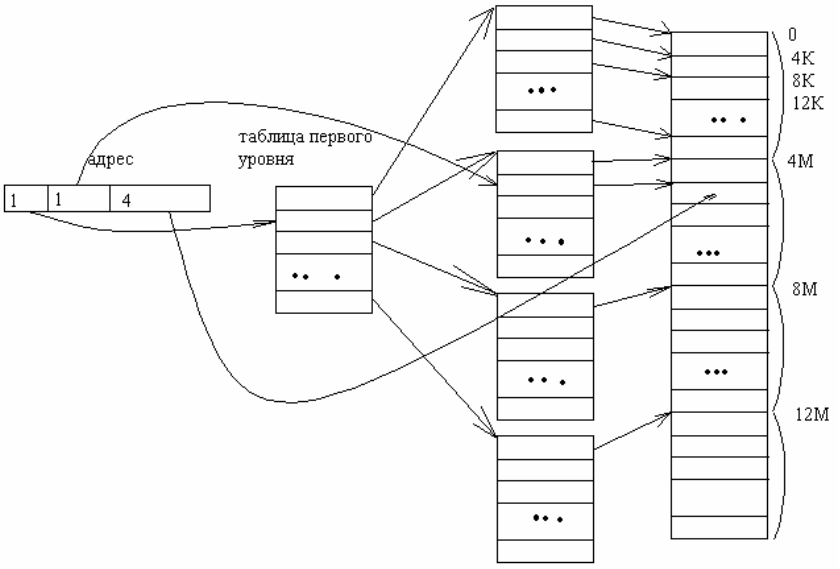


Страничный. В отличие от предыдущего варианта длина сегмента строго фиксирована и имеет заданный размер. Такие сегменты называются страницами. Часть страниц находится в ОЗУ, остальные на диске, в файле подкачки (свопп-файле). В UNIX для выгруженных страниц используется специально выделенный раздел. Аналогично используется таблица страниц. Основное отличие в том, что дескриптор страницы не имеет поля длины. Длина страницы выбирается кратной степени 2. Если размер страницы достаточно велик, то операция сложения может быть заменена сдвигом. В случае, если когда даже часто используемые страницы не помещаются в свободную область ОЗУ возникает **пробуксовка** – ситуация, когда при обращении к странице памяти происходит выгрузка текущей страницы на диск. Аналогично используется кэширование таблицы дескрипторов. Недостаток в том, что программа разбивается на страницы случайным образом, без учета логических связей между частями, что приводит к слишком частой перезагрузке страниц.



Сегментно-страничный. Как и в сегментном способе, программа разбивается на логически законченные части – сегменты. Однако память разбита на фиксированное число страниц. Виртуальный адрес представлен тремя полями: номер сегмента, номер страницы в сегменте, смещение внутри страницы. Сначала вычисляется адрес таблицы страниц сегмента, путем сложения начального адреса таблицы дескрипторов сегментов и номера сегмента, по таблице страниц сегмента и номеру страницы определяется физический адрес страницы, к которому наконец добавляется смещение. Издержки на вычисление адреса еще больше увеличиваются. Аналогично используется кэширование.

Многоуровневые таблицы страниц. Большие страницы не устраняют фрагментацию, поэтому размер страницы невелик. Тогда, если размер страницы 4К, то для 32-разрядного адреса потребуется около миллиона страниц. Размер таблицы очень велик. Чтобы не хранить ее постоянно в памяти используют многоуровневую таблицу страниц. В этом случае адрес разбивается на три части. Первая часть представляет собой номер строки в таблице верхнего уровня. В ней хранится адрес таблицы второго уровня для данного диапазона адресов. Вторая часть адреса используется как номер страницы для таблицы второго уровня, третья – как смещение. Так, если каждая из таблиц второго уровня соответствует 4 Мб, и размер страницы 4 Кб, то адрес в примере равен 1048576*4+1024+4. При такой организации нет необходимости держать в памяти таблицу страниц для всего виртуального адресного пространства в 4 Гб. Достаточно только страницы первого уровня, и N страниц второго, в зависимости от объема процесса (1 таблица второго уровня позволяет адресовать до 4 Гб). Наиболее распространен формат 32битной записи для каждой страницы. В ней содержится: номер страничного блока, бит присутствия, биты доступа, изменения и обращения, блокирования кэша. Бит присутствия определяет, находится ли данная виртуальная страница в физической памяти или нет. Биты защиты определяют права доступа, в простейшем случае – чтение-запись или только для чтения. Бит изменения устанавливается, когда содержимое страницы изменилось. Если ОС решает выгрузить такой блок, его содержимое сначала надо сохранить на диск. Бит обращения устанавливается, когда происходит обращение к данной странице. Он помогает ОС при выборе страницы для замещения. Последний бит позволяет запретить кэширование страницы. Это важно для страниц, отображающихся не на память, а на регистры устройств. Если идет ожидание ответа от устройства вв, то нужно получить именно новые данные, а не копию из кэша со старыми данными. Бит не нужен, если архитектура машины имеет отдельное адресное пространство вв, не отображаемое на память. 2 уровня – платформа Intel, 3 – Sun SPARC, DECAlpha, заданное число уровней – Motorola. Ряд архитектур, например, MIPS R2000 (RISC) не поддерживают таблицы страниц, перекладывая организацию поиска нужной страницы на ОС.



Буферы быстрого преобразования адреса TLB. Хранение в памяти таблиц оказывает значительное влияние на производительность. Однако большинство программ обращаются в основном только к небольшому четко ограниченному набору страниц, к остальным же обращение происходит редко. Машина снабжается аппаратным блоком, служащим для отображения виртуальных адресов в физические без использования таблицы страниц. Как правило, оно позволяет иметь порядка 64 записей о страницах. Фактически это ассоциативное ЗУ, дублирующее информацию об наиболее используемых страницах. При обращении к странице памяти параллельно по всей TLB проверяется номер страницы на совпадение. Если указанная страница имеется в TLB ее адрес поступает на выход. Если же ее там нет, адрес берется из обычной таблицы и вдобавок размещается в TLB. Теперь если обращение к этому адресу произойдет повторно, он уже будет храниться в быстром ассоциативном ЗУ. Многие современные RISC-компьютеры (SPARC, MIPS, Alpha, HP PA) выполняют страничное управление программно. В них записи TLB явно загружаются ОС. Если запись в буфере не найдена, диспетчер памяти вместо того, чтобы переходить к таблице страниц для поиска, формирует ошибку и передает управление ОС. Система находит страницу, замещает запись в буфере TLB и перезапускает прерванную инструкцию. В результате диспетчер памяти организуется значительно проще, что позволяет иметь на кристалле больше кэша

Инвертированные таблицы страниц. Чтобы снизить размер таблиц, используют таблицы не для виртуального адресного пространства, а для физического. Однако в этом случае перевод виртуального адреса в физический значительно усложняется.. Этот подход используется на машинах PowerPC, на ряде рабочих станций IBM и HP.

В большинстве архитектур адресное пространство для кода и данных едино, однако существуют архитектуры, в которых адресные пространства данных и кода отдельны. В этом случае оба адресных пространства могут иметь страничную организацию независимо друг от друга. Каждое из них обладает своей собственной таблицей страниц и собственным отображением виртуальных страниц на физические страничные блоки.

Подкачка страниц работает лучше, если в системе имеется достаточное количество свободных блоков, которые можно запросить при страничном прерывании. Для поддержания этого количества, во многих странично-организованных системах имеется фоновый процесс, который периодически проверяет состояние памяти. Если блоков мало, он начинает выбирать страницы в памяти для освобождения.

В ряде систем над картой памяти имеется определенный программный контроль. В этом случае программисты могут именовать область памяти, тогда один процесс сможет передать другому имя области памяти, и второй сможет ею пользоваться. В этом случае реальной становится высокая пропускная способность совместного доступа – один процесс пишет в разделяемую память, а другой читает из нее. Этот же механизм может использоваться для построения высокопроизводительных систем передачи сообщений. Еще одна разновидность совместного доступа – распределенная память совместного доступа. В этом случае несколько процессов в сети могут совместно использовать набор страниц. В случае возникновения страничного

прерывания, обработчик определяет машину, которая содержит страницу, и посылает ей сообщение с просьбой выгрузить и переслать по сети.

Обработка страничного прерывания.

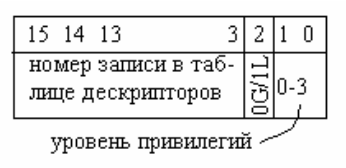
Выполняется следующая последовательность действий:

1. Аппаратное обеспечение переключает систему в режим ядра, сохраняя счетчик команд в стеке. На большинстве машин в специальных регистрах процессора сохраняется некоторая информация о состоянии текущей инструкции.
2. Запускается написанная на ассемблере программа, сохраняющая основные регистры и другую изменяющуюся информацию, защищая ее от разрушения ОС. Эта программа вызывается ОС как процедура.
3. ОС обнаруживает, что произошло страничное прерывание, и пытается найти необходимую виртуальную страницу. Часто требуемую информацию содержит один из аппаратных регистров. Если нет, ОС должна достать из стека счетчик команд, выбрать инструкцию, и программно проанализировать ее, чтобы определить, что она делала в тот момент, когда случилась ошибка.
4. Как только становится известен виртуальный адрес, вызвавший прерывание, система проверяет, имеет ли силу этот адрес, и согласуется ли защита с доступом. Если нет, то процессу посылается сигнал или процесс уничтожается. Если адрес действителен и не произошло ошибки защиты, система проверяет наличие свободных страничных блоков. Если свободных блоков нет, запускается алгоритм замещения страниц.
5. Если выбранный страничный блок был изменен, страница заносится в график записи на диск и происходит переключение контекста, приостанавливающее вызвавший прерывание процесс и позволяющее работать другому процессу до тех пор, пока не будет выполнен перенос страницы на диск. В любом случае блок отмечается как занятый, чтобы предотвратить его использование в других целях.
6. Как только страничный блок очищается, ОС ищет адрес на диске, где находится требуемая страница, и планирует дисковую операцию для ее переноса в память. Во время загрузки страницы процесс, вызвавший прерывание, все еще приостановлен, и выполняется другой пользовательский процесс, если такой доступен.
7. Когда дисковое прерывание отмечает, что страница поступила в память, обновляется таблица страниц, отражая ее позицию, а блок помечается как находящийся в нормальном состоянии.
8. Прерванная команда возвращается к тому состоянию, с которого она начиналась, и значение счетчика команд приостановленного процесса (в стеке или в системной ячейке памяти) корректируется так, чтобы указывать на эту команду.
9. Прерванный процесс вносится в график, и ОС возвращает управление ассемблерной процедуре, вызвавшей ее.
10. Эта процедура перезагружает регистры и другую информацию о состоянии и возвращает управление в пользовательское пространство для продолжения выполнения пользовательской программы, как если бы никакого прерывания не происходило.

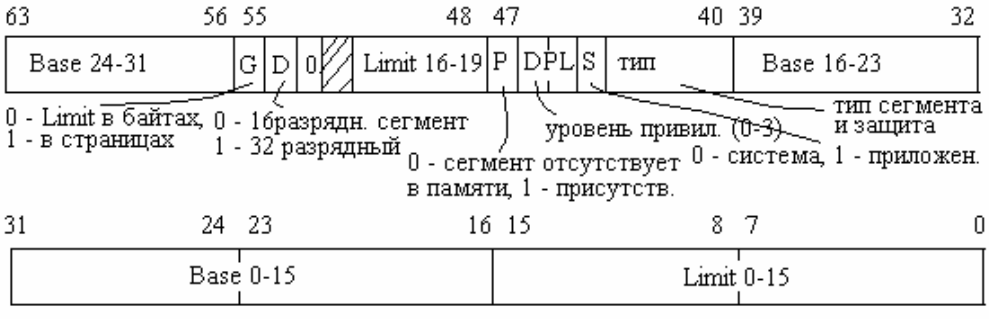
Чтобы перезапустить текущую команду, ОС должна определить, где находится первый байт команды. Но значение счетчика команд зависит от того, какой операнд вызвал ошибку, и от реализации микрокода контроллера. Поэтому зачастую ОС не в состоянии точно определить, где начиналась команда. В некоторых процессорах эта проблема решается путем наличия на кристалле скрытых регистров, в которые переносится содержимое счетчика команд перед выполнением каждой операции.

Рассмотрим еще один момент. Пусть один процесс ожидает завершения операции вв. Второй, активный процесс, вызывает страничное прерывание. Если алгоритм подкачки глобальный, то есть шанс, что для удаления из памяти будет выбрана страница, содержащая буфер ввода-вывода. Если в этот момент устройство ВВ как раз и записывало данные в буфер, то выгрузка этой страницы приведет к тому, что часть данных запишется в буфер, а часть – во вновь загруженную страницу. Решение в том, чтобы блокировать страницы, занятые ВВ, от выгрузки из памяти. Такое блокирование носит название **пришпиливания**.

Поддержка сегментно-страничной организации памяти в системах на основе Pentium.

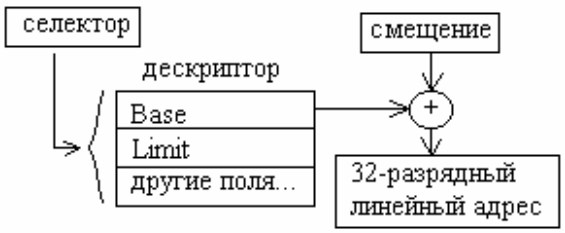


Система Pentium поддерживает 16К независимых сегментов виртуальной памяти процесса , каждый объемом до 1 млрд. 32 разрядных слов. Основа виртуальной памяти состоит из двух таблиц: локальной таблицы дескрипторов LDT и глобальной GDT. LDT имеется своя у каждого процесса, GDT одна, используемая совместно всеми процессами. LDT таблица описывает сегменты, локальные для каждой программы – код, данные, стек и т.д. GDT несет информацию о системных сегментах, включая саму ОС.



При получении доступа к сегменту, программа сначала загружает для этого сегмента в один из 6 сегментных регистров процессора селектор. Регистр CS содержит селектор для сегмента кода команд, DS - данных. Селектор представляет собой 16разрядую структуру. Один бит несет информацию, является ли данный сегмент локальным или глобальным. Еще 13 определяют номер записи в таблице дескрипторов, каждая из которых имеет длину 8 байт. Соответственно таблица дескрипторов не может иметь более 8К записей. Селектор 0 является запрещенным. Его можно загрузить в сегментный регистр, чтобы обозначить, что этот регистр недоступен. При попытке обращения к такому регистру, возникнет прерывание. После определения, в какой таблице расположен соответствующий дескриптор, селектор копируется во внутренний рабочий регистр, и три младших бита приравниваются к 0. После

этого к нему прибавляется адрес соответствующей таблицы, чтобы получить прямой указатель на дескриптор. Например, код 48h ссылается на 9 запись в глобальной таблице, которая имеет смещение 48h (9*8=72=48h) от начала таблицы. При загрузке селектора в регистр соответствующий дескриптор извлекается из таблицы GDT или LDT и сохраняется в микропрограммных регистрах, что обеспечивает к нему быстрый доступ. Дескриптор имеет размер 8 байт следующей структуры. Пара селектор-смещение при выполнении кода должна преобразовываться в физический адрес. Как только из кода микропрограммы становится ясно, какой сегментный регистр используется, во внутренних регистрах находится полный дескриптор, соответствующий этому селектору. Если сегмент не существует (селектор равен 0), или в данный момент выгружен, возникает прерывание. Далее микропрограмма проверяет, выходит ли смещение за пределы сегмента, и если это так, вызывается прерывание. Для определения размера в дескрипторе имеется поле Limit длиной 20 бит. Если поле G (granularity – детализация) =0, то Limit содержит точный размер сегмента размером до 1 Мб. Если 1 – то размер сегмента указан в страницах вместо байт. При размере страницы 4 Кб, этого достаточно для адресации сегментов размером 2^{32} байт. После всех проверок система прибавляет 32-разрядное поле Base дескриптора к смещению, формируя т.н. **линейный адрес**. Поля Base и Limit разбиты на части для совместимости с устаревшими системами, например, Base в i80286 имеет только 24 бита.



Если разбиение на страницы блокировано с помощью бита в глобальном управляющем регистре, линейный адрес интерпретируется как физический адрес и используется для чтения записи памяти. Фактически это чистая схема сегментации с базовым адресом сегмента, определяемым дескриптором.

Если страничная организация не отключена, линейный адрес интерпретируется как виртуальный и отображается на физический с помощью таблицы страниц. Однако при этом при 32 разрядном виртуальном адресе и странице размером 4 Кб, сегмент может содержать до 1 миллиона страниц, поэтому используется двухуровневое отображение чтобы уменьшить размер таблицы страниц. Линейный адрес представляется в этом случае тремя полями: каталог 10 бит, Страница 10 бит и смещение 12 бит. Таблица первого уровня, страничный каталог, содержит 1024 32 разрядных записи и располагается по адресу, хранящемуся в глобальном регистре. Поле каталог является индексом для этой таблицы (номером записи). Запись содержит адрес таблицы страниц, содержащую 1024 32-разрядных записей. Поле страница указывает номер записи в этой таблице. Запись указывает на соответствующий страничный блок, поле смещение определяет смещение относительно начала блока. Каждая таблица страниц управляет фактически 4 Мб памяти. Чтобы избежать повторного обращения к памяти, имеется небольшой буфер TLB, который напрямую отображает наиболее часто используемые комбинации каталог-страница на физический адрес страничного блока.

В случае, когда не требуется сегментной организации и достаточно только страничной, сегментные регистры все настраиваются селектором, в соответствующем дескрипторе которого Base=0 и Limit установлено на максимум. Тогда смещение команды будет линейным адресом. Все современные ОС работают именно по такой схеме, т.е. возможности сегментной организации не используются. Эти возможности поддерживала OS/2.

Элемент таблицы страниц второго уровня в Win32 имеет следующую 32 битную структуру:

Номер страничного блока (фрейма страницы)	U	P	Cw	Gl	L	D	A	Cd	Wt	O	W	V
31 – 12	11	10	9	8	7	6	5	4	3	2	1	0

Он состоит из 2 больших полей – номера страницы в физической памяти (или ее физического адреса) и поля атрибутов. Атрибуты:
U –резерв, в многопроцессорных системах указывает, можно ли записывать на эту страницу
P – резерв
Cw – резерв
Gl –Global – трансляция относится ко всем процессам
L –Large page – резерв, для элемента каталога страниц указывает, что элемент относится к 4 (2) Mb странице
D –Dirty – страница модифицирована
A –Accessed – была операция чтения с данной страницы
Cd –Cashe disabled – кэширование данной страницы отключено
Wt –Write through – отключает кэширование записи на данную страницу, в результате чего все измененные данные сбрасываются непосредственно на диск
O –Owner – указывает, доступна ли страница из кода пользовательского режима
W –Write – в многопроцессорных системах указывает, можно ли записывать на эту страницу, в однопроцессорных – тип доступа (для чтения и записи или только для чтения)
V – Valid – указывает, соответствует ли элемент странице в физической памяти
В случае, если страница не является действительной, т.е. младший бит = 0, состав и назначение остальных полей изменяется.

Современные Windows системы поддерживают механизм проецирования памяти PAE (Physical Address Extension). При соответствующей поддержке чипсетом, этот режим позволяет адресовать 64 Гб физической памяти или 1024 Гб на платформе x64. Widows ограничивает возможности этого режима до 128 Гб из-за размера таблицы страничных блоков. В этом режиме фактически используется 3-уровневая таблица страниц. Соответственно виртуальный адрес делится на 4 поля: 2 бита – индекс указателя на каталог страниц, 10 бит – номер таблицы страниц в каталоге страниц, 8 бит – номер страницы в таблице страниц, 12 бит – смещение на странице. Платформа x64 использует 4 уровневую схему таблицы страниц. Пока для виртуального адреса используется не 64 , а только 48 бит: 9,9,9,9 и 12 бит на смещение.

Таблица страничных блоков. Windows поддерживает базу данных PFN (Page Frame Number), определяющую состояние каждой страницы физической памяти. Состояния страницы могут быть следующими:

- активная (действительная)(Active/valid). Является либо частью рабочего набора процесса или ОС, либо не входит не в один рабочий набор, но на нее ссылается действительный элемент таблицы страниц (PTE – Page Table Entry).
- Переходная (Transition). Временное состояние страницы, не принадлежащей ни одному рабочему набору. Страница находится в этом состоянии в ходе операций ввода-вывода.
- Простаивающая (stand by) Страница входила ранее в рабочий набор, но теперь удалена из него. С момента последней записи на диск не изменялась. PTE все еще ссылается на нее, но уже помечен как недействительный и находящийся в переходном состоянии.
- Модифицированная (Modified). Страница входила ранее в рабочий набор, но теперь удалена из него. Однако она была изменена и еще не записана на диск. PTE все еще ссылается на нее, но уже помечен как недействительный и находящийся в переходном состоянии. Перед повторным использованием страницы она должна быть записана на диск.
- Модифицированная, но не записываемая (Modified no-write). Аналогичная ситуация, однако подсистема записи модифицированных страниц не будет записывать ее на диск. Используется драйверами файловой системы.
- Свободная (free). Свободна, но содержит какие-то данные. Нельзя передать пользовательскому процессу, пока не произойдет обнуление.
- Обнуленная (Zeroed) Свободна и инициализирована нулевыми значениями.
- Только для чтения (ROM). Ошибка страницы была вызвана из памяти только для чтения (Windows XP)
- Аварийная (Bad) Страница вызвала ошибку четности или другую аппаратную ошибку. Больше использовать нельзя.

Запись базы PFN имеет фиксированную длину, однако ее структура зависит от состояния страницы. Так, она может включать индекс рабочего набора, который содержит виртуальный адрес, по которому проецируется эта страница; Адрес PTE, указывающий на данную страницу, счетчик числа ссылок, счетчик числа пользователей, Тип страницы, флаги, исходное содержимое PTE, указывающего на страницу, что позволяет его восстанавливать, когда физическая страница более не резидентна, и ряд других полей.

В каждый момент времени программа находится на одном из имеющихся 4 уровней защиты, что отмечается 2 битовым полем в регистре слова состояния программы (PSW). Каждый сегмент системы также имеет свой уровень. Как правило, 3 уровень – это пользовательские программы, 2 – библиотеки совместного доступа, 1 – системные узлы и 0 – ядро. Разрешен доступ к данным на своем и более высоких уровнях. При попытке доступа к данным низкого уровня вызываются прерывания. Вызов процедур как высокого, так и низкого уровня допускается, однако для этого инструкция CALL должна содержать селектор вместо адреса. Этот селектор определяет дескриптор, т.н. **шлюз вызова** (call gate), который передает адрес вызываемой процедуры. Т.о., попасть в середину произвольного сегмента кода другого уровня невозможно. Могут использоваться только стандартные точки входа.

Программные и аппаратные прерывания используют подобный механизм. Они также обращаются к дескрипторам , а не к абсолютным адресам, которые указывают на определенные процедуры. Поле тип в дескрипторе позволяет различить программные сегменты, сегменты данных и различные виды шлюзов.

Win 9x. ОС этого семейства являются 32 разрядными, многопоточными ОС с вытесняющей многозадачностью. Пользовательский интерфейс – графический. При загрузке используется ОС MS DOS 7.X. В случае, если в файле MSDOS.SYS установлено BootGUI = 0, то процессор работает в реальном режиме. Распределение памяти MS-DOS этой версии не отличается от предыдущих версий DOS. При загрузке GUI перед загрузкой ядра Win 9x процессор переключается в защищенный режим и распределяет память с помощью страничного механизма, т.е. используется плоская модель памяти, при которой все возможные сегменты, доступные программисту, совпадают и имеют максимально возможный размер. Каждая прикладная программа определяется 32 битными адресами, единственный сегмент программы отображается непосредственно в область виртуального линейного адресного пространства, состоящего из страниц размером по 4 Кб.

0-64 Кб В эту область не имеют доступа 32-разрядные программы, что позволяет выполнить перехват неверных указателей, однако 16-разрядные программы могут выполнить запись в эту область.
64 Кб - 4 Мб Компоненты реального режима. Эта область используется всеми процессами. Это сделано для обеспечения совместимости с драйверами устройств реального режима, резидентными программами и некоторыми 16разрядными программами Win. Это снижает надежность
4 Мб – 2 Гб Прикладные программы Win32. У каждой прикладной программы имеется свое собственное адресное пространство. Оно невидимо для других процессов и они как правило не могут получить к нему доступ. Однако в принципе это возможно, поскольку не используются все аппаратные возможности защиты.
2 Гб - 4 Гб Отображаются в адресное пространство каждой программы и совместно используются. Это позволяет обслуживать вызовы API непосредственно в адресном пространстве прикладной программы. Естественно это снижает надежность.
2 Гб – 3 Гб Системные dll, прикладные программы Win16, совместно используемые dll. Все 16битные программы Win разделяют общее адресное пространство.
3 Гб – 4 Гб 32разрядные микропроцессоры i80x86 имеют четыре уровня (кольца) защиты. Кольцо 0 самое защищенное здесь. К нему относятся следующие компоненты: собственно ядро windows, подсистема управления виртуальными машинами, модули файловой системы, виртуальные драйверы.

Минимально допустимый объем ОЗУ, с которым Win9x может функционировать -4 Мб, однако при этом система практически висит из-за необходимости подкачки практически каждой страницы, к которой происходит обращение. Файл подкачки

по умолчанию находится в системном каталоге Windows, имеет переменный размер. При этом файл естественно фрагментируется, что снижает оперативность доступа. Файл подкачки фиксированного размера позволяет увеличить быстродействие. Соответствующие параметры прописаны в SYSTEM.INI в секции 386Enh.

```
PagingDrive = C:
PagingFile = C:\PageFile.sys        // имя и местоположение файла подкачки// Win386.swp
MinPagingFileSize = 65536           // его размер
MaxPagingFileSize = 262144
```

Win NT. Аналогично используется плоская модель памяти. Ядро системы и несколько драйверов работают в 0 кольце защиты в отдельном адресном пространстве. Остальные программные модули ОС, являясь серверными процессами по отношению к пользовательским программам, также имеют свое собственное виртуальное адресное пространство, которое недоступно пользовательским процессам.

0 – 64 Кб полностью недоступная область

64 Кб – 2 Гб Прикладные программы Win32 со своим собственным виртуальным адресным пространством. Прикладные программы полностью изолированы друг от друга, хотя могут общаться через буфер обмена (clipboard), а также механизмы DDE (Dynamic Data Exchange – механизм динамического обмена данными) и OLE (Object Linking and Embedding) – механизм связи и внедрения объектов.

В верхней части каждой 2Гб области прикладной программы размещен код системных dll 3 кольца, который перенаправляет вызовы в совершенно изолированное адресное пространство, содержащее собственно системный код. Этот системный код, выступающий в роли сервер-процесса, проверяет значения параметров, выполняет запрошенную функцию и возвращает результат назад в адресное пространство прикладной программы. Оставаясь процессом прикладного уровня, сервер-процесс полностью защищен от прикладной программы.

2 Гб - 4 Гб. Код ядра (0 кольцо защиты). Здесь располагаются низкоуровневые системные компоненты, в т.ч. ядро, планировщик потоков и диспетчер виртуальной памяти.

Для 16разрядных прикладных Win программ реализуются сеансы WOW (Windows on Windows), что позволяет выполнять 16разрядные приложения не только в разделяемом адресном пространстве, но и при необходимости в собственном пространстве памяти. Независимо от этого может использоваться механизм OLE. Может одновременно выполняться несколько сеансов DOS.

При запуске приложения создается процесс со своей информационной структурой. В его рамках запускается поток. При необходимости этот поток может запускать другие потоки. Потоки одного процесса выполняются в едином виртуальном адресном пространстве, процессы – в разных. Отображение виртуальных адресных пространств на физическую память реализует сама ОС. Процессами управления памятью управляет диспетчер виртуальной памяти VMM (virtual memory manager). При этом используется достаточно сложная стратегия учета для минимизации доступа к диску.

Каждая виртуальная страница памяти, отображаемая на физическую страницу, переносится в страничный фрейм. Прежде чем код или данные можно будет переместить с диска в память, VMM должен найти или создать свободный страничный фрейм или фрейм заполненный нулями, что отвечает требованиям безопасности уровня С2. Для замещения страниц используется дисциплина FIFO, что снижает эффективность. Размер файла подкачки по умолчанию устанавливается равным объему ОЗУ + 12 Мб.

Объекты, создаваемые и используемые ОС и приложениями, хранятся в пулах памяти. Доступ к пулам может быть получен только в привилегированном режиме работы процессора. Объекты перемещаемого пула при необходимости могут быть выгружены на диск. Неперемещаемый пул содержит объекты, которые должны постоянно находиться в памяти – например, структуры данных, используемые процедурами обработки прерываний.

Алгоритмы замещения страниц

Стратегии выборки по запросу и с упреждением. Алгоритмы замещения страниц. Алгоритм NRU, LRU, FIFO, NFU. Бит использования страницы. Понятие рабочего набора. Аномалия Билэди. Рабочие наборы в Windows

При управлении памятью используются 2 стратегии: **стратегия выборки** – в какой момент переписывать страницу из вторичной памяти в первичную. Два основных варианта – **по запросу** и с **упреждением**. В первом случае страница алгоритм работает тогда, когда программа обращается к отсутствующей странице памяти, которая находится на диске. Во втором случае помимо отсутствующей страницы загружаются и несколько окружающих ее страниц, в предположении, что ближайшие адреса тоже будут необходимы.

Стратегия **размещения** – в какой участок первичной памяти поместить новую страницу. Стратегия **замещения** – какую страницу вытолкнуть во внешнюю память, если свободной страницы для размещения нет.

Алгоритмы замещения страниц.
При отсутствии в памяти требуемой страницы, ОС должна не только найти ее и загрузить, но и еще и определить, вместо какой страницы будет подгружена новая. Простейший вариант – случайным образом – не позволяет достичь максимальной производительности.
NRU – не использовавшаяся в последнее время страница. Используется информация бит изменения и обращения. Важно реализовать изменение этих бит при каждом обращении к памяти, поэтому необходимо, чтобы они задавались аппаратно. Когда процесс запускается оба бита всех его страниц равны нулю. Периодически, например, по таймеру, бит обращения очищается, чтобы отличить страницы, к которым давно не было обращений от используемых. Когда возникает страничное прерывание, система проверяет все страницы. Они делятся на 4 класса:
0 – не было изменений и обращений
1 – не было обращений, но страница изменена
2 – было обращение, страница не изменилась
3 – были и изменения и обращения.
Алгоритм NRU (not recently used) – замещает страницу в непустом классе с наименьшим номером. Считается, что лучше выгрузить измененную страницу, к которой не было обращений хотя бы в течение последнего тика таймера, чем страницу, к которой такие обращения были.
Алгоритм FIFO. Система поддерживает список всех страниц, которые хранятся в памяти, причем в порядке их поступления. Поэтому самая первая страница является и самой старой. В результате удаляется страница, находящаяся в начале списка, а новая добавляется в его конец. В таком варианте алгоритм используется редко.
Вторая попытка. Дополнительно изучается бит обращений. Если он равен 0, старая страница удаляется, если же нет, то страница переносится в конец списка, бит обнуляется, и вновь проверяется страница, находящаяся теперь первой. Т.е. алгоритм ищет самую старую страницу, к которой не было обращений. Если обращения были ко всем страницам, алгоритм вырождается в обычный FIFO. Эффективность алгоритма снижается из-за необходимости перемещать страницы по списку.

Интуитивно кажется, что чем больше страниц на диске, тем меньше страничных прерываний вызывается программой. Рассмотрим FIFO с 3 страницами и 4:

	0	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Самая новая страница	0	1	2	3	0	1	4	4	4	2	3	3	
		0	1	2	3	0	1	1	1	4	2	2	
Самая старая страница			0	1	2	3	0	0	0	1	4	4	
	P	P	P	P	P	P	P			P	P		9 страничных прерываний

	0	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Самая новая страница	0	1	2	3	3	3	4	0	1	2	3	4	
		0	1	2	2	2	3	4	0	1	2	3	
			0	1	1	1	2	3	4	0	1	2	
Самая старая страница				0	0	0	1	2	3	4	0	1	
	P	P	P	P			P	P	P	P	P	P	10 страничных прерываний

Как видно, система с большим числом страниц при определенной стратегии замещения и определенной последовательности обращений к страницам, может вызвать большее число страничных прерываний, т.е. оказаться менее производительной. Такая ситуация называется **аномалией Биледи**.
Существует класс **магазинных алгоритмов**, у которых выполняется следующее условие: $M(m,r) \subseteq M(m+1,r)$, где m-число страничных блоков, r-индекс в последовательности обращений, M – страничный массив. Т.е. множество виртуальных страниц, после r обращений попавших в физическую память, для памяти, имеющей m страничных блоков, также попадает в физическую память, если она состоит из m+1 блока. Они не подвержены аномалии Биледи. В частности, алгоритм LRU удовлетворяет этому требованию.

Часы. Записи хранятся в списке в виде кольца, и имеется текущий указатель. При необходимости замещения проверяется та запись, на которую направлен текущий указатель. Если бит обращений равен 0, на ее место загружается новая страница, а указатель перемещается к следующей записи. Если 1 – бит сбрасывается, указатель перемещается и вновь выполняется проверка бита.

Алгоритм LRU last recently used- дольше всего не использовавшаяся страница. Замещается та страница, к которой дольше всего не было обращений. Для реализации алгоритма необходим список страниц, где последняя использовавшаяся страница находится в начале списка, а дольше всего неиспользуемая – в конце. Список должен обновляться при каждом обращении к памяти. Другой вариант реализации – в таблице добавляется поле, хранящее значение таймера. При замещении ищется страница с наименьшим значением. Еще один вариант реализации – аппаратно поддерживается матрица NxN, где N – число страниц. Изначально матрица нулевая. При обращении к блоку i, всем битам строки i присваивается 1, затем всем битам столбца i присваивается 0. В любой момент времени строка, двоичное значение которой наименьшее, является не используемой дольше всего.

Алгоритм NFU. not frequently used – редко используемая. Разновидность предыдущего алгоритма. С каждой страницей памяти связан программный счетчик, изначально равный нулю. Периодически (по таймеру) бит использования прибавляется к счетчику. При замещении выбирается страница с наименьшим значением счетчика. Проблема в том, что если какая-то часть программы работала долго, но теперь уже не используется, то последующие части все равно будут с меньшим значением счетчика, и соответственно, произойдет удаление используемых страниц.

Старение. Модификация предыдущего алгоритма: бит использования добавляется не в правый, а в левый бит счетчика, и перед добавлением счетчик сдвигается вправо.

Биты R для страниц 0-5 по тактам				
101011	110010	110101	100010	011000
Счетчики страниц по тактам				
10000000	11000000	11100000	11110000	01111000
00000000	10000000	11000000	01100000	10110000
10000000	01000000	00100000	00010000	10001000
00000000	00000000	10000000	01000000	00100000
10000000	11000000	01100000	10110000	01011000
10000000	01000000	10100000	01010000	00101000

Алгоритм Рабочий набор. Во время выполнения фрагмента кода, процесс обращается как правило к небольшой части своих страниц. Множество страниц, которое процесс использует в данный момент, называется рабочим набором. Если рабочий набор целиком находится в памяти, процесс будет работать практически не вызывая прерываний из-за отсутствия страницы. Если же в памяти не удается разместить весь рабочий набор, процесс вызовет большое количество страничных прерываний, в результате замедляя работу. При повторном запуске процесса вновь будут вызываться страничные прерывания до тех пор, пока в памяти не окажется весь страничный набор. Многие системы со страничной организацией пытаются отследить рабочий набор каждого процесса и сохраняют его до запуска нового процесса. Загрузка страниц перед тем, как разрешить процессу работу, называется опережающей подкачкой страниц. Если ОС постоянно отслеживает рабочий набор процесса, то при необходимости замены страницы в памяти можно реализовать следующую стратегию: замена той страницы, которая не входит в рабочий набор. Фактически отслеживается, какие из страниц использовались за последние k тактов таймера. Страницы, использовавшиеся в этот отрезок времени, входят в рабочий набор. Алгоритм достаточно громоздок, поскольку при каждом страничном прерывании требует проверки таблицы страниц.

Алгоритм WSClock. Основан на часовом алгоритме, но с использованием информации о рабочем наборе. Ведется структура в виде кольцевого списка страничных блоков. Вначале список пустой. По мере прихода страниц, они поступают в список, формируя кольцо. Каждая запись, кроме бит R и M, содержат время последнего использования. Если бит R равен 1, это значит, что страница использовалась за последний тик таймера, и не является оптимальным кандидатом на удаление. Если бит R равен 0, и времени от момента последнего использования прошло много, то страница не входит в рабочий набор, и в данный страничный блок просто загружается новая страница. Если у такой страницы были изменения, и ее необходимо сохранить на диск. Чтобы избежать переключения процессов, запись на диск записывается в очередь планировщика, а указатель-стрелка на начало (текущую запись)

В многозадачных системах встает следующий вопрос: при необходимости замещения страницы следует учитывать только страницы активного процесса или же все страницы памяти? В первом случае речь идет о локальных алгоритмах замещения страниц, во втором – о глобальных. В целом глобальные алгоритмы работают лучше.

Windows поддерживает **рабочие наборы** – это подмножество виртуальных страниц, резидентных в физической памяти. № вида рабочих наборов: процесса, системы, сеанса. Диспетчер памяти использует алгоритм подкачки по требованию с кластеризацией, т.е. с упреждением. Используется 2 алгоритма замещения: LRU и FIFO. По умолчанию ОС устанавливает минимальную и максимальную величину рабочего набора для процесса – 50 и 345 страниц соответственно. Функция SetProcessWorkingSetSize позволяет их изменить при наличии привилегии Increase Scheduling Priority. Однако жестко установить лимиты позволяет только Windows Server 2003, в остальных случаях диспетчер памяти позволяет как превышать допустимый размер рабочего набора при наличии достаточного объема свободной памяти, так и уменьшать его ниже лимита при отсутствии подкачки и при потребности ОС в большом объеме физической памяти. Максимально допустимый размер рабочего набора колеблется от 1984 Мб до 8192 Гб в зависимости от версии Windows и аппаратной платформы.

Подсистемы ввода-вывода

Управление вводом-выводом. Блочные и символьные операции. Синхронные и асинхронные операции. Отображение ввода-вывода на адресное пространство памяти. Прямой доступ к памяти. Кэширование операций. Упреждающее чтение. Отложенная запись. Программное обеспечение ввода-вывода. Драйверы UNIX. Псевдоустройства. Переключатели устройств. Файл устройства. Драйверы Windows. Процесс загрузки драйверов. Дерево устройств. Дисциплины оптимизации запросов чтения-записи

Система вв, способная объединить в одной модели широкий набор устройств, должна быть универсальной. Кроме того, необходимо обеспечить доступ к устройствам вв множества параллельных задач. Используется следующий принцип: любые операции по управлению вв объявляются привилегированными и могут выполняться только кодом самой ОС. За управление вв отвечает компонент ОС, называемый супервизором ОС. Он выполняет следующие действия:

- получает запросы на вв от прикладных задач и модулей ОС., проверяет их на корректность, и или обрабатывает их дальше или выдает соответствующее сообщение
- вызывает распределители каналов и контроллеров, планирует вв, помещает задачи в очередь
- инициирует операции вв, передавая управление соответствующим драйверам, если при этом используются прерывания, предоставляет процессор диспетчеру задач для смены контекста
- при получении сигналов прерываний идентифицирует их и передает управление соответствующему обработчику
- выполняет передачу сообщений об ошибках, если они произошли в процессе операции вв
- передает сообщения о завершении операции вв ожидающему процессу

Уровни вв можно представить следующей иерархией: 1. ПО вв уровня пользователя. Функции: обращение к вызовам вв, форматированный вв, спулинг. 2. Устройство-независимое ПО ОС. Ф: именование, защита, блокирование, буферизация, назначение увв. 3. Драйверы устройств. Установка регистров устройства, завершение операции вв. 4. Обработчики прерываний. активирование драйвера при завершении операции вв. 5. Аппаратура. выполнение операции вв.

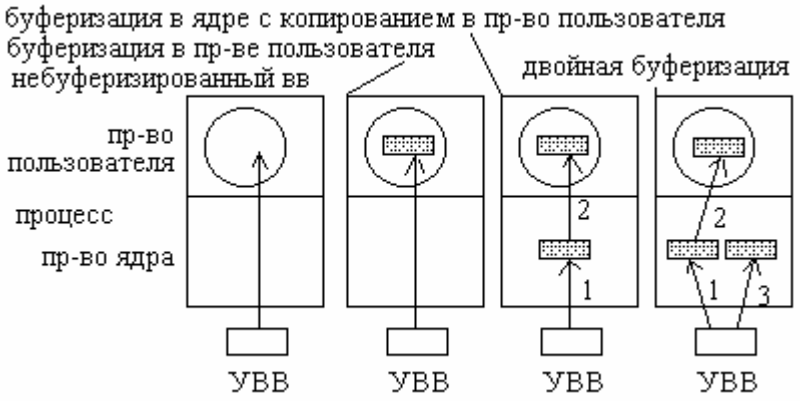
Устройства вв можно разделить на два класса: **блочные**, когда возможно чтение-запись данных блоком и имеют четко выраженную адресную структуру, например, диск, и **символьные**, которые принимают или передают поток символов без какой-либо блочной структуры, которые не являются адресуемыми и не выполняют операцию поиска. Электронная составляющая устройства вв называется **контроллером** или **адаптером**.
Интерфейс между устройством и контроллером является интерфейсом очень низкого уровня. Например, контроллер диска при чтении сектора принимает поток бит, содержащий заголовок сектора, собственно биты данных и контрольную сумму ECC. Он преобразует поток бит в последовательность байт, сравнивает ECC, после чего операция считается выполненной и передается в память.
У контроллера имеются регистры, с которыми может взаимодействовать центральный процессор для управления устройством, выбора режима и т.д. Может присутствовать буфер данных, с которым также могут выполняться операции чтения-записи. 2 способа доступа к управляющим регистрам и буферам устройств вв. 1. Каждому регистру устройства назначается номер порта, тогда операции вв могут выглядеть как IN REG, PORT или OUT PORT, REG. При такой схеме адресное пространство вв и памяти не пересекается, а существует раздельно. 2. Все регистры устройств отображаются на адресное пространство памяти, т.е. регистру присваивается номер ячейки памяти, как правило это верхние адреса диапазона (например, Motorola 680x0). Используются и гибридные варианты, например, в x86 существует адресное пространство портов вв от 0 до 64К, а адресное пространство памяти от 640К до 1М зарезервировано под буферы устройств. Отображение на адресное пространство памяти имеет следующие преимущества: не требуется ассемблерных вставок на языках высокого уровня для команд IN и OUT; не требуется специальных механизмов защиты от процессов пользователей при доступе к устройствам вв, достаточно исключить ту часть адресного пространства, на которую отображаются управляющие регистры увв из адресного пространства пользователей; при отображении разных увв на разные страницы памяти доступ пользователей можно ограничивать выборочно. Недостатки этого решения: в современных системах используется кэширование памяти, что требует запрещения кэширования отображаемого адресного диапазона, иначе при обращении к увв будут считаны данные из кэша а не из реального устройства; модули памяти должны отслеживать диапазон адресов и реагировать только на свой, также как и увв; увв не могут отследить обращения к памяти, если они происходят не по общей шине, а по прямой шине процессор-память. В x86 во время загрузки в специальные регистры моста шины PCI загружаются значения отображаемого диапазона. При обращении к памяти с этими адресами, обращение передаются не на прямую шину процессор-память, а на шину PCI.

Существует вариант **прямого доступа к памяти**, DMA (direct memory access), при котором устройство может переслать/принять данные напрямую из памяти. Для этого необходим аппаратный контроллер DMA, который и выполняет доступ к системной шине независимо от центрального процессора. Контроллер может находиться как интегрировано в увв, так и на материнской плате, обслуживая в последнем случае несколько увв. DMA контроллер содержит управляющие регистры, доступные ЦП, в которых указывается номер порта вв, направление пересылки данных, единица переноса (побайтно или пословно0), размер переносимого блока. Механизм выглядит следующим образом. ЦП программирует контроллер DMA, указывая, какие данные и куда перемещать. Далее процессором дается команда контроллеру диска прочитать данные во внутренний буфер. Как только контроллер диска сообщает, что операция выполнена, в работу включается DMA. Он выставляет запрос на перенос одного слова, получает доступ к шине и выполняет передачу, затем аналогично для следующего и т.д. Если ЦП в этот момент нужна шина, он ожидает, так как шина занята DMA. Такой механизм называется **захват цикла**, он требует выставления запроса при передаче каждого слова, забирая случайный цикл шины у ЦП и притормаживая его. Существует **пакетный режим**, когда запрос выставляется один раз на серию пересылок, в этом случае ЦП может простаивать достаточно долго, ожидая освобождения шины. По завершении операции переноса данных, контроллер DMA инициирует прерывание процессора, сообщая, что перенос данных завершен. В результате ОС нет необходимости заниматься переносом

данных в память, они уже там. Контроллеры DMA различаются по сложности. Простые могут выполнять одну операцию за один раз, более сложные имеют несколько каналов, каждый со своим набором управляющих регистров. Иногда в DMA контроллерах доступен режим, когда контроллер устройства пересылает слово данных контроллеру DMA, который затем выставляет на шину еще один запрос для передачи его, куда нужно. Такая схема позволяет передать данные напрямую между устройствами, минуя память, однако требует лишнего цикла шины. Большинство контроллеров DMA работает с физическими адресами, однако некоторые позволяют работать с виртуальным адресом (например, SPARC), тогда контроллер DMA использует менеджер памяти MMU для преобразования адреса. Для этого MMU должен быть частью памяти, а не процессора. Как правило, контроллер DMA значительно медленнее ЦП. Поэтому DMA невыгодно использовать, если ЦП в системе слабо загружен. Кроме того, машина без DMA, где все пересылки выполняются программно, оказывается дешевле, что важно, например, для встроенных систем.

Имеется 2 основных режима обмена с устройствами вв: **синхронный** (блокирующий) и **асинхронный** (управляемый прерываниями). В первом случае подав сигнал на операцию вв драйвер периодически опрашивает состояние устройства, до получения сигнала готовности. Во втором случае вместо цикла опроса управление переключается на другую задачу, а сигнал готовности трактуется как запрос на прерывание, которое позволит продолжить обработку операции вв. При этом существует механизм тайм-аута, позволяющий отреагировать на ситуацию, когда устройство не ответило в течение заданного времени. Организация обмена в режиме прерываний эффективнее, но сложнее в организационном плане. Например, в Win9x, NT драйвер печати работает через параллельный порт не в режиме прерываний, а в режиме опроса готовности – это 100 загружает процессор и снижает эффективность. Другие задачи в это время получают управление только благодаря вытесняющей многозадачности.

Как правило, при обмене используется буферизация, в общем случае это необходимо из-за различной скорости выполнения операций на стороне приемника и передатчика. **Варианты буферизации**. Без буферизации. Недостаток в том что процесс должен быть активирован при приеме каждого символа. 2 вариант. Процесс предоставляет буфер определенного размера. Обработчик прерываний активирует процесс только при заполнении буфера. Недостаток. Страница памяти с буфером может оказаться выгруженной при приходе денных. Фиксация буферов в памяти с запретом выгрузки снижает производительность. 3 вариант. Обработчик помещает данные в буфер ядра. При его заполнении данные переносятся в буфер пользователя. Недостаток в том, что данные могут поступить при переносе в буфер пользователя, когда буфер ядра заполнен. 4. Двойная буферизация. При заполнении буфера данные начинают помещаться во второй буфер. Как только данные из первого буфера поместятся в буфер пользователя, роль буферов меняется. Аналогичные схемы возможны и при передаче данных.



При чтении часто используется **кэширование**. В результате, если пользовательский процесс многократно обращается к одним и тем же данным, они только при первом запросе будут прочитаны с диска, а при последующих – из кэша. Кроме того используется механизм **упреждающего чтения**. При запросе каких либо данных, читается и несколько дополнительных следующих за ними блоков (секторов). Это ускоряет операции чтения, поскольку часто следующие затребованные задачей данные находятся в соседнем блоке. Следует различать кэш контроллера диска и кэш ОС. Кэш контроллера содержит обычно содержит блоки, на которые запрос еще не поступал, но которые удобно было прочитать, так как они оказались под головкой при чтении других блоков. Кэш ОС состоит из блоков, на которые были явные запросы, и которые по расчетам ОС могут снова понадобиться в ближайшем будущем. При записи используется механизм **отложенной записи**. Данные сначала изменяются только в кэше и помечаются как отложенная запись, в результате процессу нет необходимости ожидать завершения медленной операции вв, он может продолжить работу непосредственно. Система позже запишет данные на диск. Дополнительно такой подход позволяет оптимизировать операции записи на диск. В Win используется стратегия активного кэширования – под кэш отводится вся свободная память. В результате потенциально возможна ситуация, когда кэш вырастает настолько, что большинство страниц памяти оказываются сброшенными на диск в своп-файл. Поэтому можно ограничить допустимые размеры кэша и размеры блоков данных в кэше. В Win 9x в System.ini в разделе [vcache] задаются параметры MinFileCache (в Кб), MaxFileCache (в Кб), ChunkSize. В Win NT 4, 2000, XP это делается через реестр. В разделе HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Mamagment ключ IOPageLockLimit задает объем физической памяти для хранения буферов дискового кэша в байтах. Виртуальный размер системного кэша по умолчанию равен 64МБ. Если в системе более 4К страниц (16 Мб) физической памяти, виртуальный размер кэша равен 128 Мб + 64 Мб на каждые дополнительные 4 Мб физической памяти. Если на платформе x86 виртуальный размер кэша превышает 512 Мб, он ограничивается 512 Мб.

Для устройств, которые не которые не могут быть разделены несколькими процессами, используется запрос на **монопольное** использование, захват в случае успешного выполнения запроса и освобождение. Однако предоставление такой возможности пользовательским процессам может не только снизить производительность но и сделать устройство полностью недоступным. Для таких устройств часто используется **спулинг**. При этом создается специальный процесс **демон**, и каталог или очередь спулинга. Все данные на передачу в виде ссылок или имен файлов помещаются в очередь, а доступ к устройству реально имеет только демон.

С концепцией независимости от увв связан принцип единообразного именования. Имя файла или устройства должно быть текстовой строкой или целым числом и не должно зависеть от физического устройства. Например, в UNIX диски могут произвольно монтироваться в иерархию файловой системы и пользователь может не знать какое имя какому устройству соответствует, т.е. все файлы и устройства адресуется одним способом – по пути.

Основной принцип разработки ПО – независимость от физических устройств. Для этого вводится понятие виртуального устройства. Однако для управления конкретным устройством требуется специальная программа, называемая **драйвером**. Как правило, для каждого устройства или класса близких устройств требуется свой драйвер под каждую ОС. Чтобы получить доступ к аппаратной части устройства, т.е. к регистрам контроллера, драйвер должен быть частью ядра ОС. Возможно использование архитектур, при которых драйверы будут реализованы в пространстве пользователя, но в существующих архитектурах такой подход не используется. Драйверы пишутся иными разработчиками, нежели ОС. Значит должна существовать строго определенная модель функций драйвера и его взаимодействия с остальной частью ОС. В большинстве ОС определен стандартный интерфейс, который должны поддерживать все драйверы блочных устройств и второй стандартный интерфейс, который поддерживается драйверами символьных устройств. Интерфейсы включают набор стандартных процедур, которые могут вызываться остальной частью ОС. В ряде ОС драйверы компилируются вместе с ядром, например UNIX. При необходимости смены драйвера, ядро перекомпилируется. Другие ОС используют динамическую подгрузку драйверов при работе системы. Драйвер не только обрабатывает запросы на чтение-запись, но и при необходимости инициализирует устройство, управляет энергопотреблением и т.д. Общий план обслуживания запроса на передачу такой: проверка входных параметров, преобразование виртуальных адресов в физические, проверка, свободно ли устройство, проверка состояния, можно ли обслужить запрос сразу, определение серии необходимых команд, запись их в регистры контроллера. Далее либо ожидание, пока прерывание от устройства не разблокирует драйвер, либо устройство выполняет операцию сразу и ожидание не требуется. В любом случае, проверяется, были ли ошибки, при необходимости передача принятых данных выше по иерархии, передача информации о статусе завершения операции. При этом требуется обработка следующих ситуаций: увв прекращает операцию во время работы драйвера, поступление следующего пакета при еще не обработанном текущем, горячее отключение устройства в процессе обмена. Драйверам может быть разрешено обращение к некоторым системным процедурам. Стандартный набор операций драйвера включает:

- get – чтение символа, put – запись символа для символьных устройств
- read, write – чтение-запись блока, seek – перемещение указателя для доступа к нужному блоку, для блочных устройств
- ioctl – для передачи произвольной команды с произвольными параметрами (специфические для устройства команды)
- open – инициализация драйвера и устройства
- close – завершение работы с устройством, например, при отключении устройства
- poll – опрос состояния устройства
- halt – остановка драйвера при остановке ОС или выгрузке драйвера из памяти

Драйвер не всегда управляет физическим устройством. Он может использоваться в качестве интерфейса доступа, поддерживающего дополнительные функции. Например в UNIX драйвер mem позволяет считывать из адресов физической памяти или записывать по ним, устройство null разрешает только запись в себя, удаляя все получаемые данные, устройство zero является источником памяти, заполненной нулями. Такие устройства называют **псевдоустройствами.**

В **UNIX** драйвер делится на 2 части, верхнюю, содержащую синхронные процедуры, и нижнюю, содержащую асинхронные процедуры. Процедуры верхней половины могут обращаться к адресному пространству и области u вызывающего процесса, и если необходимо, могут переводить процесс в режим сна. Нижняя часть выполняется в системном контексте, и ее процедуры как правило не имеют отношения к текущему процессу. Структура данных, определяющая точки вхождения, поддерживаемые каждым устройством, называется **переключателем устройств**. Структура bdevsw – для блочных и cdevsw – для символьных:

```
struct bdevsw {
        int (*d_open)();
        int (*d_close)();
        int (*d_strategy)();
        int (*d_size)();
        int (*d_xhalt)();
        …
} bdevsw[];

struct cdevsw {
        int (*d_open)();
        int (*d_close)();
        int (*d_read)();
        int (*d_write)();
        int (*d_ioctl)();
        int (*d_mmap)();
        int (*d_segmap)();
        int (*d_xpoll)();
        int (*d_xhalt)();
        struct streamtab* d_str;
} cdevsw[];
```

В разных версиях структуры переключателей могут несколько отличаться. Переключатель определяет абстрактный интерфейс доступа к устройствам. Каждый драйвер предоставляет специфическую реализацию этих функций. Драйверы следуют стандартному соглашению об именовании функций переключателей, каждый драйвер использует двухбуквенную аббревиатуру для описания самого себя. Например, функции драйвера могут выглядеть как dkopen, dkclose и т.д. Устройство может поддерживать не все точки входа. Для них как правило используется общая процедура nodev(), просто производящая выход с кодом ошибки ENODEV. Для некоторых точек входа не определяется никаких действий, например, при закрытии. Тогда можно использовать общую процедуру nulldev(), которая производит выход с кодом 0.

Ядро идентифицирует каждое устройство по типу (блочное или символьное) и по паре номеров – старший и младший номер устройства. Старший номер определяет тип устройства, т.е. его драйвер. Младший номер определяет экземпляр устройства. Обычно они объединяются в одну переменную, старшие биты которой определяют старший номер, младшие – младший. Блочные и символьные устройства имеют отдельные независимые наборы старших номеров. Старший номер является индексом к таблице соответствующего переключателя. Если драйвер обслуживает несколько устройств различного типа, ему может быть назначено несколько старших номеров.

Поддерживается единый интерфейс доступа к файлам и устройствам с помощью **файла устройства**. Это специальный файл, ассоциированный с определенным устройством, и не располагаемый ни в каком определенном месте файловой системы. Все файлы устройств находятся в каталоге /dev или его подкаталогах. Пользователь может открывать или закрывать такой файл, читать и писать данные, выполнять позиционирование указателя как с обычным файлом. Стандартные потоки stdin, stdout, stderr перенаправляются командным интерпретатором в соответствующие файлы устройства. Реально файл устройства существенно отличается от обычного файла. Он не имеет блоков данных, хранимых на диске, однако обладает постоянным индексным дескриптором в той файловой системе, в которой он расположен. В отличие от обычного файла индексный дескриптор содержит не список номеров блоков данных на диске, а старший и младший номер устройства. Файл устройства может быть создан только суперпользователем с помощью привилегированного системного вызова mknod(path, mode, dev), параметры соответственно – полное имя специального файла, тип IFBLK или IFCHR и привилегии, старший и младший номера устройства.

В **Windows** поддерживается множество типов драйверов устройств и сред их программирования. Эти среды могут различаться даже для устройств одного типа. Драйверы могут работать в 2 режимах: пользовательском и ядра. Win поддерживает несколько типов драйверов пользовательского режима:

- драйверы виртуальных устройств (VDD). Используются для эмуляции 16 разрядных программ MS DOS. Они перехватывают обращения таких программ к портам вв и транслируют их в вызов соответствующих API, передаваемых реальным драйверам устройств.
- Драйверы принтеров. Транслируют аппаратно-независимые запросы на графические операции в команды, специфичные для принтера. Далее эти команды направляются драйверу режима ядра, например, драйверу параллельного порта или принтера на USB шине.

И режима ядра:

- драйверы файловой системы. Принимают и выполняют запросы на вв, выдавая специфичные запросы драйверам устройств массовой памяти или сетевым драйверам
- PnP драйверы. Драйверы, работающие с оборудованием и интегрируемые с диспетчерами электропитания и PnP. В т.ч. драйверы для устройств массовой памяти, видеоадаптеров, устройств ввода и сетевых адаптеров
- Драйверы, не поддерживающие PnP. Называются расширениями ядра. Расширяют функциональность, предоставляя доступ из пользовательского режима к сервисам и драйверам режима ядра. Не интегрируются с диспетчерами PnP.

Драйверы устройств, отвечающие спецификации Windows Driver Model (WDM), поддерживают управление электропитанием, PnP, WMI. Большинство драйверов PnP реализованы в этой модели. Три типа драйверов:

- драйверы шин. Управляют логическими или физическими шинами. Отвечают за распознавание устройств, оповещение о них диспетчера PnP и управление параметрами электропитания шины.
- Функциональные драйверы. Управляют конкретным типом устройств. Устройства предоставляются функциональным драйверам драйверами шин через диспетчер PnP.
- Драйверы фильтров. Это драйверы более высокого логического уровня, дополняющие функциональность или изменяющие поведение устройства либо другого драйвера.

В системе присутствуют два типа системных объектов – Драйвер и Устройство. Объект Драйвер представляет в системе отдельный драйвер. Объект Устройство представляет физическое или логическое устройство и описывает его характеристики. Диспетчер вв создает объект Драйвер при загрузке в систему соответствующего драйвера и вызывает его инициирующую процедуру, которая записывает в атрибуты объекта точки входа этого драйвера. Драйвер может сам создавать объекты Устройство. Однако для большинства PnP драйверов используют иной механизм. Объект Устройство создается внутренней процедурой добавления устройств, когда диспетчер PnP информирует их о присутствии управляемого ими устройства. Драйверы, не поддерживающие PnP могут создавать объект Устройство при вызове диспетчером вв их инициирующих процедур. Диспетчер вв выгружает драйвер после удаления последнего его объекта Устройство, когда ссылок на устройство больше нет. При создании объекта Устройство ему может быть присвоено имя. Этот объект помещается в пространство имен диспетчера объектов. Как правило объекты Устройство помещаются в каталог \Device пространства имен, недоступный приложениям через WinAPI.

Драйвер может загружаться явно и на основе перечисления. Явную загрузку определяет ветвь реестра HKLM\SYSTEM\CurrentControlSet\Services. На основе перечисления загрузка происходит при динамической загрузке диспетчером PnP. Параметр Start при явной загрузке устанавливается следующим образом:

- если драйвер не поддерживает PnP, Start настраивается таким образом, чтобы система загружала их на определенном этапе
- если драйвер должен загружаться системным загрузчиком при запуске ОС, Start устанавливается в 0 (запуск при загрузке системы). Например, драйвера системных шин и файловой системы
- если драйвер не требуется для загрузки системы, и распознает устройство, не перечисляемое драйвером системной шины, Start устанавливается в 1 (запуск системой). Пример – драйвер последовательного порта
- если драйвер не поддерживает PnP, или это драйвер файловой системы не обязательный для загрузки системы, Start устанавливается в 2 (автозапуск).
- если драйвер PnP, не обязательный для загрузки системы, Start устанавливается в 3 (запуск по требованию), например, драйверы сетевых адаптеров.

Загрузка по перечислению. Диспетчер PnP начинает перечисление с виртуального драйвера шины с именем Root, который представляет всю систему. Он выступает в роли драйвера шины для драйверов, не поддерживающих PnP и для HAL. Hard-Ware Abstraction Layer – загружаемый модуль ядра Hal.dll, представляющий низкоуровневый интерфейс для аппаратной платформы, на которой работает Windows. Он скрывает от ОС все, что связано с аппаратной реализацией функциональности, специфичное для архитектуры и конкретного оборудования, в т.ч. интерфейсы вв, контроллеры прерываний и механиз-

мы межпроцессорного взаимодействия. В свою очередь HAL работает как драйвер шины, перечисляющий устройства, напрямую подключенные к материнской плате и такие системные компоненты, как аккумуляторы. Он определяет основную шину, обычно PCI, устройства типа аккумуляторов и вентиляторов. Обычно HAL полагается на описание оборудования в реестре, зафиксированное программой Setup при установке ОС. Драйвер основной шины перечисляет устройства на этой шине, он может обнаружить другие шины, и процесс продолжается рекурсивно. По мере получения сообщений от драйверов шин об обнаруженных устройствах, диспетчер PnP формирует внутреннее **дерево устройств**, отражающее взаимосвязи между устройствами. Узлы этого дерева называются узлами устройств. Узел устройства содержит информацию об объектах Устройство и другую PnP информацию.

В результате процесс загрузки драйверов выглядит так:
- диспетчер вв вызывает входную процедуру для драйверов, запускаемых при загрузке системы. Если у такого драйвера есть дочерние устройства о них сообщается диспетчеру PnP. Если соответствующие драйверы также должны запускаться при загрузке системы, они конфигурируются и запускаются
- далее диспетчер PnP проходит по дереву устройств и загружает драйверы для узлов, и запускает их устройства независимо от параметра Start. Аналогично перечисляются все дочерние устройства и загружаются соответствующие им драйверы.
- далее диспетчер PnP загружает любые незагруженные драйверы, запускаемые системой. Эти драйверы определяют свои устройства и сообщают о них.
- Диспетчер управления сервисами загружает автоматически запускаемые драйверы.

Все устройства, обнаруженные системой после установки, регистрируются в подразделах раздела реестра HKLM\SYSTEM\CurrentControlSet\Enum. Драйвер класса дисков создает объекты Устройство, представляющие диски и дисковые разделы. Имена таких объектов имеют вид \Device\HarddiskX\DRX, где X – номер диска. Для идентификации разделов и создания объектов Устройство, представляющих эти разделы, используется функция IoReadPartitionTable либо IoReadPartitionTableEx.

При поступлении значительного количества запросов на чтение запись за малый промежуток времени, соседние из них могут требовать данные из областей диска, находящихся далеко друг от друга. Поэтому в современных системах используется оптимизация операции ввода-вывода. Как правило, драйверы содержат таблицу, индексированную по номерам цилиндров, в которой в единый список собираются все поступившие и ждущие обработки запросы. Основные дисциплины обслуживания:
- FCFS (First Come First Served) – первым пришел, первым обслужен.
- SSF (Shortest Seek First) – первым выполняется запрос с наименьшим временем позиционирования. Однако при этом некоторые запросы могут не обслуживаться в течении довольно долгого времени, даже если они от высокоприоритетных задач. В принципе, нет гарантии обслуживания самых крайних дорожек. Однако эта дисциплина обеспечивает максимальную пропускную способность.
- Scan – сканирование (элеваторный). Головки перемещаются от одного края к другому, по пути обслуживая подходящие запросы. При такой дисциплине гарантируется обслуживание запросов к крайним цилиндрам.
- Look. Отличие в том, что если при движении в одном направлении попутных запросов больше нет, направление изменяется на противоположное.
- NSS (Next Step Scan) – отложенное сканирование. Отличается тем, что при сканировании обслуживаются только те запросы, которые уже существовали на момент начала прохода. Поступающие записи формируют новую очередь, которая будет обслужена при обратном движении.
- C-Scan – циклическое сканирование. Головки перемещаются от наружной дорожки к внутренним, по пути обслуживаются имеющиеся запросы, после чего цикл повторяется, т.е. обратный проход является холостым. Характеризуется очень малой дисперсией времени ожидания обслуживания. Не дискриминируются крайние цилиндры.

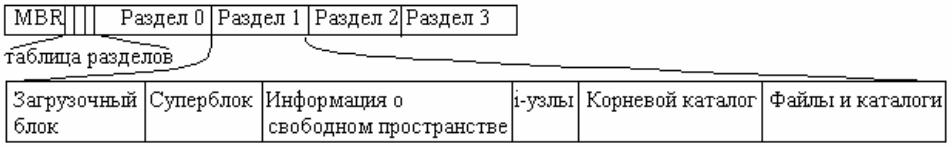
Пусть на диске 100 цилиндров. Начальное положение 63 цилиндр. Запросы на чтение цилиндров: 23, 67, 55, 14, 31, 7, 84, 10.

FCFS:	63 => 23 => 67 => 55 => 14 => 31 => 7 => 84 => 10;	40+44+12+41+17+24+77+74=329
SSF:	63 => 67 => 55 => 31 => 23 => 14 => 10 => 7 => 84;	4+12+24+8+9+4+3+77=141
Scan:	63 => 55 => 31 => 23 => 14 => 10 => 7 => 0 => 67 => 84;	8+24+8+9+4+3+7+67+17=147
Look:	63 => 55 => 31 => 23 => 14 => 10 => 7 => 67 => 84;	8+24+8+9+4+3+60+17=133
C-Scan:	63 => 55 => 31 => 23 => 14 => 10 => 7 => 0 => 99 => 84 => 67;	8+24+8+9+4+3+7+99+15+17=194

Загрузка ОС

Логическая структура жесткого диска. Главная загрузочная запись. Таблица разделов. Первичные и расширенные разделы. Таблица логических дисков. Системный загрузчик Windows. Варианты синтаксиса указания загрузочного раздела. Загрузка UNIX. Разбиение на разделы по схеме GPT. Динамические диски LDM. Жесткие и мягкие разделы. [1: 146-156; 2: 444-445, 776-778, 888-890; 7: 267-306, 659-674]

Логическая структура жесткого диска. Файловые системы позволяют хранить информацию на дисках. Большинство дисков делятся на разделы, что позволяет на одном физическом диске организовать несколько логических, на каждом из которых может находиться независимая файловая система. Для UNIX систем разделы не связаны с понятием логического диска, там ФС организована иначе. В любом случае системе требуется информация об организации разделов на диске. Сектор 0 диска **называется главной загрузочной записью** MBR master boot record и используется для загрузки компьютера. В нем располагается небольшая программа для анализа структуры диска и загрузки соответствующей ОС. Объем MBR – 512 байт. В конце MBR по смещению 0x1be содержится таблица разделов из 4 элементов объемом 16 байт на элемент. Структура элемента следующая:

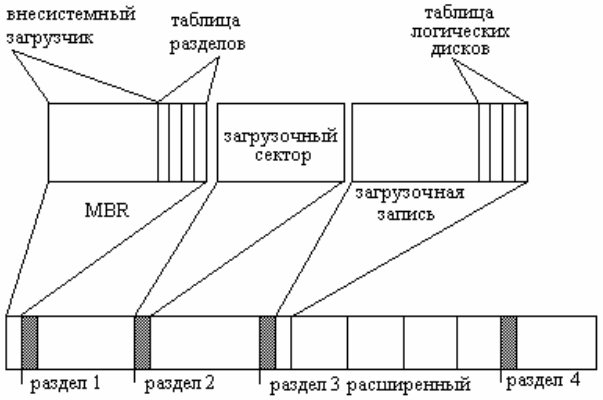


- Флаг активности раздела – 1 байт (0 – не активен, 80h-активен)
- Номер головки начала раздела – 1 байт
- Номер сектора и цилиндра загрузочного сектора раздела - 2 байт
- Код-идентификатор операционной системы – 1 байт
- Номер головки конца раздела – 1 байт
- Номер сектора и цилиндра последнего сектора раздела - 2 байта
- Младшее и старее 2байтовые слова относительного номера начального сектора – 4 байта
- Младшее и старее 2байтовые слова размера раздела в секторах – 4 байта

В таблице хранятся начальные и конечные адреса (номера блоков) каждого раздела. Один из 4 разделов является активным. При загрузке BIOS считывает и исполняет MBR-запись, код этой области определяет активный раздел, считывает его первый блок, называемый загрузочным, и исполняет его. Программа в загрузочном блоке (загрузочный сектор) загружает ОС данного раздела. Каждый дисковый раздел начинается с загрузочного сектора, даже если в нем не содержится ОС. Остальная структура раздела меняется в разных системах. Там может содержаться: суперблок, хранящий ключевые параметры ФС и считываемый при загрузке либо при первом обращении к ФС. Далее информация о свободных блоках, информация об i-узлах, являющихся массивами структур данных по одной структуре на каждый файл, далее корневой каталог и наконец прочие файлы и каталоги. Наиболее употребительные коды ОС:

00h	Раздел не используется	17h	Hidden NTFS, HPFS	82h	LinuxSwap, Solaris	A5h	Free BSD
01h	FAT12	1bh	Hidden FAT32	83h	LinuxNative	A6h	Open BSD
02h	Xenix root	1ch	Hidden FAT32 LBA	84h	OS/2 C: Hidden	A9h	Net BSD
03h	Xenix /usr	1eh	Hidden FAT16 LBA	85h	Linux Extended	abh	Apple booter
04h	FAT16 (<32Mb)	35h	OS/2 JFS	86h	FAT16 Volume Set	bbh	OS Selector
05h	Extended	3ch	Partition Magic	87h	NTFS Volume Set	beh	Solaris 8 boot
06h	FAT16	42h	LinuxSwp	8bh	FAT32 Volume Set	C2h	Hidden Linux Swap
07h	NTFS, HPFS	43h	LinuxNat	8ch	FAT32 LBA Volume Set	C3h	Hidden Linux Native
0ah	OS/2 Boot Manager	4dh	QNX 4.x first	8dh	Free Fdisk FAT12	D1h	Multiuser DOS FAT12
0bh	FAT32	4eh	QNX 4.x second	90h	Free Fdisk FAT16 (<32Mb)	D4h	Multiuser DOS FAT16(<32Mb)
0ch	FAT32 LBA	4fh	QNX 4.x third, Oberon	91h	Free Fdisk Extended	D5h	Multiuser DOS Extended
0eh	FAT16 LBA	52h	CP/M	92h	Free Fdisk FAT16	D6h	Multiuser DOS FAT16
0fh	Extended LBA	63h	UNIX	97h	Free Fdisk FAT32	D8h	CP/M – 86
11h	Hidden FAT12	64h	NetWare 2.x	98h	Free Fdisk FAT32 LBA	ebh	BeOS
14h	Hidden FAT16 (<32Mb)	65h	NetWare 3.x	9ah	Free Fdisk FAT16 LBA	F2h	DOS Secondary
16h	Hidden FAT16	70h	DiskSecure Multi-boot	9bh	Free Fdisk Extended LBA	fdh	Linux RAID

Последние 2 байта MBR имеют значение 55aah, т.е. чередующуюся последовательность 0 и 1, что позволяет проверить работоспособность линий передачи данных. Это же значение имеется в 2 последних байтах во всех загрузочных секторах. Разделы делятся на **первичные** (primary) и **расширенные** (extended). Первичных разделов не может более 4. Только один из них может быть активным. Если система поддерживает спецификации DOS, остальные первичные разделы считаются невидимыми (hidden). Расширенный раздел дополнительно также разбивается на подразделы – **логические диски,** хотя в общем случае любой логический диск может в свою очередь являться расширенным разделом. Расширенный раздел содержит вторичную загрузочную запись, которая содержит таблицу логических дисков (LDT), по



структуре аналогичных MBR.

Загрузка ОС происходит следующим образом. При включении компьютера управление получает BIOS, располагающаяся в ПЗУ по верхним адресам. Соответствующие процедуры проводят тестирование и вызывают процедуру начальной загрузки. Она определяет первое готовое устройство из списка разрешенных и доступных в соответствии с информацией BIOS. Таким устройством может быть гибкий или жесткий диск, компакт-диск, ZIP-привод, сетевой адаптер и т.д. С этого устройства в оперативную память загружается короткая главная программа-загрузчик. Для жесткого диска это внесистемный загрузчик из MBR. Далее этому загрузчику передается управление. Загрузчик определяет активный раздел на диске, загружает системный загрузчик (первый сектор) активного раздела и передает ему управление. Системный загрузчик находит и загружает необходимые файлы ОС и передает ей управление. Далее система загружает необходимые сервисы, расширяет или заменяет некоторые обработчики BIOS. В современных мультизадачных ОС большинство сервисов BIOS, изначально расположенных в ПЗУ, заменяются собственными драйверами ОС.

Существуют менеджеры загрузки, позволяющие пользователю выбрать для загрузки одну из установленных на ПК ОС. Один из наиболее известных мультизагрузчиков – BootMagic, фирмы PowerQuest. Как правило такие программы:
- поддерживают большое число ОС;
- устанавливаются на любой раздел, в т.ч. и недоступный другим ОС;
- позволяют загрузиться с дискеты независимо от установок BIOS;
- автоматически идентифицируют ОС как на первичных разделах так и на логических дисках расширенного раздела всех жестких дисков, доступных в BIOS;
- поддерживают несколько ОС на одном разделе FAT16/32, предотвращая конфликты по системным и конфигурационным файлам;
- дополнительная настройка конфигураций ОС;
- восстановление при повреждении MBR;
- поддержка больших жестких дисков во всех режимах;
- парольная защита на меню загрузки и выбранные конфигурации.

Формирование таблицы разделов осуществляется специальными утилитами, простейшие из них – fdisk от Microsoft. Один из наиболее известных разбивщиков – PartitionMagic фирмы PowerQuest. Такие программы как правило позволяют:
- создавать разделы любых типов и форматировать их под разные ФС;
- получать подробную информацию о разделах и дисках;
- удалять разделы;
- преобразовывать разделы одной ФС в другую;
- копировать и перемещать данные с одного раздела на другой;
- изменять размеры раздела;
- выбирать размер кластера для операции;
- редактировать содержимое жесткого диска посекторно.

Системные утилиты форматирования диска в MS-DOS и Win 9x создают один первичный раздел и один расширенный, в котором организуют один или несколько логических дисков. Помимо буквенного обозначения C:\ и т.д. диски получают еще и так называемый номер раздела. C имеет номер 1, D – 2 и т.д. В Win NT/2000/XP системный загрузчик считывает корневой каталог своего диска, находит в нем файл ntldr, загружает его и передает ему управление. Он уже загружает непосредственно ОС. Существует несколько версий загрузочного сектора в зависимости от формата раздела. Ntldr позволяет выбирать из установленных ОС. В файле boot.ini указывается, где находятся файлы выбранной ОС, используя номера дисков и каталог. Этот же файл отвечает и за ряд других параметров ОС (кол-во ЦП, объем ОЗУ, объем виртуальной памяти пользовательского процесса 2 или 3 Гб, частота часов реального времени). Это единственный файл с системной информацией, не содержащейся в реестре. Пример boot.ini:

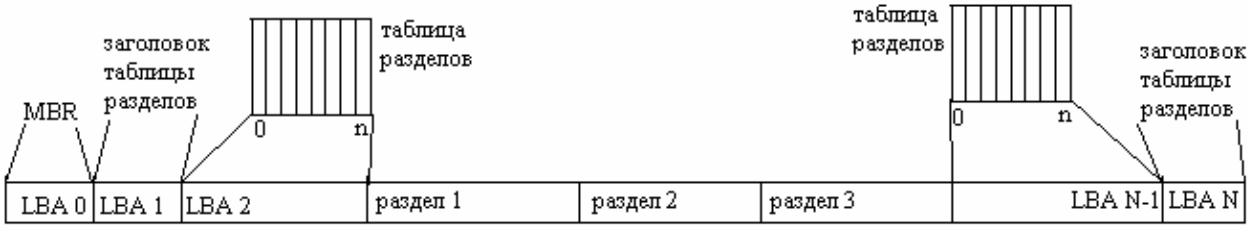
```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operation systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"  /fastdetect
C:\="Microsoft Windows"
```

1 секция описывает ОС, загружаемую по умолчанию, 2 – перечень доступных ОС. Возможны 3 варианта синтаксиса. 1 вариант (multi) указывает, что требуется загружать системные файлы через функцию прерывания Int 13h. Т.о. используется, когда у диска на котором находится загрузочный раздел, есть контроллер с поддержкой прерывания Int 13h. Синтаксис имеет формат : multi(W)disk(X)rdisk(Y)partition(Z). Здесь W- номер дискового контроллера, обычно 0, X – в этом синтаксисе всегда 0, Y – физический жесткий диск, подключенный к контроллеру (0-3 для ATA контроллера, 1-15 для SCSI), Z – номер загрузочного раздела на физическом диске, начиная с 1. Второй вариант (SCSI) сообщает, что для доступа к загрузочным файлам надо задействовать сервисы дискового в/в, предоставляемые Ntbootdd.sys. Синтаксис: scsi(W)disk(X)rdisk(Y)partition(Z). W – номер контроллера, X – физический жесткий диск, подключенный к этому контроллеру (обычно 0-15), Y – logical unit number (LUN) диска, содержащего загрузочный раздел, обычно 0, Z – номер загрузочного раздела, начиная с 1. Третий (signature) указывает, что необходимо найти диск с соответствующей сигнатурой, независимо от номера контроллера и использовать ntbootdd.sys для доступа к загрузочному разделу. Сигнатура – это глобально уникальный идентификатор GUID, извлекаемый из MBR в процессе установки и записываемый на диск. Синтаксис: signature(V)disk(X)rdisk(Y)partition(Z). V –32 разрядная сигнатура диска, X- физический жесткий диск со специфической сигнатурой, Y всегда 0, Z- номер загрузочного раздела. Этот вариант используется, если размер загрузочного раздела больше 7,8 Гб, а функции BIOS Int13h не могут обратиться ко всему разделу, либо если BIOS не поддерживает расширенное Int 13h.

Ntldr начинает работу в реальном режиме x86, т.е. трансляция виртуальных адресов отсутствует, и адреса интерпретируются как физические. Доступен лишь первый Мб физической памяти. Ntldr переключает систему в защищенный режим (пока без трансляции виртуальных адресов, но доступна уже вся память), создает таблицу страниц, включает поддержку подкачки, далее работая в полнофункциональном режиме. Далее анализирует boot.ini, и если запись выбранной ОС ссылается на ранее установленную ОС (MS DOS, Windows Me, Windows 9x), читается в память bootsect.dos, переключается обратно в реальный режим и вызывает из bootsect.dos код MBR, специфичный для данной ОС. Если же продолжается загрузка XP/Server 2003/2000, то запускает ntdetect.com для получения базовых сведений о конфигурации из BIOS, загружает hal.dll, ntoskrnl.exe, и видеодрайвер по умолчанию bootvid.dll. Далее считывается реестр, чтобы найти остальные необходимые драйверы, вносит в список драйверов драйвер ФС, отвечающий за реализацию кода для того типа раздела, на котором установлена ОС, загружает его, далее загружает остальные драйверы и управление передается ntoskrnl.exe. Выполняется ряд инициализаций, последним этапом загрузки создается первый пользовательский процесс, сеансовый менеджер smss.exe. Выполнив свою часть работы, он создает демон регистрации winlogon.exe. В этот момент ОС загружена и работает. Далее создается родительский процесс всех служебных процессов, services.exe, который создает все системные службы, запуская служебные процессы в пространстве пользователя, и загружает оставшиеся незагруженными драйверы устройств. Winlogon получает из реестра профиль пользователя и запускает оболочку, обычно explorer.exe с рядом настроенных параметров. Отредактировав реестр, можно заставить загружать другую оболочку.

. В Linux диски нумеруются иначе. Жесткий диск, подключенный к первому контроллеру IDE как master, именуется hda. Второй диск на этом же кабеле получит имя hdb. Master на втором порте контроллера будет hdc И т.д. Если раздел описан в MBR, то он имеет номер элемента таблицы разделов. Диски, созданные в расширенном разделе нумеруются начиная с 5. Разные версии загружаются по-разному, однако в общем действия выполняются следующие. Загрузочный сектор загружает автономную программу boot и передает ей управление. Она копирует себя в фиксированный адрес памяти в старших адресах, далее считывает корневой каталог, потом ядро ОС и передает ему управление. Ядро написано на ассемблере и является в значительной степени машинно-зависимым. Ядро устанавливает указатель стека, определяет тип ЦП, вычисляет объем ОЗУ, разрешает работу диспетчера памяти и запускает процедуру main, для запуска основной части ОС. Выполняется существенная инициализация, в т.ч. выделяется память под буфер сообщений, структуры данных ядра, и т.д., считываются файлы конфигурации, где описаны типы устройств, проверяется их наличие, формируя таблицу подключенных устройств. Далее загружаются драйверы устройств. Затем запускается процесс 0 , который программирует таймер реального времени, монтирует корневую файловую систему и создает процесс 1 (init) и процесс 2 страничного демона. Init запускает ОС в зависимости от флагов, выполняет ряд дополнительных действий, считывает файл etc/ttys, перечисляющий терминалы и их свойства. Для разрешенных терминалов создает копию самого себя, которая исполняет программу getty, которая читает имя пользователя и вызывает bin/login, которая запрашивает пароль и выполняет проверку, после чего запускает вместо себя оболочку пользователя.

По новой спецификации EFI (Extensible Firmware Interface) от Intel определяется среда операционной мини-системы, реализуемой в виде микрокода, как правило, зашитого в ПЗУ. Эта среда используется ОС на ранних этапах загрузки и для диагностики. Первая платформа, поддерживающая EFI, это Intel IA64. Соответствующие версии Windows используют EFI, но могут работать и по схеме MBR. EFI определяет схему разбивки на разделы – таблицу разделов



GUID (GUID Partition Table - **GPT**). Адреса секторов стали 64 разрядными, используются контрольные суммы CRC для поддержания целостности таблицы разделов и резервное копирование таблицы разделов, каждому разделу, помимо имени, назначается свой GUID. Первый сектор по-прежнему содержит MBR, чтобы защитить диск от ОС, не поддерживающих GPT. Во 2 и последнем секторах хранятся заголовки таблицы разделов, а сразу после 2 сектора и перед последним располагается сама таблица. При такой структуре отпадает необходимость в расширенных разделах. Элемент таблицы описывает начало и конец соответствующего раздела. Оба способа – и MBR и GPT определяют разделы на **базовых** дисках.

Вместе с тем в Windows 2000 появилась поддержка **динамических** разделов (LDM – Logic Disk Manager). Основное отличие в том, что LDM поддерживает одну унифицированную базу данных, где хранится информация о разделах на всех динамических дисках системы. Чтобы загрузчик мог найти системный и загрузочный разделы и каталоги, они описываются в стиле MBR или GPT, оставшаяся часть диска (или весь диск, если он не содержит загрузочного кода), помечается как раздел типа LDM. Соответственно разделы в стиле MBR или GPT называются **жесткими**, а LDM разделы **мягкими**. БД LDM-раздела размещается в зарезервированном пространстве размером 1 Мб в конце этого раздела. Она состоит из 4 областей: сектора заголовка (private header), таблицы оглавления, собственно БД, состоящей из заголовка и самих записей, и журнала транзакций, после которого следует зеркальная копия Private Header. Каждому динамическому диску LDM назначает свой GUID. Таблица оглавления занимает 16 секторов и содержит информацию о структуре БД. Длина каждой записи БД – 128 байт. Элементы БД могут быть 4 типов: раздел partition, компонент component, том volume, диск disk. На нижнем уровне находятся элементы раздела, описывающие разделы в классическом понимании, идентификаторы, хранящиеся в элементе раздела, связывают его с элементами компонентов и дисков. Элемент компонента связывает один или несколько элементов разделов и элемент тома, с которым данные разделы сопоставлены. Элемент тома хранит его GUID, суммарный размер, состояние и присвоенную ему букву диска. Элемент диска представляет динамический диск в составе группы и включает его GUID. Та-

кая структура позволяет произвольный набор разделов объединять в один том (доступ в файловой системе по одной букве), даже если они находятся на разных дисках, а также поддерживать RAID массивы.

Понятие файловой системы

Функции файловой системы. Понятие файла. Структура файла. Типы файлов. Регулярные файлы и каталоги. Специальные файлы. Файлы с последовательным и произвольным доступом. Структуры управления файлами. Атрибуты файла. Отображение файлов на память. Непрерывная организация файлов на диске, связные списки, i-узлы. Каталоги. Одноуровневая, двухуровневая и иерархическая структуры каталогов. Имена файлов. Совместно используемые файлы. Символьное связывание. Учет свободных блоков. Монтирование. Непротиворечивость файловой системы. Журналирование и каскадный откат транзакций. Кэширование. Типовые операции файловой системы

К долговременным устройствам хранения информации предъявляют следующие требования:
- устройства должны позволять хранить очень большие объемы данных
- информация должна сохраняться после прекращения работы процесса, использующего ее
- несколько процессов должны иметь возможность получения одновременного доступа к информации

Обычное решение этих задач состоит в хранении информации на дисках и других носителях в модулях, называемых **файлами.** По мере необходимости процессы могут читать и создавать файлы. Информация в файлах должна обладать устойчивостью, т.е. на нее не должны оказывать влияние создание или прекращение работы какого либо процесса. Файл должен исчезать только тогда, когда его владелец даст команду удаления файла. Файлами управляет ОС. Их структура, именование, использование, защита, реализация и доступ к ним являются важными нюансами ОС. Часть ОС, работающая с файлами и определяющая эти свойства, называется системой управления файлами или файловой системой. Помимо работы с файлами, ФС может позволять работать с недисковыми устройствами, как с файлами. ОС может позволять работать с несколькими ФС. С другой стороны, ФС называется также определенный способ организации данных на диске. Система управления файлами не существует сама по себе, она разрабатывается для работы с конкретной ФС в конкретной ОС.

Файлы относятся к абстрактному механизму и предоставляют способ сохранять информацию на диске и считывать ее. От пользователя скрываются такие детали как способ и место хранения информации, а также детали работы дисков. При создании файла процесс дает файлу имя. В разных системах используются разные правила именования, но большинство поддерживает 8-символьные текстовые строки. Часто разрешается использование в имени цифр и специальных символов, многие системы поддерживают имена длиной до 255 символов. Некоторые ОС (UNIX) различают строчные и прописные символы, другие (MS-DOS) нет. Windows использует файловую систему MS-DOS и наследует многие ее свойства, однако NT имеет свою ФС NTFS. Во многих ОС используется расширение, как правило означающее тип файла. В MS-DOS расширение содержит до 3 символов, в UNIX его длина зависит от пользователя. У файла может быть несколько расширений. Расширение файла является всего лишь декларативным и не носит обязательный характер. Т.е. exe скорее всего является исполняемым файлом, но это не является обязательным.

Файлы могут быть структурированы различными способами. 1. Файл может восприниматься как неструктурированная последовательность байт. В этом случае ОС не интересуется содержимым файла. Она видит только байты. Значение этим байтам придается в программах пользователя. Этот подход обеспечивает максимальную гибкость. Программы могут помещать в файл все что угодно и именовать любым удобным способом, а ОС не смешивается в этот процесс. 2. Файл представлен последовательностью записей фиксированной длины, каждая со своей внутренней структурой. Для таких файлов операция чтения и записи возвращает или пишет одну запись полностью. 2. Файл может представлять собой дерево записей, не обязательно одинаковой длины. Каждая запись в фиксированной позиции содержит поле ключа. Дерево отсортировано по ключу, что обеспечивает быстрый поиск. Здесь главная операция не получение следующей записи, а получение записи с указанным значением ключа.

ОС поддерживают в общем случае различные типы файлов. Например, в UNIX и Windows различают регулярные файлы и каталоги. **Регулярные** – все файлы, которые содержат информацию пользователя. **Каталоги** – это системные файлы, обеспечивающие поддержку структуры файловой системы. В UNIX поддерживаются символьные **специальные** файлы и блочные специальные файлы. ССФ используются для моделирования последовательных устройств вв, например, терминалы, принтеры, сеть и т.д. БСФ используются для моделирования дисков. Регулярные файлы в основном являются либо ASCII-файлами либо двоичными. Код конца строки различен в разных системах. Это может быть ВК, ПС (UNIX), оба (MS-DOS). ASCII файлы могут отображаться на экране и выводиться на печать так как есть, без какого-либо преобразования.

Доступ к данным может быть последовательный, когда чтение или запись могут идти только последовательно, байт за байтом, или произвольный, когда чтение или запись могут выполняться в произвольном порядке или получать доступ к записям по ключу.

У каждого файла есть имя и данные. Дополнительная информация, связанная с файлом и учитываемая ОС, называется **атрибутами**. Наиболее распространенные: защита (кто и каким образом может получить доступ к файлу), пароль, создатель (идентификатор пользователя, создавшего файл), владелец, флаг только для чтения, флаг скрытый, флаг системный, флаг архивный, флаг ASCII/двоичный, флаг произвольного доступа, временный (удаляется по окончании процесса), флаг блокировки, длина записи, позиция ключа, длина ключа, время создания, время последнего доступа, время последнего изменения, текущий размер, максимальный размер.

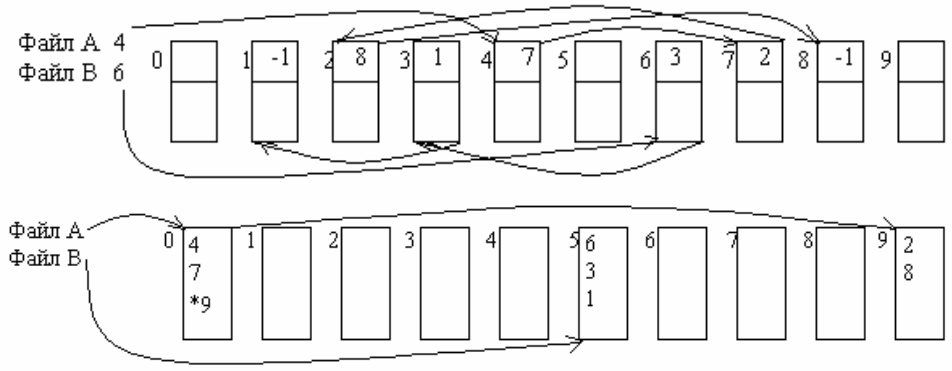
Стандартный способ доступа к файлу сильно отличен от доступа к памяти. Для того, чтобы использовать единообразный интерфейс, файлы в некоторых ОС могут отображаться на память. Это можно представить в виде двух системных вызовов – map и unmap. Первый имеет на входе два параметра – имя файла и виртуальный адрес памяти, по которому ОС отображает

этот файл. Когда процесс завершает работу, модифицированный файл из памяти выгружается на диск. В итоге все словно происходит так, как если бы использовали seek и write. При записи в файл соответствующая страница памяти помечается как модифицированная. Если эта страница памяти выгружается алгоритмом замены страниц, она записывается в соответствующую область файла. Отображение на память лучше всего работает при сегментной организации памяти. В этом случае каждый файл может отображаться на свой сегмент. Хотя отображение на память устраняет необходимость обращения к системным вызовам ввода-вывода, имеются следующие вопросы:
- ОС трудно определить длину выходного файла, т.е. ОС знает номер модифицированной страницы, но она не может определить, сколько байт было туда записано. Второй вопрос – при попытке открыть файл, уже открытый другим процессом. Каждый из процессов будет модифицировать свою копию файла в памяти, т.е. изменения, сделанные одним, не будут доступны другим, пока файл открыт. Требуется особый механизм. Третий – файл может оказаться больше сегмента памяти и даже больше, чем все доступное виртуальное адресное пространство.

Реализация файловой системы. Простейший вариант организации самого файла на диске – **непрерывный**, когда файл занимает соседние блоки диска. Такая система легко организуется, обладает высокой производительностью, однако система очень быстро подвергается фрагментации. Этот вариант организации файлов широко используется в CD_ROM. Второй вариант – **связные списки**. В начале каждого блока, принадлежащего файлу, находится указатель на следующий блок. В этом случае фрагментация не так опасна, существенно снижаются потери дискового пространства, в каталоге достаточно хранить только адрес первого блока. Однако произвольный доступ к такому файлу достаточно медленный, поскольку требуется пробежать всю цепочку блоков от начала файла, чтобы найти нужный блок. Третий вариант – организация таблицы связных списков.

Файл А	4
Файл В	6

0	
1	-1
2	8
3	1
4	7
5	
6	3
7	2
8	-1



В этом случае указатели на блоки хранятся не в самих блоках, а в отдельной таблице, загружаемой в память. Эта таблица называется FAT file allocation table. Основной недостаток в том, что вся таблица должна полностью находиться в памяти. При диске большого объема размер таблицы существенно вырастает, для его уменьшения можно использовать увеличенный размер блока, но в этом случае возрастают потери дискового пространства. Вариант **с i-узлами** состоит в связывании с каждым файлом специальной структуры данных, index node – индексный узел, содержащий атрибуты файла и несколько записей для хранения адресов блоков файла. Преимущество в том, что i-узел находится в памяти только тогда, когда соответствующий ему файл открыт. Если файл небольшой, вся информация о его размещении хранится в i-узле. В случае, если выделенного в i-узле числа записей для адресов блоков недостаточно, имеется запись, содержащая адрес блока косвенной адресации, содержащего продолжение таблицы с адресами блоков файла. Если этого недостаточно, используется запись, содержащая адрес блока двойной косвенной адресации. В этом блоке содержатся указатели на блоки, в которых содержится продолжение таблицы с адресами блоков. Если и этого недостаточно, используется запись, содержащая адрес блока тройной косвенной адресации. Такую схему использует UNIX, а также ФС HPFS, NTFS и др. Это позволяет поддерживать i-узлы фиксированного размера и вместе с тем работать с файлами размером от нескольких байт до нескольких Гб.

Файлы обычно организуются в **каталоги**, которые в свою очередь, являются файлами. Структуры каталогов. **Одноуровневая**. Имеется один каталог, в котором содержатся все файлы. Использовалась в ранних ВМ. Преимущества: простота и быстрота поиска файла. Недостатки: различные пользователи могут использовать для своих файлов одинаковые имена. **Двухуровневая**. Каждому пользователю выделяется каталог. При этом одинаковые имена файлов у разных пользователей уже не конфликтуют друг с другом. **Иерархическая**. При большом количестве файлов возникает необходимость логически группировать файлы. Т.о. необходимо дерево каталогов. Для указания файла требуется указывать путь, по которому он находится. Это либо абсолютное имя пути, начиная с корневого каталога и заканчивая тем, где расположен файл, либо относительное имя, указываемое относительно рабочего каталога. У каждого процесса свой рабочий каталог. Большинство ОС с иерархической структурой каталогов имеют специальные элементы в каждом каталоге “.” ”..”. Означающие соответственно текущий каталог и родительский каталог.

Реализация каталогов. Запись в каталоге содержит информацию, необходимую для нахождения блоков диска. В зависимости от системы это может быть дисковый адрес всего файла для непрерывных файлов, номер первого блока файла или номер i-узла. Различные атрибуты файла, поддерживаемые файловой системой, могут храниться непосредственно в записи каталога. В этом случае запись каталога имеет фиксированную длину. Системы, использующие i-узлы, могут хранить атрибуты в узлах, а не в записях каталога. В этом случае запись каталога существенно короче – имя файла и номер i-узла.

Современные ОС поддерживают длинные имена файлов. В простейшем случае длина имени файла ограничивается 255 символами, и используются уже рассмотренные схемы хранения. Однако далеко не все файлы имеют максимальную длину, и

большая часть отведенного пространства пустует. Альтернативный подход – когда поле записи, содержащее имя файла имеют переменную длину. В этом случае первое поле записи каталога хранит длину записи либо имени файла, далее следуют поля фиксированной длины с атрибутами файла, далее – имя файла, выровненное на границу слова. Недостаток такого подхода – при удалении файла остается свободная область переменной длины, в которую далеко не всегда можно поместить новую запись каталога, т.е. каталог фрагментируется. Еще один недостаток – запись может занимать несколько страниц памяти. При чтении такой записи нужной страницы может не оказаться в памяти. Еще один вариант реализации длинных имен – сделать записи каталога одинаковой длины, и хранить в нем не имя файла, а указатель на него, сами имена же хранить в куче. В этом случае фрагментация каталога отсутствует, но будет фрагментироваться память в куче. Все рассмотренные схемы предусматривают при поиске файла просмотр каталога линейно сверху вниз. Для очень больших каталогов такой поиск требует много времени. Часто для поиска используют хэш-таблицы для каждого каталога. В зависимости от имени файла ему присваивается некоторый номер из диапазона 0 - N-1, где N – длина хэш-таблицы. При добавлении файла, если элемент хэш-таблицы с данным номером свободен, туда помещается указатель на запись каталога для данного файла, если же элемент занят, то создается связный список, объединяющий все записи каталога с данным хэш-кодом. аналогично выполняется поиск: вычисляется хэш, и просматривается список, соответствующий ему.

Совместно используемые файлы. Часто удобно, чтобы один и тот же физический файл содержался в разных каталогах. В этом случае между каталогом и совместно используемым файлом устанавливается связь. Сама файловая система представляет теперь не дерево, а ориентированный ациклический граф. Такая схема создает новые проблемы. Если в каталоге содержать указатели на блоки файла, то при изменении файла из одного каталога, соответствующая ему запись другого каталога не будет содержать измененной цепочки блоков. Первый вариант решения – содержать указатели на блоки файла в особой структуре, а в каталоге содержать только ссылку на эту структуру. В этом случае даже при перемещении файла (но не i-узла) все связи останутся действующими. Второй вариант – при создании связи, во втором каталоге создается новый файл типа связь (link). Он просто содержит путь к файлу, с которым он связан. Это называется **символьным связыванием**. Дополнительное преимущество символьных связей – в том, что их можно использовать для ссылок на файлы, расположенные на удаленных машинах. Оба способа имеют проблемы. В первом случае, если файл существует в нескольких каталогах, то при удалении его из одного каталога, ОС не может найти все связи, поэтому может только удалить запись в одном каталоге, оставив и сам файл и записи других каталогов, ему соответствующих, уменьшая счетчик связей. При символьном связывании такой проблемы не возникает. Однако при этом увеличивается время поиска – поскольку сначала находится лишь файл связи, а затем, взяв из него правильный путь, ищется собственно файл. Кроме того, при перемещении файла ссылка становится нерелевантной, т.е. связь теряется.
При разработке файловой системы важным показателем является **размер блока**. Чем он больше, тем выше скорость чтения-записи файла, но тем ниже эффективность использования дискового пространства. **Учет свободных блоков** может производиться двумя разными способами – либо связный список свободных блоков, либо битовая карта, в которой один бит соответствует одному блоку диска.

Для качественной работы ФС требуется вести учет свободных блоков. Используются два подхода, как и в случае учета свободных блоков памяти – битовый вектор и связный список. Первый вариант эффективен, если битовая карта помещается в памяти целиком. Например, для диска 40 Гб и блоке в 512 б размер битового вектора составляет 10 Мб. Связный список не эффективен, если требуется получить адреса нескольких свободных блоков, поскольку это ведет к увеличению обращений к диску при поиске блоков. Используют и модифицированный вариант связного списка. В этом случае в первом свободном блоке хранятся адреса N-свободных блоков. N-1 из них действительно являются свободными, а последний адрес указывает на следующий блок с адресами свободных блоков.

Монтирование. Файловая система, хранящаяся на разделе диска, должна быть связана с существующей иерархией файловых систем, чтобы стать доступной процессам системы. Mount, umount. В общем случае, в Windows монтирование происходит автоматически, раздел подключается на корневом уровне, и ему присваивается определенная буква, являющаяся начальным путем. ФС UNIX организована иначе. На корневом уровне находится ряд каталогов – usr, bin, home, dev и т.д. Файловый системы монтируются в каталог mnt.

Непротиворечивость файловой системы. Этот вопрос относится к проблеме непротиворечивости ФС. ФС обычно читают блоки данных, изменяют их и записывают обратно. Если в системе произойдет сбой прежде, чем все измененные блоки будут записаны на диск, ФС может оказаться в противоречивом состоянии. Это особенно важно, если таким измененным и не сохраненным блоком оказывается блок i-узла, каталога или списка свободных блоков. Всоставе ОС имеется специальная утилита, проверяющая ФС на непротиворечивость. В UNIX это fsck, в Windows – scandisk. Fsck работает следующим образом. Для проверки непротиворечивости блоков создается две таблицы, каждая из которых содержит счетчик для каждого из блоков, изначально нулевые. Счетчики первой таблицы учитывают, сколько раз каждый блок присутствует в файлах. Во второй – сколько раз блок учитывается в списке свободных блоков. Считываются все i-узлы. При считывании каждого номера блока соответствующий счетчик увеличивается на 1. Далее анализируется список свободных блоков, при считывании номера блока увеличивается соответствующий счетчик во второй таблице. Если ФС непротиворечива, то каждый блок будет встречаться только один раз, либо в первой, либо во второй таблице. В случае. если для какого-то блока содержится 0 в обеих таблицах, это говорит о недостающем блоке. Ошибка корректируется добавлением этого блока в список свободных. Если какой-то блок появляется многократно в списке свободных блоков, проблема также легко решается. Хуже, если один и тот же блок окажется в разных файлах. При удалении любого из этих файлов, блок будет перемещен в список свободных, оставаясь в тоже время используемым в остальных файлах. В этом случае такой блок копируется нужное количество раз в свободные, и блок заменяется на них в этих файлах. Проверяется и структура каталогов. Используется таблица счетчиков, но не для блоков, а для файлов. Проверка идет рекурсивно начиная с корневого каталога. Символьная связь счетчик не изменяет. В результате сканирования образуется список, с информацией, в скольких каталогах содержится тот или иной файл. Эти числа сравниваются с действительным числом использований, хранящимся в i-узле. В непротиворечивой ФС счетчики совпадают.

Если счетчик связей в i-узле больше, чем рассчитано, то даже при удалении файла из всех связанных каталогов, счетчик не обнулится, в итоге в системе будет существовать потерянный файл, занимая дисковое место. Если же наоборот, счетчик связей i-узла меньше, чем рассчитано, то при удалении файла из части каталогов, счетчик обнулится и система освободит блоки файла, и в системе окажутся ссылки на несуществующий флаг. В обоих случаях для решения проблемы следует уравнивать значения счетчиков. Это не восстановит потерянную информацию, однако ФС будет непротиворечивой.

Еще один вариант повышения надежности – **журналирование** операций, и если произошел критический сбой, восстановление до непротиворечивого состояния. В ФС протоколируются не все изменения, а только изменения метаданных – i-узлов, записей в каталогах и т.д. Изменение пользовательских данных не протоколируется. Журнализация реализована в NTFS, Ext3FS, ReiserFS и др. Имеется ряд проблем, например, при необходимости выполнить откат, может оказаться, что откат может затрагивать данные, уже измененный другими файловыми операциями. Поэтому и они тоже должны быть отменены. Эта проблема называется каскадным откатом транзакций.

Увеличение производительности ФС. **Кэширование** Кэшем называется набор блоков, логически принадлежащий диску, но хранящийся в памяти для увеличения производительности. Существуют разные алгоритмы управления кэшем. Типовой вариант - перехват всех обращений к диску и проверка наличия соответствующих блоков в кэше. Если блок в кэше, он считывается оттуда, если нет, то он сначала с диска копируется в кэш, а уже затем в нужную область памяти. Последующие обращения к этому блоку будут уже выполняться не с диском, а с кэшем. Для быстрого определения, имеется ли нужный блок в кэше или нет, обычно используется хэширование номера устройства и номера блока с поиском соответствующего кода в хэш-таблице. Все блоки с одинаковыми хэш-кодами организуются в связный список. Когда требуется загрузить блок в полностью загруженный кэш, должно выполниться замещение одного из ранее загруженных блоков. Это аналогично ситуации с замещением страниц памяти и допустимы все соответствующие алгоритмы (FIFO, вторая попытка, LRU). Все блоки можно разделить на категории, например, блоки i-узлов, косвенные блоки, блоки каталогов, блоки, полностью заполненные данными и блоки, частично заполненные данными. Блоки, которые в ближайшее время вероятнее всего не потребуются, помещаются в начало списка LRU, чтобы их буферы освобождались первыми. Блоки, вероятность повторного использования которых высока, помещаются в конец списка. Кроме этого, если блок важен для сохранения непротиворечивости ФС, например, блок i-узла, то при модификации такой блок сохраняется на диске немедленно, независимо от положения в списке. Но и в этом случае хранение в кэше блоков с данными слишком долго нежелательно. В UNIX имеется системный вызов sync, принудительно сохраняющий все модифицированные блоки кэша на диск. При загрузке ОС запускается фоновая задача, обычно называющаяся update, которая периодически вызывает sync. В MS-DOS модифицированный блок записывается сразу же, это называется кэшем со сквозной записью.

Типовые системные вызовы для работы с файлами:
Create). Delete. Open – позволяет системе прочитать в память атрибуты файла и список дисковых адресов для быстрого доступа к файлу при последующих вызовах. Close. Освобождается внутренняя структура с информацией об открытом файле. Read. Write. Append. ..Seek. Get_attributes. Set_attributes. Rename.

Windows. CreateFile, CloseHandle. ReadFile, WriteFile. ReadFileGather и WriteFileGather позволяют выполнять операции чтения-записи с использованием набора буферов различного размера. DeleteFile. CopyFile. CreateHardLink – Для NT5 (2K, WS2003, XP) создание жесткой ссылки! не ярлыка!, аналогично ссылкам в UNIX. SetFilePointer. Помимо использования явного позиционирования, для обновления записи в файле, чтобы не гонять туда-сюда указатель, можно использовать параметр hEvent структуры типа OVERLAPPED, использующейся в ReadFile, WriteFile. GetFileSize. SetEndOfFile – переустанавливает размеры файла. FindFirstFile, FindNextFile, FindClose. GetFileTime.GetFileAttributes. GetTempFileName – предоставляет уникальное имя для временного файла в заданном каталоге. LockFile (LockFileEx) – блокирование файла или части файла монопольно либо разделяемо. UnlockFileEx.

Существующая блокировка	Запрашивается Разделяемая Блок.	Запрашивается монопольная блок	Чтение	Запись
Отсутствует	Предоставляется	Предоставляется	Выполняется	Выполняется
Разделяемая (одна или несколько)	Предоставляется	Отказ	Выполняется.	Не выполняется.
Монопольная	Отказ	Отказ	Выполняется, если вызывающий процесс является владельцем блокировки	Выполняется, если вызывающий процесс является владельцем блокировки

Основные операции с каталогами. Create. Delete. OpenDir. Открыть каталог для чтения. CloseDir.ReadDir – чтение очередного элемента каталога. Rename. Link. Связывание позволяет одному и тому же файлу отображаться сразу в нескольких каталогах. Файл связывается с каталогом, в котором должна появиться ссылка на него. Unlink. Windows: CreateDirectory, RemoveDirectory.SetCurrentDirectory, GetCurrentDirectory. SetVolumeMountPoint – монтирование одной ФС в точке монтирования, находящейся в другой ФС. DeleteMountPoint.

Реализации файловых систем

Файловая система FAT. Особенности и функциональные возможности файловой системы FAT. Таблица размещения файлов. Структура загрузочной записи FAT16, FAT32. Хранение длинных имен. Файловая система HPFS. Структура раздела HPFS, сбалансированные двоичные деревья. Файловая система NTFS. Структура тома. Особенности системы NTFS. Файловая система CD (ISO9660). Основной описатель тома. Рок-Ридж расширения. Расширения Joliet. Базовая файловая система UNIX System V s5fs. Индексные узлы. Суперблок. Файловая система Berkeley FFS. Файловая система Linux Ext2fs. Файловая система proc. Файловая система с журнальной структурой LFS

ФС **FAT** стала использоваться с появлением MS-DOS, и представляет собой улучшенную версию ФС CP/M. Работает только на платформах с процессором x86, не поддерживала многозадачности и использовала только реальный режим. В самой первой версии MS-DOS 1.0 FAT содержала только корневой каталог, как и CP/M. В MS-DOS 2.0 ФС приобрела иерархическую структуру, связи не допускались. В ФС FAT дисковое пространство любого логического диска делится на две области: системную и область данных. Системная создается и инициализируется при форматировании и обновляется при манипулировании файловой структурой. Системная область включает: загрузочную запись (boot record), зарезервированные сектора, таблицы FAT, корневой каталог, содержащий не более 512 записей. Загрузочная запись для FAT16 занимает 1 сектор, для FAT32 – 3 сектора.

Структура загрузочной записи

Смещ, б	Длина, б	Содержимое поля FAT16 (32)
00h	3	jump 3eh, 2 байта безусловный переход на начало загрузчика, 3 байт – операция NOP
03h	8	системный идентификатор, содержит инф. о фирме разработчике и версии ОС
0bh	2	размер сектора, байт, отсюда начинается блок параметров диска
0dh	1	число секторов в кластере
0eh	2	число зарезервированных секторов (для FAT32 содержит 32)
10h	1	число копий FAT
11h	2	максимальное число элементов корневого каталога (FAT32 - 0)
13h	2	число секторов на логическом диске, если его длина не выше 32 Mb, иначе 0 (FAT32 - 0)
15h	1	дескриптор носителя
16h	2	размер FAT, секторов (FAT32 - 0)
18h	2	число секторов на дорожке
1ah	2	число рабочих поверхностей
1ch	4	число скрытых секторов, располагающихся перед загрузочным сектором. Значение используется для вычисления абсолютного смещения корневого каталога и данных
20h	4	число секторов на логическом диске (для FAT16 если размер больше 32 Mb)

FAT16			FAT32		
24h	1	тип логического диска 00- гибкий, 80h-жесткий	24h	4	число секторов в таблице FAT
25h	1	резерв	28h	2	расширенные флаги
26h	1	маркер с кодом 29H	2ah	2	версия файловой системы
27h	4	серийный номер тома	2ch	4	№ кластера для 1 кластера корневого каталога
2bh	11	метка тома	34h	2	№ сектора с рез. копией загрузочного диска
36h	8	имя файловой системы	36h	12	резерв
3eh		загрузчик			загрузчик
1feh	2	сигнатура (код aa55h)			

Область данных содержит файлы и каталоги, подчиненные корневому. В отличие от системной области, она доступна через пользовательский интерфейс ОС. Область данных разбивается на кластеры, представляющие собой один или несколько смежных секторов (блоков 512 байт) логического диска. Файл независимо от истинного размера занимает на диске целое число кластеров. Максимальный размер раздела:

Размер кластера, Кб	FAT12, Мб	FAT16, Mb	FAT32, Tb
0,5	2		
1	4		
2	8	128	
4	16	256	1
8		512	2
16		1024	2
32		2048	2

Структура записи каталога:

№ байт	Назначение в FAT16 (FAT32)
0-10	Имя файла 8+3
11	атрибуты файла: архивный, атрибут каталога, атрибут тома, системный, скрытый, только для чтения
12	резерв (совместимость с NT, обеспечения отображения имен в правильном регистре)
13	резерв
14-15	резерв (время создания файла)
16-17	резерв (дата создания файла)
18-19	резерв (дата последнего доступа)
20-21	резерв (старшее слово номера начального кластера в таблице FAT)

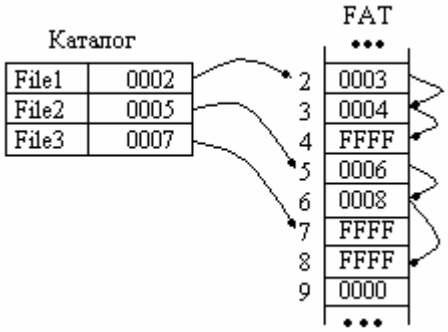
22-23 время последней модификации
24-25 дата последней модификации
26-27 номер начального кластера в таблице FAT (младшее слово номера начального кластера)
28-31 размер файла в байтах

Поле времени разбивается на 5 бит секунд, 6 минут, 5 часов, даты на 5 бит дня, 4 месяца и 7 бит года (-1980). Атрибут архивный определяет, что файл был открыт программой так, что она имеет возможность изменять его содержимое. Программы резервного копирования могут устанавливать этот бит в 0, соответственно резервные копии создаются только для файлов с установленным битом. Атрибут тома используется только для одного элемента корневого каталога, в котором хранится имя дискового тома, он же используется для длинных имен. В FAT32 поддерживаются длинные имена. Если имя файла не отвечает этому формату 8.3 (длиннее или используются недопустимые символы), то берутся первые 6 символов, при необходимости преобразуются в верхний регистр ASCII, добавляется суффикс ~1. Если такое имя уже есть, то ~2, и т.д. Дополнительно удаляются пробелы и лишние точки, ряд недопустимых символов преобразуется в подчеркивание. Имя формата 8.3 хранится непосредственно в записи каталога. Длинное имя хранится в одной или нескольких каталоговых записях, предшествующих записи с форматом имени 8.3. Каждая такая запись содержит до 13 символов формата unicode, элементы имени хранятся в обратном порядке:

№ эл-та 1 байт	5 символов имени (10 байт)	Атрибуты, 1 байт	Резерв, 1 байт	Контрольная сумма, 1 байт	6 символов имени, (12 байт)	0, два байта	2 символа имени, (4 байта)
67	26-30				31-36		37-38
2	13-17				18-23		24-25
1	символы 0-4				5-10		11-12
Стандартная запись каталога в формате имени 8.3							

Поле атрибутов для фрагментов длинного имени содержит код 0fh, который невозможен в качестве комбинации атрибутов файла для стандартной записи каталога. Старые программы записи с таким кодом интерпретируют как ошибочные и игнорируют. Последняя запись, содержащая длинное имя, имеет номер+64. Кроме того, длина имени+путь ограничена 260 символами вместо возможных 800. Поле контрольной суммы призвано отследить следующую ситуацию. ПО MS-DOS удаляет файл с длинным именем, соответствующая запись каталога будет свободной, записи содержащие длинное имя останутся. Далее на это место будет помещена запись о новом файле. При этом записи с длинным именем остались, т.к. MS-DOS их игнорирует. Соответственно Win98 может отследить подобную ситуацию с вероятностью обнаружения 255/256. Win98 не хранит в памяти всю FAT, а использует окно, накладываемое на таблицу.

Таблица **FAT** связывает кластеры, принадлежащие одному и тому же файлу. Первый кластер файла указан в каталоге, остальные – в таблице FAT. Фактически таблица состоит из 2 байтовых (4 байтовых для FAT32) элементов, № элемента соответствует кластеру с данным номером, элемент содержит номер следующего кластера в цепочке кластеров, принадлежащих файлу.



Номера кластеров 0 и 1 используются для системных целей, а для данных доступны номера кластеров, начиная со 2. В системной области имеется вторая копия FAT на случай сбоя первой, данные обновляются в обеих таблицах одновременно. В FAT свободные кластеры помечаются кодом 0, таким образом не существует отдельного списка свободных блоков. Диапазон номеров fff0-fff6 является зарезервированным, сбойные – кодом ffF7, последние в файловой цепочке – кодом ffFF.

FAT12 занимала в ОЗУ 4096 элементов по 2 байта на элемент. Поскольку MS-DOS поддерживала до 4 дисковых разделов на диске, то FAT12 максимально могла работать с дисками объемом 64Мб. В системе **FAT16** используется 16 бит для указания номера кластера, соответственно возможно не более 64К кластеров или элементов таблицы. FAT16 занимает в памяти 128Кб. При макс. разделе в 2Гб FAT16 могла работать с дисками 8Гб. Первая версия Win95 использовала FAT12 и FAT16 стандарта MS-DOS, с именами файлов 8.3. с выходом Win 95 OEM SR2 появилась поддержка длинных имен и разработана **FAT32**. Реально используются 28 разрядный адрес. Исчезли ограничения на объем логического диска 2Гб, при этом при той же емкости логического диска могут использоваться блоки (кластеры) меньшего размера. FAT32 может перемещать корневой каталог и использовать резервную копию загрузочной записи. Корневой каталог представлен в виде обычной цепочки кластеров, и может находиться в любом месте диска, а не только в системной области, что снимает ограничение предыдущих версий на 512 элементов. Система может поддерживать диски емкостью до 4Тбайт.

Хотя FAT12 не позволяет адресовать большой раздел, она используется Windows как формат для дискет. Если форматировать раздел (том) объемом менее 16Мб для FAT утилитой format или инструмента DiskManagement Windows использует не FAT16, а FAT12. Все FAT резервируют первые 2 кластера тома и 16 последних.

В Windows используются следующие локальные драйвера файловых систем: Ntfs.sys (NTFS), fastfat.sys (FAT), cdfs.sys (файловая система CD-ROM, только для чтения поддерживает ISO-9660 и расширения Joliet), udfs.sys (UDF совместимая реализация OSTA – Optical Storage Technology Association. OSTA определяет UDF как формат магнитооптических носителей, в основном DVD, это подмножество формата ISO-13346 с расширениями, для замены ISO-9660), и драйвер raw FSD, интегрированный в ntosKrnl.exe. Все эти ФС резервируют первый сектор тома как загрузочный, анализируя его, драйвер ФС может идентифицировать свой формат и найти необходимые метаданные. Распознав том, драйвер ФС создает объект Устройство, представляющий смонтированную ФС. Диспетчер вв связывает этот объект с объектом Устройство, созданным драйвером ввода-вывода через блок параметров тома. В результате диспетчер вв перенаправляет через блок параметров запросы вв, адресованные объекту Устройство вв, на объект Устройство ФС. Дополнительно локальные драйверы ФС использует диспетчер кэша. Кроме того они поддерживают демонтирование ФС, позволяющие ОС отсоединять драйвер ФС от объекта Устройство. Демонтирование происходит, когда приложение напрямую обращается к содержимому тома или при смене носителя. При первом обращении после демонтирования диспетчер вв повторно инициирует операцию монтирования.

HPFS – High Perfomance File System – высокопроизводительная ФС. Впервые появилась в ОС OS/2. Разработана лучшими специалистами IBM и Microsoft как система для многозадачного режима и обеспечения высокой производительности при работе с дисками больших объемов. Она стала первой системой, в которой реализована поддержка длинных имен. Она обладает типичной структурой каталогов, однак4о поддерживает их автоматическую сортировку и расширенные атрибуты файлов, позволяющие хранить дополнительную информацию о файле, например сопоставленное с файлом графическое изображение, описание файла, комментарий и др. В итоге такая организация позволяет упростить обеспечение безопасности и создание множественных имен. Используется несколько базовых идей: каталоги размещаются в середине диска; для поиска используются бинарные сбалансированные деревья; информация о местоположении файловых записей рассредоточено по всему диску а записи файла располагаются по возможности в смежных секторах и поблизости от данных о его местоположении. Это существенно сокращает время позиционирования головок и время ожидания пока под головкой не окажется нужный сектор. Структура раздела HPFS следующая: загрузочный блок (boot block секторы 0-15, содержит имя тома, серийный номер, блок параметров BIOS, программу начальной загрузки), дополнительный блок (super block, сектор 16, содержит указатели на список битовых карт, список сбойных блоков, полосу каталогов, файловый узел корневого каталога, дату последней проверки раздела утилитой, размер полосы), резервный блок (17 сектор, содержит указатели на карту аварийного замещения, список свободных запасных блоков каталогов для операций на сильно заполненном диске, ряд системных флагов и дескрипторов, обеспечивает высокую отказоустойчивость системы, позволяя восстанавливать поврежденные данные и переносить их в другое место), полоса 1, битовая карта 1, битовая карта 2, полоса 2, полоса 3, битовая карта 3, битовая карта 4, полоса 4 и т.д. Вначале идет несколько управляющих блоков, далее пространство разбито на области из смежных секторов – полос, в которой расположены данные файлов и вспомогательная служебная информация о свободных или занятых секторах полосы. Полоса занимает 8 Мб диска и имеет свою битовую карту занятости секторов. Такая структура позволяет разместить в непрерывном пространстве файл размером до 16Мб.

Файлы и каталоги базируются на файловом узле. Каждый файл и каталог имеют свой файловый узел, занимающий 1 сектор, расположенный поблизости от файла, обычно сразу перед ним. Он содержит размер файла, первые 15 символов имени файла, специальную служебную информацию, статистику по доступу к файлу, расширенные атрибуты и список управления доступом либо его часть. Если расширенные атрибуты занимают много место, они выносятся отдельно, а узел содержит только указатель на них. Если файл непрерывен, он описывается в узле 2 32 разрядными числами – указатель на первый блок файла, и длину непрерывного экстента (число последовательных блоков файла). Для фрагментированного файла узел содержит несколько таких пар. Для того, чтобы файл по возможности оставался непрерывным, в конце каждого из них система старается зарезервировать пространство хотя бы в 4 Кб для роста. В узле помещается информация максимум о 8 экстентах. Если их больше, в узел помещается указатель на блок размещения, содержащий до 40 указателей на экстенты или на другие блоки размещения. Полоса каталогов находится в середине диска. Если она заполняется полностью, HPFS использует и другие полосы. Поскольку обращения к каталогам наиболее часты, а размещение такой информации в середине диска в среднем снижает время позиционирования в 2 раза, то производительность уже значительно возрастает.

Структура каталога представляет сбалансированное бинарное дерево, записи в котором расположены в алфавитном порядке. Если в FAT для поиска файла требуется в среднем просмотреть N/2, в худшем - N записей, то здесь соответственно Int[$\log_2 N$] в худшем случае. Каждая запись дерева содержит атрибуты файла, указатель на соответствующий файловый узел, информацию о дате и времени создания файла, обновления и обращения, об объеме расширенных атрибутов, счетчик обращений к файлу, длина имени и само имя, другую информацию. При переименовании файла может потребоваться перебалансировка дерева. В результате, если диск переполнен, может не хватить дискового пространства для этой операции. Поэтому используется небольшой пул свободных блоков, указатель на который хранится в резервном блоке. Для исправления ошибок используется механизм аварийного замещения HotFix. Если при записи обнаруживается сбойный сектор, информация записывается в один из резервных секторов. Карта аварийного замещения представляет собой пару номеров – первый это номер сбойного сектора, второй – номер сектора для его замещения. При операциях чтения записи просматривается эта карта и при необходимости выполняется замена адреса. Это не влияет существенно на производительность, поскольку оно выполняется только при физической операции, а не при чтении данных из кэша. При проверке диска утилитой, замещенные сектора переносятся в новый обычный сектор диска, наиболее подходящий для файла, с учетом сохранения его непрерывности. Соответственно данные в карте аварийного замещения обнуляются, а номер сбойного сектора помещается в соответствующий список, который хранится в дополнительном блоке HPFS. Большинство объектов ФС, в т.ч. файловые узлы, блоки размещения и блоки каталогов имеют уникальные 32 разрядные идентификаторы и указатели на свои родительские и дочерние блоки. Анализ файловых узлов, блоков размещения и каталогов во многих случаях позволяет восстановить структуру ФС после сбоя.

ФС поддерживает управление алгоритмами оптимизации запросов, приоритетов, глубину просмотра очереди. Соответствующий файл инициализации в OS/2 поддерживает ряд соответствующих параметров, позволяющих выбрать следующие варианты оптимизации доступа: FIFO, элеваторный алгоритм, который используется и по умолчанию, либо алгоритм, выбранный менеджером дисковых операций. При поддержке приоритетов используются отдельные очереди запросов для каждого приоритета. Если приоритеты не поддерживать, используется одна общая очередь. По умолчанию приоритеты поддерживаются. Можно задать и тот вариант, который был выбран менеджером дисковых операций. Глубина просмотра очереди для оптимизации запросов задается величиной от 1 до 255. По умолчанию глубина определяется автоматически на основании рекомендации драйвера дискового адаптера.

В целом по организации, HPFS является самой высокопроизводительной ФС.

При проектировании **NTFS** особое внимание было уделено следующим характеристикам:

- надежность. Высокопроизводительные системы и серверы должны обладать повышенной надежностью. Один из способов повышения надежности – механизм транзакций, при котором выполняется журналирование файловых операций.
- расширенная функциональность. NTFS проектировалась с учетом возможности расширения. В ней повышена отказоустойчивость, имеется эмуляция других ФС, мощная модель безопасности, параллельная обработка потоков данных и создание файловых атрибутов, определяемых пользователем.

- поддержка POSIX. (Portable Operating system for computing environments) Переносимая ОС для вычислительных сред. Разработан в 1988 и с 1990 является стандартом. Представляет собой набор функций, взятых из ОС AT&T UNIX System V и Berkeley Standart Distribution UNIX. Основное внимание этого стандарта уделено интерфейсу прикладных программ с ОС. Имеется механизм жестких ссылок, позволяющий ссылаться на один и тот же файл по нескольким именам.
- гибкость. Размер кластера может изменяться в пределах от 512 байт до 64 Кб. Поддержка длинных имен файлов и имена 8.3 для совместимости с FAT.

Максимально возможные размеры тома (и файла) составляют 16 экзабайт (2^{64}). В структуру каталогов заложена модель сбалансированного бинарного дерева, имеются средства самовосстановления, поддерживается объектная модель безопасности NT, при которой все тома, каталоги и файлы рассматриваются как самостоятельные объекты. Безопасность обеспечивается на уровне файлов. Система обладает встроенными средствами сжатия.

Структура тома. NTFS делит все полезное дисковое пространство тома на кластеры, наиболее часто используется кластер в 2 или 4 Кб, поддерживая размеры от 512 байт до 64К. Дисковое пространство делится на 2 неравные части. 12% диска отводятся под зону MFT master file table. Запись в эту зону невозможна, она используется для роста метафайла MFT без фрагментации.

MFT	зона MFT	Зона для размещения файлов и каталогов	Копия первых 16 записей MFT	Зона для размещения файлов и каталогов

MFT представляет собой централизованный каталог всех остальных файлов диска, в том числе и себя самого. MFT разделен на записи размера в 1 Кб, каждая из которых соответствует какому-либо файлу. Размер файловых записей MFT для тома определяется во время форматирования и может находиться в пределах от 1 до 4 Кб. Первые 16 файлов носят служебный характер и недоступны ОС, называются метафайлами, причем первый файл – сам MFT. Эти 16 элементов имеют строго фиксированное положение и имеют копию в середине диска. Остальные части MFT могут находиться в произвольных местах диска. Метафайлы находятся в корневом каталоге NTFS тома, их имена начинаются с $.
$MFT – сам MFT
$MFTMirr – копия 16 записей в середине тома
$LogFile – файл поддержки операций журналирования
$Volume – Служебная информация – метка тома, версия файловой системы и т.д.
$AttrDef – список стандартных атрибутов файлов тома
$. – корневой каталог
$Bitmap – карта свободного места тома
$Boot – загрузочный сектор
$Quota – файл с правами пользователей на использование дискового пространства (начиная с NTFS 5.0)
$Upcase – таблица соответствия заглавных и прописных букв в именах файлов
В записях MFT хранится вся информация о файлах, кроме собственно данных, имя файла, размер, положение на диске отдельных фрагментов и т.д. Если одной записи MFT не хватает, используется несколько, не обязательно идущих подряд. Если файл небольшого размера, то он хранится в самой MFT, в свободном месте в пределах одной записи. Файл в томе идентифицируется файловой ссылкой в виде 64разрядного числа. Это номер файла, соответствующий позиции его записи в MFT и номера последовательности, который увеличивается , если эта позиция в MFT используется повторно

Файл представляется с помощью потоков. Потоками файла являются не только данные, но и его атрибуты. Т.е. сущность файла – его номер в MFT, а все остальное, в т.ч. потоки – опциональны. Соответственно файлу можно назначить новый поток, записав в него любые данные. В Windows 2000 так пишется информация об авторе и содержании файла. Эти дополнительные потоки не просматриваются стандартными средствами, например, размер файла – это размер только основного потока с данными. В результате можно удалить короткий файл, а освободится несколько Мб. Максимальная длина 1 потока – 16 Эб. Стандартные атрибуты файлов и каталогов тома NTFS имеют фиксированные имена и коды типа.

Стандартная информация о файле. – Традиционные атрибуты Read Only, Hidden, Archive, System, отметки времени создания и модификации, число каталогов, ссылающих на файл
Список атрибутов. - Список атрибутов, из которых состоит файл, файловая ссылка на файловую запись и MFT, в которой расположен каждый из атрибутов, если файлу необходимо более одной записи MFT.
Имя файла. - Имя файла в символах Unicode. Может иметь несколько атрибутов-имен. Например, если имеется связь POSIX с файлом или имеется имя формата 8.3.
Дескриптор защиты.- Структура данных, предохраняющая от несанкционированного доступа. Определяется владелец файла и кто имеет доступ.
Данные. - Собственно данные файла. У файла по умолчанию имеется один безымянный атрибут данных, он может иметь дополнительные именованные атрибуты данных. У каталога нет атрибута данных по умолчанию, но может иметь необязательные именованные атрибуты данных
Корень индекса, размещение индекса, битовая карта (только для каталогов) – атрибуты для индексов имен файлов в больших каталогах.
Расширенные атрибуты HPFS – атрибуты, используемые для реализации расширенных атрибутов HPFS для подсистемы OS/2 и OS/2 клиентов файл-серверов Windows NT.

Атрибуты файла в записях MFT расположены в порядке возрастания числовых значений кодов типа, причем некоторые атрибуты, такие как данные или имена, могут встречаться несколько раз. Обязательны атрибуты стандартной информации, имени файла, дескриптора защиты и данных. Остальные атрибуты опциональны. Блоки файла описываются аналогично HPFS – последовательностью пар, определяющих экстенты. Если свободной области в записи недостаточно для хранения всех экстентов, в запись помещаются номера записей MFT, содержащих информацию об экстентах. Имя файла может содержать любые символы Unicode, в т.ч. и символы национальных алфавитов, длиной до 255 символов. Каталог представляет собой специальный файл, хранящий ссылки на другие файлы и каталоги. Файл поделен на блоки, каждый блок содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который содержит полную информацию. Внутренняя структура каталога представляет собой бинарное дерево.

NTFS имеет следующие операции, которые могут быть разрешены для работы с файлом: чтение, запись, выполнение, удаление, изменение разрешений, получение владения файлом. Некоторые их сочетания используются ФС в качестве стандартных:

Стандартные разрешения NTFS	Соответствующие им комбинации индивидуальных разрешений	
	для каталогов	для файлов
Нет доступа – No access	Нет разрешений	нет разрешений
просмотр – list	Read, Execute	нет разрешений
чтение – read	Read, Execute	Read, Execute
добавление – add	Write, Execute	нет разрешений
чтение и добавление – Add & read	Read, write, Execute	Read, Execute
изменение – change	Read, write, Execute, delete	Read, write, Execute, delete
полный доступ – full control	все	все

Индивидуальные разрешения NTFS	Для каталога	Для файла
Чтение Read	Просмотр имен каталога, файлов в нем, разрешений на доступ к нему, атрибутов каталога и сведений о владельце	Просмолтр содержимого файла, разрешений на доступ к нему, его атрибутов и сведений о его владельце
Запись Write	Добавление в каталог файлов и папок, изменение атрибутов каталога, просмотр атрибутов каталога, сведений о владельце и разрешений на доступ к нему	Просмотр разрешений на доступ к файлу и сведений о владельце, изменение атрибутов файла, изменение и добавление данных файла
Выполнение Execute	Просмотр атрибутов каталога, изменения во вложенных папках, просмотр разрешений на доступ к каталогу и сведений о его владельце	Просмотр разрешений на доступ к файлу, его атрибутов и сведений о его владельце, запуск файла
Удаление Delete	Удаление каталога	Удаление файла
Смена разрешений Change Permissions	Изменение разрешений на доступ к каталогу	Изменение разрешений на доступ к файлу
Смена владельца Take OwnerShip	Назначение себя владельцем каталога	Назначение себя владельцем файла

NTFS поддерживает прозрачное сжатие файлов. Если файл создается в сжатом режиме, система автоматически сжимает данные при записи и распаковывает их при чтении. Алгоритм сжатия применяется независимо для серий из 16 последовательных блоков. Если при сжатии получается выигрыш хотя бы в 1 блок, записывается сжатый вариант, иначе полный. Сжатая область записывается в MFT в виде 2 экстентов, первый соответствует реальному физическому объему сжатых данных, вторая пара имеет 0 в качестве первого числа, и количество сжавшихся блоков файла в качестве второго. 0 служит идентификатором, что предыдущий экстент сжат. NTFS имеет дополнительно драйвер-фильтр шифрования, позволяющий на лету зашифровывать файлы, в настоящий момент по модифицированному алгоритму DES. NTFS не может использоваться для форматирования гибких дисков

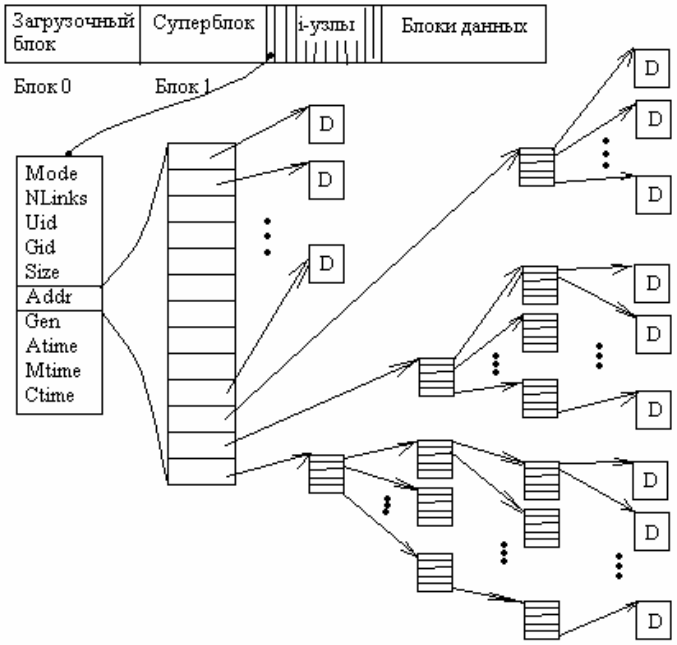
CD - ISO9660. Стандарт был принят в 1988. Стандартом были наложены ряд ограничений на формат, чтобы считать данные могла даже самая слабая из имеющихся на тот момент ОС. В отличие от жестких дисков, у CD нет концентрических цилиндров, имеется непрерывная спираль. На ней последовательно размещены биты, которые делятся на логические блоки (логические сектора) объемом 2352 байт. Часть из них используется для преамбул, коррекции ошибок и т.д. Чисто информация в блоке содержит 2048 байт. Аудиодиски содержат специальные разделительные участки между композициями и специальные заголовки и концевики. Диски могут разделяться на отдельные логические тома. Диск начинается с 16 блоков, назначение которых не определяется стандартом. Они могут использоваться для размещения загрузчика ОС. За ними идет блок, содержащий **основной описатель тома**. Он содержит помимо прочего идентификатор системы (32 байта), идентификатор тома (32 байта), идентификатор издателя (128 байт), идентификатор составителя данных (128 байт). Эти поля могут заполняться произвольным образом, однако используются только символы верхнего регистра, цифры и некоторые знаки препинания для совместимости с разными платформами. ООТ также содержит имена трех файлов, которые могут содержать краткий обзор, уведомление об авторских правах и библиографическая информация. Содержит размер логического блока, количество блоков на диске, дата создания и дата окончания срока службы диска. Содержит описатель корневого каталога, что позволяет найти его на диске. От него можно определить оставшиеся файлы и каталоги. Кроме ООТ может содержаться дополнительный описатель тома. Корневой каталог может содержать произвольное число записей. Последняя запись имеет специальный

маркер для указания, что он последний. Записи могут иметь переменную длину. Они содержат от 10 до 12 полей. Нетекстовые – двоичные – поля кодируются дважды – сначала в модели типа Pentium (младшие байты по младшему адресу), затем в модели типа SPARC (старшие байты по младшему адресу). Первое поле каталога (1 байт) –длина записи, если записи имеют расширенные атрибуты, то второй байт содержит длину записи расширенных атрибутов. Далее 8 байт (32-битный адрес) указывают номер начального блока файла. Файл хранится в виде непрерывной последовательности блоков, поэтому следующее поле (8 байт) содержит его размер. Следующее поле (7байт) содержит дату и время создания файла, годы отсчитываются от 1900. Следующий байт содержит флаги. Среди них есть флаг скрытый, разрешение расширенных атрибутов, последняя запись каталога. Далее два байта описывают чередование частей файла на диске. В простейшей версии стандарта оно не используется. Следующие 4 байта содержат номер диска в наборе, на котором расположен файл. Соответственно на диске может иметься каталог файлов для набора дисков в количестве 2^{16}. Следующий байт содержит длину имени файла в байтах. Далее идет имя файла, точка, расширение, точка с запятой, один или два байта версии файла. В имени могут использоваться прописные символы, цифры, символ подчеркивания. Допускается длина до 8 символов, расширение до трех символов. Еще два поля могут и не использоваться. Это поле заполнение, которое используется для выравнивания размера записи до четного количества байт. Если выравнивание требуется, то используется нулевой байт. И еще одно поле – системное – никак не определено стандартом, в том числе и размер. Оно должно состоять из четного числа байт.
Все записи каталога, кроме 2 первых, расположены в алфавитном порядке. Первая запись – описатель самого каталога, вторая – ссылка на родительский каталог. Максимальная глубина вложенности каталогов равна 8. Стандарт предусматривает 3 уровня ограничений. 1 уровень – самый жесткий: это формат имени 8.3, имена каталогов 8.0, файлы должны быть непрерывными. 2 уровень. Имена могут иметь длину до 31 символа из того же набора. 3 уровень. Файл может состоять из нескольких разделов, каждый представляет собой непрерывную последовательность блоков. Одна и та же последовательность может несколько раз встречаться в одном и том же файле и даже принадлежать разным файлам.
Чтобы ФС UNIX могла быть представлена на CD_ROM, были разработаны расширения. Это **Рок-Ридж** расширения. Они используют системное поле каталога для совместимости. Если система не поддерживает расширения, системное поле будет игнорировано. Расширения содержат следующие поля: PX – атрибуты POSIX, фактически стандартные биты разрешений владельца, группы, и т.д. PN – старший и младший номер устройства, ассоциированного с файлом, что позволяет сохранять каталог /dev. SL – символьная связь, позволяет файлу из одной ФС ссылаться на файл из другой ФС. NM – альтернативное имя, которое можно указывать без каких либо ограничений. CL,PL – расположение дочернего узла. RE – перераспределение. Эти три поля используются, чтобы обойти ограничение на глубину вложенности каталогов. С их помощью можно указать куда в иерархии должен быть помещен тот или иной каталог. TF – Временные штампы. Содержит времена создания, последнего изменения, последнего доступа к файлу. Такая структура расширений позволяет полностью скопировать ФС UNIX на CD_ROM, а после полностью восстановить ее.
Другую группу расширений разработала Microsoft. Это расширения **Joliet**. Они должны были позволить также полностью копировать на диск ФС Windows, а потом ее восстанавливать. Возможности этих расширений: длинные имена файлов до 64 знаков, набор символов UNICODE (т.е. имя из 64 знаков может занимать 128 байт), большая глубина вложенности каталогов, имена каталогов с расширениями.

UNIX. Изначально UNIX могла использовать только один тип ФС. Позже Sun Microsystems разработала интерфейс vnode/vfs, позволяющий сочетать различные из них. Поскольку в UNIX понятие файл включает в себя различные абстракции, в т.ч. сетевые соединения через сокеты, каналы и очереди FIFO, блочные и символьные устройства, то и в архитектуре vnode/vfs файлы и ФС являются базовыми элементами, представляющими модульный интерфейс взаимодействия с остальной частью ядра. Помимо ФС в общепринятом значении, такой подход позволил разработать ФС специализированные ФС, например, для работы с адресным пространством любого процесса.

S5FS. System V File System. Эта оригинальная ФС, поддерживаемая UNIX изначально. Раздел представляется в виде набора блоков. Размер каждого блока от 512 б и более, по степени 2. Драйвер преобразует номер блока в номера цилиндра, дорожки и секторов на диске. В начале раздела содержится загрузочная область с кодом начальной загрузки. За ней расположен суперблок, содержащий атрибуты и метаданные ФС. Далее идет список индексных узлов файлов. Размер i-узла 64 байта. В начальном блоке может быть размещено несколько i-узлов. Начальный адрес суперблока и списка i-узлов постоянен для любого раздела. Список i-узлов имеет постоянный размер, ограничивая максимальное число файлов в разделе, это задается при создании ФС на разделе. За таблицей i-узлов идет область данных. Она содержит непосредственно файлы данных и каталоги, а также блоки косвенной адресации, содержащие указатели на блоки данных файлов.
Суперблок содержит размер ФС в блоках, размер списка i-узлов в блоках, количество свободных блоков и i-узлов., список свободных блоков, список свободных i-узлов. При этом не содержится полный список свободных узлов, а только какая-то их часть. Когда известные узлы заполнятся, ядро сканирует диск для поиска других свободных узлов и добавляет их в список. Для блоков это невозможно. Полный список свободных блоков занимает несколько блоков диска. Суперблок содержит только первую часть этого списка. Первый элемент списка в блоке указывает на следующий блок, содержащий список.
Каждый файл имеет свой i-узел, содержащий служебную информацию или метаданные файла. При открытии файла информация из i-узла считывается в память в специальную структуру, которая имеет ряд дополнительных полей. Поля i-узла диска:

di_mode 2 байта тип файла, привилегии и т.д.
di_nlinks 2 количество жестких ссылок на файл
di_uid 2 идентификатор владельца
di_gid 2 идентификатор группы владельца
di_size 4 размер файла в байтах
di_addr 39 массив адресов блоков файла
di_gen 1 генерируемый номер (инкрементируется при запросе индексного дескриптора для нового файла)
di_atime 4 время последнего доступа
di_mtime 4 время последней модификации
di_ctime 4 время последнего изменения индексного дескриптора



4 старших бита типа файла позволяют указать обычный ли это файл, каталог, блочное либо символьное устройство и т.д. Младшие 9 бит определяют права доступа для владельца, группы и т.д. Поле addr позволяет хранить 13 элементов массива блоков файла: номер блока диска занимает 3 байта. Первые 10 элементов содержат номера блоков с данными. Если этого недостаточно, то 11 элемент содержит номер блока, содержащего не данные, а остальные номера блоков файла, т.н. блок косвенной адресации. Если и этого недостаточно, то 12 элемент содержит номер блока двойной косвенной адресации, который содержит номер блока, содержащего адреса блоков, содержащих номера остальных блоков файла, а если и этого мало, то 13 элемент содержит номер блока тройной косвенной адресации. Если какой-либо блок файла не содержит данных, например, в результате перемещения указателя для записи сразу за него, то такой блок на диске не хранится, соответствующий адрес блока в массиве устанавливается в 0.
Каталог содержит список файлов и подкаталогов. Размер записи – 16 байт. Первые 2 содержат номер i-узла, соответствующего файлу, еще 14 – имя файла. В результате на диске не может быть более 2^{16}=65535 файлов – по количеству возможных номеров i-узлов. Если имя файла меньше 14 символов, оно завершается 0. Каталог, поскольку тоже является файлом, имеет и собственный i-узел. Первые два элемента каталога – это сам каталог и родительский каталог.
Недостатком является отсутствие копии суперблока, что снижает надежность, далее все i-узлы находятся в начале диска, а данные – в оставшейся части, что вызывает дополнительное перемещение головок при чтении файла. Кроме этого ограничение на общее число файлов и длину имени.

В результате была разработана **FFS** Fast File System в лаборатории Berkeley. В целом она предоставляет те же возможности, однако оптимизирована по быстродействию и снят ряд ограничений. Каждый раздел делится на одну или несколько групп цилиндров. Информация суперблока делится на 2 части. Первая содержит сведения о ФС в целом, эта информация может измениться только при форматировании. Каждая группа цилиндров содержит структуру с информацией о группе, в т.ч. списки свободных i-узлов и блоков. Кроме этого, каждая группа цилиндров хранит дубликат суперблока. Причем эти копии в каждой группе находятся на разном смещении. В результате суперблок распределен по разделу. Размер блока установлен в 4К, либо 8К в отличие от 512байт и 1К s5fs. Это позволило отказаться от блоков тройной косвенной адресации. Для маленьких файлов такой размер блока вызывает существенные потери дискового пространства. Поэтому для блоков, не попавших в косвенную адресацию, блок делится на 1,2,4 или 8 фрагментов, объемом минимум 512 байт. Файл должен занимать целые дисковые блоки, за исключением последнего. Он может содержать только один или несколько последовательных фрагментов. В результате один блок может содержать несколько конечных фрагментов разных файлов, что снижает потери. Помимо этого система оптимизирует размещение файлов и каталогов по группам цилиндров. Например, блоки данных система пытается разместить в той же группе цилиндров, что и его i-узел. Длина записи каталога может быть различна. Постоянная часть содержит номер i-узла, размер переменной части, тип файла, размер имени файла, далее идет имя файла, заканчивающееся 0, длина имени ограничена 255 символами. Каталог делится на 512-байтовые области, 1 элемент не может занимать несколько таких областей. Система стала поддерживать символические ссылки. Поддержка FFS была включена в SVR4.

Linux **Ext2fs**. В целом похожа на FFS. Поскольку для современных дисков, скрывающих физическую геометрию, и предоставляющих виртуальный интерфейс, разделение на группы цилиндров ни к чему не приводит, то после загрузочного блока, система делит раздел на группы блоков. Каждая группа начинается с суперблока, в котором содержится информация сколько блоков и i-узлов находится в данной группе, размере группы и т.д. Далее

идет описатель группы, хранящий информацию о расположении битовых массивов, количестве свободных блоков и i-узлов в группе а также каталогов в группе. В битовых массивах ведется учет свободных блоков и i-узлов. Размер каждого из массивов – 1 блок. За битовыми массивами располагаются сами i-узлы, размером по 128 байт каждый Это позволило хранить 12 прямых и 3 косвенных дисковых адреса длиной по 4 байта, а не по 3. Имеются поля, зарезервированные для указателей на списки управления доступом, но это на будущее. Дисковые блоки используются фиксированного размера в 1 Кб. Система

также пытается оптимизировать расположение блоков файла на диске. Так, новый блок файла по возможности помещается в ту же группу, что и остальные блоки, желательно сразу за ними. Новый файл – в той же группе блоков, что и блоки каталога. Новые каталоги равномерно распределяются по диску.
Кроме того, Linux использует ФС **proc**. Для каждого процесса в каталоге /proc создается подкаталог с именем, равным PID процесса в десятичном виде. В подкаталоге содержатся файлы, хранящие информацию о процессе –командную строку, строки окружения, маски сигналов и т.д. Реально таких файлов на диске нет. Многие расширения, реализованные в Linux, расположены в /proc. Это позволяет пользовательскому процессу читать системную информацию безопасным для системы образом.

ФС с журнальной структурой. LFS. Log-structured File System Идея в том, что по мере увеличения скорости процессоров и объема ОЗУ кэширование становится все выгоднее. Становится возможным удовлетворить существенную часть всех дисковых запросов непосредственно из кэша ФС без обращения к диску. Следовательно, большинство обращений к диску будут обращениямина запись, а не на чтение. Поэтому алгоритм опережающего чтения становится малоэффективным. Далее, в большинстве ФС запись выполняется небольшими блоками данных, что также неэффективно, поскольку помимо собственно записи выполняется еще и относительно длинный поиск цилиндра. Например, в UNIX для записи в файл требуется: выполнить операции записи в i-узел каталога, блок каталога, i-узел файла и блок самого файла. Система LFS пытается учесть эти особенности. Идея в том, что диск используется как журнал. Периодически, когда возникает необходимость, все буферизированные в памяти блоки, которые должны быть записаны, собираются в единый сегмент, и он записывается на диск единым блоком в конец журнала. Этот сегмент может содержать i-узлы, блоки каталогов, блоки данных, перемешанные друг с другом. В начале каждого сегмента создается оглавление сегмента. Если средний размер сегмента довести до 1 Мб, пропускная способность диска может бть использована практически на 100%. Для быстрого поиска i-узлов, поскольку теперь они могут располагаться в произвольной области, создается массив, j элемент которого хранит указатель на j i-узел. Этот массив хранится на диске и вкэше. Поскольку постепенно весь объем диска будет использован журналом, то в такой ФС определен чистящий поток, постоянно сканирующий журнал, чтобы делать его более компактным. Поток считывает содержимое сегмента журнала, определяя какие i-узлы и файлы в нем находятся. Далее проверяется текущий массив i-узлов, чтобы определить, являются ли i-узлы текущими и используются ли все еще блоки файлов. Если нет, то такая информация отбрасывается, а все еще используемые узлы и блоки считываются в память, чтобы быть записанными в следующий сегмент. Исходный сегмент отмечается как свободный, и может быть использован для новых данных.Чистильщик двигается по журналу от сегмента к сегменту, а диск фактически представляет большой кольцевой буфер, в котором пишущий поток добавляет сегменты с одного конца, а чистильщик удаляет их с другого.

Семестр 2. Теория компиляторов

Современные системы программирования

История возникновения трансляторов. Классификация трансляторов. Трансляторы, компиляторы, интерпретаторы. Этапы трансляции. Технология и средства разработки программ. Краткая история систем программирования. Структура современных систем программирования. Классификация языков программирования. Непроцедурные языки программирования. Принципы организации SQL. Технологии 4GL. Принципы организации технологий на основе Java, HTML

Литература

1. Компаниец Р.И., Маньков Е.В., Филатов Н.Е. Системное программирование. Основы построения трансляторов. Спб., Корона принт, 2004.
2. Соколов А.П. Системы программирования: теория, методы, алгоритмы. Учеб. пособие. М., Финансы и статистика, 2004.
3. Молчанов А.Ю. Системное программное обеспечение. Учебник для ВУЗов. Спб., Питер, 2003.
4. Карпов Ю.Г. Теория автоматов. Учебник для ВУЗов. СПб., Питер, 2003.
5. Опалева Э.А., Самойленко В.П. Языки программирования и методы трансляции. СПб, БХВ-Петербург, 2005.
6. Мозговой М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. СПб, Наука и техника, 2006.

История возникновения трансляторов.

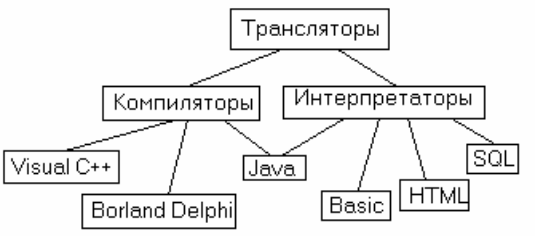
Первые программы для МП разрабатывались фактически в машинных кодах - т.е. непосредственно использовалась система команд МП. Например, для процессора Intel Pentium команда сброса флага переноса имеет код **F8**. Команда получения информации о текущем процессоре является двухбайтовой, ее код **0F A2**. Команда чтения байта из порта с заданным номером имеет 2 байтовый код вида **E4 <номер порта>**. Команда увеличения на 1 значения в регистре AX имеет код **40**, BX – **41**. Более сложные команды, такие как пересылки имеют несколько разновидностей, например, регист-память, регистр-регистр, память-регистр, более того при этом может использоваться как явная, так и неявная адресация регистров, также как и адресация памяти может быть прямой и косвенной. Каждый из этих вариантов имеет свой код в системе команд. Очевидно, что писать программу непосредственно в кодах очень и очень трудоемкая операция. Каждый МП имеет свою собственную систему команд. Это значит, что программа, написанная для одного МП не могла быть перенесена на другой МП с помощью элементарных действий. Кроме того с появлением нового МП программисту требовалось переобучение.

Как результат, появились первые языки Assembler, использующие мнемонические обозначения команды. Так, вышеприведенные команды будут иметь вид**:**

CLC	CPUID	IN AL, <номер порта>	INC AX	INC BX	INC <регистр>	MOV <назначение>, <источник>
F8	0F A2	E4 <номер порта>	40	41	...	

С помощью мнемонических обозначений программы стали значительно нагляднее, хотя язык ассемблера попрежнему низкоуровневый и ориентирован на архитектуру конкретного МП. Возникла очевидная необходимость в программах-переводчиках с языка Assemblera, а в дальнейшем и с языков более высокого уровня, в машинные коды. Эти программы получили название *транслятора* – от англ. Translate – переводить.

Классификация трансляторов.

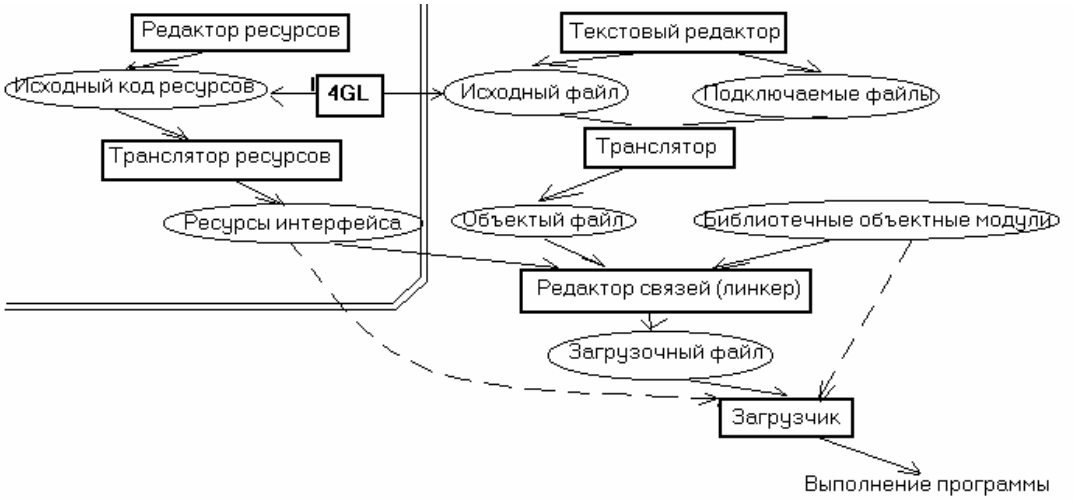


Трансляторы – это программные средства, выполняющие преобразование программ, представленных на одном языке, в эквивалентную ей программу на другом языке. Все трансляторы делятся на 2 большие группы – *компиляторы и интерпретаторы*. **Компиляторы** переводит программу с исходного языка на язык более низкого уровня. Чаще всего в машинные коды. На входе компиляторы получают исходный текст программы, а на выходе выдают готовую программу в машинном, объектном или ином промежуточном коде. Пример компиляторов – C, C++, Pascal. Компилятор с языка Assembler традиционно называется **ассемблером**. **Кросс-компилятор** выполняет трансляцию программы на одной платформе, формирую объектный код для другой платформы. Еще одна разновидность компиляторов – это **компилятор для построения компиляторов**. В этом случае разрабатываемый язык описывается в терминах формальных грамматик, а на выходе компилятора формируется текст программы на языке высокого уровня, как правило, C, позволяющей выполнять компиляцию программ на разрабатываемом языке. Примерами таких компиляторов служат LEX, YACC. **Интерпретаторы** не формируют готовой программы, выполнение исходной программы происходит по частям, по мере ее обработки. Примером интерпретатора может служить транслятор языка Visual Basic. Очевидно, что процесс выполнения программы в этом случае медленнее. **Препроцессор** – это транслятор для макрорасширений языка, который переводит их в программу на входном языке. Препроцессор для ассемблера называется **макрогенератором**. **Детрансляторы** выполняют обратную трансляцию с языков более низкого уровня к языкам более высокого уровня. Детранслятор на язык ассемблера называется **дизассемблером**.

Этапы трансляции.

Трансляция традиционно разбивается на несколько этапов. Если входной язык допускает макроописания, выполняется обработка препроцессором. Далее следует этап лексического анализа. Его задача – проверка правильности лексики (написания) основных элементарных конструкций языка (лексем) – констант, идентификаторов, ключевых слов. Далее в дело

вступает синтаксический анализатор. Здесь выполняется правильность синтаксических конструкций, сформированных из лексем. Семантический анализ – слабоформализуемая часть трансляции, состоящая в жесткой проверке контекстных зависимостей, сводящаяся в большинстве случаев к проверке соблюдения правил объявления данных до их использования и иных подобных правил. Распределение памяти заключается в назначении адресов для данных программы, размещенных транслятором во внутренние таблицы. Этап оптимизации призван добиться оптимальной генерации в соответствии с критерием объема или быстродействия. Большинство задач оптимизации нетривиальны и требуют формализации семантики программы. Поэтому на практике решаются только простейшие из них – например, обнаружение неиспользуемых данных. Оптимизация может отсутствовать. На этапе генерации кода окончательно генерируется программа на выходном языке транслятора, эквивалентная исходной программе. Современные трансляторы, в отличие от ранних, не имеют такой жесткой ступенчатой иерархии, этапы в значительной степени интегрированы с синтаксическим анализом.



Структура современных систем программирования. Трансляторы являются основной, однако далеко не единственной частью современных систем программирования. **Система программирования** – это комплекс программных средств, предназначенных для кодирования, тестирования и отладки программного обеспечения. Программа на входном языке вводится с помощью **текстового редактора**. В общем случае текстовый редактор может быть произвольным, однако современные СП предоставляют встроенные редакторы, обладающие дополнительными возможностями. В ССП практически все действия по редактированию, отладке и запуску программы ведутся в среде текстового редактора. Он обладает встроенными функциями отображения ошибок, обнаруженных на этапах компиляции и компоновки, пошаговым отладчиком. Интеграция текстового редактора с лексическим анализатором позволило выделять графически в тексте программы лексемы в соответствии с их типом – ключевые слова, идентификаторы, константы и т.д. Эта же интеграция позволяет производить **лексический анализ на лету** – по мере ввода исходного текста. При запуске программы на компиляцию лексический анализ в этом случае оказывается уже выполненным. Создаваемые при этом таблицы идентификаторов могут использоваться встроенной в редактор системой подсказок и гиперссылок. Например, после ввода имени функции эта система выводит подсказку о передаваемых аргументах – типе, порядке следования и т.д. А при вводе имени экземпляра какого-либо класса выдавать перечень свойств и методов класса, которые могут быть использованы в данном контексте. Кроме того текстовый редактор позволяет вызывать справку по синтаксису используемого языка программирования. Структура и назначение **компилятора** уже рассматривались выше. Можно лишь отметить, что в ССП разработчик как правило не обращается к компилятору напрямую, вызывая его как функцию интегрированной оболочки. ССП могут иметь в своем составе не только компилятор исходного текста, но и ряд других компиляторов: например, транслятор на язык ассемблера, компилятор ассемблера. **Компоновщик (редактор связей)** предназначен для связывания между собой объектных файлов, созданных компилятором и библиотек СП. Фактически компоновщик устанавливает связи (т.е. определяет адреса) при вызове внешних по отношению в модулю функций и констант, объединяя все используемые модули в один исполняемый файл. В ССП компоновщик включает в исполняемый файл не только код объектных модулей, но и описание ресурсов пользовательского интерфейса. Поскольку программа может быть загружена ОС по любым доступным адресам виртуального адресного пространства пользователя, то компоновщик работает с относительными адресами переменных и функций. В ССП компоновщик как правило подключает не весь объектный код используемых библиотек, а только используемые ее части, что снижает объем исполняемого файла. Компоновщики в общем случае могут подключать и объектный код, созданный другой СП. За трансляцию относительных адресов в абсолютные отвечает **загрузчик**. В состав исполняемого модуля включается таблица, содержащая ссылки на адреса, которые необходимо транслировать. При запуске программы загрузчику уже известны реальные адреса и на основе информации из этой таблицы он изменяет относительные адреса в абсолютные. Загрузчик, выполняющий трансляцию адресов в момент запуска программы, называется **настраивающим**. В современных ОС трансляция адресов может происходить уже непосредственно при выполнении программы. Такая возможность обусловлена архитектурой МП. Загрузчик как правило является частью ОС. Различные ОС имеют различный формат таблиц настройки, что учитывается компоновщиком. В состав ОС входит и динамический загрузчик, отвечающий за обеспечение работы с динамически подключаемыми библиотеками и модулями. В ССП используются **ресурсы интерфейса** – множество данных, обеспечивающих внешний вид интерфейса пользователя, не связанных напрямую с логикой выполнения программы. Как правило функции работы с ресурсами пользовательского интерфейса входят в состав динамических библиотек ОС. ОС содержит и набор наиболее часто используемых ресурсов, которыми пользователь также может воспользоваться. ССП имеют в своем составе графические средства редактирования ресурсов, на основе чего и строится описание ресурса на специальном языке. Еще одной современной технологией является использование **4GL** языков (Fourth Generation Languages) и систем быстрой разработки приложений **RAD** (Rapid Application Development).

Классификация языков программирования.

Традиционно ЯП классифицируются по уровню – т.е. сложности решения задач с помощью этого языка. Чем проще записывается решение, чем нагляднее представляются сложные операции и понятия, чем меньше объем исходного текста, тем выше уровень. По этому признаку ЯП классифицируются следующим образом:

Процедурные языки позволяют указать **что и как** необходимо сделать для решения задачи. **Непроцедурные** языки позволяют указать **что** необходимо сделать, а как это сделать должна решить система программирования. Типичным примером непроцедурных языков является HTML и SQL. Рассмотрим для примера решение простой задачи с помощью SQL – structured query language – язык структурированных запросов, используемый при работе с БД. Этот язык позволяет создавать, модифицировать, удалять таблицы БД и других ее объектов, таких как триггеры, последовательности, представления, профили и т.д., а также добавлять, изменять и удалять записи в таблицах. ССП позволяют выполнить эти действия как с помощью SQL так и с помощью обычного процедурного языка. Рассмотрим, как решается простая задача – в таблице Money увеличить значение поля Salary на 10, если оно больше 100:

Код на Delphi

```
While not (Money.Eof) do begin
      If Money['Salary']>100 then begin
            Money.Edit;
            Money['Salary']:= Money['Salary']+10;
            Money.Post;
      End;
      Money.Next;
End;
```

Запрос на SQL

```
update money
set salary=salary+10
where salary>100
```

Язык SQL является типичным интерпретатором, однако его расширения, например, PL/SQL в Oracle, являются компилируемыми (правда это расширения является процедурным), что позволяет в архитектурах "клиент-сервер" хранить на сервере скомпилированный код хранимых процедур и триггеров, а также представлений. В ряде случаев для поступивших на сервер SQL запросов сохраняется дерево синтаксического разбора, что при поступлении однотипного запроса позволяет значительно снизить временные затраты на синтаксический анализ. **4GL** представляет собой набор средств, позволяющих проектировать и разрабатывать ПО не на основе синтаксических конструкций языка и элементов интерфейса, а при помощи представляющих их графических образов. Простейшие приложения в этом случае может разрабатывать квалифицированный пользователь, обладающий минимальными знаниями в области программирования. Пользователю не требуется определять все этапы выполнения программы, необходимые для решения поставленной задачи, достаточно задать необходимые параметры, на основе которых система программирования автоматически выполняет генерацию приложения. Среди языков 4GL можно выделить **языки представления информации и генераторы приложений**. К языкам представления информации относятся языки запросов, генераторы отчетов, форм и т.д. Типичный пример – MS Access, имеющий встроенные мастера построения отчетов, запросов и форм. Генератор форм – интерактивный инструмент, предназначенный для быстрого создания шаблонов ввода и отображения данных в экранных формах. Позволяет пользователю определить внешний вид формы, ее содержимое, место положения на экране. Некоторые ГФ позволяют выполнять проверку вводимых значений, а также создавать вычисляемые атрибуты с использованием арифметических операторов или агрегирующих функций. ГО предназначен для создания отчетов на основе имеющейся информации. Позволяют определить внешний вид печатаемого документа. В ССП как правило имеются ГФ и ГО. Генераторы приложений позволяют непосредственно создать каркас программы по описанию ее структуры, могут иметь в своем составе набор внутренних функций для выполнения типовых действий, разработчик указывает какие из требуемых свойств должно иметь разрабатываемое приложение, а генератор создает соответствующий код на основе имеющихся вызовов. Примером может служить мастер создания проекта в Vusual C, когда в зависимости от выбранного типа приложения генерируется свой исходный код приложения. Полученное описание программы транслируется в обычный текст и файл ресурсов, которые могут быть скорректированы профессиональным программистом для внедрения необходимой функциональности приложения.

Принципы технологий на основе Java, HTML.

Для выполнения приложений в среде Internet традиционные подходы на основе компиляции не могут быть применимы, поскольку Inet характеризуется большим разнообразием платформ: как аппаратных архитектур, так и составом используемых ОС. Поэтому для интернет-технологий основой стало использование интерпретаторов. Рассмотрим **HTML** (HyperText Markup Language – язык разметки гипертекста). Браузер HTML, являющийся по сути своей интерпретатором, позволяет отображать HTML страницы в соответствии с заданным описанием. В случае статических HTML страниц каждая страница представляет отдельный файл, сервер передает его клиенту, и браузер на стороне клиента интерпретирует его. В случае динамических страниц сервер сначала формирует текст страницы, после чего отправляет его клиенту. В результате возможна реакция на введенные пользователем данные или настройка внешнего вида страницы. В простейшем случае для подготовки динамических страниц на сервере используется специальный исполняемый файл. На его вход поступают некоторые параметры, переданные клиентом (например, введенные в форме данные), а на выходе его формируется HTML текст. Технологиями подобного рода являются CGI – common gateway interface –интерфейс общих шлюзов и ISAPI – internet server aplication programming interface – интерфейс программирования приложений интернет-сервера. Интерфейс CGI определяет способ взаимодействия сценариев с Web-серверами. Сервер создает несколько переменных среды, сценарий получает их и читает стандартный входной поток stdin. Затем он выполняет обработку данных и передает их в стандартный выходной поток stdout. Сценарий отвечает за отправку клиенту заголовка, помещаемого перед основной частью выходного потока. Сценарии CGI можно создавать практически на любом языке, позволяющем читать и писать значения переменных среды ОС и выполнять чтение-запись с помощью стандартных потоков ввода-вывода. Для UNIX это PHP, Java, C, Forth и др., для

Windows – VisualBasic, C/C++, Delphi и т.д. Отличие CGI от ISAPI в том, что в первом случае создаются отдельные приложения в виде полноценных программ, а во втором – динамически подключаемые к серверу библиотеки. ССП позволяют создавать приложения и библиотеки для этих стандартов. Недостаток этих методов в том, что при изменении содержимого HTML страницы требуется изменение и перекомпиляция CGI или ISAPI приложения. Для генерации HTML страниц можно использовать не скомпилированное приложение, а исходный текст программы и интерпретатор. В таких технологиях используется Perl (Practical Extraction and Report Language), PHP (personal home pages), ASP (active server pages). PHP или ASP скрипты представляют собой вставки в обычные HTML страницы, при необходимости сгенерировать страницу для клиента, сервер находит скрипты, интерпретирует их, формируя на выходе код HTML, который и вставляется в страницу на место сценария.

Java в отличие от HTML является полноценным языком программирования. Однако он не компилируется непосредственно в машинные коды. Платформонезависимость Java достигается тем, что исходный код программы компилируется в промежуточный язык кодов для JVM, называемый Java-апплетом. Второй частью является собственно JVM - платформозависимый интерпретатор. Компилятор Java-апплета находится на сервере, а клиенты должны иметь JVM-интерпретатор. Такой подход снижает трафик в сети, поскольку не передаются текстовые описания команд. Основной недостаток – необходимость JVM-интерпретатора на стороне клиента. Основное достоинство Java – независимость исходного кода от архитектуры системы, в т.ч. и от ОС. Java Script – подмножество языка Java, встраиваемое в виде скрипта в текст HTML страниц. В этом случае роль JVM исполняет браузер. Java скрипты могут исполняться как на стороне сервера, так и на стороне клиента. Аналогичной технологией является VBScript – процедурный интерпретируемый язык сценариев на основе Visual Basic. Аналогична по организации технология Flash, однако она не является полноценным языком программирования и предназначена в основном для анимации.

Выводы. ССП развились в достаточно сложные системы, в большинстве своем позволяющие интерактивно на основе элементов языков 4GL строить основу приложения. Наблюдается широкое взаимопроникновение и интеграция различных технологий и языков, ССП прибегают как к процедурным, так и непроцедурным языкам, позволяя использовать их совместно. Ядром СП является транслятор, преобразующий исходный код в объектный. Большинство ССП позволяют сочетать ряд различных языков, например, использовать Asm вставки или вставки на Java. Математический аппарат построения трансляторов хорошо разработан, что позволяет быстро строить требуемые компиляторы, в т.ч и компиляторы компиляторов.

Формальные языки и грамматики

Понятие алфавита. Цепочки символов и формальные языки. Конкатенация, обращение, подстановка, итерация цепочек. Замыкание алфавита. Способы задания языков. Лексика, синтаксис и семантика языка. Особенности языков программирования. Понятие грамматики. Нетерминальный и терминальный словари грамматики, аксиома грамматики, продукции. Вывод цепочек, сентенциальная форма. Запись правил грамматик. Форма Бэкуса-Наура. Синтаксические диаграммы. Принцип рекурсии. Общая схема распознавателя. Конфигурация распознавателя. Виды распознавателей. Задача разбора

Понятие алфавита. Цепочки символов и формальные языки

В общем случае язык состоит из знаковой системы (множество допустимых последовательностей знаков), множества смыслов этой системы, соответствия между последовательностями знаков и смыслами. В общем случае знаками могут быть буквы алфавита, математические обозначения, звуки и т.д. **Символ** (буква) – это простой неделимый знак. **Алфавит** – это счетное множество допустимых символов языка. Обозначим алфавит символом **A**. Алфавит не обязательно должен быть конечным множеством, хотя все существующие языки строятся на основе конечных алфавитов. **Цепочка символов** (строка) – это произвольная упорядоченная конечная последовательность символов алфавита. Цепочки будем обозначать греческими буквами **α, β, γ** и т.д. Произвольность означает, что в цепочку может входить любая допустимая последовательность символов, она не обязательно имеет смысл. Упорядоченность означает, что цепочка имеет определенный состав входящих в нее символов, их количество и порядок следования. Цепочки символов α и β равны **α=β**, если они имеют один и тот же состав символов, их количество и порядок следования. Количество символов в цепочке называется ее **длиной** и обозначается как **|α|**. Если **α=β => |α|=|β|**. **Конкатенацией** Θ цепочек α, β называется цепочка **γ=αΘβ=αβ**. => |γ|=|α|+|β|. Цепочка α является префиксом, β - суффиксом строки γ. Конкатенация не транзитивна: **∃α,β: αβ≠βα.** Конкатенация ассоциативна: **γ(αβ)=(γα)β**. Любую цепочку символов можно представить как конкатенацию составляющих ее частей, причем может существовать большое число вариантов такой разбивки. Цепочка ω называется **подцепочкой** γ, если γ=αωβ. **Заменой** (подстановкой) цепочки γ=αωβ называется новая цепочка **γ'=αφβ**. *Пример*. **Обращение** цепочки α^R– это запись символов цепочки α в обратном порядке. $\alpha^{RR}=\alpha$. $\alpha^R=\alpha\alpha^{R-1}=\alpha^{R-1}\alpha$. **∀α,β: $(\alpha\beta)^R=\beta^R\alpha^R$**. **Итерация** цепочки α^n - это конкатенация цепочки α самой с собой n раз, **n∈N, n≥0**. **Пустая цепочка** символов **λ (ε)** не содержит ни одного символа. $\alpha^0=\lambda$. Для пустой цепочки справедливо: **|λ|=0**. **∀α: λα=αλ=α**. $\lambda^R=\lambda$. **∀n≥0: $\lambda^n=\lambda$**. $\alpha^0=\lambda$. Цепочка символов α является **цепочкой над алфавитом A: α(A),** если для ∀ символа x∈α, x ∈ A. **Замыкание** алфавита A – это множество всех возможных цепочек из символов алфавита A (над алфавитом A): Если A={a,b,c}, то A^*={λ, a, b, c, aa, ab, ac, bb, ba, bc, cc, ca, cb, aaa, aab, aac,…}.

$\mathbf{A^*= A^0\cup A^1\cup A^2\cup \dots =}\bigcup_{n=0}^{\infty} A^n$ **,** где A^0={λ}, A^1=A. Множество непустых цепочек над алфавитом A определяется следующим образом: $\mathbf{A^+= A^1\cup A^2\cup \dots =}\bigcup_{n=1}^{\infty} A^n$. **$A^*= A^+\cup${λ}**.

Язык L над алфавитом A: **L(A)** это некоторое счетное подмножество цепочек конечной длины из множества всех цепочек алфавита A: **L(A)⊆A^***. Множество цепочек языка не обязательно должно быть конечным. Длина цепочки символов может быть сколь угодно большой. Все существующие языки подпадают под это определение. Всего языков над алфавитом A может быть сколь угодно много. Язык L(A) включает в себя язык L'(A): **L'(A) ⊆L(A)**, если **∀α∈L'(A): α∈L(A)**. Т.е. множество цепочек языка L'(A) является подмножеством множества цепочек языка L(A). Языки L(A) и L'(A) эквивалентны: **L'(A)=L(A)**, если **L'(A)⊆L(A) и L(A)⊆L'(A)**. Или **∀α∈L'(A): α∈L(A) и ∀β∈L(A): β∈L'(A)**. Для эквивалентных языков множества допустимых цепочек равны. Языки L(A) и L'(A) почти эквивалентны: **L'(A)≅L(A)**, если **L'(A)∪{λ}=L(A)∪{λ}**. Для них множества допустимых цепочек могут различаться только на пустую цепочку символов. Над языками могут выполняться все операции, допустимые для множеств. Особые случаи конкатенации с множеством, содержащим только пустую строку и с пустым множеством (это разные понятия !!!): L{λ}={λ}L=L; L∅=∅L=∅. Конкатенация произвольного числа цепочек формального языка L носит название **замыкания Клини**:

$\mathbf{L^*= L^0\cup L^1\cup L^2\cup \dots =}\bigcup_{n=0}^{\infty} L^n$ **,** где **L^0={λ}.** При n≥1 справедливо **$L^n=LL^{n-1}=L^{n-1}L$**. Такое замыкание означает ноль или более сцеплений языка L. **Позитивное замыкание** означает одно или более сцеплений языка L: $\mathbf{L^+=L^1\cup L^2\cup \dots =}\bigcup_{n=1}^{\infty} L^n$, или **$L^+=LL^*=L^*L$, $L^*=L^+\cup${λ}**.

Способы задания языков. Синтаксис и семантика языка.

В общем случае язык можно задать тремя способами:

- перечислением всех допустимых цепочек языка;
- указанием способа порождения цепочек языка (задание грамматики);
- определение метода распознавания цепочек языка.

Лексика – это совокупность слов (словарный запас) языка. **Лексема** (слово, лексическая единица) языка – это конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций. Например, для русского языка лексемами являются слова русского языка, знаки препинания и пробелы это разделители, не образующие лексем. Для алгебры лексемами являются числа, знаки операций, обозначения функций и неизвестных величин. Для ЯП – ключевые слова, идентификаторы, константы, метки, знаки операций. **Синтаксис** – набор правил, определяющий допустимые конструкции языка, т.е. задает набор цепочек символов, которые принадлежат языку. В виде строгого набора правил можно описать только формальные языки, для большинства ЯП набор заданных синтаксических правил нуждается в дополнительных пояснениях. Математический аппарат для изучения синтаксиса языков называется теорией формальных грамматик. **Семантика** –

это раздел языка, определяющий значение предложений языка, т.е. задает смысл всех допустимых цепочек языка. Как правило, не поддается формальному определению. Конкретная программа несет в себе некоторое воздействие на транслятор, т.н. прагматизм. Синтаксис, семантика и прагматизм в совокупности образуют **семиотику** языка. ЯП занимают промежуточное положение между формальными и естественными языками. Как и формальные языки, они имеют строгие синтаксические правила. Из естественных языков ЯП позаимствовали ряд слов, выражающих ключевые слова. Для задания ЯП необходимо:

- определить множество допустимых символов языка;
- определить множество правильных программ языка;
- задать смысл для каждой правильной программы.

Первый вопрос решается легко, при задании алфавита языка. Для ЯП это как правило набор символов, который можно ввести с клавиатуры. Второй вопрос решается в теории формальных языков только частично. Для ЯП существуют правила, определяющие синтаксис, но их недостаточно, чтобы строго определить все допустимые предложения ЯП. Семантические ограничения накладываются неформально, например, необходимость предварительного описания переменных, требование соответствия типов данных и т.д. Третий вопрос в принципе в теории формальных языков не решается.

Формальное определение грамматики.

Язык L над алфавитом A – это подмножество цепочек A^*. Это очень общее определение , не позволяющее выделять среди множества языков отдельные их классы. **Соотношения Туэ** – это правила, согласно которым любой цепочке $\alpha=\gamma\xi\delta$ из множества A^* ставится в соответствие цепочка $\beta=\gamma\eta\delta$ из того же множества. Эти соотношения приводят к ассоциативным исчислениям, но и они достаточно общи. Ограничения на них в виде введения односторонних правил привели к созданию формального математического аппарата – **формальным грамматикам**. Теория формальных грамматик занимается описанием, распознаванием и переработкой языков. Они позволяют ответить на ряд прикладных вопросов – могут ли языки из некоторого класса Z быть легко распознанными, принадлежит ли данный язык классу Z, существуют ли алгоритмы, определяющие принадлежность цепочки α языку L и т.д. Существует два основных способа описания отдельных классов языков: с помощью порождающей процедуры и с помощью распознающей процедуры. Первая из них задается с помощью конечного множества правил, называемых **грамматикой**. Распознающая процедура задается с помощью абстрактного распознающего устройства – автомата. При построении транслятора используются оба эти способа – грамматика как средство описания синтаксиса ЯП, а автомат как модель алгоритма распознавания предложений языка, который и кладется в основу транслятора. Т.е. вначале строится грамматика, а по ней строится алгоритм распознавания.

Грамматику можно описать, используя формальное описание грамматики, построенное на основе системы правил. **Правило** (продукция) – упорядоченная пара цепочек символов (α, β). Как правило их записывают в виде $\alpha\to\beta$ - α порождает β. Грамматика ЯП содержит правила 2 типов – поддающиеся формальному описанию, такие как определяющие синтаксические конструкции языка, и неформально описываемые, определяющие семантические ограничения.

Язык, заданный грамматикой G, обозначается как L(G). Грамматики G и G' называются **эквивалентными**, если определяют один и тот же язык: **L(G) = L(G').** Грамматики почти эквивалентны, если заданные им языки различаются не более чем на пустую цепочку символов: $\mathbf{L(G)\cup\{\lambda\}=L(G')\cup\{\lambda\}}$.

Формально грамматика определяется как четверка **G(T, N, P, S),** где T - конечное непустое множество терминальных символов языка (терминальный или основной словарь грамматики G), N – конечное непустое множество нетерминальных символов (нетерминальный или вспомогательный словарь), P – конечное множество правил (продукций) грамматики вида $\alpha\to\beta$, где $\alpha\in(N\cup T)^+$, $\beta\in(N\cup T)^*$, S – начальный символ (**аксиома**) грамматики, $S\in N$, обозначает главный нетерминал, цель грамматики G. Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: $N\cap T=\varnothing$. Т.е. символ не может быть терминальным и нетерминальным одновременно. Начальный символ – всегда нетерминальный. Множество $N\cup T$ называют **полным алфавитом** (объединенным словарем) грамматики G. Множество терминальных символов включает в себя символы, входящие в алфавит языка, порождаемого грамматикой. Нетерминальные символы определяют слова, понятия, конструкции языка. Правила грамматики обычно строятся так, чтобы в левой части правил был хотя бы один нетерминальный символ. Элементы нетерминального словаря будем обозначать прописными латинским буквами **A,B,C,**… терминальные символы – строчными – **a,b,c**…. Произвольные цепочки – греческими α,β,γ,…

Цепочка ω' **непосредственно выводима** из цепочки ω в грамматике G ($\omega\Rightarrow\omega'$), если $\boldsymbol{\omega=\xi_1\varphi\xi_2}$ $\boldsymbol{\omega'=\xi_1\psi\xi_2}$ **и** $\boldsymbol{\exists(\varphi\to\psi)\ni P}$. Цепочка ω' **выводима** из цепочки ω в грамматике G **($\omega\Rightarrow^*\omega'$)**, если $\exists$ цепочка последовательностей $\boldsymbol{\omega=\omega_0,\omega_1,\ldots,\omega_n=\omega'}$**:** $\boldsymbol{\omega_{i+1}\Rightarrow\omega_i}$**,** $\boldsymbol{i=0,1,\ldots,n-1}$, либо $\omega=\omega'$. Последовательность цепочек $\omega=\omega_0,\omega_1,\ldots,\omega_n$ называется **выводом** цепочки ω_n из цепочки ω_0 в грамматике G. Здесь ω_n может быть выведена и за ноль шагов ($\omega=\omega'$). Для уточнения, что вывод цепочки осуществляется не менее чем за один шаг, обозначается $\boldsymbol{\omega\Rightarrow^+\omega'}$**.** Отношение $\Rightarrow^*$ называется **транзитивным замыканием**. Каждая строка, которую можно вывести из аксиомы – **сентенциальная форма**. Сентенциальная форма, состоящая только из терминалов, представляет собой строку языка.

Язык L, порождаемый грамматикой G – это множество всех цепочек терминальных символов, выводимых из аксиомы грамматики: $\mathbf{L(G)=\{\chi \mid S\Rightarrow^*\chi,\ \chi\in T^*\}}$

Пример. G=<N,T,P,S>, N={S,D,C}, T={>,<,=,!}, P={C→D=, C→S, D→!, D→=, D→S, S→<, S→>}, S={C}.

Грамматике не присуща детерминированность – т.е. конкретный порядок подстановки правил (алгоритм) является произвольным, не определенным строго. Это обеспечивает компактность. Алгоритм можно зафиксировать различными способами, поэтому формальная грамматика потенциально задает множество алгоритмов порождения языка.

Форма Бэкуса-Наура

Форма БН явилась исторически первой для записи в сжатом виде привил грамматики. В ней точно так же используются основной и вспомогательный словари. В терминологии ФБН они называются соответственно как основные символы языка и металингвистические переменные. Каждая металингвистическая формула (форма) в левой части находится металингвистическая переменная, обозначающая соответствующую конструкцию, в правой части указывается один или несколько спосо-

бов построения конструкции. Варианты в правой части разделяются символом | , обозначающем ИЛИ. Правая и левая части разделяются связкой **::=**, что означает определяется как. Металингвистические переменные обозначаются словами, поясняющими смысл описываемой конструкции и заключаются в угловые скобки **<>**. Пример для вышеописанной грамматики:
<compare>::=<double>= | <single> <double>::= ! | = | <single> <single>::= < | >
Возможность описания бесконечного множества цепочек языка с помощью конечного набора правил достигается за счет **рекурсий**. Рекурсия может быть **явной**, когда символ определяется сам через себя (пример) и **неявной** (косвенной), когда тоже самой происходит через цепочку правил (пример). Чтобы рекурсия не была бесконечной, должны существовать и другие правила, определяющие тот же символ.
Формы БН не позволяют описывать контекстные зависимости ЯП. Например, ограничение вида: идентификатор не может быть описан более одного раза в одном и том же блоке. В этом случае используются другие, **метасемантические** языки. Однако, как правило, их ядром является форма БН.
Правила грамматики могут быть записаны и в иной форме. Например, для описания синтаксиса КОБОЛа и ПЛ/1 использовалось расширение формы БН:

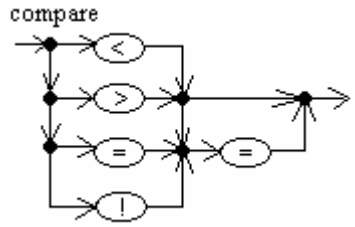
1. Необязательный элемент внутри правила заключается в квадратные скобки []
2. Альтернативные варианты обозначаются вертикальным списком, заключенным в фигурные скобки {}
3. Необязательные альтернативные варианты обозначаются вертикальным списком, заключенным в квадратные скобки
4. Повторяющийся элемент обозначается списком из одного элемента (если необходимо, заключенного в квадратные или фигурные скобки), за которым следует многоточие …
5. Обязательные ключевые слова подчеркиваются, а необязательные нет.

Пример (два разных варианта описания):

$$\text{<compare>::=}\left\{\begin{array}{c}< double >=\\ < \sin gle >\end{array}\right\} \quad \text{<double>::=2}\left\{\begin{array}{c}!\\ =\\ < \sin gle >\end{array}\right\} \quad \text{<single>::=}\left\{\begin{array}{c}<\\ >\end{array}\right\} \quad \text{<compare>::=}\left\{\begin{array}{l}\left\{\begin{array}{c}<\\ >\end{array}\right\}[=]\\ \left\{\begin{array}{c}!\\ =\end{array}\right\}=\end{array}\right\}$$

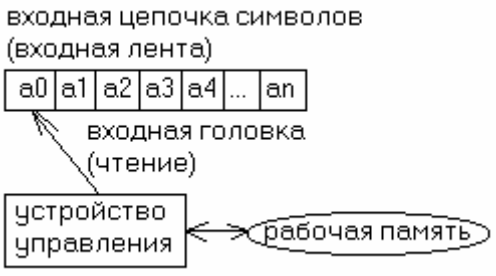
Существуют и другие нотации с использованием метасимволов. Еще один способ описания – в виде **синтаксических диаграмм**. Впервые была использована при описании языка Pascal. Эта запись доступна для тех грамматик, в которых в левой части присутствует не более одного символа. Этого достаточно для описания существующих ЯП. В этой форме каждому нетерминалу соответствует диаграмма в виде направленного графа Граф имеет несколько типов вершин:

- точка входа. На диаграмме не обозначается, из нее просто начинается входная дуга графа
- нетерминал. На диаграмме обозначается прямоугольником, в котором написано его обозначение
- цепочка терминалов. Обозначается овалом, внутри которого записана цепочка
- узловая точка. Обозначается точкой или закрашенным кружком
- точка выхода. Никак не обозначается, в нее просто входит выходная дуга графа.



Каждая диаграмма имеет только одну точку входа и одну точку выхода. Вершин остальных типов может быть сколько угодно. Вершины соединяются направленными дугами. Из входной точки дуги только выходят, в выходную – только входят. Остальные вершины должны иметь как минимум один вход и один выход. (пример)

Общая схема распознавателя



Распознаватель – это специальный автомат, который позволяет определить принадлежность цепочки символов некоторому языку.
Входная лента представляет собой последовательность ячеек, каждая из которых содержит один символ некоторого конечного входного алфавита. Входная головка в каждый момент времени видит только одну ячейку. За такт головка может сдвигаться на один символ вправо или влево. УУ задает конечное множество состояний распознавателя и определяет переходы между ними в зависимости от текущего символа ленты и содержимого рабочей памяти. УУ имеет конечную память для хранения своего состояния и некоторой промежуточной информации. Объем памяти не ограничивается, у некоторых типов распознавателей может отсутствовать. Может быть организована в виде стека. Распознаватель работает с символами своего алфавита. Он конечен и включает в себя все допустимые символы входных цепочек и некоторый дополнительный алфавит символов, которые могут обрабатываться УУ и храниться в рабочей памяти. Распознаватель может выполнять следующие действия:

- чтение очередного символа с ленты
- анализ входного символа, содержимого памяти и текущего состояния
- перемещение при необходимости входной головки
- изменение содержимого памяти
- изменение состояния распознавателя.

Поведение распознавателя отслеживается по его **конфигурациям**. Конфигурация распознавателя определяется:

- состоянием УУ
- содержимым входной ленты и положением головки
- содержимым рабочей памяти.

Конфигурация называется **начальной**, если УУ находится в заданном начальном состоянии, входная головка установлена на первый символ последовательности и рабочая память имеет заранее установленное начальное содержимое. Конфигурация

заключительная, если УУ находится в одном из заданных заключительных состояний, входная головка находится за концом исходной цепочки.

Распознаватель допускает входную цепочку ω, если находясь в начальной конфигурации и получив на вход эту цепочку он может проделать последовательность шагов, заканчивающуюся одной из его заключительных конфигураций.. Язык, определяемый распознавателем – это множество всех цепочек, которые допускает распознаватель.

Виды распознавателей

По типу считывающего устройства разделяются на **односторонние** и **двусторонние** (может ли считывающая головка двигаться в обе стороны). Распознаватели разделяются на **левосторонние** и **правосторонние** – соответственно с какой стороны начинается чтение входной цепочки. По виду УУ распознаватели разделяются на **детерминированные** и **недетерминированные**. Для детерминированных распознавателей для каждой допустимой конфигурации, которая может возникнуть на некотором шаге, существует единственно возможная конфигурация, в которую распознаватель перейдет на следующем шаге. Для недетерминированного существует несколько допустимых конфигураций в которые распознаватель может перейти из текущей конфигурации. По виду рабочей памяти разделяются на:

- Р. без внешней памяти
- Р. с ограниченной внешней памятью
- Р. с неограниченной внешней памятью

В случае ограниченной памяти ее объем зависит от длины входной цепочки символов. Зависимость может быть произвольной – линейной, полиномиальной, экспоненциальной и др. Может быть указан способ организации памяти, например, стековая. Для Р. с неограниченным объемом памяти предполагается произвольный метод доступа.

Для случая преобразования одного языка в другой можно вести речь о **преобразователях**. Преобразователь пишет строку символов на выходную ленту. Если входная строка переводит преобразователь из начальной конфигурации в заключительную, строка на выходной ленте считается переводом входной строки.

Задача разбора.

Для ЯП важно не только уметь построить текст программы на данном языке, но и определить принадлежность входного текста к данному языку. Именно эту задачу наряду с задачей построения эквивалентной выходной цепочки решают компиляторы. Для описания языка используется грамматика, для распознавания – распознаватель. Фактически это два независимых метода определения языков. На практике возникает задача связывания их между собой.

Разработчики компилятора имеют дело с уже описанным языком программирования, грамматика для которого известна. Таким образом задача разбора в общем виде состоит в построении распознавателя для некоторого языка на основе уже имеющейся грамматики. Заданная грамматика и распознаватель должны быть эквивалентны, т.е. определять один и тот же язык. В общем виде эта задача решается не для всех языков. Для синтаксических конструкций ЯП задача разбора решаема. И найдены формальные методы ее решения. Язык L(G) называется **распознаваемым**, если существует алгоритм, который за конечное число шагов позволяет определить., принадлежит ли произвольная цепочка над основным словарем грамматики α языку L(G). Если при этом число шагов алгоритма зависит от длины цепочки и может быть оценено до начала его выполнения, язык L(G) называется **легко распознаваемым**.

Классификация грамматик

Классификация грамматик по Хомскому. Контекстно-зависимые и контекстно-свободные грамматики. Неукорачивающие грамматики. Регулярные грамматики и языки. Праволинейные и леволинейные грамматики. Автоматные грамматики. Классификация языков. Классификация распознавателей. Машина Тьюринга.

Классификация грамматик по Хомскому.

Американский лингвист Ноам Хомски предложил классифицировать грамматики по структуре их правил. Если все без исключения правила грамматики удовлетворяют некоторой заданной структуре, то грамматика принадлежит заданному типу. По классификации Хомского грамматики делятся на 4 типа:

Тип 0. Грамматики с фразовой структурой. На структуру правил не накладывается никаких ограничений: Для грамматики вида G(T,N,P,S), V=N∪T правила имеют вид: $\alpha\to\beta$, $\alpha\in V^+$, $\beta\in V^*$. Это самый общий тип грамматик. Все формальные грамматики принадлежат этому классу. Грамматики, которые относятся к типу 0, и не могут быть отнесены к другим типам, имеют самую сложную структуру. Практического применения этот класс грамматик не имеет.

Тип 1. Контекстно-зависимые (КЗ) грамматики. Для грамматики вида G(T,N,P,S), V=N∪T правила имеют вид: $\alpha_1 A\alpha_2\to\alpha_1\beta\alpha_2$, α_1, $\alpha_2\in V^*$, $A\in N$, $\beta\in V^+$. Структура этих грамматик такова, что один и тот же нетерминальный символ может быть заменен на ту или иную цепочку символов в зависимости от контекста, в котором он встречается. Цепочки α_1, α_2 обозначают контекст, они могут быть пустой цепочкой в общем случае.

Разновидность грамматик этого класса – **неукорачивающие** грамматики. Для грамматики вида G(T,N,P,S), V=N∪T правила имеют вид: $\alpha\to\beta$, α, $\beta\in V^+$, $|\beta|\geq|\alpha|$. Любая цепочка символов в этом случае заменяется на цепочку не меньшей длины.

Доказано, что эти два класса грамматик эквивалентны. Для любого языка, заданного КЗ грамматикой можно построить неукорачивающую грамматику, задающую эквивалентный язык, и наоборот. При построении компиляторов эти грамматики не применяются, поскольку синтаксические конструкции ЯП имеют более простую структуру и могут быть построены с помощью грамматик других типов.

Тип 2. Контекстно-свободные (КС) грамматики. Для грамматики вида G(T,N,P,S), V=N∪T правила имеют вид: $A\to\beta$, $A\in N$, $\beta\in V^+$. КС грамматики имеют в правой части как минимум один символ. Они являются неукорачивающими. Фактически получаются из условий грамматик класса 1, у которых $\alpha_1=\alpha_2=\lambda\in V^*$, т.е. отсутствует контекст.

Почти эквивалентный им класс – **укорачивающие** КС грамматики. Для грамматики вида G(T,N,P,S), V=N∪T правила имеют вид: $A\to\beta$, $A\in N$, $\beta\in V^*$.

Внутри класса КС грамматик выделяют множество различных классов. Синтаксис большинства ЯП основан на КС грамматиках. Их используют для построения синтаксического анализатора компиляторов.

Тип 3. Регулярные (автоматные) грамматики. К ним относятся два эквивалентных класса грамматик: леволинейные и праволинейные. **Леволинейные** грамматики G(T,N,P,S), V=N∪T правила имеют вид: $A\to B\gamma$, $A\to\gamma$, $A,B\in N$, $\gamma\in T^*$. Т.е. при выводе нетерминальный символ если и остается, то слева. **Праволинейные** грамматики G(T,N,P,S), V=N∪T правила имеют вид: $A\to\gamma B$, $A\to\gamma$, $A,B\in N$, $\gamma\in T^*$. Леволинейные и праволинейные грамматики эквивалентны. Для любого языка, заданного леволинейной грамматикой, может быть построена праволинейная, определяющая эквивалентный язык, и наоборот. Чаще используются леволинейные грамматики, поскольку их построение отвечает порядку построения предложений ЯП. Используются при описании простейших конструкций ЯП: идентификаторов, констант, строк, комментариев и т.д. На их основе строятся лексические анализаторы компиляторов.

Среди регулярных грамматик выделяется класс **автоматных грамматик**, которые так же могут быть праволинейными и леволинейными: Леволинейные: G(T,N,P,S), V=N∪T правила имеют вид: $\mathbf{A\to Bt,\ A\to t,\ A,B\in N,\ t\in T,}$ праволинейные: G(T,N,P,S), V=N∪T правила имеют вид: $\mathbf{A\to tB,\ A\to t,\ A,B\in N,\ t\in T}$. Основное отличие в том, что где в правилах регулярных грамматик может присутствовать цепочка символов, в автоматных может присутствовать только один терминальный символ. Классы терминальных и автоматных грамматик почти эквивалентны. В автоматных грамматиках допускается дополнительное правило вида $S\to\lambda$, где S - аксиома. При этом S не должен встречаться в правых частях других правил грамматики. В этом случае язык, заданный автоматной грамматикой, может включать в себя пустую цепочку, и такая автоматная грамматика полностью эквивалентна регулярной.

В общем случае одна и та же грамматика может быть отнесена к нескольким классам. Для классификации всегда выбирается тип с максимальным номером. Сложность грамматики обратно пропорциональна их типу. Класс 0 допускает самые сложные грамматики, класс 3 – самые простые.

Классификация языков.

Языки классифицируются в соответствии с типом грамматики, с помощью которых они задаются. Поскольку один и тот же язык может быть задан с помощью большого количества грамматик, относящихся к разным типам, то из всех этих грамматик берется та, которая имеет максимальный номер класса. Например, если язык L задается грамматиками G1 (класс 1) и G2 (класс 2), то он относится к типу 2.

Тип 0. Языки с фразовой структурой. Самые сложные языки, задаваемые грамматиками типа 0. К этому типу относятся все естественные языки. Структура и значение фразы ЕЯ зависит не только от контекста данной фразы, но и от содержания текста, где встречается эта фраза. Слово может иметь разное значение, может играть разные роли в предложении. Поэтому и проблематично построить соответствующий компилятор (в т.ч. переводчик на другой язык).

Тип 1. Контекстно-зависимые языки (КЗ). Языки и грамматики этого типа используются при анализе и переводе текстов на естественных языках. В компиляторах КЗ-языки не используются.

Тип 2. Контекстно-свободные языки (КС). Эти языки лежат в основе большинства современных ЯП.

Тип 3. Регулярные языки. Самый простой тип языков. Эти языки лежат в основе простейших конструкций ЯП (идентификаторы, константы), на их основе строятся многие мнемокоды машинных команд (ассемблеры) и т.д.

Классификация распознавателей.

Для каждого из типов языков существует свой тип распознавателя.

Для языков класса 0 необходим распознаватель, соответствующий машине Тьюринга, т.е. недетерминированный двусторонний автомат с неограниченной внешней памятью. Поэтому для такого языка невозможно построить компилятор, который гарантированно выполнял бы разбор предложений за ограниченное время на основе ограниченных вычислительных ресурсов.

Для КЗ языков распознаватели представляют собой двусторонние недетерминированные автоматы с линейно ограниченной внешней памятью. Для этого типа время распознавания в общем случае экспоненциально зависит от длины исходной цепочки символов.

Для КС языков распознаватели представляют собой односторонние недетерминированные автоматы с магазинной (стековой) внешней памятью (микропрограммные автоматы). Время на распознавание в общем случае полиномиально зависит от длины входной цепочки. В зависимости от класса языка это либо кубическая либо квадратичная зависимость. Для многих языков этого класса такая зависимость является линейной.

Здесь можно выделить подкласс детерминированных автоматов с магазинной памятью. Для соответствующих языков имеется алгоритм работы распознавателя с квадратичной сложностью. Для построения компиляторов интерес представляют линейные распознаватели (время работы линейно зависит от длины входной цепочки). Синтаксические конструкции большинства ЯП могут быть отнесены к соответствующему классу.

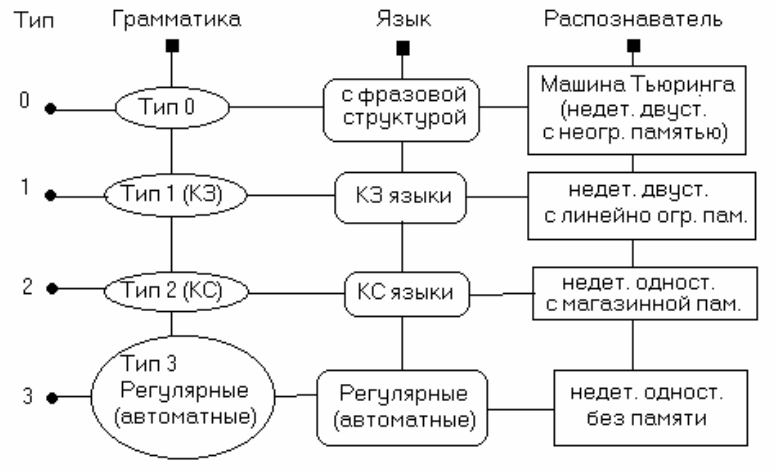
Для регулярных языков распознаватель представляет собой односторонний недетерминированный автомат без внешней памяти (конечный автомат). Время распознавания линейно зависит от длины входной цепочки символов. В компиляторах конечные автоматы используются для выделения в исходном коде лексем, что позволяет сократить объем входной информации для синтаксического анализатора. Конечные автоматы находят применение не только в компиляторах. Многие командные процессоры функционируют на их основе.

Пример. $G=<N,T,P,S>$, $N=\{S,D,C\}$, $T=\{>,<,=,!\}$, $P=\{C\to D=, C\to S, D\to !, D\to =, D\to S, S\to <, S\to >\}$, $S=\{C\}$.
Для типа 1 (КЗ): $\alpha_1 A\alpha_2\to\alpha_1\beta\alpha_2$, α_1, $\alpha_2\in V^*$, $A\in N$, $\beta\in V^+$. Смотрим. $\alpha_1=\alpha_2=\lambda\in V^*$, тогда $A=\{C\}\in N$, $\beta=\{D=\}\in V^+=(N\cup T)^+$. Все условия выполняются. Аналогично остальные продукции. Грамматика G соответствует классу 1.
Для неукорачивающих. $\alpha\to\beta$, α, $\beta\in V^+$, $|\beta|\geq|\alpha|$. Смотрим. $\{C\}\in V^+$,$\{D=\}\in V^+$, $|D=|=2 > |C|=1$. Соответствует.
Для типа 2 (КС): $A\to\beta$, $A\in N$, $\beta\in V^+$. Смотрим. $A=\{C\}\in N$, $\beta=\{D=\}\in V^+=(N\cup T)^+$. Соответствует.
Для типа 3 (регулярные). Леволинейные $A\to B\gamma$, $A\to\gamma$, $A,B\in N$, $\gamma\in T^*$. Праволинейные $A\to\gamma B$, $A\to\gamma$, $A,B\in N$, $\gamma\in T^*$. Смотрим. $A=\{C\}\in N$, $B=\{D\}\in N$, $\gamma=\{=\}\in T^*$. Леволинейная.
Для регулярных. $A\to Ba$, $A\to a$, $A,B\in N$, $a\in T$. Смотрим. $A=\{C\}\in N$, $B=\{D\}\in N$, $a=\{=\}\in T$. Доходим до продукции $C\to S$. Здесь: $A=\{C\}\in N$, $B=\{S\}\in N$, $a=\{\lambda\}\notin T$. Не автоматная.

Теорема 1. Язык L(G), порождаемый неукорачивающей грамматикой G, легко распознаваем.

Теорема 2. Если язык L(G) регулярный, то он контекстно свободный. Если язык L(G) контекстно свободный, то язык $L(G)\backslash\lambda$ контекстно-зависимый. Если язык L(G) контекстно-зависимый, то он язык класса 0.

Иерархия языков, грамматик и автоматов:



Лексический анализ и регулярные грамматики

Принципы построения лексических анализаторов. Регулярные множества и регулярные выражения. Свойства регулярных языков. Способы задания регулярных языков. Свойства регулярных языков. Теорема Клини. Лемма о разрастании языка. Преобразование регулярной грамматики к автоматному виду. Преобразование регулярной грамматики к регулярному выражению

Лексический анализатор –часть компилятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка. Это необязательная часть компилятора, поскольку все его функции могут быть выполнены на этапе синтаксического анализа. Однако практически все компиляторы имеют в своем составе лексический анализатор по следующим причинам:

- лексический анализатор структурирует поступающий исходный текст программы и устраняет избыточную, ненужную информацию, что упрощает структуру синтаксического анализатора;
- для выделения и разбора лексем возможно использовать простую, эффективную и теоретически хорошо проработанную технику анализа, тогда как на этапе синтаксического анализа используются более сложные алгоритмы разбора;
- лексический анализатор позволяет отстранить синтаксический анализатор от работы с исходным кодом, и при модификации лексики входного языка позволяет достаточно быстро перенастроить компилятор заменив лексический анализатор и не трогая синтаксический А.

Какие конкретно функции выполняет лексический анализатор и какие типы лексем он должен выделять, решают разработчики компилятора. Как правило это: устранение комментариев, незначащих пробелов и иных незначащих символов, выделение лексем следующих типов: идентификаторы, константы, ключевые слова, знаки операций и разделители.

Лексический анализ – это процесс предварительной обработки исходной программы, на котором основные лексические единицы программы – лексемы – приводятся к единому формату и заменяются условными кодами или ссылками на соответствующие таблицы.

Результат работы ЛА – поток образов лексем-дескрипторов и таблицы, в которых хранятся значения выделенных в программе лексем. Дескриптор – это пара вида (<тип лексемы>,<указатель>), где тип лексемы – это как правило числовой код класса лексемы, а указатель – это либо начальный адрес области памяти, в которой хранится адрес этой лексемы, либо число, адресующее элемент таблицы, в которой хранится значение лексемы. В общем случае все выделяемые классы являются либо конечными – ключевые слова, разделители, - это классы фиксированных слов для данного ЯП, либо же бесконечными (или очень большими) – идентификаторы, константы, метки - это классы переменных слов для данного ЯП. Коды дескрипторов из конечных классов всегда одни и те же для данного компилятора независимо от исходной программы. Коды дескрипторов из бесконечных классов различны для разных программ. В процессе ЛА значения лексем из бесконечных классов помещаются в таблицы соответствующих классов. Числовые константы перед их помещением в таблицы могут переводиться из символьного во внутреннее машинное представление. Пример:

```
long factorial (long x)
{if(x==1) return x;
 return factorial(x-1)*x;}
```

таблицы конечных классов:
(внутренние таблицы ЛА)

01. Ключевые слова	
№ п/п	**Ключевое слово**
1	long
2	if
3	return
…	…

02. Разделители	
№ п/п	**Разделитель**
1	(
2	)
3	{
4	==
5	;
6	-
7	*
8	}
…	…

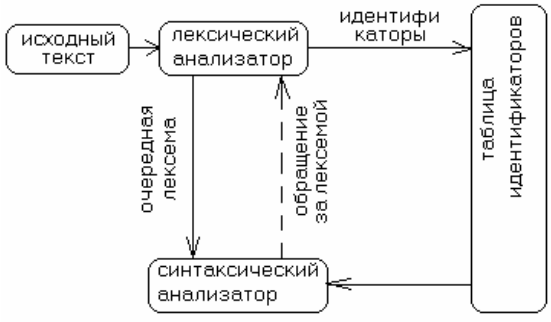
таблицы бесконечных классов
(формируемые таблицы ЛА)

03. Идентификаторы	
№ п/п	**Идентификатор**
1	factorial
2	x

04. Константы	
№ п/п	**Значение**
1	1

Выходной поток дескрипторов (таблица лексем):
(01,1)(03,1)(02,1)(01,1)(03,2)(02,2)
(02,3)(01,2)(02,1)(03,2)(02,4)(04,1)(02,2)(01,3)(03,2)(02,5)
(01,3)(03,1)(02,1)(03,2)(02,6)(04,1)(02,2)(02,7)(03,2)(02,5)(02,8)

Язык лексем может быть описан с помощью регулярных грамматик, распознавателями для которых являются конечные автоматы. КА определяет, принадлежит ли заданная входная цепочка символов языку, определяемому автоматом. Помимо этой задачи, ЛА должен уметь определять границы лексем, которые в тексте явно не указаны, и должен сохранять информацию об обнаруженной лексеме, т.е. запись найденной лексемы в таблицу лексем, поиск найденной лексемы в таблице идентификаторов и запись в нее. Определение границ лексем – не такая тривиальная задача, например, выражение x+++y можно трактовать двояко: (x++)+y либо x+(++y).



Выходной поток с ЛА поступает на вход СА. Имеется два варианта организации связи ЛА и СА:

- раздельный (последовательный), когда выходной поток ЛА формируется полностью и затем передается СА;

- нераздельный (параллельный), когда СА вызывает ЛА если ему требуется очередной образ лексемы.

Первый вариант проще в реализации и обеспечивает более высокую скорость компиляции. Однако в силу неоднозначности определения границ лексем из-за недостатка информации требуется использование параллельной организации. Она характерна для однопроходных трансляторов, но влечет за собой большие накладные расходы.

ЛА выделяет очередную лексему и передает ее СА. Тот проводит разбор очередной конструкции языка, подтверждает правильность выделенной лексемы и просит следующую. Если же разбор СА оказался ошибочен, то он сообщает ЛА о необходимости повторить выделение лексемы и дополнительно указывает, какую лексему следует ожидать. Так может быть перебрано несколько вариантов выделения лексемы, и только если ни один из них не подошел, генерируется ошибка. С целью упрощения СА и ЛА разработчики компиляторов сознательно отсекают некоторые вполне допустимые но трудноанализируемые цепочки, что в ряде случаев позволяет использовать последовательную схему взаимодействия.

Регулярные множества.

Регулярные грамматики служат для формального определения регулярных языков: язык называется **регулярным**, если он может быть порожден регулярной грамматикой. Регулярный язык L в некотором алфавите A представляет собой регулярное множество строк.

Пусть дан алфавит A и $P \in A^*$, $Q \in A^*$. Тогда:

Конкатенацией PQ называется **PQ={pq | ∀p∈P, ∀q∈Q};**

Итерацией P^* называется **P^*={p | ∀p∈P};**

Тогда для алфавита A регулярные множества определяются рекурсивно:

1. ∅ - РМ;
2. {λ} – РМ;
3. {a} – РМ **∀a∈A;**
4. если P и Q произвольные РМ, то множества $P \cup Q$, PQ, P^* - также являются РМ;
5. Ничто другое не является РМ.

Регулярные выражения.

РМ принято обозначать с помощью регулярных выражений. Это удобное средство формального определения регулярных языков. РМ – это множество цепочек, а РВ – это формула, схематично показывающая, как было построено соответствующее ей РМ с помощью допустимых операций. РВ вводятся следующим образом:

1. 0 – РВ, обозначающее ∅;
2. λ - РВ, обозначающее {λ}
3. a – РВ, обозначающее {a} **∀a∈A;**
4. если p и q – РВ, обозначающие РМ P и Q, то p+q (p|q), pq, p^* - РВ, обозначающие РМ $P \cup Q$, PQ, P^* соответственно.

Приоритеты операций: итерация, конкатенация (сцепление), альтернатива (или).

Два РВ α,β равны, α=β, если они обозначают одно и то же РМ. Каждое РВ обозначает только одно РМ, но для одного РМ может существовать сколь угодно много РВ. Пример РМ и РВ:

РМ	{01}	{0,1}	$\{1\}^*$	$\{0,1\}^*$	$\{0\}\{1\}^*$	$\{0,\{1\}^*\}$	$\{0,\{1\}\{0\}^*\}$	$\{0,1\}^*\{011\}$	$\{\{a\}^*\{b\},\{c\}\{a\}^*\}$
РВ	01	0|1	1^*	$(0\|1)^*$	01^*	$0\|1^*$	$(0\|(1(0^*)))=0\|10^*$	$(0\|1)^*011$	$a^*b\|ca^*$
Пр.	01	0,1	1,111	0,1,010	0,01,011	0,1,1111	0,1,10,10000	011010011	b,ab,aaab,c,ca,caaa

Свойства РВ.

№ п/п	Свойство	№ п/п	Свойство
1	$\alpha\|\beta=\beta\|\alpha$	10	$(\alpha^*)^*=\alpha^*$
2	$0^*=\lambda$	11	$\alpha\|\alpha=\alpha$
3	$\alpha\|(\beta\|\gamma)=(\alpha\|\beta)\|\gamma$	12	$\alpha\|0=\alpha$
4	$\alpha(\beta\gamma)=(\alpha\beta)\gamma$	13	$\alpha^*\alpha^*=\alpha^*$
5	$\alpha(\beta\|\gamma)=\alpha\beta\|\alpha\gamma$	14	$\alpha\alpha^*=\alpha^*\alpha$
6	$(\alpha\|\beta)\gamma=\alpha\gamma\|\beta\gamma$	15	$(\alpha^*\|\beta^*)^*=(\alpha^*\beta^*)^*=(\alpha\|\beta)^*$
7	$\alpha\lambda=\lambda\alpha=\alpha$	16	$(\alpha\beta)^*\alpha=\alpha(\beta\alpha)^*$
8	$0\alpha=\alpha0=0$	17	$(\alpha^*\beta)^*\alpha^*=(\alpha\|\beta)^*$
9	$\alpha^*=\alpha\|\alpha^*=\alpha\alpha^*\|\lambda$	18	$(\alpha^*\beta)^*=(\alpha\|\beta)^*\beta\|\lambda$

На основе РВ можно построить уравнения с регулярными коэффициентами. Простейшие УРК будут выглядеть:

$X=\alpha X|\beta$; $X=X\alpha|\beta$; где $\alpha,\beta \in A^*$, $X \notin A$.

Решением таких уравнений будут РМ. Т.е., если взять РМ, являющееся решением уравнения, обозначить его в виде РВ и подставить в уравнение, получим тождество. Решение 1 уравнения будет $\alpha^*\beta$:

$\alpha X|\beta=\alpha(\alpha^*\beta)|\beta=(\alpha\alpha^*)\beta|\beta=(\alpha\alpha^*)\beta|\lambda\beta=(\alpha\alpha^*|\lambda)\beta=\alpha^*\beta=X$, решением 2 уравнения будет $X=\beta\alpha^*$. Для уравнений вида $X=\alpha X$; $X=X\alpha$ решением будет $X=\alpha^*$, уравнение $X=\beta$ само по себе является решением.

Некоторые свойства регулярных языков

Множество называется замкнутым относительно некоторой операции, если в результате выполнения этой операции над любыми элементами, принадлежащими данному множеству, получается новый элемент, принадлежащий тому же множеству. Регулярные множества замкнуты относительно операций: пересечения, объединения, дополнения, итерации, конкатенации, гомоморфизма (изменения имен символов и подстановки цепочек вместо символов). Т.е. если L_1 и L_2 – регулярные языки, то замыкание Клини L_1^*, сцепление L_1L_2 и объединение $L_1 \cup L_2$ – тоже регулярные языки.

Теорема Клини. Каждому регулярному языку из A^* соответствует регулярное выражение над множеством A.

Лемма о разрастании языка. (лемма о накачке) Пусть L – регулярный язык: **$\forall \alpha \in L, \exists \delta,\beta,\gamma \in V^*, \exists p \in N>0$ | $\alpha=\delta\beta\gamma$, $|\alpha| \geq p$, $0<|\beta| \leq p$, $\alpha' = \delta\beta^i\gamma$, $\forall i \in N \geq 0$, $\alpha' \in L$.** В любой достаточно длинной строке регулярного языка всегда можно найти непустую

подстроку, повторение которой произвольное кол-во раз порождает новые строки того же языка. Если для данного языка выполняется эта лемма, то он регулярный, ели не выполняется, то он нерегулярный. Например, язык $L=\{0^m1^n \mid m,n\geq 0\}$ регулярный, а язык $L=\{0^n1^n \mid n\geq 1\}$ нерегулярный.

Доказано, что для регулярных языков разрешимы следующие проблемы:
- проблема эквивалентности. Даны два регулярных языка $L_1(A)$ и $L_2(A)$. Существует алгоритм проверки их на эквивалентность.
- Проблема принадлежности цепочки языку. Дан регулярный язык L(A) и цепочка $\alpha\in A^*$. Существует алгоритм проверки цепочки на принадлежность языку.
- Проблема пустоты языка. Дан регулярный язык L(A). Существует алгоритм проверки, является ли этот язык пустым, т.е. существует ли хотя бы одна цепочка $\alpha\neq\lambda$, $\alpha\in L(A)$.

Регулярные (праволинейные и леволинейные) грамматики , конечные автоматы и регулярные множества (регулярные выражения) – это три способа задания регулярных языков.

Преобразование регулярной грамматики к автоматному виду.
Имеется регулярная грамматика G(T,N,P,S), надо преобразовать ее в почти эквивалентную автоматную грамматику G'(T',N',P',S'). Будем использовать леволинейную грамматику. Алгоритм преобразования:
1) $\forall A\in N$, $N'=N'\cup\{A\}$. Т.е. все нетерминальные символы грамматики G переносятся в новую грамматику G'.
2) $\forall a\in T$, $T'=T'\cup\{a\}$. Т.е. все терминальные символы грамматики G переносятся в новую грамматику G'.
3) $\forall p_i\in P$: $A\to Ba_1$, либо $A\to a_1$, $A,B\in N$, $a_1\in T \Rightarrow P'=P'\cup\{p_i\}$. Т.е. правила этого вида переносятся в G' без изменений.
4) $\forall p_i\in P$: $A\to Ba_1a_2\ldots a_n$, $n>1$, $A,B\in N$, $a_k\in T$, $k=1,2,\ldots,n \Rightarrow$
$N'=N'\cup\{A_1,A_2,\ldots,A_{n-1}\}$, $P'=P'\cup\{A_k\to A_{k-1}a_k,\ k=1,2,\ldots,n$, где $A_n=A$, $A_0=B\}$. Добавляется n-1 нетерминал и n новых правил.
5) $\forall p_i\in P$: $A\to a_1a_2\ldots a_n$, $n>1$, $A\in N$, $a_k\in T$, $k=1,2,\ldots,n \Rightarrow$
$N'=N'\cup\{A_1,A_2,\ldots,A_{n-1}\}$, $P'=P'\cup\{A_k\to A_{k-1}a_k,\ k=1,2,\ldots,n$, где $A_n=A$, $A_0=\lambda\}$. Добавляется n-1 нетерминал и n новых правил.
6) $\forall p_i\in P$: $A\to B$, либо $A\to\lambda$, $A,B\in N \Rightarrow P'=P'\cup\{p_i\}$. Т.е. правила этого вида переносятся в G' без изменений.
7) $\forall p_i\in P'$: $A\to B$, и $\exists\ B\to C \mid B\to Ca \mid B\to a \mid B\to\lambda$, $\forall A,B,C\in N'$, $a\in T' \Rightarrow P'=P'\cup\{A\to C\}|\{A\to Ca\}|\{A\to a\}|\{A\to\lambda\}$ соответственно. $P'=P'\setminus\{A\to B\}$. Т.е. правило заменяется "синонимами" и удаляется.
8) $\forall p_i\in P'$: $A\to\lambda$, и $\exists\ B\to A \mid B\to Aa$, $\forall A,B,\in N'$, $A\neq S$, $a\in T' \Rightarrow P'=P'\cup\{B\to\lambda\}|\{B\to a\}$ соответственно. $P'=P'\setminus\{A\to\lambda\}$. Т.е. правило заменяется "синонимами" и удаляется.
9) Если шаги 7,8 были выполнены хотя бы для одного правила, то вернуться к шагу 7
10) $S'=S$. Т.е. аксиома сохраняется.

Если грамматика не содержит цепных правил вида $A\to B$, либо $A\to\lambda$ то шаги 6-9 не выполняются. Как правило, реальные регулярные грамматики не содержат цепных правил.

Пример. Грамматика описывает строковые выражения, соответствующие комментариям в С.
G({/,*,a,↓,←},{S,C,K},P,{S}), здесь ↓ - символ перевода строки (chr(13)), ← - символ возврата каретки (chr(10)), a – любой символ, кроме /,*,↓,←. Парой ↓,← в текстовых файлах отмечается конец строки.

P: $S\to C*/ \mid K\downarrow\leftarrow$
$C\to /* \mid C/ \mid C* \mid Ca \mid C\downarrow\leftarrow$
$K\to // \mid K/ \mid K* \mid Ka$

1. $N'=\{S,C,K\}$
2. $T'=\{/,*,a,\downarrow,\leftarrow\}$
3. $P'=P'\cup\{ C\to C/ \mid C* \mid Ca,\ K\to K/ \mid K* \mid Ka \}$
4. $A\to Ba_1a_2\ldots a_n, \Rightarrow N'=N'\cup\{A_1,A_2,\ldots,A_{n-1}\}$, $P'=P'\cup\{A_k\to A_{k-1}a_k,\ k=1,2,\ldots,n$, где $A_n=A$, $A_0=B\}$.
4.1. $S\to C*/ : N'=N'\cup\{A_1\}$, $\mathbf{A_k\to A_{k-1}a_k,\ A_n=A,\ A_0=B}$: $A_1\to A_0*$, $A_2\to A_1/ \Rightarrow A_1\to C*$, $S\to A_1/ \ \Rightarrow\ P'=P'\cup\{ A_1\to C*,\ S\to A_1/\}$.
4.2. $S\to K\downarrow\leftarrow : N'=N'\cup\{A_2\}$, $P'=P'\cup\{ A_2\to K\downarrow,\ S\to A_2\leftarrow\}$.
4.3. $C\to C\downarrow\leftarrow : N'=N'\cup\{A_3\}$, $P'=P'\cup\{ A_3\to C\downarrow,\ C\to A_3\leftarrow\}$.
5. $A\to a_1a_2\ldots a_n, \Rightarrow N'=N'\cup\{A_1,A_2,\ldots,A_{n-1}\}$, $P'=P'\cup\{A_k\to A_{k-1}a_k,\ k=1,2,\ldots,n$, где $A_n=A$, $A_0=\lambda\}$.
5.1. $C\to /* : N'=N'\cup\{A_4\}$, $\mathbf{A_k\to A_{k-1}a_k,\ A_n=A,\ A_0=\lambda}$: $A_1\to A_0/$, $A_2\to A_1* \Rightarrow A_4\to /$, $C\to A_4* \ \Rightarrow\ P'=P'\cup\{ A_4\to /,\ C\to A_4*\}$.
5.2. $K\to // : N'=N'\cup\{A_5\}$, $P'=P'\cup\{ A_5\to /,\ K\to A_5/\}$.
6-9. Правил вида $A\to B$, либо $A\to\lambda$ нет
10. $S'=\{S\}$

$G'=(\{/,*,a,\downarrow,\leftarrow\},\ \{S,C,K,A_1,A_2,A_3,A_4,A_5\},\ P',\ S)$
P': $S\to A_1/ \mid A_2\leftarrow$
$C\to C/ \mid C* \mid Ca \mid A_3\leftarrow \mid A_4*$
$K\to K/ \mid K* \mid Ka \mid A_5/$
$A_1\to C*$
$A_2\to K\downarrow$
$A_3\to C\downarrow$
$A_4\to /$
$A_5\to /$

$G'=(\{/,*,a,\downarrow,\leftarrow\},\ \{S,C,K,A,B,D,E\},\ P',\ S)$
P'= $S\to A/ \mid B\leftarrow$
$C\to C/ \mid C* \mid Ca \mid D\leftarrow \mid E*$
$K\to K/ \mid K* \mid Ka \mid E/$
$A\to C*$
$B\to K\downarrow$
$D\to C\downarrow$
$E\to /$

Построение регулярного выражения, соответствующего регулярной грамматике. В общем виде алгоритм состоит из двух частей: строится система уравнений с регулярными коэффициентами; решается полученная система, решение, соответствующее аксиоме грамматики и будет искомым регулярным выражением:

1. Переименовываются нетерминальные символы грамматики: $N=\{X_1, X_2, \dots X_n\}$, соответственно правила грамматики переписываются в виде: $X_i \to X_j\gamma$, $X_i \to \gamma$, $X_i, X_j \in N, \gamma \in T^*$. Для праволинейной грамматики нетерминалы и терминальные цепочки в правой части меняются местами.
2. Строится система УРК:

$X_1 = \alpha_{01} + X_1\alpha_{11} + X_2\alpha_{21} + \dots + X_n\alpha_{n1}$
$X_2 = \alpha_{02} + X_1\alpha_{12} + X_2\alpha_{22} + \dots + X_n\alpha_{n2}$
…
$X_n = \alpha_{0n} + X_n\alpha_{1n} + X_2\alpha_{2n} + \dots + X_n\alpha_{nn}$

Коэффициенты α_{ji} выбираются следующим образом: $\alpha_{ji} = (\gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_m) \mid \forall p \in P: X_i \to X_j\gamma_1 \mid X_j\gamma_2 \mid \dots \mid X_j\gamma_m$, $i>0, j\geq 0, X_0 = \lambda$. Если правил такого вида в грамматике не существует, то $\alpha_{ji}=\varnothing$. Для праволинейной грамматики нетерминалы и терминальные цепочки в правой части меняются местами.

3. Решается система уравнений:
3.1. i=0;
3.2. i++; Уравнение для X_i переписывается в виде $X_i = X_i\alpha_{ii} + \beta_i$, $\beta_i = \alpha_{i0} + X_{i+1}\alpha_{ii+1} + \dots + X_n\alpha_{in}$
3.3. Находится решение уравнения в виде $X_i = \beta_i\alpha_{ii}^* = (\alpha_{i0} + X_{i+1}\alpha_{ii+1} + \dots + X_n\alpha_{in})\alpha_{ii}^*$, для праволинейной грамматики решение будет в виде $X_i = \alpha_{ii}^*\beta_i$.Следует отметить, что если i=n, то решение для X_n будет конечным, т.е. α_{ii} и β_i не будут содержать в своем составе переменных X_m.
3.4. $\forall k \mid i<k\leq n$, в уравнениях для X_k выполняется подстановка вида $X_i \to \beta_i\alpha_{ii}^*$.
3.5. Если i<n, перейти к шагу 3.2.
3.6. i--
3.7. $\forall k \mid i<k\leq n$, в уравнениях для X_i выполняется подстановка окончательных решений для X_k .
3.8. Если i>1, перейти к шагу 3.6.
3.9. Решение для $X_ь$, соответствующего аксиоме грамматики, и будет искомым регулярным выражением.

Пример. Грамматика описывает объявление многомерных массивов в C.
G({[,],n,i,0},{R,L,K},P,{R}), здесь n – любой символ 1..9, 0 не включается, чтобы исключить объявления вида x[0], i – идентификатор.

P: $R \to L]$
$L \to L0 \mid Ln \mid Kn$
$K \to i[\mid R[$

1. $R=X_1$, $L=X_2$, $K=X_3$; P: $X_1 \to X_2]$; $X_2 \to X_20 \mid X_2n \mid X_3n$; $X_3 \to i[\mid X_1[$.
2. $X_1 = X_2]$ $= \varnothing + X_1\varnothing + X_2] + X_3\varnothing$
   $X_2 = X_20 + X_2n + X_3n$ $= \varnothing + X_1\varnothing + X_2(0+n) + X_3n$
   $X_3 = i[+ X_1[$ $= i[+ X_1[+ X_2\varnothing + X_3\varnothing$
3.2. i=1. $X_1 = X_2]$; $\alpha_{11}=\varnothing$; $\beta_1= X_2]$;
3.3. Решение будет само уравнение $X_1 = X_2]$.
3.4. В уравнении для X_2 подстановки не требуется. $X_3 = i[+ X_1[= i[+ X_2][$
3.2. i=2. $X_2 = X_20 + X_2n + X_3n = X_2(0+n) + X_3n$; $\alpha_{22}=0+n$; $\beta_2= X_3n$;
3.3. Решением будет $X_2 = \beta_2\alpha_{22}^* = X_3n(0+n)^*$.
3.4. $X_3 = i[+ X_2][= i[+ X_3n(0+n)^*][$.
3.2. i=3. $X_3 = i[+ X_3n(0+n)^*][= X_3n(0+n)^*][+ i[$; $\alpha_{33}= n(0+n)^*][$; $\beta_3= i[$.
3.3. Решением будет $X_3 = \beta_3\alpha_{33}^* = i[\ (n(0+n)^*][)^*$.
3.7. i=2. $X_2 = X_3n(0+n)^* = i[\ (n(0+n)^*][)^*n(0+n)^*$
3.7. i=1. $X_1 = X_2] = i[\ (n(0+n)^*][)^*n(0+n)^*]$
3.9. Аксиоме соответствует X_1. Следовательно грамматике соответствует регулярное выражение для X_1: $= i[(n(0|n)^*][)^*n(0|n)^*]$.

Конечные автоматы

Конечные автоматы. Детерминированные и недетерминированные КА. Диаграмма состояний КА. Связь КА и синтаксических диаграмм. Построение КА на основе леволинейной и праволинейной грамматик. Построение леволинейной и праволинейной грамматик на основе КА. Преобразование КА к детерминированному виду. Минимизация КА. Устранение недостижимых и эквивалентных состояний КА. Автоматизация построения лексических анализаторов. Программа LEX

Конечные автоматы.

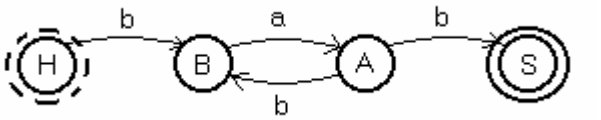
Конечным автоматом называется пятерка следующего вида: **M (Q, V, δ, q_0, F),** где Q – конечное множество состояний автомата, V – конечное множество допустимых входных символов (алфавит автомата), δ - функция переходов, отображающая декартово произведение множеств $V^{\times}Q$ во множество подмножеств Q: **δ(a,q)=R, a∈V, q∈Q, R⊆Q**; q_0 – начальное состояние автомата, $q_0 \in Q$; F – непустое множество конечных состояний автомата, $F \subseteq Q$, $F \neq \varnothing$.

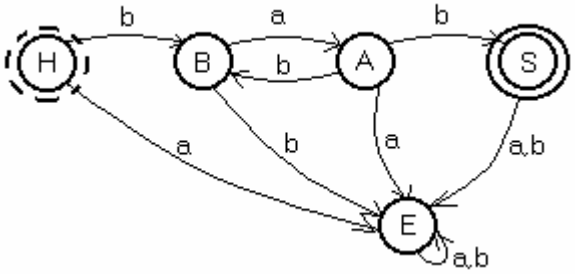
КА **полностью определен**, если в каждом его состоянии существует функция перехода для всех возможных входных символов: **∀a∈V, ∀q∈Q ∃ δ(a,q)=R, R⊆Q.**

В начале работы автомат всегда находится в состоянии q_0. На каждом такте он под воздействием очередного символа входной цепочки либо переходит в новое состояние либо остается в текущем. Если функция перехода допускает несколько возможных состояний, то КА может перейти в любое из них. Работа КА продолжается до тех пор, пока на его вход поступают символы из входной цепочки.

КА M (Q, V, δ, q_0, F) **принимает цепочку символов** $\omega \in V^+$, если получив на вход эту цепочку, он из начального состояния может перейти в одно из конечных состояний f∈F. **Язык L(M), заданный КА** – это множество всех цепочек символов, которые принимаются этим автоматом. Два КА **эквивалентны**, если они задают один и тот же язык. Все КА являются распознавателями для регулярных языков.

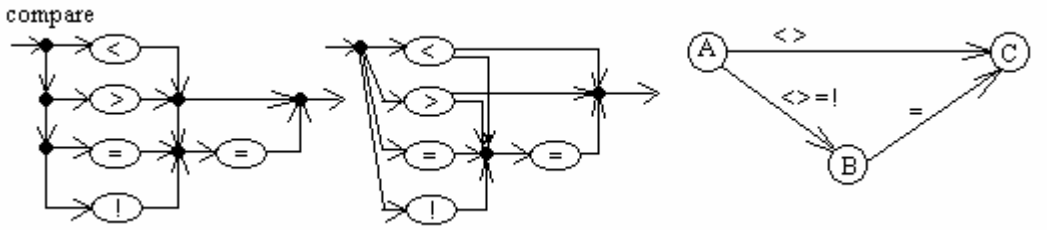
Граф переходов КА – это ориентированный граф, в котором состояния КА соответствуют вершинам, а переходам δ(p,a)=q, a∈V, p,q∈Q - дугам (p,q), помеченным символом a. Вершины, соответствующие начальному и конечным состояниям, выделяются, обычно двойной границей. На рисунке представлен КА: M({H,A,B,S},{a,b},δ,H,{S}). δ: δ(H,b)=B, δ(B,a)=A, δ(A,b)={B,S}. Чтобы исключить ситуации, из которых нет переходов по входным символам, в КА добавляют еще одно состояние, на которое замыкают все неопределенные переходы, преобразуя автомат к полностью определенному виду. Это дополнительное состояние соответствует состоянию ошибки.





КА M (Q, V, δ, q_0, F) называется **детерминированным** КА, если **∀a∈V, ∀q∈Q: δ(q,a)={r}, r∈Q либо δ(q,a)=∅,** т.е. в каждом из его состояний для любого входного символа функция перехода содержит не более одного состояния. Иначе КА **недетерминированный**.

Связь КА и синтаксических диаграмм.



Построение КА по леволинейной грамматике. Имеется леволинейная грамматика G(T,N,P,S), задающая язык L(G). Необходимо построить эквивалентный ей КА M(Q,V,δ,q_0,F), задающий тот же язык L(M)=L(G). Задача решается в 2 этапа. 1. Исходная леволинейная грамматика G приводится к автоматному виду G'. 2. Искомый автомат M(Q,V,δ,q_0,F) строится на основе полученной автоматной грамматики G'(T',N',P',S'). Алгоритм построения КА:

1) Q=N∪{H}. Множество состояний соответствует нетерминальным символам грамматики, к которым добавляется еще один символ H.
2) V=T. Входной алфавит соответствует множеству терминальных символов.
3) $\forall p_i \in P$: A→t, A∈N, t∈T ⇒ δ(H,t)= δ(H,t)∪{A}. В функцию переходов для состояния H добавляется состояние A.
4) $\forall p_i \in P$: A→Bt, A,B∈N, t∈T ⇒ δ(B,t)= δ(B,t)∪{A}. В функцию переходов для состояния B добавляется состояние A.
5) q_0=H. Начальное состояние соответствует добавленному на первом шаге состоянию H.
6) F={S}. Множество конечных состояний соответствует аксиоме S.

Пример.

G=({/,*,a,↓,←}, {S,C,K,A,B,D,E}, P, S)

P= { S → A/ | B←; C → C/ | C* | Ca | D← | E*; K → K/ | K* | Ka | E/; A → C*; B → K↓; D → C↓; E → / }

1. Q={S,C,K,A,B,D,E,H}
2. V={/,*,a,↓,←}
3. E → / ⇒ δ(H,/)= δ(H,/)∪{E}.

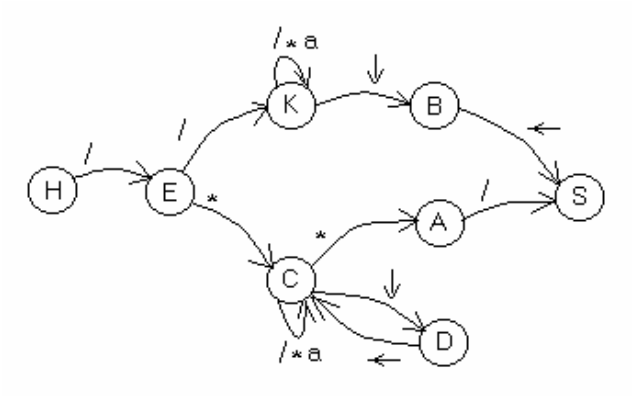
4.1. S → A/ ⇒ δ(A,/)= δ(A,/)∪{S}; S → B← ⇒ δ(B, ←)= δ(B, ←)∪{S}

4.2. δ(C,/)= δ(C,/)∪{C}; δ(C,*)= δ(C,*)∪{C}; δ(C,a)= δ(C,a)∪{C}; δ(D, ←)= δ(D, ←)∪{C}; δ(E,*)= δ(E*,)∪{C}

4.3. δ(K,/)= δ(K,/)∪{K}; δ(K,*)= δ(K,*)∪{K}; δ(K,a)= δ(K,a)∪{K}; δ(E,/)= δ(E,/)∪{K}

4.4. $\delta(C,*)= \delta(C,*)\cup\{A\}$; $\delta(K,\downarrow)= \delta(K,\downarrow)\cup\{B\}$; $\delta(C,\downarrow)= \delta(C,\downarrow)\cup\{D\}$
5. $q_0=H$
6. $F=\{S\}$

δ	S	C	K	A	B	D	E	H
/		C	K	S			K	E
*		C,A	K				C	
a		C	K					
↓		D	B					
←					S	C		



Видно, что автомат недетерминированный, т.к. $\delta(C,*)=\{C,A\}$. Колонки, соответствующие конечному состоянию в общем случае могут и не быть пустыми. Автомат не полностью определен. Построим граф.

Построение КА по праволинейной грамматике. Имеется праволинейная грамматика G(T,N,P,S), задающая язык L(G). Необходимо построить эквивалентный ей КА M(Q,V,δ,q_0,F), задающий тот же язык L(M)=L(G). Аналогично предыдущему случаю, исходная грамматика приводится к автоматному виду, после чего строится автомат. Алгоритм построения КА:

1) $Q=N\cup\{H\}$. Множество состояний соответствует нетерминальным символам грамматики, к которым добавляется еще один символ H.
2) $V=T$. Входной алфавит соответствует множеству терминальных символов.
3) $\forall p_i \in P: A\to t, A\in N, t\in T \Rightarrow \delta(A,t)= \delta(A,t)\cup\{H\}$. В функцию переходов для состояния A добавляется состояние H.
4) $\forall p_i \in P: A\to tB, A,B\in N, t\in T \Rightarrow \delta(A,t)= \delta(A,t)\cup\{B\}$. В функцию переходов для состояния A добавляется состояние B.
5) $q_0=S$. Начальное состояние соответствует аксиоме S.
6) $F=\{H\}$. Множество конечных состояний соответствует добавленному на первом шаге состоянию H.

Построение леволинейной грамматики по КА. Имеется КА M(Q,V,δ,q_0,F), определяющий язык L(M). Необходимо построить эквивалентную ему леволинейную грамматику G(T,N,P,S), задающую тот же язык L(G)=L(M). Алгоритм построения леволинейной грамматики:

1) $T=V$. Множество терминальных символов строится из входного алфавита автомата.
2) $N=Q\backslash\{q_0\}$. Множество нетерминалов грамматики соответствует множеству состояний автомата за исключением начального.
3) $A_i = q_0 \mid \delta(A_i,t)=\{B_1, B_2,\ldots, B_n\}, n>0, B_k\in Q, k=1,2,..,n, t\in V \Rightarrow P=P\cup\{B_k\to t\}$. Пополняется множество правил.
4) $\forall A_i \neq q_0, A_i \in Q \mid \delta(A_i,t)=\{B_1, B_2,\ldots, B_n\}, n>0, B_k\in Q, k=1,2,..,n, t\in V \Rightarrow P=P\cup\{B_k\to A_i t\}$. Пополняется множество правил.
5) Если $F=\{F_0\} \Rightarrow S=\{F_0\}$. Если у автомата одно конечное состояние, оно становится аксиомой грамматики.
6) Если $F=\{F_1, F_2,\ldots, F_n\}, n>1 \Rightarrow N=N\cup\{S\}, P=P\cup\{S\to F_1 \mid F_2 \mid\ldots\mid F_n\}$. Если у автомата несколько конечных состояний, то добавляется новый нетерминал S, и пополняется множество правил.

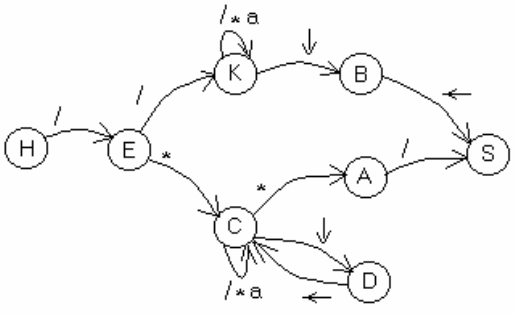
Построение праволинейной грамматики по КА. Имеется КА M(Q,V,δ,q_0,F), определяющий язык L(M). Необходимо построить эквивалентную ему праволинейную грамматику G(T,N,P,S), задающую тот же язык L(G)=L(M). Алгоритм:

1) $T=V$. Множество терминальных символов строится из входного алфавита автомата.
2) $N=Q$. Множество нетерминалов грамматики соответствует множеству состояний автомата.
3) $\forall A_i \in Q: \delta(A_i,t)=\{B_1, B_2,\ldots, B_n\}, n>0, B_k\in Q, B_k\notin F, k=1,2,..,n, t\in V \Rightarrow P=P\cup\{A_i\to tB_k\}$. Пополняется множество правил.
4) $\forall A_i \in Q: \delta(A_i,t)=\{B_1, B_2,\ldots, B_n\}, n>0, B_k\in Q, B_k\in F, k=1,2,..,n, t\in V \Rightarrow P=P\cup\{A_i\to t\}$. Пополняется множество правил.
5) $\forall A_i \in Q: \delta(A_i,t)=\{B_1, B_2,\ldots, B_n\}, n>0, B_k\in Q, B_k\in F, k=1,2,..,n, t\in V, \exists\delta(B_k,t')=\{C_1, C_2,\ldots, C_m\}\ C_j\in Q, q=1,2,..,m, t'\in V \Rightarrow P=P\cup\{A_i\to tB_k\}$. Пополняется множество правил, если есть переходы из конечных состояний.
6) $S=\{q_0\}$

Пример.

M({S,C,K,A,B,D,E,H},{/,*,a,↓,←},δ,H,{S})

δ	S	C	K	A	B	D	E	H
/		C	K	S			K	E
*		C,A	K				C	
a		C	K					
↓		D	B					
←					S	C		



1. $T=\{/,*,a,\downarrow,\leftarrow\}$
2. $N=\{S,C,K,A,B,D,E,H\}$
3.1. $\delta(C,/)=\{C\} \Rightarrow P=P\cup\{C\to/C\}$; $\delta(C,*)=\{C,A\} \Rightarrow P=P\cup\{C\to *C \mid *A\}$;
$P=P\cup\{C\to aC\}$; $P=P\cup\{C\to\downarrow D\}$;
3.2. $P=P\cup\{K\to/K \mid *K \mid aK \mid \downarrow B\}$;
3.3. $P=P\cup\{D\to\leftarrow C\}$;
3.4. $P=P\cup\{E\to/K \mid *C\}$;
3.5. $P=P\cup\{H\to/E\}$
4. $\delta(A,/)=\{S\} \Rightarrow P=P\cup\{A\to/\}$; $\delta(B,\leftarrow)=\{S\} \Rightarrow P=P\cup\{B\to\leftarrow\}$.
5. Если бы из состояния S были переходы, то добавились бы $A\to/S$; $B\to\leftarrow S$.
6. $S=\{H\}$

P: {A→/; B→←; C→/C | *C | *A | aC | ↓D; K→/K | *K | aK | ↓B; D→←C; E→/K | *C; H→/E;}

Преобразование КА к детерминированному виду. Для любого КА можно построить эквивалентный ему ДКА. Алгоритм преобразования КА $M(Q,V,\delta,q_0,F)$ в эквивалентный ему ДКА $M'(Q',V',\delta',q_0',F')$:

1) $V'=V$;
2) $q_0'= q_0$;
3) $Q'=\{q_0\}$; Начинаем с начального состояния
4) $\forall q_i' \in Q' \Rightarrow \delta(q_i', v_j')= \cup\{\delta(q_k,v_j)\}=q_{ij}'$, $q_k \in q_i'$, $j=1,2,..,|V|$; $Q'=Q' \cup q_{ij}'$; Итерационно формируем новые состояния и переходы;
5) $\forall q_i' \in Q'$, $q_k \in F$, $q_k \in q_i' \Rightarrow F'=F' \cup \{q_i'\}$; Множество конечных состояний.

Пример. ИД см выше.

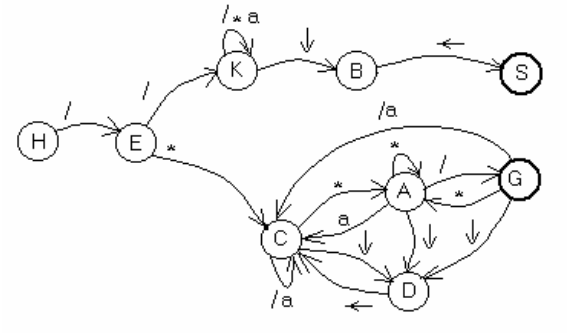
1. $V'=\{/,*,a,\downarrow,\leftarrow\}$;
2. $q_0'=\{H\}$;

3-4.

δ	H	E	K	C	B	C,A	D	S	C,S
/	E	K	K	C		C,S			C
*		C	K	C,A		C,A			C,A
a			K	C		C			C
↓			B	D		D			D
←					S		C		

δ	H	E	K	C	B	A	D	S	G
/	E	K	K	C		G			C
*		C	K	A		A			A
a			K	C		C			C
↓			B	D		D			D
←					S		C		

5. $F'=\{\{S\},\{CS\}\}$;



$F'=\{S,G\}$;

Минимизация КА. Минимизация заключается в построении эквивалентного КА с меньшим числом состояний. Рассмотрим два алгоритма: устранение недостижимых состояний и объединение эквивалентных состояний. Состояние $q \in Q$ КА $M(Q,V,\delta,q_0,F)$ **недостижимое**, если при $\forall \omega \in V^+$ невозможен переход КА из начального состояния в состояние q. Алгоритм устранения недостижимых состояний:

1) $Q'=\{q_0\}$;
2) $\forall q_i \in Q'$: $\delta(q_i, v_k)=q_j$, $k=1,2,..,|V|$; $\Rightarrow$ $Q'=Q' \cup q_j$ Т.е. итерационно включаются все состояния, в которые можно попасть из начального;
3) $\delta'=\delta' \cup \delta(q_i,v_k)$, $\forall q_i \in Q'$; т.е. остаются только соответствующие им переходы;
4) $F'=F \cap Q'$.

Состояния q, $q' \in Q$ называются **n-эквивалентными**, $n \geq 0$, если, находясь в одном из этих состояний и получив на вход произвольную цепочку $\omega \in V^*$, $|\omega| \leq n$, КА может перейти в одно и то же множество конечных состояний. 0-эквивалентными состояниями КА будут F и Q\F. Множества эквивалентных состояний называются классами эквивалентности, а их совокупность – множеством классов эквивалентности R(n), и $R(0)=\{F, Q\backslash F\}$. Алгоритм построения эквивалентных состояний:

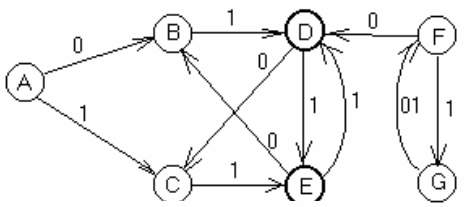
1) n=0, Строится R(n)
2) n=n+1. Строится R(n): $R(n)=\{r_i(n): \{q_i \in Q: \forall a \in V, \delta(q_i,a) \subseteq r_j(n-1)\}\}$. Т.е. в новые классы входят те состояния,, которые для одних и тех же входных символов переходят в одно и то же n-1 эквивалентные состояния.
3) Если $R(n) \neq R(n-1)$, то повторить шаг 2.

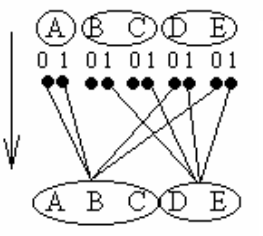
Далее каждый класс эквивалентности становится состоянием минимизированного КА, функции переходов в котором строятся очевидным образом.

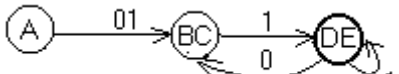
Пример. $M(\{A,B,C,D,E,F,G\}, \{0,1\}, \delta, A, \{D, E\})$

δ	A	B	C	D	E	F	G
0	B			C	B	D	F
1	C	D	E	E	D	G	F

1. $Q'=\{A\}$;
2.1. $Q'=Q' \cup \{B,C\}$
2.2. $Q'=Q' \cup \{D,E\}$
2.3. $Q'=Q' \cup \{B,C,D,E\}$
4. $F'=\{DE\} \cap \{A,B,C,D,E\}=\{D,E\}$







1. $R(0)=\{\{A,B,C\},\{D,E\}\}$
2. $R(1)=\{0:\{A\},\{BC\},\{D,E\}; 1:\{A\},\{BC\},\{D,E\}\}$;
3. $R(2)=\{0:\{A\},\{BC\},\{D,E\}; 1:\{A\},\{BC\},\{D,E\}\}$.

Лексический анализ используется не только при построении компиляторов, но и в различных других областях, связанных с обработкой текста. Например, подсветка синтаксиса ЯП в текстовом редакторе, командные процессоры. Поскольку задачи лексического анализа четко формализуются, имеется возможность автоматизировать процесс разработки ЛА. Самой известной программой подобного рода является **LEX**. Входной язык содержит описания лексем в терминах регулярных выражений. Результатом является программа на каком-либо ЯП, которая позволяет читать входной файл (или стандартный поток ввода) и выделять из него лексемы, соответствующие заданным регулярным выражениям. Полученный код сканера можно дополнять необходимыми функциями по усмотрению разработчика.

КС-грамматики. Приведение КС-грамматик

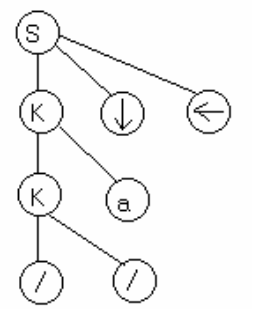
Контекстно-свободные языки. Лемма о разрастании КС-языка. Дерево синтаксического разбора. Однозначность и рекурсивность грамматики. Нормальные формы Хомского и Грейбах. Преобразование грамматики в нормальную форму Хомского. Преобразование грамматики в нормальную форму Грейбах. Приведение КС-грамматик (устранение недостижимых и бесполезных символов, устранение λ-правил, устранение цепных правил). Устранение левой рекурсии

Синтаксический анализатор – это часть компилятора, которая отвечает за выявление и проверку синтаксических конструкций входного языка. СА выполняет:
- Поиск и выделение синтаксических конструкций в тексте исходной программы
- Установку типа и проверку правильности синтаксической конструкции
- Представление синтаксических конструкций в виде, удобном для генерации результирующего кода.

СА – основная часть компилятора на этапе анализа. На вход СА поступает таблица лексем, сформированная ЛА. СА разбирает ее в соответствии с грамматикой входного языка.

Лемма о разрастании КС-языка.) Пусть L – КС-язык: $\forall\alpha\in L, \exists\delta,\beta_1,\varphi,\beta_2,\gamma\in V^*, \exists p\in N>0 \mid \alpha=\delta\beta_1\varphi\beta_2\gamma, |\alpha|\geq p, 0<|\beta_1\beta_2|\leq p, \alpha' = \delta\beta_1^i\varphi\beta_2^i\gamma, \forall i\in N\geq 0, \alpha'\in L$. В достаточно длинной строке КС языка всегда можно найти две подстроки с ненулевой суммарной длиной, одновременное повторение которых произвольное кол-во раз порождает новые строки того же языка. Например, язык $L=\{0^n1^n \mid m,n\geq 1\}$ КС, а язык $L=\{0^n1^n2^n \mid n\geq 1\}$ не КС.

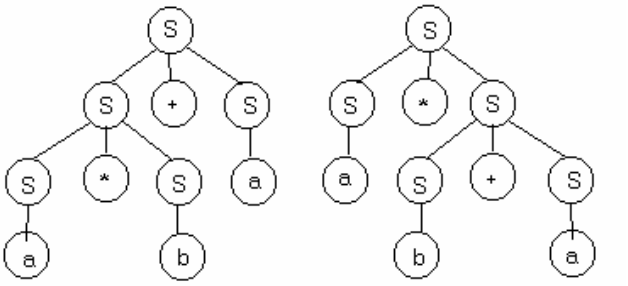
Деревом вывода грамматики G(T,N,P,S) называется дерево (граф), которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:
- каждая вершина обозначается символом грамматики $V\in(T\cup N\cup\{\lambda\})$
- корнем дерева является вершина, обозначенная аксиомой S
- листьями являются вершины, обозначенные символом $t\in(T\cup\{\lambda\})$
- если некоторый узел обозначен символом $A\in N$, а связанные с ним узлы символами $V_1,V_2,\ldots,V_n$, $n>0$, $V_i\in(T\cup N\cup\{\lambda\})$, то в грамматике G существует правило $A\rightarrow V_1,V_2,\ldots,V_n\in P$



В таком виде дерево вывода всегда можно построить для КС и регулярных грамматик. Для других типов – только частные случаи.. Пример.

G({/,*,a,↓,←},{S,C,K},P,{S}), P:{S → C*/ | K↓←; C → /* | C/ | C* | Ca | C↓←; K → // | K/ | K* | Ka}

Для цепочки //a↓← дерево вывода будет иметь следующий вид:



Для построения дерева достаточно иметь цепочку вывода. **Левосторонний** вывод имеет место, когда правило применяется всегда к самому левому нетерминалу, **правосторонний** – к самому правому. Грамматика называется **однозначной**, если для каждой цепочки символов языка, заданного этой грамматикой можно построить единственный левосторонний и единственный правосторонний вывод, т.е. для каждой цепочки символов языка существует единственное дерево вывода. Однако иногда одна и та же цепочка может иметь разные деревья вывода, например:

G({(,a,)},{S},P,{S}), P:{S → S+S | S*S | (S) | a | b}, цепочка a*b+a.

Возможные варианты вывода:

$S\rightarrow S+S\rightarrow S*S+S\rightarrow a*S+S\rightarrow a*b+S\rightarrow a*b+a$ или

$S\rightarrow S*S\rightarrow a*S\rightarrow a*S+S\rightarrow a*b+S\rightarrow a*b+a$ для левостороннего вывода.

Имеется неоднозначность. Для построения компиляторов грамматики не должны допускать подобного. Неоднозначность может устраняться заданием приоритетов. Однозначность – это свойство грамматики, а не языка. Т.е. для языка, заданного неоднозначной грамматикой, может найтись однозначная грамматика, задающая тот же самый язык. Если в грамматике имеются правила вида:

$S\rightarrow SS \mid \alpha$

$S\rightarrow S\alpha S \mid \beta$

$S\rightarrow \alpha S \mid S\beta \mid \gamma$

$S\rightarrow \alpha S \mid \alpha S\beta S \mid \gamma$, то она является неоднозначной. Это необходимое, но не достаточное условие однозначности, т.е. отсутствие правил такого вида еще не гарантирует однозначности.

***Рекурсия** может быть **явной**, когда символ определяется сам через себя (пример) и **неявной** (косвенной), тогда тоже самой происходит черезцепочку правил (пример).*

Грамматика называется **леворекурсивной**, если в ней имеются выводы вида: $\exists A\in N: A\Rightarrow^* A\alpha, \alpha\in V^+$.

Праворекурсивной: $\exists A\in N: A\Rightarrow^* \alpha A, \alpha\in V^+$.

Самовставляющей (самовложенной) $\exists A\in N: A\Rightarrow^* \alpha A\beta, \alpha,\beta\in V^+$.

В грамматике G(N,T,P,S) символ $A\in N$ называется **бесполезным** (бесплодным), если не существует вывода вида: $S\Rightarrow^*\alpha A\beta\Rightarrow\alpha b\beta$, $\alpha,b,\beta\in T^*$. Т.е. из него нельзя вывести ни одной цепочки терминальных символов.

Символ A называется **недостижимым**, если $A\neq S$ и не существует вывода вида: $S\Rightarrow^*\alpha A\beta$, $\alpha,\beta\in V^*$. Т.е. при любом выводе этот нетерминал не может появиться в цепочке.

Грамматика G(N,T,P,S) называется **грамматикой без λ-правил**, (λ-свободной) если множество P не содержит правил вида $A\rightarrow\lambda$, $A\in N$, либо есть ровно одно правило $S\rightarrow\lambda$, и аксиома S не встречается в правых частях других правил.

КС Грамматика G(N,T,P,S) называется **грамматикой без циклов**, если в ней нет выводов вида $A\Rightarrow^* A$, $A\in N$. Циклы возможны в грамматике только если в ней присутствуют **цепные правила** вида $A\rightarrow B$, $A,B\in N$.

С одной стороны при описании грамматики естественно желание ее максимально упростить. Устранение недостижимых и бесполезных символов направлены именно на это. С другой стороны, хотелось бы, чтобы СА строился максимально просто.

Устранение цепных правил (как следствие циклов) и λ-правил упрощают построение распознавателя, хотя эти методы могут несколько усложнить грамматику. Грамматика называется **приведенной**, если она не имеет бесполезных и недостижимых символов, λ-правил, цепных правил. Любая приведенная КС-грамматика, не содержащая самовложений, эквивалентна регулярной грамматике. Алгоритмы выполняются в следующем порядке: удаление бесполезных символов, удаление недостижимых символов, удаление λ-правил, удаление цепных правил.

Для КС грамматик существует несколько стандартных способов задания, т.н. нормальных форм. Каждая КС грамматика G(N,T,P,S) эквивалентна КС грамматике G'=(N',T,P',S) в **НФ Хомского**, все порождающие правила из множества P' имеют вид: $A\to BC \mid A\to a$, $A,B,C\in N'$, $a\in T$; либо $S\to\lambda$, если $\lambda\in L(G)$ и аксиома S не встречается в правых частях правил. Существует алгоритм перевода приведенной КС грамматики в НФ Хомского:

1. T'=T, N'=N, S'=S;
2. $\forall p_i\in P$: $A\to BC$, $A\to a$, $S\to\lambda$, $A,B,C\in N$, $a\in T \Rightarrow P'=P'\cup\{A\to BC \mid A\to a \mid S\to\lambda\}$;
3. $\forall p_i\in P$: $A\to Ba$, $A,B\in N$, $a\in T \Rightarrow N'=N'\cup\{A'\}$, $P'=P'\cup\{A\to BA', A'\to a\}$;
4. $\forall p_i\in P$: $A\to aB$, $A,B\in N$, $a\in T \Rightarrow N'=N'\cup\{A'\}$, $P'=P'\cup\{A\to A'B, A'\to a\}$;
5. $\forall p_i\in P$: $A\to A_1A_2\ldots A_k$, $k>2 \Rightarrow N'=N'\cup\{B_1, B_2,\ldots,B_{k-2}, A_1', A_2',\ldots A_k'\}$, $P'=P'\cup\{B_{i-1}\to A_i'B_i\}$, $i=1,2,\ldots,k-1$, $B_0=A$, $B_{k-1}=A_k$', если $A_k\in N$, $A_k'=A_k$, если $A_k\in T$, $P'=P'\cup\{A_k'\to A_k\}$ где A_k'– новый нетерминал

Пример. G=({a,q,w},{X,D,E},{X→DaqEw},X).

1. T'={a,q,w}, N'={X,D,E}, S'=X; 5. K=5, N'=N'∪{B_1, B_2, B_3, A_1', A_2', A_3', A_4', A_5'}={B_1, B_2, B_3, D, A_2', A_3', E, A_5'};
P'=P'∪{$B_0\to A_1'B_1$, $B_1\to A_2'B_2$, $B_2\to A_3'B_3$, $B_3\to A_4'B_4$}={$X\to A_1'B_1$, $B_1\to A_2'B_2$, $B_2\to A_3'B_3$, $B_3\to A_4'A_5'$}=
{ $X\to DB_1$, $B_1\to A_2'B_2$, $B_2\to A_3'B_3$, $B_3\to EA_5'$}, P'=P'∪{ $A_2'\to A_2$, $A_3'\to A_3$, $A_5'\to A_5$}={ $A_2'\to a$, $A_3'\to q$, $A_5'\to w$}
G'=({a,q,w},{X,D,E, B_1, B_2, B_3, A_2', A_3', A_5'},{ $X\to DB_1$, $B_1\to A_2'B_2$, $B_2\to A_3'B_3$, $B_3\to EA_5'$, $A_2'\to a$, $A_3'\to q$, $A_5'\to w$ },X)

Другой нормальной формой является **НФ Грейбах**. Все порождающие правила в ней имеют вид: $A\to b\alpha$, $A\in N$, $b\in T$, $\alpha\in N^*$ либо $S\to\lambda$, если $\lambda\in L(G)$ и аксиома S не встречается в правых частях правил. Алгоритм перевода приведенной КС-грамматики без левой рекурсии:

1. G'=G;
2. Нетерминалы упорядочиваются $N'=\{A_1,A_2,\ldots,A_n\}$ таким образом, что $\forall p\in P'$: $A_i\to A_j\alpha$, $A_i,A_j\in N$, $\alpha\in V^*$, $i<j$; $k=|N'|-1$;
3. $A_k\to A_j\alpha$, $A_j\to\beta_1|\beta_2|\ldots|\beta_m$ $\forall A_j\in N'$, $\beta_d\in V^* \Rightarrow A_i\to\beta_1\alpha|\beta_2\alpha|\ldots|\beta_m\alpha$; правило заменяется для всех A_j, где β_d, d=1,2,…m – весь набор правил для A_j. k--; если k!=0, повторить п.3
4. $\forall p_i\in P$: $A\to b\gamma_1\gamma_2\ldots\gamma_m$, $A\in N'$, $b\in T'$, $\gamma_i\in V \Rightarrow A\to bY_1Y_2\ldots Y_m$, $\forall\gamma_i\in T'$, $N'=N'\cup\{Y_i\}$, $P'=P'\cup\{Y_i\to\gamma_i\}$; $\forall\gamma_i\in N'$, $Y_i=\gamma_i$,

Пример. G={{*,n},{S,A,B},{S→B*A, B→n | A*B, A→n},S}

2. $S^1\to B^2*A^3$, $B^2\to n \mid A^3*B^2$, $A^3\to n$; k=3-1=2;
3. $B^2\to n \mid A^3*B^2 \Rightarrow B^2\to n \mid n* B^2$; k=1
3. $S^1\to B^2*A^3 \Rightarrow S^1\to n* A^3 \mid n* B^2*A^3$; k=0;
4. $A^3\to n$; $B^2\to n$; $B^2\to n*B^2\Rightarrow B^2\to nY_1B^2$, $N'=N'\cup\{Y_1\}$, $P'=P'\cup\{Y_1\to *\}$; $S^1\to n*A^3\Rightarrow S^1\to nY_1A^3$; $S^1\to n*B^2*A^3\Rightarrow S^1\to nY_1B^2Y_1A^3$

G'=({*,n},{S,A,B,Y},{ A→n; B→n | nYB; Y→*; S→nYA | nYBYA}, S)

Алгоритм удаления бесполезных символов. Алгоритм работает со специальным множеством нетерминальных символов Y_i. Вначале в него попадают только те нетерминалы, из которых можно непосредственно вывести терминальные цепочки. Далее оно итерационно пополняется.

1. $Y_0=\varnothing$; i=1;
2. $\forall A\in N$: $\exists(A\to\alpha)\in P$, $\alpha\in (Y_{i-1}\cup T)^* \Rightarrow Y_i = Y_i\cup\{A\}$; т.е. на каждом шаге включаются все нетерминалы из левых частей правил, в правых частях которых только терминалы или нетерминалы, включенные на предыдущем шаге;
3. Если $Y_i\neq Y_{i-1}$, то i=i+1 и перейти к шагу 2
4. $N'=Y_i$; T'=T; S'=S;
5. $P'=P'\cup\{p_i\}$, p_i: $A\to\alpha$, $A,\alpha\in(T\cup Y_i)^*$; остаются только правила, которые содержат используемые нетерминалы

Алгоритм удаления недостижимых символов. Алгоритм работает с множеством достижимых символов Y_i. Вначале это аксиома грамматики, затем множество итерационно пополняется. Все символы, которые в итоге не войдут в это множество, являются недостижимыми и могут быть удалены.

1. $Y_0=\{S\}$; i=1;
2. $\forall A\in Y_{i-1}$: $\exists(A\to\alpha x\beta)\in P$, $\alpha,\beta\in V^*$, $x\in V \Rightarrow Y_i = Y_i\cup\{x\}\cup Y_{i-1}$; т.е на каждом шаге включаются терминалы и нетерминалы из правых частей правил, в левых частях которых только нетерминалы, включенные на предыдущем шаге;
3. Если $Y_i\neq Y_{i-1}$, то i=i+1 и перейти к шагу 2
4. $N'=N\cap Y_i$; $T'=T\cap Y_i$; S'=S;
5. $P'=P'\cup\{p_i\}$, p_i: $A\to\alpha$, $A,\alpha\in Y_i$; остаются только правила, которые содержат используемые символы

Пример. G({a,b,c},{A,B,C,D,E,F,G,S},P,S); P: { S→aAB | E; A→aA | bB; B→ACb | b; C→A | bA | cC | aE; E→cE | aE | Eb | ED | FG; D→a | c | Fb; F→BC | EC | AC; G→Ga | Gb;}

Удаляем бесполезные символы:

1. $Y_0=\varnothing$; i=1;
2-3.1. $A\to\alpha$, $\alpha\in (Y_0\cup T)^* = \{a,b,c\} \Rightarrow Y_1 = Y_1\cup\{B,D\}$; $Y_1\neq Y_0$; i=2;
2-3.2. $A\to\alpha$, $\alpha\in (Y_1\cup T)^* = \{B,D,a,b,c\} \Rightarrow Y_2 = Y_2\cup\{A,B,D\}$; $Y_2\neq Y_1$; i=3;
2-3.3. $A\to\alpha$, $\alpha\in (Y_2\cup T)^* = \{A,B,D,a,b,c\} \Rightarrow Y_3 = Y_3\cup\{S,A,B,C,D\}$; $Y_3\neq Y_2$; i=4;
2-3.4. $A\to\alpha$, $\alpha\in (Y_3\cup T)^* = \{S,A,B,C,D,a,b,c\} \Rightarrow Y_4 = Y_4\cup\{S,A,B,C,D,F\}$; $Y_3\neq Y_2$; i=5;
2-3.5. $A\to\alpha$, $\alpha\in (Y_4\cup T)^* = \{S,A,B,C,D,F,a,b,c\} \Rightarrow Y_5 = Y_5\cup\{S,A,B,C,D,F\}$; $Y_4=Y_5$;
4. $N'=Y_5=\{S,A,B,C,D,F\}$; $T'=T=\{a,b,c\}$; $S'=S=\{S\}$;
5. P'={ S→aAB; A→aA | bB; B→ACb | b; C→A | bA | cC; D→a | c | Fb; F→BC | AC;}

Удаляем недостижимые символы:

1. $Y_0=\{S\}$; i=1;

2-3.1. $\forall A \in Y_0=\{S\}$: $A \to \alpha x \beta$, $\alpha,\beta \in V^*$, $x \in V \Rightarrow Y_1 = Y_1 \cup \{a,A,B\} \cup \{S\} = \{a,A,B,S\}$; $Y_1 \neq Y_0$; i=2;
2-3.2. $\forall A \in Y_1=\{a,A,B,S\}$: $A \to \alpha x \beta$, $\alpha,\beta \in V^*$, $x \in V \Rightarrow Y_2 = Y_2 \cup \{a,A,B,b,C\} \cup \{a,A,B,S\} = \{a,b,A,B,C,S\}$; $Y_2 \neq Y_1$; i=3;
2-3.3. $\forall A \in Y_2=\{a,b,A,B,C,S\}$: $A \to \alpha x \beta$, $\alpha,\beta \in V^*$, $x \in V \Rightarrow Y_3 = Y_3 \cup \{a,A,B,b,C,c\} \cup \{a,b,A,B,C,S\} = \{a,b,c,A,B,C,S\}$; $Y_3 \neq Y_2$; i=4;
2-3.4. $\forall A \in Y_3=\{a,b,c,A,B,C,S\}$: $A \to \alpha x \beta$, $\alpha,\beta \in V^*$, $x \in V \Rightarrow Y_4 = Y_4 \cup \{a,A,B,b,C,c\} \cup \{a,b,c,A,B,C,S\} = \{a,b,c,A,B,C,S\}$; $Y_4 = Y_3$;
4. $N'=N \cap Y_i=\{A,B,C,S\}$; $T'=T \cap Y_i=\{a,b,c\}$; $S'=S=\{S\}$
5. p_i: $A \to \alpha$, $A,\alpha \in Y_i$; $P'=\{ S \to aAB;\ A \to aA \mid bB;\ B \to ACb \mid b;\ C \to A \mid bA \mid cC; \}$

Алгоритм устранения λ-правил. Алгоритм работает со специальным множеством нетерминальных символов Y_i. Вначале это нетерминалы, допускающие λ-правила. Затем оно итерационно пополняется, после чего изменяется набор правил грамматики для нетерминалов этого множества.

1. $Y_0 = \{\forall A: \exists (A \to \lambda)\}$; i=1;
2. $\forall A: \exists (A \to \alpha) \in P$, $\alpha \in Y_{i-1}^* \Rightarrow Y_i = Y_{i-1} \cup \{A\}$; т.е на каждом шаге включаются нетерминалы из левых частей правил, в правых частях которых только нетерминалы, включенные на предыдущем шаге;
3. Если $Y_i \neq Y_{i-1}$, то i=i+1 и перейти к шагу 2
4. N'=N; T'=T; S'=S; $P'=P \setminus \{A \to \lambda\}$;
5. $\forall p_i \in P$: $A \to X_1\beta_1X_2\beta_2 \ldots X_n\beta_n$, $\exists X_j \in Y_i \Rightarrow P=P \cup \{A \to W_1\beta_1W_2\beta_2 \ldots W_n\beta_n$: $W_k=\{\{X_k$ либо λ, если $X_k \in Y_i\}\{X_k$, если $X_k \notin Y_i\}\} \setminus \{A \to \lambda,\ A \to A;\}\}$. Т.е. для всех правил, в правых частях которых встречаются нетерминалы из множества Y_i к правилам добавляется множество, в котором правые части представляют собой все возможные комбинации, в которых эти нетерминалы заменяются на λ.
6. Если $S \in Y_i \Rightarrow N'=N' \cup \{S'\}$, $P'=P' \cup \{S' \to \lambda \mid S\}$, $S'=\{S'\}$; т.е если $\lambda \in L(G)$, то вводится новый нетерминал, который становится аксиомой и два новых правила

Пример. G({a,b,c},{A,B,C,S},P,S); P:{S→AaB | aB | cC; A→AB | a | b | B; B→Ba | λ; C→AB | c;}
1. $Y_0 = \{B\}$; i=1;
2-3.1. $\exists (A \to B)$, $B \in Y_0^* \Rightarrow Y_1 = Y_0 \cup \{A\} = \{A,B\}$; $Y_1 \neq Y_0$; i=2;
2-3.2. $\exists (C \to AB;\ A \to AB;\ A \to B)$, $A,B \in Y_1^* \Rightarrow Y_2 = Y_1 \cup \{A,B,C\} = \{A,B,C\}$; $Y_2 \neq Y_1$; i=3;
2-3.2. $\exists (C \to AB;\ A \to AB;\ A \to B)$, $A,B \in Y_2^* \Rightarrow Y_3 = Y_2 \cup \{A,B,C\} = \{A,B,C\}$; $Y_3 = Y_2$;
4. $N'=N=\{A,B,C,S\}$; $T'=T=\{a,b,c\}$; $P'=\{S \to AaB \mid aB \mid cC;\ A \to AB \mid a \mid b \mid B;\ B \to Ba;\ C \to AB \mid c;\}$
5.1. $S \to AaB$, $A,B \in Y_3$; $\Rightarrow P=P \cup \{S \to AaB \mid Aa \mid aB \mid a\}$;
5.2. $S \to aB$; $B \in Y_3$; $\Rightarrow P=P \cup \{S \to aB \mid a\}$;
5.3. $S \to cC$; $C \in Y_3$; $\Rightarrow P=P \cup \{S \to cC \mid c\}$;
5.4. $A \to AB$; $A,B \in Y_3$; $\Rightarrow P=P \cup \{A \to AB \mid A \mid B\} \setminus \{A \to A\}$;
5.5. $A \to B$; $B \in Y_3$; $\Rightarrow P=P \cup \{A \to B\}$;
5.6. $B \to Ba$; $B \in Y_3$; $\Rightarrow P=P \cup \{B \to Ba \mid a\}$;
5.7. $C \to AB$; $A,B \in Y_3$; $\Rightarrow P=P \cup \{C \to AB \mid A \mid B\}$;
6. $S \notin Y_3$ $P'=\{ S \to AaB \mid aB \mid cC \mid Aa \mid a \mid c;\ A \to AB \mid a \mid b \mid B;\ B \to Ba \mid a;\ C \to AB \mid c \mid A \mid B;\}$

Алгоритм устранения цепных правил. Для каждого нетерминала X строится специальное множество цепных символов Y^x, на основе которых выполняется преобразование правил грамматики.

1. $\forall X \in N$:
   2. $Y^x_0=\{X\}$; i=1;
   3. $\forall p_k \in P$: $A \to B$, $A \in Y^x_{i-1} \Rightarrow Y^x_i = Y^x_{i-1} \cup \{B\}$; т.е. включаются все нетерминалы, которые непосредственно выводятся из нетерминалов множества, полученного на предыдущем шаге;
   4. Если $Y^x_i \neq Y^x_{i-1}$, то i=i+1, и перейти к шагу 3;
   5. $Y^x = Y^x_i \setminus \{X\}$;
6. N'=N; T'=T; $P'=P \setminus \{A \to B\}$; S'=S;
7. $\forall p_i \in P'$: $A \to \alpha \Rightarrow P'=P' \cup \{B \to \alpha\}$, $A \in Y^B$, $B \neq A$; Идет замена цепных правил непосредственно на нетерминал-источник.

Пример. G=({+,-,/,*,a,b},{S,T,E},P,S); P={S→S+T | S-T | T; T→T*E | T/E | E; E→(S) | a | b};
1.1. Y^s:
   2.1. $Y^s_0=\{S\}$; i=1;
   3-4.1.1. $S \to T$, $S \in Y^s_0 \Rightarrow Y^s_1 = Y^s_0 \cup \{T\} = \{S,T\}$; $Y^s_1 \neq Y^s_0$; i=2;
   3-4.1.2. $S \to T$, $T \to E$, $S,T \in Y^s_1 \Rightarrow Y^s_2 = Y^s_1 \cup \{T,E\} = \{S,T,E\}$; $Y^s_2 \neq Y^s_1$; i=3;
   3-4.1.3. $S \to T$, $T \to E$, $S,T \in Y^s_2 \Rightarrow Y^s_3 = Y^s_2 \cup \{T,E\} = \{S,T,E\}$; $Y^s_3 = Y^s_2$;
   5.1. $Y^s = Y^s_3 \setminus \{S\} = \{T,E\}$;
1.2. Y^T:
   2.2. $Y^T_0=\{T\}$; i=1;
   3-4.2.1. $T \to E$, $T \in Y^T_0 \Rightarrow Y^T_1 = Y^T_0 \cup \{E\} = \{T,E\}$; $Y^T_1 \neq Y^T_0$; i=2;
   3-4.2.2. $T \to E$, $T \in Y^T_1 \Rightarrow Y^T_2 = Y^T_1 \cup \{E\} = \{T,E\}$; $Y^T_2 = Y^T_1$;
   5.2. $Y^T = Y^T_2 \setminus \{T\} = \{E\}$;
1.3. Y^E:
   2.2. $Y^E_0=\{E\}$; i=1;
   3-4.2.1. $Y^E_1 = Y^E_0$
   5.2. $Y^E = Y^E_1 \setminus \{E\} = \varnothing$;
6. N'=N={S,T,E}; T'=T={+,-,/,*,a,b}; P'={S→S+T | S-T; T→T*E | T/E; E→(S) | a | b}; S'=S={S};
7.1. S→S+T | S-T; $S \notin Y^S$, $S \notin Y^T$ $S \notin Y^E$;
7.2. T→T*E | T/E; $T \in Y^S$; $\Rightarrow P'=P' \cup \{ S \to T*E \mid T/E\}$;
7.3. E→(S) | a | b; $E \in Y^S$, $E \in Y^T$; $\Rightarrow P'=P' \cup \{ S \to (S) \mid a \mid b;\ T \to (S) \mid a \mid b \}$;

$P'=\{ S \to S+T \mid S-T \mid T*E \mid T/E \mid (S) \mid a \mid b;\ T \to T*E \mid T/E \mid (S) \mid a \mid b\ ;\ E \to (S) \mid a \mid b \}$

Устранение левой рекурсии. Любая КС-грамматика может быть как леворекурсивной, так и праворекурсивной, а также одновременно и право- и леворекурсивной. Полностью исключить рекурсию невозможно, однако один вид рекурсии можно заменить на другой. Значительные неудобства чаще всего создает левая рекурсия. Рассмотрим алгоритм ее устранения.

1. $\forall A_i \in N$, i=1,2… |N|: т.е. для каждого нетерминального символа выполняются действия
   2. $\forall p_k \in P$: $A_i \to B\beta$, $B \in V$, $\beta \in V^*$, $A_i \neq B \Rightarrow$ P'=P'$\cup p_k$; N'=N'$\cup A_i$; т.е. если все правила для A_i не содержат левой рекурсии, они переносятся без изменений;
   3. $\exists p_k \in P$: $A_i \to A_i\beta$, $\beta \in V^*$: $\Rightarrow A_i=A_i\alpha_1 \mid A_i\alpha_2 \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_p$, где $\forall j$, $1 \le j \le p$, не$\exists \beta_j = A_k\gamma$, $k \le i \Rightarrow$ P'=P'$\cup \{A_i \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p \mid \beta_1 A_i' \mid \beta_2 A_i' \mid \ldots \mid \beta_p A_i'$; $A_i' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A_i' \mid \alpha_2 A_i' \mid \ldots \mid \alpha_m A_i'$;}; N'=N'$\cup \{A_i, A_i'\}$; т.е. если есть хотя бы одно леворекурсивное правило для A_i все правила для A_i переписываются с добавлением нового нетерминала
   4. Если i=|N|, то все нетерминалы рассмотрены, перейти к п.9;
   5. i=i+1; j=1;
   6. $\forall p_k \in P$: $A_i \to A_j\gamma$, $\gamma \in V^* \Rightarrow$ P=P$\cup \{A_i \to \omega_1\gamma \mid \omega_2\gamma \mid \ldots \mid \omega_q\gamma$: $\{A_j \to \omega_1 \mid \omega_2 \mid \ldots \mid \omega_q\} \in P'\} \backslash p_k$; т.е. для правил данного вида каждое из них меняется на множество правил;
   7. Если j=i-1, то перейти к п.2;
   8. j=j+1; перейти к п.6
9. S'=S;

Пример G({a,b,c,d,z},{S,A,B,C},P,S); P:{ S→Aa; A→Bb; B→Cc | d; C→Az | a;}

1. i=1; A_i=S;
2-3.1. S→Aa ⇒ P'=P'∪{S→Aa}; N'=N'∪{S}; P'={ S→Aa }
4.1. i≠4
5.1. i=2; A_i=A; j=1; A_j=S;
6.1.1. правил в P вида A→Sγ нет
7.1.1. j=i-1, переход к п.2

2-3.2. A→Bb ⇒ P'=P'∪{A→Bb}; N'=N'∪{A}; P'={ S→Aa; A→Bb }
4.2. i≠4
5.2. i=3; A_i=B; j=1; A_j=S;
6.2.1. правил в P вида B→Sγ нет
7.2.1. j≠i-1
8.2.1. j=2; A_j=A переход к п.6.
6.2.2. правил в P вида B→Aγ нет
7.2.2. j=i-1; переход к п.2

2-3.3. B→Cc | d ⇒ P'=P'∪{ B→Cc | d }; N'=N'∪{B}; P'={ S→Aa; A→Bb; B→Cc | d }
4.3. i≠4
5.3. i=4; A_i=C; j=1; A_j=S;
6.3.1. правил в P вида C→Sγ нет
7.3.1. j≠i-1
8.3.1. j=2; A_j=A переход к п.6.
6.3.2. {C→Az}∈P, {A→Bb}∈P' ⇒ P=P∪{C→Bbz}\{C→Az} P={ S→Aa; A→Bb; B→Cc | d; C→Bbz | a}
7.3.2. j≠i-1
8.3.2. j=3; A_j=B переход к п.6.
6.3.3. {C→Bbz}∈P, {B→Cc | d} ∈P' ⇒ P=P∪{ C→{Cc | d}bz = Ccbz | dbz}\{ C→Bbz }
P={ S→Aa; A→Bb; B→Cc | d; C→ Ccbz | dbz | a }
7.3.3. j=i-1; переход к п.2

2-3.4. C→ Ccbz | dbz | a ⇒ P'=P'∪{ C→dbz | a | dbzD | aD; D→cbz | cbzD }; N'=N'∪{C,D};
4.4. i=4; переход к п.9 P'={ S→Aa; A→Bb; B→Cc | d; C→dbz | a | dbzD | aD; D→cbz | cbzD }
9. S'=S; N'={S,A,B,C,D}

Алгоритмы построения синтаксических анализаторов

Автоматы с магазинной памятью. Синтаксические анализаторы КС-языков. Синтаксический анализ сверху вниз. Распознаватели с возвратом. Алгоритм с подбором альтернатив. Построение МП-автомата с возвратом по заданной КС-грамматике. Синтаксический анализ снизу вверх. Алгоритм "перенос-свертка". Построение расширенного МП-автомата. Построение автомата по НФ Грейбах. Распознаватели без возвратов. Алгоритм по методу рекурсивного спуска. Построение автомата для разбора S-грамматики

Язык называется **контестно-свободным**, если он определяется грамматикой G(N,T,P,S), в которой правила имеют вид $A\to\beta$, где $A\in N$, $\beta\in V^*$. Распознавателями КС языков являются **автоматы с магазинной (стековой) памятью**. МП-автомат определяется следующим образом: $R(Q,A,Z,\delta, q_0, z_0, F)$, где Q – множество состояний автомата, A – алфавит входных символов, Z – специальный конечный алфавит магазинных символов автомата, $A\subseteq Z$, δ - функция переходов автомата, отображающая множество $Qx(A\cup\{\lambda\})xZ$ на конечное множество подмножеств $P(QxZ^*)$, $q_0\in Q$ – начальное состояние автомата, $z_0\in Z$ – начальный символ магазина , $F\subseteq Q$ – множество конечных состояний.

В отличие от обычного КА МП-автомат имеет стек, в который можно помещать специальные магазинные символы, обычно это терминальные и нетерминальные символы грамматики языка. Переходы между состояниями зависят не только от входного символа, но и от символа на вершине стека. В итоге конфигурация автомата определяется тремя параметрами: состоянием автомата, цепочкой еще не прочитанных символов, содержимым стека (q,α,ω). Один такт работы автомата описывается в виде $(q,a\alpha, z\omega)\Rightarrow(q',\alpha,\gamma\omega)$, где $(q',\gamma)\in\delta(q,a,z)$, $q,q'\in Q$, $a\in A\cup\{\lambda\}$, $\alpha\in A^*$, $z\in Z\cup\{\lambda\}$, $\gamma,\omega\in Z^*$. При выполнении такта в стеке заменяется символ, соответствующий условию перехода, на цепочку, соответствующую правилу перехода. Первый символ этой цепочки становится новой вершиной стека. Допускаются переходы, при которых входной символ игнорируется, оставаясь на следующий такт. Такие переходы (такты) называются **λ-переходами**. Далее, автомат может и не извлекать символ из стека ($\gamma=z$).

Начальная конфигурация определяется как (q_0, α, z_0), $\alpha\in A^*$, а множество конечных конфигураций как (q,λ,ω) $q\in F$, $\omega\in Z^*$. МПА **допускает** (принимает) цепочку символов, если получив ее на вход и находясь в начальной конфигурации, он может перейти в одну из конечных конфигураций – когда автомат находится в одном из конечных состояний, а стек содержит определенную цепочку. Язык, определяемый МПА – множество всех цепочек, которые допускает автомат. Два МПА R1 и R2 эквивалентны, если они определяют один и тот же язык L(R1)=L(R2). МПА допускает цепочку **с опустошением магазина**, если по окончании разбора автомат находится в одном из конечных состояний, а магазин пуст (конфигурация (q,λ,λ)). Для любого МПА всегда можно построить эквивалентный ему МПА, допускающий цепочки с опустошением стека. **Расширенный** МПА в отличие от обычного, может заменять не один символ на вершине стека, а цепочку символов на вершине стека. Функция переходов для него отображает множество $Qx(A\cup\{\lambda\})xZ^*$. Для любого расширенного МПА всегда можно построить эквивалентный ему обычный. Для произвольной КС-грамматики всегда можно построить МПА, задающий тот же язык. Для произвольного МПА всегда можно построить КС-грамматику, задающую тот же язык.

МПА называется **детерминированным**, если из каждой его конфигурации возможно не более одного перехода в другую конфигурацию. Формально для ДМПА функция переходов δ может иметь один из трех видов:

1. $\delta(q,a,z)$ содержит 1 элемент: $\delta(q,a,z)=\{(q',\gamma)\}$, $\gamma\in Z^*$, $\delta(q,\lambda,z)=\varnothing$;
2. $\delta(q,a,z)=\varnothing$, $\delta(q,\lambda,z)$ содержит 1 элемент $\delta(q,\lambda,z)=\{(q',\gamma)\}$, $\gamma\in Z^*$;
3. $\delta(q,a,z)=\varnothing$, $\delta(q,\lambda,z)=\varnothing$;

Класс ДМПА и соответствующих им языков значительно уже, чем весь класс МПА и КС-языков. В отличие от КА, для которого всегда можно построить эквивалентный ему ДКА, не для каждого МПА можно построить эквивалентный ему ДМПА. ДМПА определяют очень важный подкласс КС-языков – **детерминированные** КС-языки. Все языки, принадлежащие к этому подклассу, могут быть построены с помощью однозначных КС-грамматик. Однако не всякий язык, задаваемый однозначной КС-грамматикой, является детерминированным. Большинство практически используемых распознавателей, относятся к классу детерминированных КС-языков.

Все распознаватели для КС-языков можно разделить на 2 большие группы – **нисходящие и восходящие**. Нисходящие просматривают входную цепочку символов слева направо и порождают левосторонний вывод. Дерево вывода таким распознавателем строится от корня к листьям (сверху вниз). Восходящие также просматривают входную цепочку слева направо, но порождают правосторонний вывод. Дерево вывода при этом строится от листьев к корню (снизу вверх). Для моделирования этих групп распознавателей используются 2 алгоритма: для нисходящих алгоритм с подбором альтернатив, для восходящих – алгоритм "сдвиг-свертка". Для ЯП в большинстве случаев легче построить правосторонний восходящий распознаватель. Однако на основе левостороннего (нисходящего) синтаксического анализатора легче организовать процесс порождения цепочек результирующего языка, проще в этом случае и обнаружение и локализация ошибок в исходном тексте. На практике используются оба варианта. Конкретный выбор зависит от реализации конкретного компилятора и сложности грамматики входного языка.

Нисходящий синтаксический анализ. В основе лежит левосторонний разбор. При анализе сверх вниз вывод заданной входной цепочки строят исходя из аксиомы, называемой целью. Анализируются первые символы цепочки и принимается решение, какое правило должно быть применено, после чего выполняется попытка распознать элементы правой части правила, определяя их как подцели. И так до тех пор, пока очередные подцели не приведут к сравнению символов цепочки с терминалами, что и позволит сделать вывод о принадлежности цепочки языку. Отличительная особенность алгоритмов нисходящего разбора в том, что текущая цель (подцель) используется как вспомогательная информация для принятия решения. В общем случае при нисходящем анализе возникают следующие проблемы:

1. Наличие леворекурсивных правил. Пусть цель – A, и первое же правило для A имеет вид $A\to A\gamma$. Раз так, то будет установлена подцель A. Она опять потребует подцели A и т.д. Для нисходящего разбора леворекурсивные правила должны быть исключены из грамматики.
2. Пусть при нисходящем анализе необходимо заменить самый левый нетерминал, определяемый правилом вида:

$A\to\alpha_1 \mid \alpha_2 \mid\ldots\mid \alpha_n$. Какую из подстановок выбрать? Общего подхода не существует. Есть следующие варианты: упорядочить альтернативы так, чтобы наиболее вероятные варианты испытывались первыми, обычно первой испытывают самую длинную альтернативу; проверка следующих 2-3 символов, что позволяет точнее выбрать альтернативу; учет уже проверенных альтернатив, если ряд альтернатив имеет общие префиксы и неприемлемость какой-либо из них устанавливается на основе этих префиксов, то все эти альтернативы могут быть пропущены; ограничение глубины просмотра.

3. Вопрос локализации ошибок. Компилятор должен не только обнаружить, но и локализовать ошибку. Однако ошибка обнаруживается, если все правила проверены и подходящего не найдено. Для локализации ошибки все альтернативы подходят одинаково, поэтому в грамматику требуется вставлять правила, описывающие ошибки (неправильные конструкции).

Синтаксический распознаватель с возвратом. Это самый простой тип нисходящих распознавателей для КС-языков, они моделируют недетерминированный МПА. Поэтому на некотором шаге может возникнуть ситуация, когда существует несколько допустимых следующих состояний автомата. Алгоритм, по которому функционирует такой распознаватель, называется **алгоритмом с подбором альтернатив**. Алгоритм запоминает все возможные следующие состояния, выбирает одно из них, переходит в него и так до тех пор, пока не будет достигнуто конечное состояние, тогда строка принимается, либо пока автомат не перейдет в такую конфигурацию, когда следующее состояние не будет определено, в этом случае автомат возвращается на несколько шагов назад, где возможен выбор другого варианта следующего состояния, выбирает другой вариант и продолжает работу. Если все возможные варианты перебраны, а конечное состояние не достигнуто, то строка не принадлежит языку. Время выполнения алгоритма с возвратом имеет экспоненциальную зависимость от длины входной цепочки, а требуемый объем памяти – линейную зависимость. Поэтому, хотя алгоритмы и просты в реализации, но практически применимы только для КС-языков с малой длиной входных предложений языка.

Построение МП автомата с возвратом по заданной КС-грамматике. Алгоритм позволяет построить МП-автомат, выполняющий левосторонний разбор, т.е. начинающийся с аксиомы грамматики. Стек используется для размещения сентенциальной формы, в начале это аксиома. Очередная сентенциальная форма получается заменой верхнего нетерминала стека (вершина стека слева). Автомат обладает только одним состоянием и принимает входную строку опустошением стека. Дано: КС-грамматика $G=(T,N,P,S)$. Строится МПА $M=(Q,A,Z,\delta, q_0, z_0, F)$, такой, что $L(M)=L(G)$. Полученный МПА в общем случае не является детерминированным, что при распознавании создает проблему выбора нужного правила. В примере будем нумеровать все альтернативы и выбирать альтернативу с меньшим номером. Для учета выбранных альтернатив алгоритму потребуется второй стек. Для реализации алгоритма работы такого МП-автомата исходная грамматика не должна быть леворекурсивной. Вершину стека будем считать слева.

1. $Q=\{q\}$; $q_0=q$; $F=\varnothing$; $Z=T\cup N$, $A=T$, $z_0=S$.
2. $\forall p_i\in P$: $X\to\beta$, $X\in N$, $\beta\in V^*$ $\Rightarrow\delta(q,\lambda,X)=\delta(q,\lambda,X)\cup(q,\beta)$. Эти функции позволяют замещать нетерминал на вершине стека по правилу грамматики. Здесь λ означает, что из входной цепочки символ не берется, т.е. цепочка остается без изменений
3. $\forall a\in T$ $\Rightarrow\delta(q,a,a)=(q,\lambda)$. Эти функции выталкивают со стека символ, совпадающий со входным и перемещают читающую головку.

Пример. $G=(\{+,(,),i\},\{E,F\},P,E)$. $P=\{E\to F+E \mid F;\ F\to(E) \mid i\}$;

1. $Q=\{q\}$; $q_0=q$; $A=\{+,(,),i\}$; $Z=\{+,(,),i,E,F\}$; $z_0=E$; $F=\varnothing$;
2. **1.** $\delta(q,\lambda,E)=\cup(q,F+E)$; **2.** $\delta(q,\lambda,E)=\cup(q,F)$; **3.** $\delta(q,\lambda,F)=\cup(q,(E))$; **4.** $\delta(q,\lambda,F)=\cup(q,i)$;
3. **5.** $\delta(q,+,+)=(q,\lambda)$; **6.** $\delta(q,(,()=(q,\lambda)$; **7.** $\delta(q,),))=(q,\lambda)$; **8.** $\delta(q,i,i)=(q,\lambda)$;

Пусть автомат распознает цепочку (i), a, тогда изменение конфигураций по тактам следующее :

$(q,(i),E)\Rightarrow^{?1}(q,(i),F+E)\Rightarrow^{?3}(q,(i),(E)+E)\Rightarrow^{6}(q,i),E)+E)\Rightarrow^{?1}(q,i),F+E)+E)\Rightarrow^{?3}(q,i),(E)+E)+E)\Rightarrow^{?-}$

$(q,i),F+E)+E)\Rightarrow^{?4}(q,i),i+E)+E)\Rightarrow^{8}(q,),+E)+E)\Rightarrow^{?-}$

$(q,i),F+E)+E)\Rightarrow^{?-}$

$(q,i),E)+E)\Rightarrow^{?2}(q,i),F)+E)\Rightarrow^{?3}(q,i),(E))+E)\Rightarrow^{?-}$

$(q,i),F)+E)\Rightarrow^{?4}(q,i),i)+E)\Rightarrow^{8}(q,),)+E)\Rightarrow^{7}(q,\lambda,+E)\Rightarrow^{?-}$

$(q,i),F)+E)\Rightarrow^{?-}$

$(q,i),E)+E)\Rightarrow^{?-}$

$(q,(i),F+E)\Rightarrow^{?4}(q,(i),i+E)\Rightarrow^{?-}$

$(q,(i),F+E)\Rightarrow^{?-}$

$(q,(i),E)\Rightarrow^{?2}(q,(i),F)\Rightarrow^{?3}(q,(i),(E))\Rightarrow^{6}(q,i),E))\Rightarrow^{?1}(q,i),F+E))\Rightarrow^{?3}(q,i),(E)+E))\Rightarrow^{?-}$

$(q,i),F+E))\Rightarrow^{?4}(q,i),i+E))\Rightarrow^{8}(q,),+E))\Rightarrow^{?-}$

$(q,i),F+E))\Rightarrow^{?-}$

$(q,i),E))\Rightarrow^{?2}(q,i),F))\Rightarrow^{?3}(q,i),(E)))\Rightarrow^{?-}$

$(q,i),F))\Rightarrow^{?4}(q,i),i))\Rightarrow^{8}(q,),))\Rightarrow^{7}(q,\lambda,\lambda)$

Строка принята, т.к. магазин пуст. Фактически всюду, где в примере стоял ? автомат перебирает альтернативы, пытаясь достигнуть конечного состояния.

$(q,(i),E)\Rightarrow^{?2}(q,(i),F)\Rightarrow^{?3}(q,(i),(E))\Rightarrow^{6}(q,i),E))\Rightarrow^{?2}(q,i),F))\Rightarrow^{?4}(q,i),i))\Rightarrow^{8}(q,),))\Rightarrow^{7}(q,\lambda,\lambda)$

Восходящий синтаксический анализ. В основе лежит правосторонний разбор. Исходной сентенциальной формой является разбираемая строка языка, а целью – аксиома. Обычно восходящий анализ выполняется как последовательность операций **перенос** и **свертка**. Перенос добавляет очередной символ строки в стек. Свертка производит замену верхних символов стека, совпадающих с правой частью правила, на нетерминал левой части. Если в результате серии этих операций получена аксиома грамматики, то распознаватель принял входную строку, иначе строка не принадлежит языку. Главная проблема восходящего анализа – обеспечение однозначности определения строки в вершине стека для свертки. Проблемы возникают и когда правые части правил одинаковы. Как и в случае нисходящего анализа, эти проблемы решаются путем использования грамматик с особыми свойствами. Для реализации алгоритма работы такого МП-автомата исходная грамматика не должна содержать циклов и λ-правил. Здесь также могут возникнуть альтернативы, когда нужно определить, сколько символов с вер-

шины стека заменяются на левую часть правила, либо когда имеется несколько правил с одинаковой правой частью, для которой требуется выполнить свертку. Для учета выбранных альтернатив алгоритму потребуется второй стек.

Построение расширенного МП автомата по заданной КС-грамматике. Алгоритм позволяет построить расширенный МП-автомат, выполняющий правосторонний разбор. Стек используется для размещения левой части текущей сентенциальной формы, в начале он пуст (используется особый маркер #), вершина стека справа. На каждом шаге автомат замещает нетерминалом строку верхних символов стека в соответствии с правилами грамматики или помещает в стек очередной входной символ. Автомат обладает только одним состоянием и имеет одно заключительное состояние, в котором стек пуст. Дано: КС-грамматика G=(T,N,P,S). Строится расширенный МПА M=(Q,A,Z,δ, q_0, z_0, F), такой, что L(M)=L(G).

1. Q={q, r}; q_0=q; F={r}; Z=T∪N∪{#},A=T,z_0=#.
2. $\forall p_i \in P$: X→β, X∈N, β∈V^* ⇒ δ(q,λ,β)=δ(q,λ,β)∪(q,X). Эти функции позволяют замещать на нетерминал правую часть правила, находящуюся на вершине стека
3. ∀a∈T ⇒δ(q,a,λ)=(q,a). Эти функции помещают очередной символ в стек, если только в нем не находится правая часть какого-либо правила и перемещают читающую головку.
4. δ(q,λ,#S)=(r,λ). Добавляется функция для перевода автомата в заключительное состояние.

Пример. G=({+,(,),i},{E,F},P,E). P={E→F+E | F; F→(E) | i};

1. Q={q,r}; q_0=q; A={+,(,),i}; Z={+,(,),i,E,F,#}; z_0=#; F=r;
2. **1.** δ(q,λ,F+E)=∪(q,E); **2.** δ(q,λ,F)=∪(q,E); **3.** δ(q,λ,(E))=∪(q,F); **4.** δ(q,λ,i)=∪(q,F);
3. **5.** δ(q,+,λ)=(q,+); **6.** δ(q,(,λ)=(q,(); **7.** δ(q,),λ)=(q,)); **8.** δ(q,i,λ)=(q,i);
4. **9.** δ(q,λ,#E)=(r,λ).

Пусть автомат распознает цепочку (i), тогда изменение конфигураций по тактам следующее :
(q,(i),#) $\Rightarrow^6$ (q,i),#)) $\Rightarrow^8$ (q,),#(i)$\Rightarrow^4$ (q,),#(F) $\Rightarrow^2$ (q,),#(E)$\Rightarrow^7$ (q,λ,#(E)) $\Rightarrow^3$ (q,λ,#F) $\Rightarrow^2$ (q,λ,#E) $\Rightarrow^9$ (r,λ). Строка принята, т.к. автомат в заключительном состоянии и магазин пуст. В этот раз не возникало никаких неопределенностей.

Построение МП автомата по НФ Грейбах. Алгоритм позволяет построить недетерминированный МП-автомат. Стек используется для размещения правой части текущей сентенциальной формы, в начале это аксиома. Вершина стека слева. Автомат обладает только одним состоянием и принимает входную строку опустошением стека. Дано: КС-грамматика G=(T,N,P,S) в НФ Грейбах. Строится недетерминированный МПА M=(Q,A,Z,δ, q_0, z_0, F), такой, что L(M)=L(G).

1. Q={q}; q_0=q; F=∅; Z=N,A=T,z_0=S.
2. $\forall p_i \in P$: X→bα, b∈T, X∈N, α∈N^* ⇒ δ(q,b,X)= δ(q,b,X)∪(q,α). Эти функции позволяют замещать нетерминал на вершине стека на цепочку.

Пример. G=({+,(,),i},{E,F,G},P,E). P={E→(EFGE | iGE | (EF | i ; F→); G→+};

1. Q={q}; q_0=q; F=∅; A={+,(,),i}; Z={E,F,G}; z_0=E;
2. **1.** δ(q,(,E)=∪(q,EFGE); **2.** δ(q,i,E)=∪(q,GE); **3.** δ(q,(,E)=∪(q,EF); **4.** δ(q,i,E)=∪(q,λ); **5.** δ(q,),F)=∪(q,λ); **6.** δ(q,+,G)=∪(q,λ); Или иначе: δ(q,(,E)={(q,EFGE), (q,EF)}; δ(q,i,E)={(q,GE), (q,λ)}; δ(q,),F)= (q,λ); δ(q,+,G)= (q,λ);

Пусть автомат распознает цепочку (i), тогда изменение конфигураций по тактам следующее:
(q,(i),E) $\Rightarrow^{?1}$ (q,i),EFGE) $\Rightarrow^{?2}$ (q,),GEFGE) $\Rightarrow^{?-}$
(q,i),EFGE)$\Rightarrow^{?4}$ (q,),FGE) $\Rightarrow^5$ (q,λ,GE) $\Rightarrow^{?-}$
(q,i),EFGE)$\Rightarrow^{?-}$
(q,(i),E) $\Rightarrow^{?3}$ (q,i),EF) $\Rightarrow^{?2}$ (q,),GEF) $\Rightarrow^{?-}$
(q,i),EF) $\Rightarrow^{?4}$ (q,),F) $\Rightarrow^5$ (q,λ,λ). Строка принята, т.к. автомат в заключительном состоянии и магазин пуст.

Распознаватели без возвратов. Распознаватели без возвратов основаны на определении метода, по которому выбирается одна из возможных альтернатив. Остальные альтернативы при этом не рассматриваются. Время работы такого алгоритма обладает не экспоненциальной, а линейной зависимостью от длины входной цепочки, что позволяет использовать алгоритм для значительно более широкого спектра практических задач. Такие алгоритмы могут потребовать дополнительных ограничений на правила грамматики.

Нисходящий распознаватель S-грамматики. КС-грамматика G(N,T,P,S) называется S-грамматикой, если ее правила удовлетворяют следующим требованиям:

1. $\forall p_i \in P$: A→aβ, A∈N, a∈T, β∈V^*; т.е. правая часть правил начинается с терминала,
2. $\forall p_i \in P$: A→ $a_1\beta_1 \mid a_2\beta_2 \mid \ldots \mid a_n\beta_n$, A∈N, a_i∈T, β_i∈V^*, $a_i \neq a_j$, i,j=1, 2,..., n; i≠j; т.е. правые части правил, определяющие один и тот же нетерминал, начинаются с разных терминалов.

Первое свойство S-грамматики обеспечивает выбор очередного правила грамматики, а второе делает этот выбор однозначным, позволяя построить детерминированный распознаватель. Реализовать разбор можно различными методами. В алгоритме разбора по **методу рекурсивного спуска** для каждого нетерминала A грамматики G строится процедура разбора, получающая на вход цепочку символов α и положение считывающей головки i (номер символа). Если для символа A определено более одного правила, процедура ищет среди них правило, первый символ правой части которого бы совпадал с текущим символом входной цепочки. Если такого нет, то цепочка не принимается. Если правило найдено, то запоминается его номер, считывающая головка передвигается, и для каждого нетерминала правой части рекурсивно вызывается процедура разбора этого нетерминала. Другие методы позволяют обойтись без рекурсии и обрабатывать входную строку в цикле. Наличие на входе S-грамматики достаточное, но не необходимое условие. Т.е. и иная произвольная КС-грамматика может задавать язык, распознаваемый методом рекурсивного спуска, однако не существует алгоритма, позволяющего это проверить. Кроме того, не существует и алгоритма, позволяющего проверить, преобразуется ли произвольная КС-грамматика в S- грамматику, и алгоритма, выполняющего такое преобразование. В общем случае исключение λ-правил, левой рекурсии, цепных правил, выполнение левой факторизации могут способствовать приведению грамматики к требуемому виду, но не гарантируют этого. Алгоритм распознавания S-грамматики просты и эффективны, но имеют ограниченную применимость, поскольку только узкий класс грамматик отвечает заданным требованиям.

Распознаватель для S-грамматики строится следующим образом (вершина стека слева):

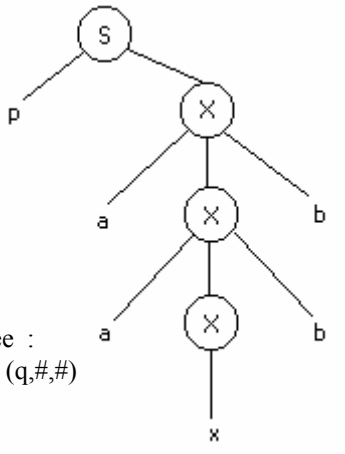
1. $A=T\cup\{\#\}$; $Z=N\cup\{t: t\in T, A\to t\alpha\notin P\}\cup\{\#\}$; $Q=\{q\}$; $q_0=q$; $F=\varnothing$; $z_0=\{S\#\}$. Здесь S- аксиома, # - символ конца строки.
2. $\forall p_i\in P$: $X\to a\beta$, $a\in T$, $X\in N$, $\beta\in V^*$ $\Rightarrow$ $\delta(q,a,X)=(q,\beta)$. Эти функции позволяют замещать нетерминал на вершине стека на цепочку;
3. $\forall$ t: $t\in T$, $A\to t\alpha\notin P$ $\Rightarrow$ $\delta(q,t,t)=(q,\lambda)$; эти функции позволяют вытолкнуть из стека терминал совпадающий с входным символом.

Алгоритм работы распознавателя на каждом из шагов выглядит так:

1. Читается очередной символ t входной цепочки;
2. Если на вершине стека находится такой же терминал, он удаляется из стека, читающая головка перемещается к следующему символу, шаг завершается (функция перехода типа 3);
3. Если на вершине стека нетерминал X, то для правила $X\to t\alpha$ он замещается цепочкой α, читающая головка перемещается, шаг завершается (функция перехода типа 2)
4. Если автомат находится в конфигурации (q, #, #), т.е. входная строка закончилась и стек пуст, строка принимается
5. Во всех остальных случаях цепочка не принимается.

Пример. G({a, b, d, p, q, x, y},{S, X, Y}, P, S), P: {$S\to pX \mid qY$; $X\to aXb \mid x$; $Y\to aYd \mid y$};

1. $A=\{a, b, d, p, q, x, y, \#\}$; $Z=\{S, X, Y, b, d, \#\}$; $Q=\{q\}$; $q_0=q$; $F=\varnothing$; $z_0=\{S\#\}$.
2. $S\to pX \Rightarrow \delta(q,p,S)=(q,X)$; 1
   $S\to qY \Rightarrow \delta(q,q,S)=(q,Y)$; 2
   $X\to aXb \Rightarrow \delta(q,a,X)=(q,Xb)$; 3
   $X\to x \Rightarrow \delta(q,x,X)=(q,\lambda)$; 4
   $Y\to aYd \Rightarrow \delta(q,a,Y)=(q,Yd)$; 5
   $Y\to y \Rightarrow \delta(q,y,Y)=(q,\lambda)$; 6
3. b: $\Rightarrow \delta(q,b,b)=(q,\lambda)$;
   d: $\Rightarrow \delta(q,d,d)=(q,\lambda)$;

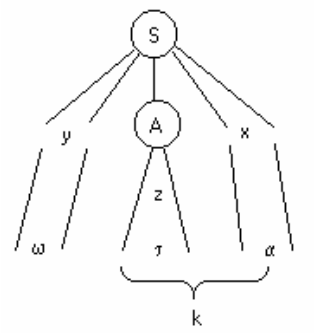


Пусть требуется распознать строку paaxbb, тогда изменение конфигураций по тактам следующее :
$(q,paaxbb\#,S\#)\Rightarrow^{1}(q,aaxbb\#,X\#)\Rightarrow^{3}(q,axbb\#,Xb\#)\Rightarrow^{3}(q,xbb\#,Xbb\#)\Rightarrow^{4}(q,bb\#,bb\#)\Rightarrow^{7}(q,b\#,b\#)\Rightarrow^{7}(q,\#,\#)$
Поскольку автомат перешел в конечную конфигурацию, строка принимается.
Номера правил из второй части (1-6) позволяют построить дерево разбора.

LL(k) грамматики

Обозначение грамматик по Кнуту. Понятие LL(k) грамматики. Сильно LL(k) грамматики. Множества направляющих цепочек, цепочек-предшественников и цепочек-последователей. Формальное определение LL(1) грамматики. Признаки LL(1) грамматики. Алгоритм построения множества символов-предшественников. Алгоритм построения множества символов-последователей. Алгоритм построения распознавателя для LL(1) грамматики.

Существует класс грамматик, допускающий детерминированный разбор сверху вниз, т.е. без возвратов.
Кнутом было предложено следующее обозначение грамматик: {L | R}{L | R}(k). Здесь первый символ указывает, в каком направлении читается входная цепочка символов – L – слева направо, R – справа налево, второй символ - левосторонний (L) или правосторонний (R) вывод используется, k – число предварительно просматриваемых символов входной строки для выбора очередного правила грамматики. Чем больше k, тем более сложные языки описывает грамматика. И тем более сложные распознаватели для них требуются.
LL(k)-грамматики. Распознаватели этих грамматик просматривают входную цепочку слева направо и строят левосторонний вывод, просматривая один символ входной цепочки. Грамматики LL(k) класса обеспечивают детерминированный нисходящий разбор. КС-грамматика G называется LL(k)-грамматикой, если для любой цепочки $\omega A\alpha \in V^*$ и первых k терминальных символов, выводимых из подцепочки $A\alpha$, существует не более одной подстановки, которую можно применить к нетерминалу A, чтобы получить левый вывод цепочки, начинающийся с ω и продолжающийся k терминальными символами.



Рассмотрим частичное дерево вывода для некоторой LL(k)-грамматики. Здесь ω - уже разобранная часть входной цепочки $\omega\tau\alpha$, построенная на основе левой части дерева y. Правая часть дерева x – еще не разобранная часть, A – текущий нетерминальный символ на вершине стека Мп-автомата. Цепочка x содержит как терминальные, так и нетерминальные символы. После завершения вывода нетерминал A раскрывается в часть входной цепочки τ, а правая часть x в часть входной цепочки α.Однозначный выбор альтернативы для нетерминала A может быть сделан на основе k первых символов цепочки $\tau\alpha$, являющейся частью входной цепочки. Основные свойства LL(k) грамматик:

- любая LL(k)-грамматика для k>0 является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LL(k)-грамматикой для строго определенного k.

Кроме того, все грамматики, допускающие разбор по методу рекурсивного спуска, являются подклассом LL(1) грамматик. Однако:

- не существует алгоритма проверки, является ли заданная КС-грамматика LL(k)-грамматикой для произвольного числа k;
- не существует алгоритма преобразования произвольной КС-грамматики к виду LL(k) грамматики для некоторого k

Для LL(k)-грамматики при k>1 вовсе не обязательно, чтобы все правые части правил грамматики для каждого нетерминального символа начинались с k различных терминальных символов. Грамматики, для которых все правые части правил для каждого нетерминального символа начинаются с k различных терминальных символов, называются **сильно LL(k)-грамматики.** Для них распознаватель строится очень просто, однако такие грамматики достаточно редко встречаются. На практике широко применяются LL(1) грамматики. S-грамматики являются их подклассом. В общем случае LL(1) грамматики накладывают менее жесткие ограничения по сравнению с S грамматиками: LL(1) грамматики допускают в правой части цепочки, начинающиеся с нетерминала, а также λ-правила. LL(1)-грамматика не может содержать для любого нетерминального символа $A \in N$ правил, начинающихся с одного и того же терминального символа.
Формальное определение LL(1)-грамматики. Множество направляющих цепочек **DS(k,A,α)** для правила $A\to\alpha$, $A\in N$, $\alpha\in V^*$ определим как объединение двух множеств $DS(k,A,\alpha) = S(k,\alpha)\cup F(k,A)$. Здесь **S(k,α)** – множество цепочек-предшественников для строки α длины k, определяемое как $S(k,\alpha)=\{\beta \mid \beta\in T^*, |\beta|=k, \alpha\in V^+, \alpha\Rightarrow^*\beta\gamma, \gamma\in V^*\}$. Фактически это множество терминальных цепочек, выводимых из непустой строки α, укороченных до k символов. **F(k,A)** – множество цепочек-последователей для нетерминала A длины k, определяемое как $F(k,A)=\{\beta \mid \beta\in T^*, |\beta|=k, S\Rightarrow^*\omega A\beta\gamma$, S- аксиома, $A\in N$, $\omega,\gamma\in V^*, \alpha\Rightarrow^*\lambda\}$. Фактически это множество терминальных цепочек, которые могут следовать в сентенциальных формах непосредственно за нетерминалом A, укороченных до k символов. Причем множество F(k,A) участвует в объединении для построения множества направляющих цепочек, только если для правила $A\to\alpha$ существует вывод $\alpha\Rightarrow^*\lambda$ (т.е. вывод пустой строки из правой части правила для нетерминала A).
Для LL(1) грамматик определение упрощается следующим образом: множество направляющих символов **DS(A,α)** для правила $A\to\alpha$, $A\in N$, $\alpha\in V^*$ $DS(A,\alpha) = S(\alpha)\cup F(A)$. **S(α)** – множество символов-предшественников $S(\alpha)=\{b \mid b\in T, \alpha\in V^+, \alpha\Rightarrow^* b\gamma, \gamma\in V^*\}$. **F(A)** – множество символов-последователей $F(A)=\{b \mid b\in T, S\Rightarrow^*\omega Ab\gamma$, S- аксиома, $A\in N$, $\omega,\gamma\in V^*, \alpha\Rightarrow^*\lambda\}$.
КС-грамматика называется **LL(1) грамматикой**, если множества направляющих символов для правил, определяющих один и тот же нетерминал грамматики, не пересекаются. $\forall A\in N$: $A\to\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \in P$, $DS(A,\alpha_i)\cap DS(A,\alpha_j)=\varnothing$, $i\neq j$, $i,j=1,2,..,n$.
Проверку принадлежности КС-грамматики классу LL(1)-грамматик можно выполнить на основе определения LL(1) грамматики, т.е. вычисляя множества направляющих символов, и их пересечения. Кроме того, имеются признаки, позволяющие установить, что КС-грамматика не является LL(1) грамматикой:

- одинаковые головные символы в правых частях правил для одного и того же нетерминала: $A\to a\alpha \mid a\beta\ldots$
- наличие прямой либо косвенной левой рекурсии $A\Rightarrow^* A\alpha$

При вычислении множества направляющих символов при наличии в грамматике λ-правил может возникать неопределенность, проявляющаяся как отсутствие символа-последователя. Проблема решается введением вспомогательного символа – маркера конца ввода #. Все строки языка L(G) представляются в виде α#, и строки выводятся не из аксиомы S, а из строки S#.

Алгоритм построения множества символов-предшественников. $S(\alpha)=\{b \mid b\in T,\ \alpha\in V^{+},\ \alpha\Rightarrow^{*}b\gamma,\ \gamma\in V^{*}\}$. В случае, когда $\alpha=d\beta$, $d\in T$, $\beta\in V^{*}$, т.е. цепочка α начинается с терминала, $S(\alpha)=\{d\}$. В случае, когда $\alpha=A\beta$, $A\in N$, $\beta\in V^{*}$, т.е. цепочка α начинается с нетерминала, $S(\alpha)=S(A)$. Алгоритм построения множества S(A):

1. Исходная грамматика G(N,T,P,S) преобразуется в G'(N'; T', P', S'), не содержащую λ-правил.
2. $\forall A\in N$: $S_0(A)=\{X \mid A\to X\alpha\in P,\ X\in V,\ \alpha\in V^{*}\}$, i=0. Вначале вносятся все первые символы правых частей правил нетерминала A
3. $\forall A\in N$: $S_{i+1}(A)= S_i(A)\cup S_i(B)$, $B\in S_i(A)\cap N$.
4. Если $\exists\ A\in N$: $S_{i+1}(A)\neq S_i(A) \Rightarrow$ i=i+1; шаг 3
5. $\forall A\in N$: $S(A)= S_i(A)\backslash N$. В результирующее множество не включаются нетерминалы.

Алгоритм построения множества символов-последователей. F(A) – множество символов-последователей $F(A)=\{b \mid b\in T,\ S\Rightarrow^{*}\omega Ab\gamma$, S- аксиома, $A\in N$, $\omega,\gamma\in V^{*}$, $\alpha\Rightarrow^{*}\lambda\}$.

1. $\forall A\in N$: $F_0(A)=\{X \mid \exists B\to\alpha AX\beta\in P,\ B\in N\ X\in V,\ \alpha,\beta\in V^{*}\}$, i=0. Изначально вносятся все символы, которые в правых частях правил встречаются непосредственно за A.
2. $F_0(S)=F_0(S)\cup\{\#\}$. Вносится пустой символ для аксиомы.
3. $\forall A\in N$: $F'_i(A)=F_i(A)\cup S(B)$, $B\in F_i(A)\cap N$.
4. $\forall A\in N$: $F''_i(A)=F'_i(A)\cup F'_i(B)$, $B\in F'_i(A)\cap N$, $\exists B\to\lambda$
5. $\forall A\in N$: $F_{i+1}(A)=F''_i(A)\cup F''_i(B)$, $B\in N$, $\exists B\to\alpha A$, $\alpha\in V^{*}$
6. Если $\exists\ A\in N$: $F_{i+1}(A)\neq F_i(A) \Rightarrow$ i=i+1; шаг 3
7. $\forall A\in N$: $F(A)= F_i(A)\backslash N$. В результирующее множество не включаются нетерминалы.

Пример. G({+,-,/,*,a,b,(,)},{S,R,T,F,E},P,S), P: {S→TR; R→λ | +TR | -TR; T→EF; F→λ | *EF | /EF; E→ (S) | a | b}

Проверим, является ли она LL(1) грамматикой. Построим множества символов предшественников и последователей.

1. Удаляем λ-правила. G'({+,-,/,*,a,b},{S,R,T,F,E},P,S), P: {S→T | TR; R→+T | -T | +TR | -TR; T→E | EF; F→*E | /E | *EF | /EF; E→ (S) | a | b}

2. $S_0(S)=\{T\}$; $S_0(R)=\{+,-\}$; $S_0(T)=\{E\}$; $S_0(F)=\{*,/\}$; $S_0(E)=\{(,a,b\}$;i=0;
3. $S_1(S)= S_0(S)\cup S_0(T)=\{T,E\}$; $S_1(R)=\{+,-\}$; $S_1(T)= S_0(T)\cup S_0(E)=\{E,(,a,b\}$; $S_1(F)=\{*,/\}$; $S_1(E)=\{(,a,b\}$;
4. i=1; шаг 3
3. $S_2(S)= S_1(S)\cup S_1(T)\cup S_1(E)=\{T,E,(,a,b\}$; $S_2(R)=\{+,-\}$; $S_2(T)= S_1(T)\cup S_1(E)=\{E,(,a,b\}$; $S_2(F)=\{*,/\}$; $S_2(E)=\{(,a,b\}$;
4. i=2; шаг 3
3. $S_3(S)= S_2(S)\cup S_2(T)\cup S_2(E)=\{T,E,(,a,b\}$; $S_3(R)=\{+,-\}$; $S_3(T)= S_2(T)\cup S_2(E)=\{E,(,a,b\}$; $S_3(F)=\{*,/\}$; $S_3(E)=\{(,a,b\}$;
4. i=2
5. $S(S)=\{(,a,b\}$; $S(R)=\{+,-\}$; $S(T)=\{(,a,b\}$; $S(F)=\{*,/\}$; $S(E)=\{(,a,b\}$;

S→TR; S(TR)=S(T)={(,a,b}	R→λ ; S(λ)=∅	R→+TR; S(+TR)={+}	R→-TR; S(-TR)={-}	
T→EF; S(EF)=S(E)={(,a,b}	F→λ; S(λ)=∅	F→*EF; S(*EF)={*}	F→/EF; S(/EF)={/}	
E→(S); S((S))={(	E→a; S(a)={a}	E→b; S(b)={b}		

1. $F_0(S)=\{)\}$; $F_0(R)=\varnothing$; $F_0(T)=\{R\}$; $F_0(F)=\varnothing$; $F_0(E)=\{F\}$; i=0;
2. $F_0(S)=\{),\#\}$;

3
$F'_i(A)=F_i(A)\cup S(B)$, $B\in F_i(A)\cap N$
$F'_0(S)=\{),\#\}$;
$F'_0(R)=\varnothing$;
$F'_0(T)= F_0(T)\cup S(R)=\{R,+,-\}$;
$F'_0(F)=\varnothing$;
$F'_0(E)=F_0(E)\cup S(F)=\{F,*,/\}$;

4
$F''_i(A)=F'_i(A)\cup F'_i(B)$, $B\in F'_i(A)\cap N, \exists B\to\lambda$
$F''_0(S)=\{),\#\}$;
$F''_0(R)=\varnothing$;
$F''_0(T)= F'_0(T)\cup F'_0(R)=\{R,+,-\}$;
$F''_0(F)=\varnothing$;
$F''_0(E)=F'_0(E)\cup F'_0(F)=\{F,*,/\}$;

5
$F_{i+1}(A)=F''_i(A)\cup F''_i(B)$, $B\in N$, $\exists B\to\alpha A$, $\alpha\in V^{*}$
$F_1(S)=\{),\#\}$;
$F_1(R)=F''_0(R)\cup F''_0(S)$ {:$\exists S\to TR$}$=\{),\#\}$;
$F_1(T)=\{R,+,-\}$;
$F_1(F)=F''_0(F)\cup F''_0(T)$ {:$\exists T\to EF$}$=\{R,+,-\}$;
$F_1(E)=\{F,*,/\}$;

6. i=1; шаг 3.

3
$F'_1(S)=\{),\#\}$;
$F'_1(R)=\{),\#\}$;
$F'_1(T)= F_1(T)\cup S(R)=\{R,+,-\}$;
$F'_1(F)= F_1(F)\cup S(R)=\{R,+,-\}$;
$F'_1(E)=F_1(E)\cup S(F)=\{F,*,/\}$;

4
$F''_1(S)=\{),\#\}$;
$F''_1(R)=\{),\#\}$;
$F''_1(T)= F'_1(T)\cup F'_1(R)=\{R,+,-,),\#\}$;
$F''_1(F)= F'_1(F)\cup F'_1(R)=\{R,+,-,),\#\}$;
$F''_1(E)=F'_1(E)\cup F'_1(F)=\{R,+,-,F,*,/\}$;

5
$F_2(S)=\{),\#\}$;
$F_2(R)=F''_1(R)\cup F''_1(S)$ {:$\exists S\to TR$}$=\{),\#\}$;
$F_2(T)=\{R,+,-,),\#\}$;
$F_2(F)=F''_1(F)\cup F''_1(T)$ {:$\exists T\to EF$}$=\{R,+,-,),\#\}$;
$F_2(E)=\{R,+,-,F,*,/\}$;

6. i=2; шаг 3.

3
$F'_2(S)=\{),\#\}$;
$F'_2(R)=\{),\#\}$;
$F'_2(T)=F_2(T)\cup S(R)=\{R,+,-,),\#\}$;
$F'_2(F)=F_2(F)\cup S(R)=\{R,+,-,),\#\}$;
$F'_2(E)=F_2(E)\cup S(R)\cup S(F)=\{R,+,-,F,*,/\}$;

4
$F''_2(S)=\{),\#\}$;
$F''_2(R)=\{),\#\}$;
$F''_2(T)= F'_2(T)\cup F'_2(R)=\{R,+,-,),\#\}$;
$F''_2(F)= F'_2(F)\cup F'_2(R)=\{R,+,-,),\#\}$;
$F''_2(E)=F'_2(E)\cup F'_2(F)\cup F'_2(R)=\{R,+,-,F,*,/,),\#\}$;

5
$F_3(S)=\{),\#\}$;
$F_3(R)=F''_2(R)\cup F''_2(S)$ {:$\exists S\to TR$}$=\{),\#\}$;
$F_3(T)=\{R,+,-,),\#\}$;
$F_3(F)=F''_2(F)\cup F''_2(T)$ {:$\exists T\to EF$}$=\{R,+,-,),\#\}$;
$F_3(E)=\{R,+,-,F,*,/,),\#\}$;

6. i=3; шаг 3.

3
$F'_3(S)=\{),\#\}$;
$F'_3(R)=\{),\#\}$;
$F'_3(T)=F_3(T)\cup S(R)=\{R,+,-,),\#\}$;
$F'_3(F)=F_3(F)\cup S(R)=\{R,+,-,),\#\}$;
$F'_3(E)=F_3(E)\cup S(R)\cup S(F)=\{R,+,-,F,*,/,),\#\}$;

4
$F''_3(S)=\{),\#\}$;
$F''_3(R)=\{),\#\}$;
$F''_3(T)= F'_3(T)\cup F'_3(R)=\{R,+,-,),\#\}$;
$F''_3(F)= F'_3(F)\cup F'_3(R)=\{R,+,-,),\#\}$;
$F''_3(E)=F'_3(E)\cup F'_3(F)\cup F'_3(R)=\{R,+,-,F,*,/,),\#\}$;

5
$F_4(S)=\{),\#\}$;
$F_4(R)=F''_3(R)\cup F''_3(S)$ {:$\exists S\to TR$}$=\{),\#\}$;
$F_4(T)=\{R,+,-,),\#\}$;
$F_4(F)=F''_3(F)\cup F''_3(T)$ {:$\exists T\to EF$}$=\{R,+,-,),\#\}$;
$F_4(E)=\{R,+,-,F,*,/,),\#\}$;

6. $\forall A\in N$: $F_{i+1}(A)=F_i(A)$

7. F(S)={),#}; F(R)={),#}; F(T)={+,-,),#}; F(F)={+,-,),#}; F(E)={+,-,*,/,),#}

A→α: DS(A,α) = S(α)∪F(A). S(α)={a | a∈T, α∈V$^+$, α⇒*aγ, γ∈V F(A)={a | a∈T, S⇒*ωAaγ, S- аксиома, A∈N, ω,γ∈V*, α⇒*λ}.

S→TR; DS(S,TR)=S(TR)={(,a,b,}
R→λ ; DS(R,λ)=S(λ)∪F(R)={),#} R→+TR;DS(R,+TR)=S(+TR)={+} R→-TR; DS(R,-TR)=S(-TR)={-}
T→EF; DS(T,EF)=S(EF)={(,a,b}
F→λ; DS(F, λ)=S(λ)∪F(F)={+,-,),#} F→*EF; DS(F,*EF)=S(*EF)={*} F→/EF; DS(F,/EF)=S(/EF)={/}
E→(S); DS(E,(S))=S((S))={(} E→a; DS(E,a)=S(a)={a} E→b; DS(E,b)=S(b)={b}

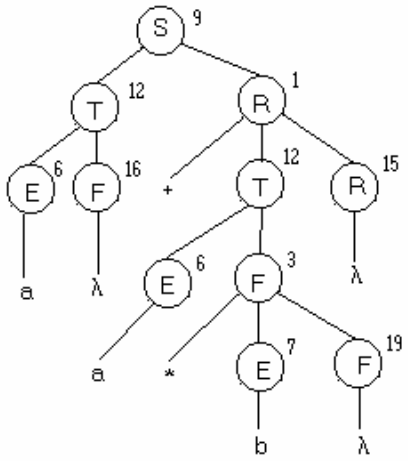
DS(R,λ)∩ DS(R,+TR)=∅; DS(R,λ)∩ DS(R,-TR)=∅; DS(R,-TR)∩ DS(R,+TR)=∅;
DS(F,λ)∩ DS(F,*EF)=∅; DS(F,λ)∩ DS(F,/EF)=∅; DS(F,*EF)∩ DS(F,/EF)=∅;
DS(E,(S))∩ DS(E,a)=∅; DS(E,(S))∩ DS(E,b)=∅; DS(E,a)∩ DS(E,b)=∅; Множества направляющих символов для каждого из нетерминалов не пересекаются. Это LL(1) грамматика.

Алгоритм построения распознавателя для LL(1) грамматики. R(Q,A,Z,δ, q_0, z_0, F) Вершина стека слева. В грамматике не должно быть левой рекурсии (поскольку грамматика с левой рекурсией не является LL(1) грамматикой)

4. Q={q}; A=T∪{#}; Z=N∪{t | t∈T, B→tα∉P, B∈N, α∈V*}∪{#}; q_0=q; z_0={S#}. F=∅; Здесь S- аксиома, # - символ конца строки.
5. ∀p∈P: X→aβ, a∈T, X∈N, β∈V* ⇒ δ(q, a, X)=(q,β);сдвиг. Эти функции позволяют замещать нетерминал на вершине стека на цепочку;
6. ∀p∈P: X→Aβ, X,A∈N, β∈V* ⇒ ∀a∈DS(X,Aβ) δ(q, a, X)=(q, Aβ); нет сдвига;
7. ∀p∈P: X→λ, X∈N ⇒ ∀a∈DS(X,λ) δ(q,a,X)=(q,λ); нет сдвига;
8. ∀ t: t∈T, A→tα∉P, A∈N, α∈V* ⇒ δ(q,t,t)=(q,λ); сдвиг. Эти функции позволяют вытолкнуть из стека терминал совпадающий с входным символом.

На каждом такте читается текущий входной символ и символ на вершине стека, и в соответствии с функцией перехода выполняются соответствующие действия, при этом учитывается, необходимо ли перемещать считывающую головку. Если автомат переходит в конфигурацию (q,#,#), то строка принимается, если он попадает в ситуацию, для которой не определено функции перехода, строка не принимается.

Пример. Грамматика из предыдущего примера. Цепочка a+a* и a+a*b.



G({+,-,/,*,a,b,(,)},{S,R,T,F,E},P,S),
P: {S→TR; R→λ | +TR | -TR; T→EF; F→λ | *EF | /EF; E→ (S) | a | b}
1. A={+,-,/,*,a,b,(,),#}; Z={S,R,T,F,E,),#}; Q={q}; q_0=q; z_0={S#}
2.1. R→ +TR ⇒ δ(q,+,R)=(q,TR);сдвиг 1
2.2. R→ -TR ⇒ δ(q,-,R)=(q,TR);сдвиг 2
2.3. F→ *EF ⇒ δ(q,*,F)=(q,EF);сдвиг 3
2.4. F→ /EF ⇒ δ(q,/,F)=(q,EF);сдвиг 4
2.5. E→ (S) ⇒ δ(q,(,E)=(q,S));сдвиг 5
2.6. E→ a ⇒ δ(q,a,E)=(q,λ);сдвиг 6
2.7. E→ b ⇒ δ(q,b,E)=(q, λ);сдвиг 7
3.1. S→TR ⇒ DS(S,TR)={(,a,b} ⇒ δ(q,(,S)=(q,TR) (8); δ(q,a,S)=(q,TR) (9); δ(q,b,S)=(q,TR) (10);
3.2. T→EF ⇒ DS(T,EF)={(,a,b} ⇒ δ(q,(,T)=(q,EF) (11); δ(q,a,T)=(q,EF) (12); δ(q,b,T)=(q,EF) (13);
4.1. R→λ ⇒ DS(R,λ)={),#} ⇒ δ(q,),R)=(q, λ) (14); δ(q,#,R)=(q, λ) (15);
4.2. F→λ ⇒ DS(F,λ)={+,-,),#} ⇒ δ(q,+,F)=(q, λ) (16); δ(q,-,F)=(q, λ) (17);δ(q,),F)=(q, λ) (18); δ(q,#,F)=(q, λ) (19);
5. δ(q,),))=(q, λ);

(q,a+a*#,S#)⇒^{9}(q,a+a*#,TR#)⇒^{12}(q,a+a*#,EFR#)⇒^{6}(q,+a*#,FR#)⇒^{16}(q,+a*#,R#)⇒^{1}(q,a*#,TR#)⇒^{12}(q,a*#,EFR#)⇒^{6}(q,*#,FR#)⇒^{3}(q,#,EFR#)⇒переход не определен, строка не принимается

(q,a+a*b#,S#)⇒^{9}(q,a+a*b#,TR#)⇒^{12}(q,a+a*b#,EFR#)⇒^{6}(q,+a*b#,FR#)⇒^{16}(q,+a*b#,R#)⇒^{1}(q,a*b#,TR#)⇒^{12}(q,a*b#,EFR#)
⇒^{6}(q,*b#,FR#)⇒^{3}(q,b#,EFR#)⇒^{7}(q,#,FR#)⇒^{19}(q,#,R#)⇒^{15}(q,#,#)⇒автомат перешел в конечную конфигурацию, строка принимается. Дерево разбора для этого варианта.

LR(k) грамматики

LR(k) грамматики. Понятие основы. Пополненная КС-грамматика. LR(0) и LR(1) грамматики. Понятие ситуации. Построение последовательности ситуаций для LR(0) грамматики. Построение управляющей таблицы распознавателя LR(0) грамматики. Последовательность ситуаций для LR(1) грамматики. Управляющая таблица распознавателя LR(1) грамматики. SLR(1) и LALR(1) грамматики. Распознаватель SLR(1) грамматики. Иерархия КС-грамматик. Программа YACC

LR(k)-грамматики. Распознаватели этих грамматик просматривают k символов входной цепочки слева направо и строят правосторонний вывод. Грамматики LR(k) класса обеспечивают детерминированный восходящий разбор. При этом требуется, чтобы при работе алгоритма "перенос-свертка" на каждом шаге можно было однозначно ответить на вопросы:

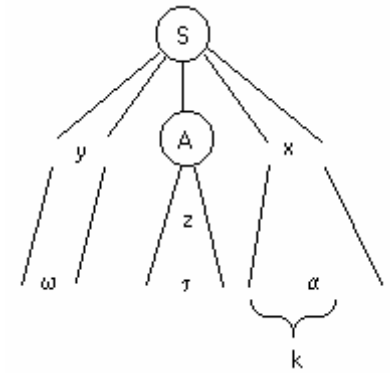
- какая операция (перенос или свертка) должна быть выполнена
- какой длины цепочку брать из стека для свертки
- какое правило брать для свертки с цепочкой α, если существует несколько альтернатив вида $A_1\to\alpha$, $A_2\to\alpha$,…, $A_n\to\alpha$

КС-грамматика G называется LR(k)-грамматикой, если для некоторого $k\geq 0$ на каждом шаге вывода для однозначного решения вопроса о выполняемом действии в алгоритме "перенос-свертка" достаточно знать содержимое верхней части стека и k-первых символов входной цепочки.

Основа цепочки α - это вхождение правой части последнего примененного правила в правом выводе этой цепочки. Пусть G(N,T,P,S) – КС-грамматика, в которой имеется правый вывод: $S \Rightarrow^* \alpha A\omega \Rightarrow \alpha\beta\omega \Rightarrow^* \gamma\omega$, $A\in N$, $\alpha,\beta\in V^*$, $\gamma,\omega\in T^*$. Тогда правовыводимая сентенциальная форма $\alpha\beta\omega$ левосвертываема к правовыводимой сентенциальной форме $\alpha A\omega$ с помощью правила $A\to\beta$, где β является основой. Т.о. основа – это самая левая подлежащая свертке подстрока сентенциальной формы правостороннего вывода.

Пример: Грамматика G({S,L,I},{i, , ,r},{S→rL; L→L,I | I; I→i},S). Для входной цепочки ri,i правый вывод будет иметь вид: $S \Rightarrow^1 rL \Rightarrow^2 rL,I \Rightarrow^4 rL,i \Rightarrow^3 rI,i \Rightarrow^4 ri,i$. Просмотр вывода в обратном порядке можно интерпретировать как разбор входной цепочки: ri,i ⇒ rI,i ⇒ rL,i ⇒ rL,I ⇒ rL ⇒ S, при этом на каждом шаге вхождение правой части некоторого правила (основа) заменяется нетерминалом из левой части правила: r[i],i ⇒ r[I],i ⇒ rL,[i] ⇒ r[L,I] ⇒ [rL] ⇒ S

При работе распознавателя выполняется перенос входных символов в стек до тех пор, пока на вершине стека не окажется основа, к которой затем и применяется свертка.



Рассмотрим частичное дерево вывода для некоторой LR(k)-грамматики. Здесь ω - уже разобранная часть входной цепочки ωτα, построенная на основе левой части дерева y. Правая часть дерева x – еще не разобранная часть, A – текущий нетерминальный символ, к которому на очередном шаге будет свернута цепочка символов z, находящаяся на вершине стека Мп-автомата. В нее входит прочитанная, но пока не разобранная часть входной цепочки τ. Однозначный выбор на каждом шаге алгоритма может быть сделан на основе цепочки τ и k первых символов цепочки α. Действием на каждом такте будет являться либо свертка цепочки z к нетерминалу A, либо перенос первого символа из α. и добавление его к z.

Основные свойства LR(k) грамматик:

- любая LR(k)-грамматика для $k\geq 0$ является однозначной;
- существует алгоритм, позволяющий проверить, является ли заданная грамматика LR(k)-грамматикой для строго определенного k.

Однако:

- не существует алгоритма проверки, является ли заданная КС-грамматика LR(k)-грамматикой для произвольного числа k;
- не существует алгоритма преобразования произвольной КС-грамматики к виду LR(k) грамматики для некоторого k

Пополненная КС-грамматика. Грамматика G' называемая пополненной, строится на основе грамматики G(N,T,P,S) следующим образом:

- G'=G, если аксиома S не встречается в правых частях правил: не $\exists\, p\in P$: $A\to\alpha S\beta$, $A,S\in N$, $\alpha,\beta\in V^*$;
- G'={N∪{S'},T,P∪{S'→S},S'}, если аксиома S встречается в правых частях правил: $\exists\, p\in P$: $A\to\alpha S\beta$, $A,S\in N$, $\alpha,\beta\in V^*$.

Пополненная грамматика необходима для того, чтобы свертка к целевому символу (аксиоме) служила для распознавателя сигналом завершением алгоритма. т.е. не возникало неоднозначности – продолжать ли разбор или нет, если была выполнена свертка к аксиоме.

Время распознавания для LR(k)грамматики линейно зависит от длины входной цепочки. Для любого детерминированного КС-языка может быть построена LR(1) грамматика, задающая этот язык. Поэтому на практике используются LR(0) и LR(1) грамматики, для k>1 LR(k) грамматики практически не применяются. Любая LR(1) грамматика задает детерминированный КС-язык. Если бы проблема преобразования КС-грамматик была решена, распознаватели LR(1) грамматик могли бы стать универсальным механизмом при построении трансляторов. Класс LR-грамматик значительно шире, чем класс LL-грамматик. Для любого КС-языка, заданного LL-грамматикой, может быть построена LR-грамматика, задающая тот же язык. Но не наоборот. При этом вовсе необязательно, что LL(k) грамматике будет соответствовать LR(k) грамматика, т.е число k может быть другим.

Для LR(0) грамматики текущий символ входной цепочки не участвует в анализе. Решение принимается только на основе содержимого стека. При этом требуется обеспечить непротиворечивость управляющей таблицы распознавателя, т.е. не должно возникать конфликта между выполняемым действием (сдвиг или свертка) и между различными вариантами при выполнении свертки. При выполнении свертки к какому-либо нетерминалу, в стеке перед ним будут находиться только те символы, которые могут встретиться слева от него. Это т.н. **левый контекст**. Для LR(0) грамматики во внимание принимается

только он. Очевидно, что непротиворечивость выбора на основе только левого контекста обеспечить сложнее, чем на основе левого и правого контекстов. Поэтому класс LR(0) грамматик уже, чем класс LR(1) грамматик, для которых учитывается и правый контекст тоже.

Ситуация представляет собой множество правил КС-грамматики, в которых указывается положение считывающей головки МП-автомата, которое может возникнуть при разборе сентенциальной формы этой грамматики. Это положение обозначается в правой части правила специальным символом •. Этот символ не входит в алфавит грамматики. Если S – аксиома некоторой Для КС-грамматики G(T, N, P, S), где $S\to\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n \in P$, $\alpha_i \in V^*$, **начальной ситуацией** будет множество правил $R=\{S\to\bullet\alpha_1 , S\to\bullet\alpha_2 , \dots , S\to\bullet\alpha_n \}$. Последовательность ситуаций строится по следующим правилам:

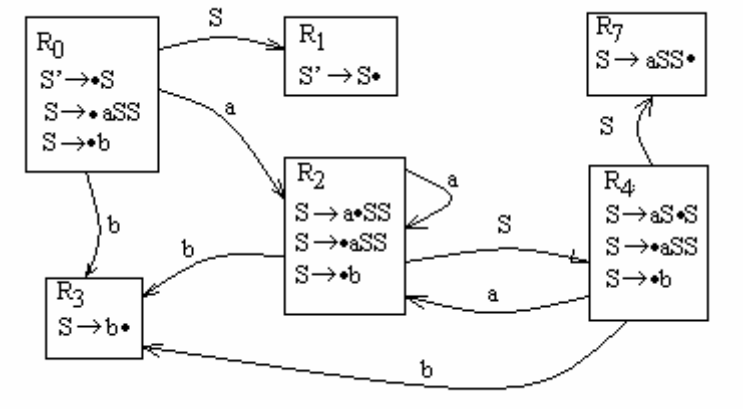
1. $\forall\ r\in R : A\to\gamma\bullet B\beta$, $A,B\in N$, $\gamma,\beta\in V^*$, $\forall p\in P : B\to\alpha$, $\alpha\in V^*$ $\Rightarrow$ $R=R\cup\{B\to\bullet\alpha\}$, т.е. ситуация пополняется новыми правилами
2. $\forall\ r\in R : A\to\gamma\bullet x\beta$, $A\in N$, $\gamma,\beta\in V^*$, $x\in V$ $\Rightarrow$ $R'=\{A\to\gamma x\bullet\beta\}$, и $R\to^x R'$ т.е. строится новая ситуация, знак $\to^x$ означает, что новая ситуация следует из текущей по символу x.

Множество связанных между собой по разным символам ситуаций и является последовательностью ситуаций. Она может быть изображена в виде графа. Поскольку количество правил грамматики конечно, то и последовательность ситуаций тоже конечна. Для построения управляющего автомата все ситуации в последовательности нумеруются: $R_0, R_1, R_2, \dots, R_n$. Здесь R_0 – начальная ситуация. Пример:

G=({a,b},{S}, {S→aSS | b},S). Построим последовательность ситуаций.
Вначале строим пополненную грамматику, поскольку аксиома встречается в правой части правил:
G'=({a,b},{S,S'}, {S'→S, S→aSS | b},S').

0. $R_0=\{ S'\to\bullet S\}$.
1. Поскольку существуют правила для S, то $R_0= R_0\cup\{ S\to\bullet aSS , S\to\bullet b\}$ $R_0=\{ S'\to\bullet S, S\to\bullet aSS , S\to\bullet b\}$
2. $R_0 \to^s R_1 = \{ S'\to S\bullet\}$ Других правил в эту ситуацию добавить нельзя
2. $R_0 \to^a R_2 = \{ S\to a\bullet SS \}$.
1. Поскольку существуют правила для S, то $R_2= R_2\cup\{ S\to\bullet aSS , S\to\bullet b\}$ $R_2=\{ S\to a\bullet SS, S\to\bullet aSS , S\to\bullet b\}$
2. $R_0 \to^b R_3 = \{ S\to b\bullet \}$. Других правил в эту ситуацию добавить нельзя
2. $R_2 \to^s R_4 = \{ S\to aS\bullet S \}$
1. Поскольку существуют правила для S, то $R_4= R_4\cup\{ S\to\bullet aSS , S\to\bullet b\}$ $R_4=\{ S\to aS\bullet S, S\to\bullet aSS , S\to\bullet b\}$
2. $R_2 \to^a R_5 = \{ S\to a\bullet SS \}$.
1. Поскольку существуют правила для S, то $R_5= R_5\cup\{ S\to\bullet aSS , S\to\bullet b\}$ $R_5=\{ S\to a\bullet SS, S\to\bullet aSS , S\to\bullet b\}=R_2$
2. $R_2 \to^b R_6 = \{ S\to b\bullet \}$. Других правил в эту ситуацию добавить нельзя $R_6=R_3$
2. $R_4 \to^s R_7 = \{ S\to aSS\bullet \}$ Других правил в эту ситуацию добавить нельзя
2. $R_4 \to^a R_8 = \{ S\to a\bullet SS \}$.
1. Поскольку существуют правила для S, то $R_8= R_8\cup\{ S\to\bullet aSS , S\to\bullet b\}$ $R_8=\{ S\to a\bullet SS, S\to\bullet aSS , S\to\bullet b\}=R_2$
2. $R_4 \to^b R_9 = \{ S\to b\bullet \}$. Других правил в эту ситуацию добавить нельзя $R_9=R_3$

Итого получилось 6 ситуаций: 0,1,2,3,4,7. Граф:



Построение распознавателя. Распознаватель для LR(0) грамматики функционирует на основе управляющей таблицы. Она строится следующим образом: строками являются все ситуации грамматики, столбцы содержат все символы словаря (как терминальные так и нетерминальные и маркер конца строки). Элементом ij таблицы является операция, которую будет выполнять автомат, если он находится в состоянии i, а текущим символом является j. Автомат использует 2 стека – стек символов и стек состояний. Возможны 4 типа операции:

- перенос(сдвиг) П(s). Текущий символ помещается в стек символов. Если это терминал – перемещение головки чтения. Символ входной строки фиксируется в качестве текущего. В стек состояний заносится s. Автомат переходит в состояние s.
- свертка С(n, A, k). Свертка по правилу k: $A\to\alpha$. Из вершин обоих стеков удаляется по n символов, где $n = |\alpha|$. Нетерминал A фиксируется в качестве текущего символа. Номер правила (k) заносится в разбор грамматики. Переход в состояние, указанное на вершине стека состояний.

- Ошибка E. Разбор не может быть продолжен. Строка отвергается.
- Конец. Разбор окончен, строка принята.

Управляющая таблица $U = \{u_{ij}\}$ строится на основе последовательности ситуаций.

1. Для каждой ситуации R_i создается строка таблицы i.
2. Для каждой ситуации R_i выполняется следующее:
   - $\forall r \in R_i : A \to \gamma\bullet, \gamma \in V^*, A \in N$, k – номер правила $A \to \gamma \Rightarrow u_{ij} = П(|\gamma|, A, k)$, для всех $j \neq S$
   - $\forall r \in R_i : A \to \gamma\bullet x\beta, \gamma,\beta \in V^*, x \in V, A \in N, R_i \to^x R_j \Rightarrow u_{ix} = C(j)$
3. u_{0S} = Конец
4. Во все оставшиеся пустыми ячейки таблицы заносится Ошибка. При этом каждой ячейке можно поставить в соответствие свое диагностическое сообщение, локализующее и классифицирующее ошибку.

Если таблицу удалось заполнить непротиворечивым образом, т.е. для одной и той же ячейки таблицы нет взаимоисключающих действий, то рассматриваемая грамматика является LR(0) грамматикой. Перед началом работы распознаватель устанавливается в состояние 0, головка чтения – на первый символ входной строки, который принимается в качестве текущего. Стек символов содержит #, стек состояний 0. Дальнейшее функционирование происходит на основе таблицы. Пример:

Ситуация	S’	S	a	b	#
0	Конец	П(1)	П(2)	П(3)	E
1	E	C (1, S’, 1)	C (1, S’, 1)	C (1, S’, 1)	C (1, S’, 1)
2	E	П(4)	П(2)	П(3)	E
3	E	C (1, S, 3)	C (1, S, 3)	C (1, S, 3)	C (1, S, 3)
4	E	П(7)	П(2)	П(3)	E
7	E	C (3, S, 2)	C (3, S, 2)	C (3, S, 2)	C (3, S, 2)

Пример разбора цепочки abababb:

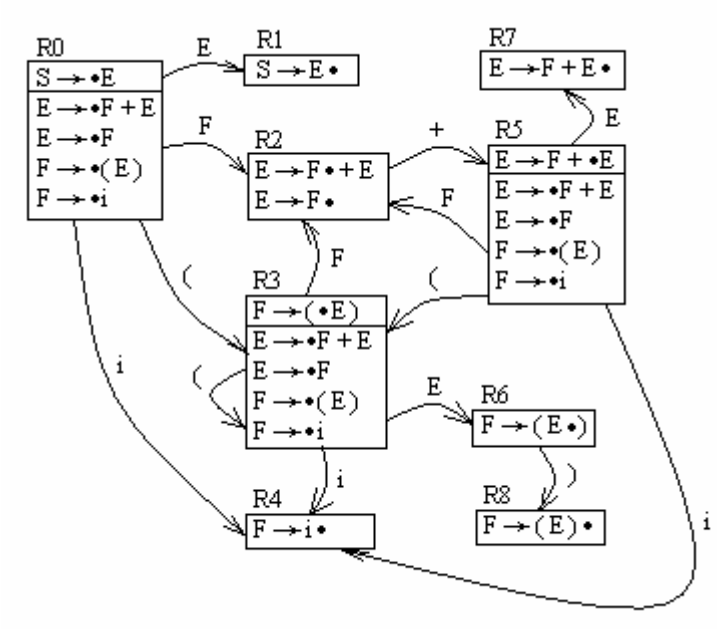
Входная строка	Состояние	Текущий символ	Стек символов	Стек состояний	Разбор
abababb#	0	a	#	0	
bababb#	2	b	#a	02	
ababb#	3	a	#ab	023	
ababb#	2	S	#a	02	3
ababb#	4	a	#aS	024	3
babb#	2	b	#aSa	0242	3
abb#	3	a	#aSab	02423	3
abb#	2	S	#aSa	0242	33
abb#	4	a	#aSaS	02424	33
bb#	2	b	#aSaSa	024242	33
b#	3	b	#aSaSab	0242423	33
b#	2	S	#aSaSa	024242	333
b#	4	b	#aSaSaS	0242424	333
#	3	#	#aSaSaSb	02424243	333
#	4	S	#aSaSaS	0242424	3333
#	7	#	#aSaSaSS	02424247	3333
#	4	S	#aSaS	02424	33332
#	7	#	#aSaSS	024247	33332
#	4	S	#aS	024	333322
#	7	#	#aSS	0247	333322
#	0	S	#	0	3333222
#	1	#	#S	01	3333222
#	0	S’	#	0	33332221

Разберем еще один **пример**. G({+, (,), i},{S, E, F},{S→E; E→F+E; E→F; F→(E); F→i}, S)

0. R_0={ S→•E}.
1. Поскольку существуют правила для E, то $R_0 = R_0 \cup$ { E→•F+E , E→•F} R_0={ S→•E, E→•F+E, E→•F }
1. Поскольку существуют правила для F, то $R_0 = R_0 \cup$ { F→•(E) , F→•i} R_0={ S→•E, E→•F+E, E→•F, F→•(E) , F→•i}
2. $R_0 \to^E R_1$ = { S→E•}Других правил в эту ситуацию добавить нельзя
2. $R_0 \to^F R_2$ = { E→F•+E, E→F• } Других правил в эту ситуацию добавить нельзя
2. $R_0 \to^(R_3$ = { F→(•E)}
1. Поскольку существуют правила для E, то $R_3 = R_3 \cup$ { E→•F+E , E→•F} R_3={ F→(•E), E→•F+E , E→•F }
1. Поскольку существуют правила для F, то $R_3 = R_3 \cup$ { F→•(E) , F→•i} R_3={ F→(•E), E→•F+E , E→•F, F→•(E) , F→•i }
2. $R_0 \to^i R_4$ = { F→i•} Других правил в эту ситуацию добавить нельзя
2. $R_2 \to^+ R_5$ = { E→F+•E }
1. Поскольку существуют правила для E, то $R_5 = R_5 \cup$ { E→•F+E , E→•F} R_5={ E→F+•E, E→•F+E , E→•F }
1. Поскольку существуют правила для F, то $R_5 = R_5 \cup$ { F→•(E) , F→•i} R_5={ E→F+•E, E→•F+E , E→•F, F→•(E) , F→•i }
2. $R_3 \to^E R_6$ = { F→(E•) } Других правил в эту ситуацию добавить нельзя
2. $R_3 \to^F$ R = { E→F•+E, E→F• } Других правил в эту ситуацию добавить нельзя, она совпадает с R_2
2. $R_3 \to^($ R = { F→(•E)}

1. Поскольку существуют правила для E, то R= R∪{ E→•F+E , E→•F} R={ F→(•E), E→•F+E , E→•F }
1. Поскольку существуют правила для F, то R= R∪{ F→•(E) , F→•i} R={ F→(•E), E→•F+E , E→•F, F→•(E) , F→•i }
ситуация совпадает с R_3
2. $R_3 \rightarrow^{i} R$ = { F→i• } Других правил в эту ситуацию добавить нельзя, она совпадает с R_4
2. $R_5 \rightarrow^{E} R_7$ = { E→F+E• } Других правил в эту ситуацию добавить нельзя
2. $R_5 \rightarrow^{F} R$ = { E→F•+E, E→F• } Других правил в эту ситуацию добавить нельзя, она совпадает с R_2
2. $R_5 \rightarrow^{(} R$ = { F→(•E)}
1. Поскольку существуют правила для E, то R= R∪{ E→•F+E , E→•F} R={ F→(•E), E→•F+E , E→•F }
1. Поскольку существуют правила для F, то R= R∪{ F→•(E) , F→•i} R={ F→(•E), E→•F+E , E→•F, F→•(E) , F→•i }
ситуация совпадает с R_3
2. $R_5 \rightarrow^{i} R$ = { F→i• } Других правил в эту ситуацию добавить нельзя, она совпадает с R_4
2. $R_6 \rightarrow^{)} R_8$ = { F→(E)•} Других правил в эту ситуацию добавить нельзя

Итого получилось 8 ситуаций. Граф:



Ситуация	S	E	F	+	(	)	i	#
0	Конец	П(1)	П(2)	Е	П(3)	Е	П(4)	Е
1	Е	С(1,S,1)	С(1,S,1)	С(1,S,1)	С(1,S,1)	С(1,S,1)	С(1,S,1)	С(1,S,1)
2	Е	С(1,E,3)	С(1,E,3)	С(1,E,3) П(5)	С(1,E,3)	С(1,E,3)	С(1,E,3)	С(1,E,3)
3	Е	П(6)	П(2)	Е	П(3)	Е	П(4)	Е
4	Е	С(1,F,5)	С(1,F,5)	С(1,F,5)	С(1,F,5)	С(1,F,5)	С(1,F,5)	С(1,F,5)
5	Е	П(7)	П(2)	Е	П(3)	Е	П(4)	Е
6	Е	Е	Е	Е	Е	П(8)	Е	Е
7	Е	С(3,E,2)	С(3,E,2)	С(3,E,2)	С(3,E,2)	С(3,E,2)	С(3,E,2)	С(3,E,2)
8	Е	С(3,F,4)	С(3,F,4)	С(3,F,4)	С(3,F,4)	С(3,F,4)	С(3,F,4)	С(3,F,4)

В позиции (2,+) возник конфликт перенос-свертка. Соответственно данная грамматика не является LR(0). Рассмотрим построение распознавателя для LR(1) грамматик.

Ситуация для LR(1) грамматики задается несколько сложнее. Если S – аксиома некоторой Для КС-грамматики G(T, N, P, S), где $S\rightarrow\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \in P$, $\alpha_i \in V^*$, **начальной ситуацией** будет множество правил R={$S\rightarrow\bullet\alpha_1$ /#, $S\rightarrow\bullet\alpha_2$ /#, … , $S\rightarrow\bullet\alpha_n$ /#}. Таким образом, в правилах ситуаций указывается еще и правый контекст. Последовательность ситуаций строится по следующим правилам:

1. $\forall r\in R : A\rightarrow\gamma\bullet Bb\beta$ /a, $A,B\in N$, $a,b\in T$, $\gamma,\beta\in V^*$, $\forall p\in P : B\rightarrow\alpha$, $\alpha\in V^* \Rightarrow$ R=R∪{$B\rightarrow\bullet\alpha$ /b}, т.е. ситуация пополняется новыми правилами с учетом правого контекста
2. $\forall r\in R : A\rightarrow\gamma\bullet B$ /a, $A,B\in N$, $a\in T$, $\gamma\in V^*$, $\forall p\in P : B\rightarrow\alpha$, $\alpha\in V^* \Rightarrow$ R=R∪{$B\rightarrow\bullet\alpha$ /a}
3. $\forall r\in R : A\rightarrow\gamma\bullet BC\beta$ /a, $A,B,C\in N$, $a\in T$, $\gamma,\beta\in V^*$, $\forall p\in P : B\rightarrow\alpha$, $\alpha\in V^*$, $\forall c\in S(C) \Rightarrow$ R=R∪{$B\rightarrow\bullet\alpha$ /c}, где S(C) – Множество символов предшественников
4. $\forall r\in R : A\rightarrow\gamma\bullet x\beta$ /a, $A\in N$, $\gamma,\beta\in V^*$, $a\in T$, $x\in V \Rightarrow$ R'={$A\rightarrow\gamma x\bullet\beta$ /a}, и $R\rightarrow^{x}R'$

Пример: G({+, (,), i},{S, E, F},{S→E; E→F+E; E→F; F→(E); F→i}, S)

0. R_0={ S→•E /#}
1. Поскольку существуют правила для E, то R_0= R_0∪{ E→•F+E /# , E→•F /#} R_0={ S→•E /#, E→•F+E /# , E→•F /#}
1. Поскольку существуют правила для F, то R_0= R_0∪{ F→•(E) /+ , F→•i /+, F→•(E) /# , F→•i /# }

R_0={ S→•E /#, E→•F+E /# , E→•F /#, F→•(E) /+ , F→•i /+, F→•(E) /# , F→•i /# }

2. $R_0 \to^E R_1$ = { S→E• /#} Других правил в эту ситуацию добавить нельзя

2. $R_0 \to^F R_2$ = { E→F•+E /#, E→F• /# } Других правил в эту ситуацию добавить нельзя

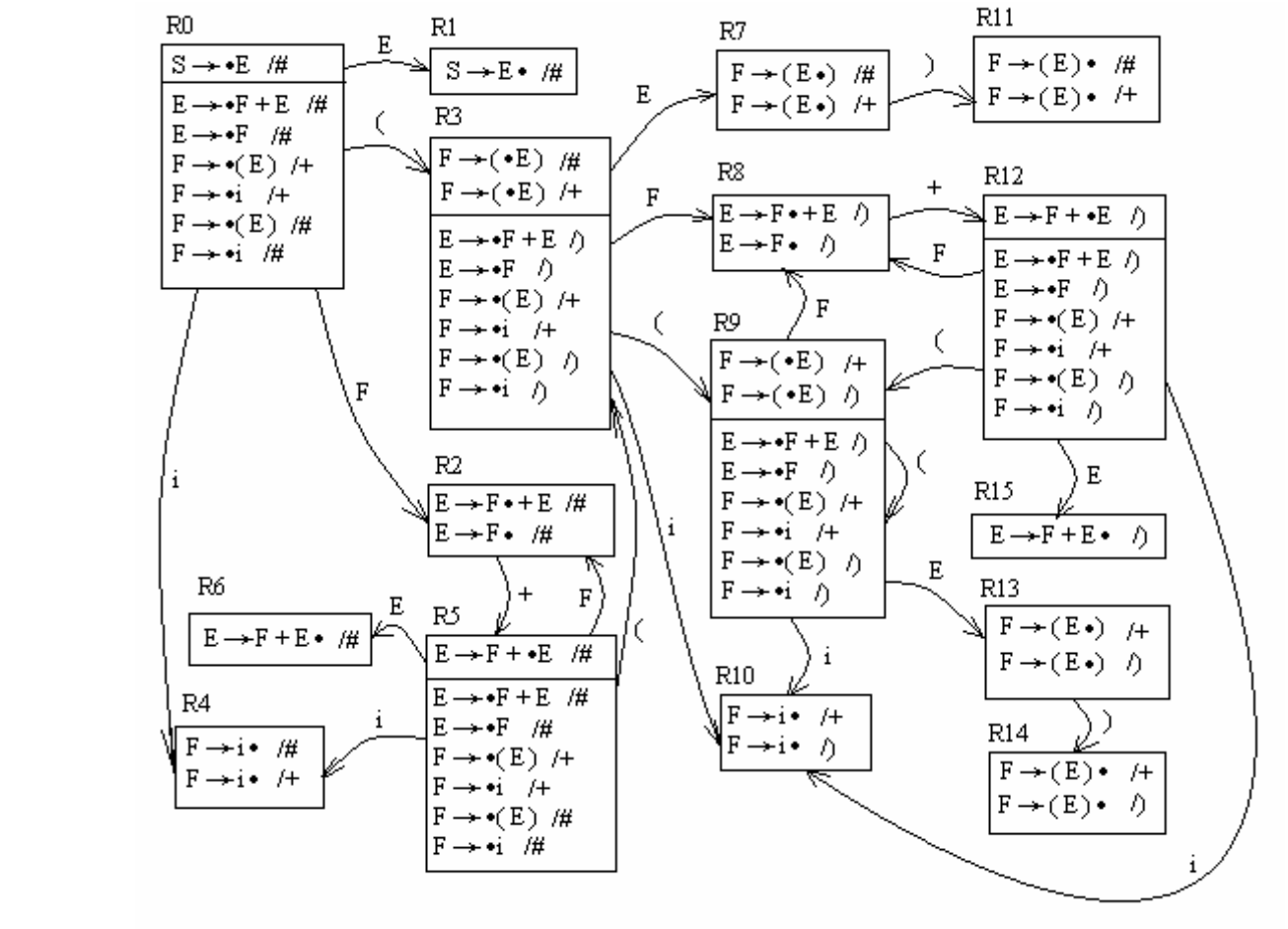
2. $R_0 \to^(R_3$ = { F→(•E) /+ , F→(•E) /# }

1. Т.к. существуют правила для E, то R_3= $R_3 \cup$ { E→•F+E /) , E→•F /)} R_3={ F→(•E) /+ , F→(•E) /#, E→•F+E /) , E→•F /) }

1. Поскольку существуют правила для F, то R_3= $R_3 \cup$ { F→•(E) /+ , F→•i /+, F→•(E) /) , F→•i /) }

R_3={ F→(•E) /+ , F→(•E) /#, E→•F+E /) , E→•F /), F→•(E) /+ , F→•i /+, F→•(E) /) , F→•i /) }

Продолжая построения аналогичным образом, получим 15 ситуаций. Граф:



Построение распознавателя для LR(1) грамматики. Распознаватель для LR(1) грамматики строится по похожим принципам, что и для LR(0). Управляющая таблица U = {u_{ij}} строится следующим образом.

1. Для каждой ситуации R_i создается строка таблицы i.
2. Для каждой ситуации R_i выполняется следующее:
- $\forall$ r $\in R_i$: A→γ• /a, γ$\in V^*$, A$\in$N, a$\in$T, k – номер правила A→γ $\Rightarrow u_{ia} = \Pi(|\gamma|, A, k)$
- $\forall$ r $\in R_i$: A→γ•xβ /a, γ,β$\in V^*$, x$\in$V, A$\in$N, a$\in$T, $R_i \to^x R_j \Rightarrow u_{ix} = C(j)$

2. u_{0S} = Конец

3. Во все оставшиеся пустыми ячейки таблицы заносится Ошибка.

Если таблицу удалось заполнить непротиворечивым образом, то рассматриваемая грамматика является LR(1) грамматикой. Перед началом работы распознаватель устанавливается в состояние 0, головка чтения – на первый символ входной строки, который принимается в качестве текущего. Стек символов содержит #, стек состояний 0. Дальнейшее функционирование происходит на основе таблицы. Пример:

Ситуация	**S**	**E**	**F**	**+**	**(**	**)**	**i**	**#**
0	Конец	П(1)	П(2)	Е	П(3)	Е	П(4)	Е
1	Е	Е	Е	Е	Е	Е	Е	С(1,S,1)
2	Е	Е	Е	П(5)	Е	Е	Е	С(1,E,3)
3	Е	П(7)	П(8)	Е	П(9)	Е	П(10)	Е
4	Е	Е	Е	С(1,F,5)	Е	Е	Е	С(1,F,5)
5	Е	П(6)	П(2)	Е	П(3)	Е	П(4)	Е
6	Е	Е	Е	Е	Е	Е	Е	С(3,E,2)
7	Е	Е	Е	Е	Е	П(11)	Е	Е
8	Е	Е	Е	П(12)	Е	С(1,E,3)	Е	Е
9	Е	П(13)	П(8)	Е	П(9)	Е	П(10)	Е
10	Е	Е	Е	С(1,F,5)	Е	С(1,F,5)	Е	Е
11	Е	Е	Е	С(3,F,4)	Е	Е	Е	С(3,F,4)

12	E	П(15)	П(8)	E	П(9)	E	П(10)	E
13	E	E	E	E	E	П(14)	E	E
14	E	E	E	C(3,F,4)	E	C(3,F,4)	E	E
15	E	E	E	E	E	C(3,E,2)	E	E

Как видно из примера, число состояний значительно увеличилось, более того, число элементов в каждой ситуации тоже больше, чем для LR(0) грамматики. Посмотрим, как происходит разбор строки (i)+i:

Входная строка	**Состояние**	**Текущий символ**	**Стек символов**	**Стек состояний**	**Разбор**
(i)+i#	0	(	#	0	
i)+i#	3	i	#(	0 3	
)+i#	10	)	#(i	0 3 10	
)+i#	3	F	#(	0 3	5
)+i#	8	)	#(F	0 3 8	5
)+i#	3	E	#(	0 3	5 3
)+i#	7	)	#(E	0 3 7	5 3
+i#	11	+	#(E)	0 3 7 11	5 3
+i#	0	F	#	0	5 3 4
+i#	2	+	#F	0 2	5 3 4
i#	5	i	#F+	0 2 5	5 3 4
#	4	#	#F+i	0 2 5 4	5 3 4
#	5	F	#F+	0 2 5	5 3 4 5
#	2	#	#F+F	0 2 5 2	5 3 4 5
#	5	E	#F+	0 2 5	5 3 4 5 3
#	6	#	#F+E	0 2 5 6	5 3 4 5 3
#	0	E	#	0	5 3 4 5 3 2
#	1	#	#E	0 1	5 3 4 5 3 2
#	0	S	#	0	5 3 4 5 3 2 1

Конфликт при построении LR(0) распознавателя в одной-единственной ячейке привел к необходимости построения LR(1) грамматики, для которой и ситуации строятся значительно сложнее и число состояний значительно возрастает. Поиск методов упрощения построения распознавателей при подобных конфликтах привел к появлению SLR(1) (от simple - простая) и LALR(1) (от look ahead – заглядывание вперед) грамматик. Если проанализировать конфликтную позицию (2,+) {C(1,E,3), П(5)} таблицы LR(0) распознавателя, то видно, что свертка должна производиться только при последующем символе #, т.е. конфликт разрешается в пользу переноса. Грамматики, для которых подобный анализ позволяет устранить конфликты, относятся к классу SLR(1) грамматик. LALR(1) грамматики используют более сложный анализ контекста.

Формально ситуации для **SLR(1) грамматик** строятся так же, как и для LR(0) грамматик, но управляющая таблица заполняется иначе:

Управляющая таблица $U = \{u_{ij}\}$ строится на основе последовательности ситуаций.

1. Для каждой ситуации R_i создается строка таблицы i.
2. Для каждой ситуации R_i выполняется следующее:
   - $\forall r \in R_i : A \to \gamma\bullet,\ \gamma \in V^*$, $A \in N$, k – номер правила $A \to \gamma \Rightarrow u_{ij} = П(|\gamma|, A, k)$, для всех $j \in F(A)$, где F(A) – множество символов-последователей (этот пункт можно оставить таким же, как и для грамматики LR(0), а изменения принимать во внимание только для конфликтных строк)
   - $\forall r \in R_i : A \to \gamma\bullet x\beta,\ \gamma,\beta \in V^*$, $x \in V$, $A \in N$, $R_i \to^x R_j \Rightarrow u_{ix} = C(j)$
3. u_{0S} = Конец
4. Во все оставшиеся пустыми ячейки таблицы заносится Ошибка.

Если таблицу удалось заполнить непротиворечивым образом, то рассматриваемая грамматика является SLR(1) грамматикой. Перед началом работы распознаватель устанавливается в состояние 0, головка чтения – на первый символ входной строки, который принимается в качестве текущего. Стек символов содержит #, стек состояний 0. Дальнейшее функционирование происходит на основе таблицы. Пример:

Пример. Возьмем имеющуюся конфликтную последовательность ситуаций. Сначала построим множество символов – последователей для каждого из нетерминалов: F(S)={#}; F(E)={),#}; F(F)={+,),#}.

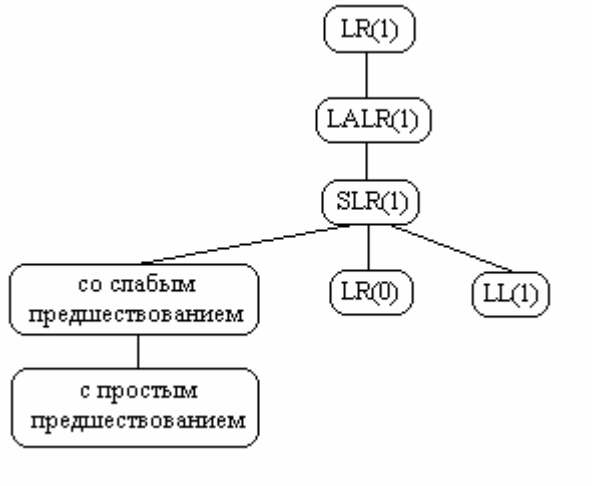
Заполним управляющую таблицу:

Ситуация	**S**	**E**	**F**	**+**	**(**	**)**	**i**	**#**
0	Конец	П(1)	П(2)	E	П(3)	E	П(4)	E
1	E	E	E	E	E	E	E	C(1,S,1)
2	E	E	E	П(5)	E	C(1,E,3)	E	C(1,E,3)
3	E	П(6)	П(2)	E	П(3)	E	П(4)	E
4	E	E	E	C(1,F,5)	E	C(1,F,5)	E	C(1,F,5)
5	E	П(7)	П(2)	E	П(3)	E	П(4)	E
6	E	E	E	E	E	П(8)	E	E
7	E	E	E	E	E	C(3,E,2)	E	C(3,E,2)
8	E	E	E	C(3,F,4)	E	C(3,F,4)	E	C(3,F,4)

Входная строка	Состояние	Текущий символ	Стек символов	Стек состояний	Разбор
(i)+i#	0	(	#	0	
i)+i#	3	i	#(	0 3	
)+i#	4	)	#(i	0 3 4	
)+i#	3	F	#(	0 3	5
)+i#	2	)	#(F	0 3 2	5
)+i#	3	E	#(	0 3	5 3
)+i#	6	)	#(E	0 3 6	5 3
+i#	8	+	#(E)	0 3 6 8	5 3
+i#	0	F	#	0	5 3 4
+i#	2	+	#F	0 2	5 3 4
i#	5	i	#F+	0 2 5	5 3 4
#	4	#	#F+i	0 2 5 4	5 3 4
#	5	F	#F+	0 2 5	5 3 4 5
#	2	#	#F+F	0 2 5 2	5 3 4 5
#	5	E	#F+	0 2 5	5 3 4 5 3
#	7	#	#F+E	0 2 5 7	5 3 4 5 3
#	0	E	#	0	5 3 4 5 3 2
#	1	#	#E	0 1	5 3 4 5 3 2
#	0	S	#	0	5 3 4 5 3 2 1

Как видим, разбор строки идентичен распознавателю LR(1) с точностью до номера состояния.

Рассмотрим **иерархию КС-грамматик:**



Чем уже класс грамматик, тем более простой распознаватель будем иметь на выходе, и тем проще его синтез. С другой стороны, приведение грамматики к более узкому классу требует больших усилий. Иногда, если часть правил не описывается выбранным классом грамматик, они анализируются отдельно, по дополнительному алгоритму.

Любая SLR(1) грамматика является LR(1) грамматикой, но не наоборот. Класс языков, задаваемый SLR(1) грамматикой уже, чем для LR(1), а это значит, что не всякий детерминированный КС-язык может быть задан SLR(1) грамматикой.

Любая SLR(1) грамматика является LALR(1) грамматикой, но не наоборот. Любой детерминированный КС-язык может быть задан LALR(1) грамматикой. Это значит, что классы языков, задаваемых LALR(1) и LR(1) грамматиками, совпадают, однако не всякая LR(1) грамматика является LALR(1) грамматикой.

С другой стороны, как правило, чем уже класс грамматик в этой иерархии, тем меньше возможностей по диагностированию ошибок он предоставляет, при этом снижается точность локализации и классификации ошибок.

Автоматизация построения синтаксических анализаторов решается с помощью различных пакетов, например **YACC** (Yet Another Compiler Compiler). Язык описывается в форме, близкой к форме Бэкуса-Наура, результатом является текст программы синтаксического анализатора. YACC реализует восходящий распознаватель на основе LALR(1) грамматики.

Грамматики предшествования

Грамматики предшествования. Грамматики простого предшествования. Отношения предшествования. Обратимые грамматики. Построение матрицы предшествования. Распознаватель для грамматики простого предшествования. Операторная грамматика. Грамматика операторного предшествования. Построение матрицы предшествования для грамматики операторного предшествования. Распознаватель для грамматик операторного предшествования. Остовная грамматика и остовный вывод

Грамматики предшествования также позволяют построить детерминированный анализатор на базе алгоритма перенос-свертка. Для этого для каждой упорядоченной пары символов в грамматике устанавливается отношение, называемое отношением предшествования. В процессе разбора МП-автомат сравнивает текущий символ входной цепочки с одним из символов на вершине стека и проверяет отношение между ними. В зависимости от этого выполняется либо перенос, либо свертка. Если отношение отсутствует, то генерируется ошибка. Различают грамматики простого, расширенного, слабого, операторного предшествования и др.

Приведенная КС-грамматика является **грамматикой простого предшествования**, если:

1. Для каждой упорядоченной пары терминальных и нетерминальных символов выполняется не более чем одно **отношение предшествования**:

$B_i =. B_j$ $(\forall B_i, B_j \in V) \Leftrightarrow \exists\, p \in P : A \to \alpha B_i B_j \gamma$, $A \in N$, $\alpha, \gamma \in V^*$

$B_i <. B_j$ $(\forall B_i, B_j \in V) \Leftrightarrow \exists\, p \in P : A \to \alpha B_i D \gamma$, $\exists\, D \Rightarrow^* B_j \omega$, $A, D \in N$, $\alpha, \gamma, \omega \in V^*$

$B_i .> B_j$ $(\forall B_i, B_j \in V) \Leftrightarrow \exists\, p \in P : A \to \alpha C B_j \gamma$, $\exists\, C \Rightarrow^* \omega B_i$, либо $\exists\, p \in P : A \to \alpha C D \gamma$, $\exists\, C \Rightarrow^* \omega B_i$, $\exists\, D \Rightarrow^* B_j \tau$, $A, C, D \in N$, $\alpha, \gamma, \omega, \tau \in V^*$

2. $\forall p_i \in P : A_i \to \alpha_i$, $\alpha_i \neq \alpha_j$, $i,j = 0,1,\ldots,|P|$, $i \neq j$. Т.е. правила имеют различные правые части. Грамматики, обладающие этим свойством, называют **обратимыми**.

Любая грамматика простого предшествования является однозначной. Легко проверить, является ли произвольная КС-грамматика грамматикой простого предшествования. Однако не существует алгоритма, позволяющего преобразовать произвольную КС-грамматику в грамматику простого предшествования.

Алгоритм основан на том факте, что между двумя соседними символами входной строки возможны следующие варианты отношений предшествования:

$B_i <. B_{i+1}$, если символ B_{i+1} крайний левый символ некоторой основы, т.е. B_i предшествует основе

$B_i .> B_{i+1}$, если символ B_i крайний правый символ некоторой основы, т.е. B_i следует за основой

$B_i =. B_{i+1}$, если символы B_i , B_{i+1} принадлежат одной основе

α	γ	β	δ
a		b	

<. =. .>

Рассмотрим рисунок. Изображена цепочка $\alpha\gamma\beta\delta$ в момент выполнения свертки подцепочки γ.

Символ a – последний в подцепочке α, b – первый в подстроке β. Если отношения предшествования в грамматике непротиворечивы, то сдвиг можно выполнять всегда, пока между символом на вершине стека и текущим символом входной цепочки существует отношение <. или =. . Как только обнаружено отношение .> , необходимо выполнить свертку, причем из стека выбираются все символы, связанные отношением =. . На основании отношений предшествования строится **матрица предшествования**. Строкам матрицы соответствует символ i пары (i,j), столбцам – j. Для построения матрицы удобно воспользоваться двумя дополнительными множествами:

$L(A) = \{X \mid \exists A \Rightarrow^* X\omega\}$, $A \in N$, $X \in V$, $\omega \in V^*$ - множество крайних левых символов для нетерминала A

$R(A) = \{X \mid \exists A \Rightarrow^* \omega X\}$, $A \in N$, $X \in V$, $\omega \in V^*$ - множество крайних правых символов для нетерминала A

Эти множества строятся по следующему алгоритму:

1. $\forall A \in N : R_0(A) = \{X \mid A \to \omega X\}$, $X \in V$, $\omega \in V^*$; $L_0(A) = \{X \mid A \to X\omega\}$, $X \in V$, $\omega \in V^*$; $i=1$;
2. $\forall A \in N : R_i(A) = R_{i-1}(A) \cup R_{i-1}(B)$, $\forall B \in (R_{i-1}(A) \cap N)$; $L_i(A) = L_{i-1}(A) \cup L_{i-1}(B)$, $\forall B \in (L_{i-1}(A) \cap N)$;
3. Если $\exists\, R_i(A) \neq R_{i-1}(A)$ или $\exists\, L_i(A) \neq L_{i-1}(A) \Rightarrow i=i+1$; повторить п.2.

Построение матрицы предшествования. Для построения матрицы предшествования $B=\{b_{ij}\}$ добавляют символы концевых маркеров (начала строки и конца строки). В качестве этих маркеров можно использовать один и тот же символ (#). Матрица заполняется на основе следующих выражений:

1. $\forall\, p \in P : A \to \alpha i j \gamma$, $A \in N$, $i,j \in V$, $\alpha, \gamma \in V^* \Rightarrow b_{ij} = =.$
2. $\forall\, p \in P : A \to \alpha i D \gamma$, $j \in L(D)$, $A, D \in N$, $i,j \in V$, $\alpha, \gamma \in V^* \Rightarrow b_{ij} = <.$
3. $\forall\, p \in P : A \to \alpha C j \gamma$, $i \in R(C)$, $A, C \in N$, $i,j \in V$, $\alpha, \gamma \in V^* \Rightarrow b_{ij} = .>$
4. $\forall\, p \in P : A \to \alpha C D \gamma$, $i \in R(C)$, $j \in L(D)$, $A, C, D \in N$, $i,j \in V$, $\alpha, \gamma \in V^* \Rightarrow b_{ij} = .>$
5. $\forall\, j \in L(S)$: S-аксиома, $A \in N$, $j \in V$, $\Rightarrow b_{\#j} = <.$
6. $\forall\, i \in R(S)$: S-аксиома, $A \in N$, $i \in V$, $\Rightarrow b_{i\#} = .>$

Пример: G({t, i, **,** ;},{S,N},{S→tN N→i**,**N N→i**;**},S)

$R_0(S)=\{N\}$; $R_0(N)=\{N, ;\}$; $L_0(S)=\{t\}$; $L_0(N)=\{i\}$;

$R_1(S)= R_0(S) \cup R_0(N)=\{N, ;\}$; $R_1(N)=\{N, ;\}$; $L_1(S)=\{t\}$; $L_1(N)=\{i\}$;

$R_2(S)= R_1(S) \cup R_1(N)=\{N, ;\}$; $R_2(N)=\{N, ;\}$; $L_2(S)=\{t\}$; $L_2(N)=\{i\}$;

Рассмотрим заполнение ячеек:

S→tN 1) b(t,N)= =. 2) L(N)={i}, b(t,i) = <.

N→i**,**N 1) b(i, **,**)= =. b(**,**, N)= =. 2) L(N)={i}, b(**,**, i) = <.

N→i**;** 1) b(i, **;**)= =.

5) L(S)={t}, b(#,t)= <. 6) R(S)={N, ;}, b(N, #)= .> b(;, #)= .>

Получим таблицу:

	S	**N**	**t**	**i**	**,**	**;**	**#**
S							

N							>.
t		=.		<.			
i					=.	=.	
,		=.		<.			
;							>.
#			<.				

Распознаватель функционирует следующим образом: вначале в стеке = маркер конца, флаг свертки сброшен, далее на каждом такте выполняются следующие действия:

1. При сброшенном флаге свертки: текущий символ – первый символ строки; иначе – первый символ строки для свертки.
2. По таблице определяется отношение между символом на вершине стека и текущим символом; если он отсутствует – выдать ошибку;
3. При сброшенном флаге свертки:
   3.1. Для =. : текущий символ помещается в стек, указатель чтения сдвигается;
   3.2. Для <. : текущий символ помещается в стек, указатель чтения сдвигается;;
   3.3. Для .> : установить флаг свертки, символ с вершины стека вытолкнуть в строку для свертки;

   При установленном флаге свертки:
   3.4. Для =. : символ с вершины стека вытолкнуть в строку для свертки;
   3.5. Для <. : определить правило для свертки, очистить строку свертки, поместить нетерминал из левой части правила в стек, сбросить флаг свертки;
4. Если входная строка содержит только маркер, а стек – только маркер и аксиому, завершить разбор сигналом Успешно.

Пример:

Входная строка	Стек	Текущ символ	Отношение	Строка свертки	Флаг	Разбор
ti,i;#	#	t	<.		0	
i,i;#	#t	i	<.		0	
,i;#	#ti	,	=.		0	
i;#	#ti,	i	<.		0	
;#	#ti,i	;	=.		0	
#	#ti,i;	#	.>		0	
#	#ti,i	;	=.	;	1	
#	#ti,	i	<.	i;	1	
#	#ti,N	#	.>		0	3
#	#ti,	N	=.	N	1	3
#	#ti	,	=.	,N	1	3
#	#t	i	<.	i,N	1	3
#	#tN	#	.>		0	3 2
#	#t	N	=.	N	1	3 2
#	#	t	<.	tN	1	3 2
#	#S	#			0	3 2 1

Операторная грамматика – это КС-грамматика без λ правил, в которой правые части всех правил не содержат смежных нетерминальных символов. Для таких грамматик отношения предшествования можно задать на множестве терминальных символов. **Грамматикой операторного предшествования** называется операторная КС-грамматика, для которой выполняются следующие условия:

1. Для каждой упорядоченной пары терминальных символов выполняется не более чем одно **отношение предшествования**:
$a_i =. a_j$ $(\forall a_i , a_j \in T) \Leftrightarrow \exists\, p\in P :\ A\to\alpha a_i a_j\gamma$, либо $A\to\alpha a_i B a_j\gamma$ $A,B\in N$, $\alpha,\gamma\in V^*$
$a_i <. a_j$ $(\forall a_i , a_j \in T) \Leftrightarrow \exists\, p\in P :\ A\to\alpha a_i D\gamma$, $\exists\, D\Rightarrow^* a_j\omega$, либо $\exists\, D\Rightarrow^* B a_j\omega$, $A,D,B\in N$, $\alpha,\gamma,\omega\in V^*$
$a_i .> a_j$ $(\forall a_i , a_j \in T) \Leftrightarrow \exists\, p\in P :\ A\to\alpha C a_j\gamma$, $\exists\, C\Rightarrow^* \omega a_i$, либо $\exists\, C\Rightarrow^* \omega a_i B$, $A,C,B\in N$, $\alpha,\gamma,\omega\in V^*$
2. $\forall p_i\in P : A_i\to\alpha_i$, $\alpha_i \neq \alpha_j$, $i,j=0,1,\ldots,|P|$, $i \neq j$.
3. В грамматике отсутствуют λ правила.

Любая грамматика операторного предшествования задает детерминированный КС язык. Существует алгоритм проверки, является ли произвольная КС грамматика грамматикой операторного предшествования. Однако не существует алгоритма, позволяющего преобразовать произвольную КС грамматику в грамматику операторного предшествования.

Принцип работы распознавателя аналогичен предыдущему случаю, однако отношения проверяются только между терминальными символами. Матрица предшествования содержит только терминальные символы. Аналогично опрелеляются множества:

$L^t(A) = \{t \mid \exists A\Rightarrow^* t\omega$ либо $\exists A\Rightarrow^* Bt\omega\}$, $A,B\in N$, $t\in T$, $\omega\in V^*$ - множество крайних левых терминальных символов для нетерминала A
$R^t(A) = \{t \mid \exists A\Rightarrow^* \omega t$ либо $\exists A\Rightarrow^* \omega tB\}$, $A,B\in N$, $t\in T$, $\omega\in V^*$ - множество крайних правых терминальных символов для нетерминала A
Эти множества строятся по следующему алгоритму:
1. Определяются множества L(A) и R(A) – рассмотрено ранее.

2. $\forall A\in N : R^t_0(A) = \{t \mid A\to \omega t$ либо $A\to \omega tX\}$, $t\in T$, $X\in N$, $\omega\in V^*$; $L^t_0(A) = \{t \mid A\to t\omega$ либо $A\to Xt\omega\}$, $t\in T$, $X\in N$, $\omega\in V^*$; i=1; Во множество L включаются самые левые терминальные символы правой части правил, во множество R – самые правые.

3. $\forall A\in N : R^t_i(A) = R^t_{i-1}(A) \cup R^t_{i-1}(B)$, $\forall B\in (R(A)\cap N)$; $L^t_i(A) = L^t_{i-1}(A) \cup L^t_{i-1}(B)$, $\forall B\in (L(A)\cap N)$;

4. Если $\exists R^t_i(A) \neq R^t_{i-1}(A)$ или $\exists L^t_i(A) \neq L^t_{i-1}(A) \Rightarrow$ i=i+1; повторить п.3.

Построение матрицы предшествования. Для построения матрицы предшествования $B=\{b_{ij}\}$ добавляют символы концевых маркеров (начала строки и конца строки). В качестве этих маркеров можно использовать один и тот же символ (#). Матрица заполняется на основе следующих выражений:

1. $\forall p\in P : A\to\alpha ij\gamma$ либо $A\to\alpha iBj\gamma$, $A,B\in N$, $i,j\in T$, $\alpha,\gamma\in V^* \Rightarrow b_{ij} = =$.
2. $\forall p\in P : A\to\alpha iD\gamma$, $j\in L^t(D)$, $A,D\in N$, $i,j\in T$, $\alpha,\gamma\in V^* \Rightarrow b_{ij} = <$.
3. $\forall p\in P : A\to\alpha Cj\gamma$, $i\in R^t(C)$, $A,C\in N$, $i,j\in T$, $\alpha,\gamma\in V^* \Rightarrow b_{ij} = .>$
5. $\forall j\in L^t(S)$: S-аксиома, $A\in N$, $j\in T$, $\Rightarrow b_{\#j} = <$.
6. $\forall i\in R^t(S)$: S-аксиома, $A\in N$, $i\in T$, $\Rightarrow b_{i\#} = .>$

Пример: G({t, i, **,** ;},{S,N},{S→tN N→i**,**N N→i**;**},S)

1. R(S)= {N, ;}; R(N)={N, ;}; L(S)={t}; L(N)={i};
2. $R^t_0(S)$={t}; R^t_0(N)={**,** , ;}; L^t_0(S)={t}; L^t_0(N)={i}; i=1
3.1. $R^t_1(S) = R^t_0(S) \cup R^t_0(N)$= {t, **,** , ;}; $L^t_1(S) = L^t_0(S)$ ={t};
$R^t_1(N) = R^t_0(N)$={**,** , ;}; $L^t_1(N) = L^t_0(N)$={i}; i=2
3.2. $R^t_2(S) = R^t_1(S) \cup R^t_1(N)$= {t, **,** , ;}; $L^t_2(S) = L^t_1(S)$ ={t};
$R^t_2(N) = R^t_2(N)$={**,** , ;}; $L^t_2(N) = L^t_1(N)$={i};

Рассмотрим заполнение ячеек:

S→tN 1) $L^t(N)$={i}, b(t,i)= <. N→i**,**N 1) b(i, **,**)= =. 2) $L^t(N)$={i}, b(**,** i) = <.

N→i**;** 1) b(i, **;**)= =. 5) $L^t(S)$={t}, b(#,t)= <. 6) $R^t(S)$={t, **,** , ;}, b(t, #)= .> b(**,** , #)= .> b(;, #)= .>

Получим таблицу:

	t	**i**	**,**	**;**	**#**
t		<.			>.
i			=.	=.	
,		<.			>.
;					>.
#	<.				

Как видим, эта таблица не получается простым вычеркивание строк и столбов, соответствующих нетерминалам. Распознаватель функционирует аналогично грамматике простого предшествования. Однако в работе имеются две особенности. При определении отношения между символами, если на вершине стека нетерминал, то для анализа используется следующий за ним символ. При переносе символа в строку для свертки, если символ на вершине стека нетерминал, то в строку для свертки переносится и следующий за ним терминальный символ. В остальном все аналогично.

Пример:

Входная строка	Стек	Текущ символ	Отношение	Строка свертки	Флаг	Разбор
ti,i;#	#	t	<.		0	
i,i;#	#t	i	<.		0	
,i;#	#ti	,	=.		0	
i;#	#ti,	i	<.		0	
;#	#ti,i	;	=.		0	
#	#ti,i;	#	.>		0	
#	#ti,i	;	=.	;	1	
#	#ti,	i	<.	i;	1	
#	#ti,N	#	.>		0	3
#	#ti	,	=.	,N	1	3
#	#t	i	<.	i,N	1	3
#	#tN	#	.>		0	3 2
#	#	t	<.	tN	1	3 2
#	#S	#			0	3 2 1

Управляющая таблица получается компактнее, а обработка строки происходит несколько быстрее. К сожалению, класс языков операторного предшествования уже, чем класс языков простого предшествования, поэтому упростить распознаватель не всегда возможно.

Распознаватель операторного предшествования полностью игнорирует нетерминалы. Поэтому, в принципе, их можно заменить одним и тем же нетерминалом. Грамматика, полученная таким образом, называется **остовной грамматикой**. Эта грамматика не является эквивалентной исходной и строится исключительно для упрощения алгоритма работы. Более того, она не является и однозначной, однако однозначность в распознаватель закладывается при построении управляющей таблицы. Из нее удаляются правила вида S→S, однако нумерация сохраняется в полном соответствии с исходной грамматикой. Результат разбора на основе остовной грамматики называется **остовным выводом**. В нем отсутствуют цепные правила и не учитываются типы нетерминалов. Однако компилятору в этом и нет необходимости, более того, это только усложняет обработку. Поэтому остовный вывод для компилятора более результативен.

Генерация кода. Распределение памяти

Семантический анализ. Проверка контекстных условий. Таблицы идентификаторов. Перевод конструкций на промежуточный язык. Локальные и глобальные данные. Статическая, автоматическая, динамическая память. Управление функциями. Возврат значений. Исключительные ситуации. Алгоритмы генерации машинного кода. Препроцессорная обработка

Полный распознаватель ЯП можно построить на основе распознавателя КЗ-языка. При этом КЗ-грамматики, как правило, не используются. Реально существующие компиляторы выполняют анализ программы в 2 этапа: синтаксический анализ на основе КС-языка и семантический анализ. Для проверки семантической правильности необходимо иметь всю информацию об обнаруженных лексических единицах языка. Эта информация хранится в таблицах лексем. Кроме этого, на этапе семантического анализа используются различные варианты описания синтаксических деревьев, построенных в результате разбора. Семантический анализ может выполняться на двух стадиях: каждый раз после анализа очередной синтаксической конструкции либо после полного завершения синтаксического разбора. На этапе семантического анализа выполняется:
- проверка соблюдения во входной программе семантических соглашений входного языка
- дополнение внутреннего представления программы операторами и действиями, неявно предусмотренными семантикой входного языка
- проверка элементарных семантических норм ЯП, напрямую не связанных со входным языком.

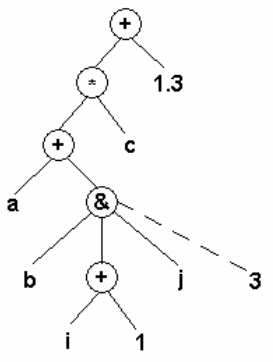
Проверка соблюдения семантических соглашений заключается в сопоставлении входных цепочек с требованиями семантики входного языка, например:
- любая метка, на которую есть ссылка, должна один раз присутствовать в программе
- каждый идентификатор в пределах блока (области видимости) должен быть описан ровно один раз
- все операнды в выражениях и операциях должны иметь допустимые для этих выражений и операций типы
- типы переменных в выражении должны быть согласованы между собой
- при вызове процедур и функций число и тип фактических параметров должны быть согласованы с числом и типом формальных параметров и т.д.

Дополнение внутреннего представления как правило связаны с добавлением преобразования типов операндов в выражениях и при передаче аргументов в функции а также с операциями вычисления адреса для сложных структур данных и др.
Проверка семантических норм. Поскольку компилятор не способен полностью понять и оценить смысл программы, это доступно только разработчику, эти требования не являются обязательными. Необязательность этих требований приводит к тому, что они не могут трактоваться как ошибка. Пример таких требований:
- каждая переменная или константа должна хотя бы один раз использоваться
- каждая переменная должна быть определена до ее первого использования при любом ходе выполнения программы
- результат функции должен быть определен при любом ходе ее выполнения
- каждый оператор программы должен иметь возможность хотя бы один раз выполниться
- операторы условия и выбора должны иметь возможность хода выполнения по каждой из ветвей
- операторы цикла должны иметь возможность завершения и т.д.

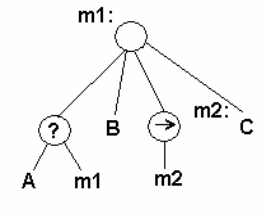
Идентификация переменных, типов, процедур, функций и др. лексических единиц ЯП – это установление однозначного соответствия между лексическими единицами и их именами в тексте программы. Чаще всего выполняется на этапе семантического анализа. Как правило, в ЯП существуют требования, чтобы имена лексических единиц не совпадали между собой и с ключевыми словами ЯП. Для идентификации компилятор может выполнять следующие действия:
- дополнять имена локальных переменных именами блоков (функций, процедур), в которых они описаны
- дополнять имена внутренних переменных и функций модулей программы именами модулей
- дополнять имена процедур и функций объектов (классов) именами типов объектов (классов)
- модифицировать имена процедур и функций в зависимости от типов их формальных аргументов.

Правила модификации имен компилятором могут понадобиться при связывании объектных модулей или библиотек, созданных на другом ЯП или с помощью другого компилятора. Таблицы идентификаторов являются одной из основных структур данных любого транслятора. Они значительно усложняются для ЯП, имеющих структуру вложенных блоков. Поэтому может использоваться и блочная структура таблиц идентификаторов.

На этапе семантического анализа может решаться задача перевода программы на промежуточный язык, что упрощает переход к машинно-зависимым этапам трансляции. Нелинейной формой промежуточной программы, является, например, синтаксическое дерево в виде связной списочной структуры. Часто в трансляции применяется разновидность постфиксной записи – обратная польская запись. Она получается путем обхода дерева по принципу левая ветвь – правая ветвь – узел. Эффективной формой представления промежуточной программы являются триады, которые содержат операцию и ее операнды. Например: (a-b)*(c+e). 1) – a b 2) + c e 3) * (1) (2). Распространена и тетрадная форма промежуточной программы. В ней присутствуют 4 элемента: операция, операнды и обозначение результата. Например: 1) – a b r1 2) + c e r2 3) * r1 r2. Иногда применяют ассемблерную форму промежуточной программы (**псевдокод**)– в этом случае вместо операндов операций языка ассемблер используются ссылки на таблицы компилятора. Помимо алгоритмов перевода, основанных на обходе дерева, существуют и алгоритмы перевода обычной формы записи в польскую, либо в форму триад или тетрад. При этом используется один или несколько стеков и задается приоритет операций. В зависимости от приоритета символа на вершине стека и типа очередного входного символа выполняется то либо иное действие. Существуют более сложные формы записи, позволяющие вычислять например адрес элемента массива или использовать условные операторы. Пример. Введем операцию Addr, вычисляющую адрес элемента массива. Операция содержит следующие операнды: собственно имя массива, индексы, число индексов. Рассмотрим выражение (a+b[1+1,j])*c+1.3. Дерево :



Здесь знаком & обозначена операция вычисления адреса. Видно, что в дереве появляется элемент, которого не было явно в исходной строке – число элементов. Обход дерева даст результат: a b i 1 + j 3 & + c * 1.3 +. Разумеется, чтобы получить корректный алгоритм перевода (или построения дерева), необходимо учитывать приоритеты квадратных скобок и запятой, а также правильно определить соответствующие действия. Для записи условного оператора понадобится возможность разметки дерева, и операции условного и безусловного перехода. Обозначим условный переход по false как ?, и безусловный переход как ->. Тогда дерево для выражения if A then B else C будет выглядеть следующим образом:



Условный переход имеет 2 операнда: условие, и метка, куда выполняется переход, если условие ложно. Безусловный переход имеет один операнд. Результат записи будет следующий:
A m1 ? B m2 -> C. С учетом меток запись приобретет вид: A m1 ? B m2 -> m1: C m2:

Распределение памяти – это процесс, который ставит в соответствие лексическим единицам исходной программы адрес, размер и атрибуты области памяти, необходимой для этой лексической единицы. Исходными данными служит, как правило, таблица идентификаторов и декларативная часть программы (область описаний). Поскольку декларативная часть может явно не присутствовать, и могут существовать дополнительные правила для описания констант и переменных, распределение памяти выполняется после семантического анализа. Современные компиляторы работают не с абсолютными, а с относительными адресами памяти. По роли области памяти в результирующей программе она бывает **глобальная** и **локальная**. Глобальная область памяти выделяется один раз при инициализации результирующей программы и действует все время выполнения результирующей программы. Как правило, может быть доступна из любой части программы. Локальная область памяти выделяется в начале выполнения некоторого фрагмента результирующей программы и может быть освобождена по завершении этого фрагмента. Доступ к этой памяти запрещен за пределами данного фрагмента. По способу распределения область памяти бывает **статическая** или **динамическая**. Статическая область памяти – это память, размер которой известен на этапе компиляции. Поэтому компилятор всегда может выделить эту область памяти и связать ее с соответствующим элементом. Для статической памяти компилятор порождает некоторый адрес. В этом случае говорят о **статическом связывании** области памяти и лексической единицы языка. Для динамической памяти размер на этапе компиляции неизвестен. Он станет известен в процессе выполнения программы. Поэтому для такой памяти компилятор порождает фрагмент кода, который отвечает за ее распределение. В этом случае происходит **динамическое связывание**. Динамическая память может распределяться разработчиком программы или компилятором автоматически. В первом случае разработчик непосредственно использует соответствующие функции, во втором – разработчик использует типы данных, операции над которыми предполагают перераспределение памяти, например, строки, динамические массивы и многие операции над объектами, и память распределяется компилятором автоматически. Можно выделить **автоматическую** память – когда распределение памяти планируется при компиляции, а выделяется и освобождается при выполнении программы. Многие компиляторы используют специальный менеджер памяти, который при первом же требовании на выделение памяти запрашивает у ОС область памяти значительного большего объема, а при последующих запросах он самостоятельно выделяет память из этого блока, и только при невозможности этого вновь обращается к ОС. Менеджер памяти может обладать функциями сборщика мусора – поиска и освобождения фрагментов, которые заняты, но не используются. При использовании менеджера памяти сокращается число обращений к ОС, что несколько увеличивает быстродействие, кроме того, сокращается фрагментация оперативной памяти. Код менеджера либо включается в код результирующей программы либо поставляется в виде динамически подгружаемой библиотеки.

Размер памяти, необходимый для лексической единицы скалярного (базового) типа, считается заранее известным. К сожалению, не удается полностью исключить зависимость результирующей программы от архитектуры вычислительной системы. Для более сложных структур используются правила, определяемые семантикой этих структур. Для массивов это произведение числа элементов на размер памяти для одного элемента, для структур – сума размеров всех полей, для объединений – размер максимального поля и т.д. Архитектура современных систем позволяет выполнять обработку данных эффективнее, если адрес, по которому выбираются данные, кратен 2,4,8,16 байтам. Компиляторы могут использовать это свойство, как правило, опционально.

Дисплей памяти процедуры – это область данных, доступных для обработки в этой процедуре. Он включает следующие составляющие: глобальные данные всей программы, формальные аргументы процедуры, локальные данные процедуры. Адрес возврата – это адрес того фрагмента кода результирующей программы, куда должно быть передано управление после завершения данной процедуры. Процедура должна работать со своими данными однотипным образом вне зависимости от того, откуда она была вызвана. Современные системы ориентированы как правило, на **стековую организацию дисплея памяти** процедуры. Она основана на том, что для хранения параметров процедур и функций , их локальных переменных и адреса возврата в результирующей программе выделяется одна на всю программу специальная область памяти, организованная в виде стека. Этот стек называется **стеком параметров**. Объектный код процедуры адресуется к параметрам и локальным переменным по смещению относительно вершины стека. Такой подход значительно упрощает и ускоряет выполнение вызовов при стековой организации дисплея памяти процедуры. Недостатки, присущие этой схеме – невозможно точно оценить необходимый размер стека, кроме того нет стандартного механизма реализации дисплея памяти процедуры. Так, параметры могут помещаться в стек в прямом или обратном порядке. Извлекаться из стека они могут либо в момент возврата из процедуры либо непосредственно после возврата. Так, в С параметры помещаются в обратном порядке и извлекаются после возврата из функции, а в Pascal – в прямом порядке и извлекаются в момент возврата из процедуры. Кроме этого, часть параметров могут передаваться в регистрах процессора. Эти вопросы становятся важны, если необходимо связать код, скомпилированный на разных языках или компиляторах.

Исключительная ситуация – это нештатная ситуация, возникающая в ходе выполнения программы, предусматривающая, что в момент ее возникновения выполнение программы будет прервано и управление будет передано специальному обработчику. По умолчанию весь код исходной программы помещается компилятором внутрь одного блока обработки исключительной ситуации, при этом компилятор сам порождает соответствующий код обработчика. Все современные компиляторы

используют стековую модель дисплея памяти процедуры для обработки исключительных ситуаций. При обработке исключительных ситуаций происходит следующее:
- выполнение программы прерывается
- вычисляется адрес обработчика исключительной ситуации
- управление передается обработчику, при этом:
- - все локальные данные стека параметров должны быть изъяты из него вне зависимости от глубины вложенности
- - для всех данных, для которых компилятор (не разработчик) выделял динамическую память, память должна быть освобождена.
- обработчик проверяет, соответствует ли ему тип произошедшей ситуации, если нет, то его выполнение прерывается и обработка повторяется сначала, т.е. происходит передача управления обработчику более высокого уровня

Для виртуальных функций адрес вызываемой функции становится известен только в момент выполнения программы, это называется **поздним связыванием**. Адрес вызова зависит от того типа данных, для которого она вызывается. В современных компиляторах предусмотрены специальные структуры для организации таких вызовов. Компилятор предусматривает, что программа может обрабатывать не только переменные, константы и структуры данных, но и информацию о типе данных, описанных в исходной программе. Эта информация называется RTTI – Run Time Type Information. Она определяется семантикой языка и реализацией компилятора.

Генерация машинного кода как правило делится на две подзадачи. 1. Выделение операций промежуточного языка, эквивалентных машинным и представление программы в виде псевдокода. Псевдокод – это такое представление команд, при котором операции могут быть заимствованы из ассемблера, а некоторые операнды по-прежнему являются ссылками на таблицы компилятора. 2. Формирование машинных команд по псевдокоду. Для каждой операции в большинстве случаев используются соответствующие заготовки, например, для выражения a+b будет использована заготовка вида:

```
mov ax, a
add ax, b
mov res, ax
```

Разумеется, вместо имен переменных реально используется связывание с соответствующими ссылками. Существуют специальные алгоритмы генерации псевдокода, например, по обратной польской записи. Алгоритм формирования машинного кода по псевдокодам не очень сложен, но громоздок. Код операции в общем случае может зависеть не только от типа операции (мнемоники) но и от типа используемых операндов.

Препроцессор выполняет предварительную обработку входных данных для другой программы. Препроцессор компилятора транслирует программу, написанную на расширенном входном языке компилятора, в программу на базовом входном языке. Языковые средства расширения базового языка называют **макросредствами**. Макрообработка выполняется обычно за один проход до основных этапов компиляции.

Синтаксически управляемый перевод

Схемы компиляции. Многоадресный код с явно и неявно именуемым результатом. Синтаксически-управляемые схемы. СУ-перевод. МП-преобразователи. Построение промежуточной программы. Транслирующие грамматики. Активные цепочки. МП-преобразователь для Т-грамматики. Атрибутные Т-грамматики. Синтезируемые и наследуемые атрибуты. Свойства АТ-грамматик. Формирование АТ-грамматик

Чтобы компилятор мог построить код результирующей программы для синтаксической конструкции входного языка, в большинстве случаев использую метод, называемый **синтаксически управляемым переводом** (СУ перевод). Его основная идея в том, что синтаксис и семантика языка взаимосвязаны. Следовательно, смысл предложений языка зависит от синтаксической структуры предложения. Теорию СУ-перевода предложил Хомски. Входной язык компилятора имеет бесконечное множество допустимых предложений, задать смысл каждого из них невозможно. Однако предложения строятся на основе конечного множества правил грамматики, и для каждого из них можно определить семантику. То же самое можно сказать и относительно выходного языка. Если по отношению к исходной программе компилятор выступает в качестве распознавателя, то для результирующей программы он является генератором предложений выходного языка. Задача состоит в том, чтобы найти порядок правил выходного языка, по которым необходимо выполнить генерацию. В первом приближении идея в том, чтобы каждому правилу входного языка компилятора сопоставить некоторое количество (в т.ч. одно или ни одного) правил выходного языка, обладающие той же семантикой (смыслом). Будем считать результатом синтаксического анализа дерево синтаксического анализа. Суть подхода СУ-перевода в следующем: с каждой вершиной дерева N связывается цепочка некоторого промежуточного кода C(N). Возможно модель компилятора, в котором фаза СУ-перевода совмещается непосредственно с синтаксическим анализом. Для таких компиляторов используется термин **СУ-компиляция**. Схемы перевода на основе СУ-компиляции можно построить , помимо прочих, для LR и LL языков. В общем случае при СУ-переводе выполняются следующие действия:

- помещаются данные в выходной поток в виде машинных кодов или команд ассемблера
- пользователю выдаются сообщения об ошибках
- порождаются и выполняются команды, указывающие, что компилятор должен выполнить некоторые действия.

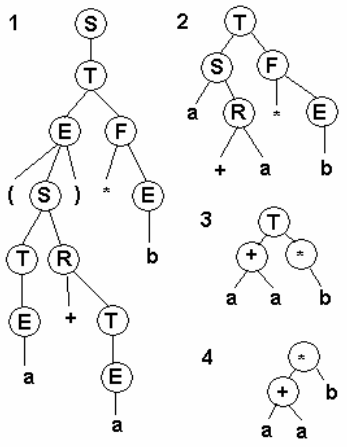
Возможны следующие формы внутреннего представления программ:

- связные списочные структуры в виде синтаксических деревьев
- многоадресный код с явно именуемым результатом (тетрады)
- многоадресный код с неявно именуемым результатом (триады)
- постфиксная запись операций (обратная польская запись)
- ассемблерный или машинный коды.

В каждом конкретном случае компилятор может пользоваться одной из этих форм, как правило, на разных фазах компиляции могут использоваться разные формы, которые по мере выполнения проходов компилятора преобразуются одна в другую.

Синтаксическое дерево – это граф в виде дерева, вершины которого соответствуют операциям, а листья представляют собой операнды. Эти деревья уже использовались в предыдущей лекции. Как правило, листья синтаксического дерева связаны с записями в таблице идентификаторов. Дерево операций можно построить непосредственно из дерева вывода. Для этого достаточно исключить из дерева цепочки нетерминальных символов, а также узлы, не несущие семантической нагрузки, например, разделители и скобки. То, какой узел дерева является операцией, а какой – операндом – определяется исходя из семантики исходного языка. Поэтому только разработчик определяет, как при построении дерева должны различаться операции и операнды, и что не несет семантической нагрузки. Упрощенный алгоритм преобразования дерева синтаксического разбора в синтаксическое дерево:

1. Если в дереве больше нет узлов, помеченных нетерминальными символами, завершить алгоритм.
2. Выбрать крайний левый узел дерева, помеченный нетерминалом, и сделать его текущим.
3. Если текущий узел имеет только один нижележащий узел, то текущий узел удалить из дерева, связанный с ним узел присоединить к узлу вышележащего уровня, в этом случае вернуться к п.1
4. Если текущий узел имеет нижележащий узел, помеченный терминалом, который не несет семантической нагрузки, лист удаляется из дерева, в этом случае переход к п.3
5. Если текущий имеет один нижележащий узел, помеченный терминалом, обозначающим знак операции, а остальные узлы помечены как операнды, то лист – знак операции удаляется из дерева, текущий узел помечается знаком данной операции, в этом случае переход к п.1.
6. Если среди нижележащих узлов текущего узла есть узлы, помеченные нетерминалами, выбирается крайний левый среди этих узлов, делается текущим и выполняется п.3, иначе алгоритм завершается.



Пример. G ({+, - , / ,*, (,), a ,b}, {S, R, T, F, E}, P, S), где P:

$S \rightarrow TR \mid T$

$R \rightarrow +T \mid -T \mid +TR \mid -TR$

$T \rightarrow EF \mid E$

$F \rightarrow *E \mid /E \mid *EF \mid /EF$

$E \rightarrow (S) \mid a \mid b$

Для цепочки (a+a)*b исходное дерево разбора, промежуточные деревья при работе алгоритма и результат будут иметь вид:

Недостаток синтаксических деревьев в том, что они не могут быть тривиальным образом преобразованы в линейную последовательность команд результирующей программы.

Базовая форма для записи тетрад выглядит следующим образом: <операция> (<операнд1> <операнд2> <результат>). Операция заданная тетрадой, выполняется над ее операндами и результат помещается в переменную, заданную результатом тетрады. Тетрады представляют собой линейную последовательность команд. Если операнд отсутствует, то он либо опускается, либо заменяется пустым операндом. Результат тетрады не может быть опущен. Порядок вычисления тетрад может быть изменен, только если есть специальные тетрады, целенаправленно его изменяющие. Для тетрад легко написать тривиальный алгоритм перевода в результирующую программу. Недостаток тетрад в том, что их сложно преобразовывать в машинный код, так как в наборах команд современных процессоров редко используются операции с тремя операндами.

Особенностью триад является то, что операнды могут быть ссылками на другую триаду. Они также представляют собой линейную последовательность команд. Результат выполнения триады нужно хранить во временной памяти, так как он может использоваться по ссылке из другой триады. В остальном они подобны тетрадам. При реализации алгоритма перевода дополнительно требуется распределение памяти.

Рассмотрим простейшую схему СУ-компиляции для перевода выражения в обратную польскую запись. Будем считать, что имеется выходная цепочка символов R и известно текущее положение указателя в ней p. Распознаватель, выполняя свертку к очередному нетерминалу или подбор альтернативы по правилу грамматики, может записывать символы в выходную цепочку и менять положение указателя. Пример:

$S \to S+T$	R(p)=”+”	shift(p)
$S \to S-T$	R(p)=”-”	shift(p)
$S \to T$		
$T \to T*E$	R(p)=”*”	shift(p)
$T \to T/E$	R(p)=”/”	shift(p)
$T \to E$		
$E \to (S)$		
$E \to a$	R(p)=”a”	shift(p)
$E \to b$	R(p)=”b”	shift(p)

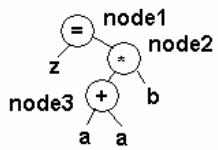
При генерации кода по дереву необходимо определять тип узла дерева. Он соответствует типу операции, которой помечен узел. Кроме того, необходимо различать четыре комбинации нижележащих узлов: оба они – листья, только левый – лист, только правый – лист, оба они не являются листьями. Пусть функция, которая реализует перевод узла дерева в последовательность команд ассемблера, называется NodeGen, параметром для нее является узел, который ей необходимо перевести. Тогда для 4 вариантов узлов для операции XXX будут представлены следующим результатом:

1. Оба нижележащих узла – операнды (op1 - левый, op2 - правый). Результат: mov ax, op1; XXX ax, op2
2. Правый нижележащий узел – не операнд (node), левый – операнд op1. Результат: NodeGen(node); mov dx,ax; mov ax, op1; XXX ax, dx
3. Левый нижележащий узел – не операнд (node), правый – операнд op1. Результат: NodeGen(node); XXX ax, op2
4. Оба нижележащих узла – не операнды (node1 - левый, node 2 - правый). Результат: NodeGen(node1); push(ax); NodeGen(node2); mov dx, ax; pop ax; XXX ax, dx.

В качестве операции XXX может использоваться любая, например, *, /, + , - (mul, div, add, sub).

Генерация кода для операции присваивания (=) несколько отличается. Здесь возможны два варианта.

1. Оба нижележащих узла – операнды (op1 – левый, op2- правый). Результат: mov ax, op2; mov op1, ax;
2. Левый нижележащий узел – операнд op1, правый – не операнд (node). Результат: NodeGen(node); mov op1, ax.



Для выражения z=(a+a)*b дерево имеет вид:

Тогда генерация выглядит следующим образом:

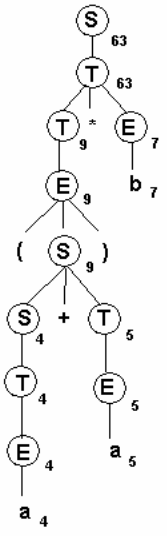
NodeGen (node2)	NodeGen(node3)	mov ax, a
mov z, ax	mov dx, ax	add ax, a
	mov ax, b	mov dx, ax
	mul ax, dx	mov ax, b
	mov z, ax	mul ax, dx
		mov z, ax

В примере было сделано несколько допущений. Во-первых, ассемблер должен воспринимать строку mul ax, op2. Поскольку для x86 платформы первый множитель всегда должен находиться в ax, то правильный синтаксис команды следующий: mul op2. Т.е. в реальных условиях необходимо учитывать особенности мнемоник ассемблера реальной платформы. Кроме того, в зависимости от типа операндов могут использоваться различные регистры процессора или даже требоваться иная серия команд, например, для чисел с плавающей запятой. Эти аппаратные зависимости устраняются, если порождать код в виде триад либо тетрад.

СУ-схемой называется пятерка D=(T, N, Δ, R, S), где T – конечный входной алфавит (терминалы), N – конечное множество нетерминалов, Δ - конечный выходной алфавит, R – конечное множество правил вида $A \to \alpha,\beta$, где $\alpha \in V^*$, $\beta \in (N \cup \Delta)^*$; S - аксиома схемы. СУ-схема называется **простой**, если в каждом правиле $A \to \alpha,\beta$ одноименные нетерминалы встречаются в α и β в одном и том же порядке. СУ-схема называется **постфиксной**, если $\beta \in N^*\Delta^*$ в каждом правиле $(A \to \alpha,\beta) \in R$. **СУ-переводом** τ, определяемым СУ-схемой D=(T, N, Δ, R, S), называется множество пар $\tau(D)=\{(\omega,y) \mid (S,S) \Rightarrow^* (\omega,y), \omega \in T^*, y \in \Delta^*\}$. Грамматика $G_{вх}=(T,N,P,S)$, где $P=\{A \to \alpha \mid (A \to \alpha,\beta) \in R\}$, называется **входной грамматикой** СУ-схемы. Грамматика $G_{вых}=(\Delta,N,P',S')$, где $P'=\{A \to \beta \mid (A \to \alpha,\beta) \in R\}$, называется **выходной грамматикой** СУ-схемы. **МП-преобразователем** называют восьмерку вида M=(Q, A, Z, Δ, δ, q_0, z_0, F), где Q – конечное множество состояний преобразователя ,A – конечный входной алфавит, Z – конечный магазинный алфавит, Δ - конечный выходной алфавит, δ - отображение множества $(Q \times (A \cup \{\lambda\} \times Z)$ в множество всех подмножеств множества $(Q \times Z^* \times \Delta^*)$, т.е. $\delta:(Q \times (A \cup \{\lambda\} \times Z) \to M(Q \times Z^* \times \Delta^*)$, q_0 – начальное состояние преобразователя, $q_0 \in Q$, z_0 – начальное содержимое магазина, $z_0 \in Z$, F – множество заключительных состояний

преобразователя, $F \subseteq Q$. Конфигурация преобразователя определяется четверкой $(q,\omega,\alpha,y) \in (Q \times A^* \times Z^* \times \Delta^*)$. Строка y будет выходом МП-преобразователя для строки ω, если существует путь от начальной до заключительной конфигурации при поступлении ω на входпреобразователя: $(q_0, \omega, z_0, \lambda) \Rightarrow^* (q, \lambda, \alpha, y)$, $q \in F$, $\alpha \in Z^*$. Переводом (преобразованием) τ, определяемым МП-преобразователем называется множество $\tau(M)=\{(\omega,y) \mid (q_0, \omega, z_0, \lambda) \Rightarrow^* (q, \lambda, \alpha, y), q \in F, \alpha \in Z^*\}$.
Если D – простая СУ-схема с входной грамматикой LL(k), то СУ-перевод можно осуществить детерминированным МП-преобразователем. Если D – простая постфиксная СУ-схема с входной грамматикой LR(k), то перевод можно выполнить детерминированным МП-преобразователем.

В простой СУ-схеме можно объединить не только левые, но и правые части правил входной и выходной грамматик. Такие объединенные грамматики называют **транслирующими** (Т-грамматики). Символы выходного алфавита Δ в Т-грамматике называются операционными символами. Операционные символы каким-либо образом выделяют. Например:
G(N={S,T,E},T={+,-,*,/,(,),a,b},Δ={+,-,*,/,a,b},P,S)
$S \rightarrow S+T\underline{+} \mid S-T\underline{-} \mid T$
$T \rightarrow T*E\underline{*} \mid T/E\ \underline{/} \mid E$
$E \rightarrow (S) \mid a\ \underline{a} \mid b\ \underline{b}$
Т-грамматика описывает правила образования строк из терминалов и операционных символов. Эти строки **называют активными цепочками**. Если из активной цепочки удалить операционные символы, получится ее входная часть. Если из активной цепочки удалить терминалы, то получится ее выходная часть. Пример: (a $\underline{a}$ + a $\underline{a\ +}$)* b $\underline{b\ *}$. Этой активной цепочке соответствует входная цепочка (a + a)* b и выходная цепочка $\underline{aa+b*}$, что соответствует обратной польской записи входной цепочки. МП-преобразователь для Т-грамматики строится подобно распознавателю, с учетом того, что в выходную цепочку помещаются операционные символы. Дальнейшее развитие Т-грамматик связано со значением символов.



Найти значение строки КС-языка можно, вычислив так называемые **атрибуты** в каждом узле дерева ее разбора. Атрибуты вычисляются по формулам, связанным с правилами грамматики. Атрибуты подразделяют на **синтезируемые** и **наследуемые**. Синтезируемые атрибуты в некотором узле дерева зависят только от атрибутов узлов-потомков. Они служат для передачи информации по дереву снизу вверх, т.е. из правой части правил грамматики в левую. Наследуемые атрибуты в некотором узлеявляются функциями атрибутов его узла-предка и(или) атрибутов узлов-потомков этого предка. Они передают информацию в противоположном направлении. Значение выражения формируется из значений его подвыражений по очевидным формулам, соответствующим правилам грамматики. Имена атрибутов обычно записывают в виде подстрочных индексов. Пример:
$S_p \rightarrow S_a+T_b\underline{+} \mid S_a-T_b\underline{-} \mid T_q$ формулы: p=a+b; p=a-b; p=q;
$T_p \rightarrow T_a*E_b\underline{*} \mid T_a/E_b\ \underline{/} \mid E_q$ формулы p=a*b, p=a/b; p=q
$E_p \rightarrow (S_q) \mid a_q\ \underline{a} \mid b_q\ \underline{b}$ формулы p=q; p=q; p=q;

Пусть на вход поступает строка $(a_4 + a_5)*b_7$, в качестве атрибута используется значение переменных. Тогда дерево разбора для данной цепочки будет следующим:
Полученные таким образом правила называют атрибутными, а грамматику с атрибутными правилами и формулами для атрибутов – атрибутной транслирующей грамматикой (**АТ-грамматикой**). АТ-грамматика – это транслирующая грамматика, дополненная следующим образом:
1. Каждый терминал, нетерминал и операционный символ имеет конечное множество атрибутов, и каждый атрибут имеет множество (в т.ч. бесконечное) допустимых значений
2.Все атрибуты нетерминалов и операционных символов делятся на наследуемые и синтезируемые
3. Наследуемые атрибуты вычисляются следующим образом:
3.1. значение наследуемого атрибута из правой части правила грамматики вычисляется как функция некоторых других атрибутов символов, входящих в правую или левую часть данного правила
3.2. начальные значения наследуемых атрибутов аксиомы грамматики полагаются известными
4. Синтезируемые атрибуты вычисляются следующим образом:
4.1. значение синтезируемого атрибута нетерминала из левой части правила грамматики вычисляется как функция некоторых других атрибутов символов из левой или правой части данного правила
4.2. значение синтезируемого атрибута операционного символа вычисляется как функция некоторых других атрибутов этого символа
5. Значения атрибутов терминалов считаются заданными, они не относятся ни к синтезируемым, ни к наследуемым.

Атрибутное дерево строится следующим образом:
1. По Т-грамматике построить дерево разбора активной цепочки, без атрибутов
2. Присвоить значение атрибутам терминалов, входящих в дерево разбора
3. Присвоить начальные значения наследуемым атрибутам аксиомы грамматики на дереве разбора
4. Вычислять значения атрибутов символов на дереве, пока это возможно, по правилу: найти атрибут, которого еще нет на дереве, но аргументы для функции его вычисления уже известны, вычислить значение этого атрибута, разместить его на дереве
5. Если по окончании п.5. значения всех атрибутов всех символов дерева оказываются вычисленными, то такое дерево называется **завершенным**.
АТ-грамматика, обеспечивающая завершенность любого дерева разбора, называется **корректной**.

Верификация и оптимизация кода

Методы оптимизации кода. Свертка выражений. Оптимизация линейного участка. Свертка объектного кода. Оптимизация передачи параметров. Оптимизация циклов. Машинно-зависимые методы оптимизации. Методы анализа свойств корректности программ. Автоматизация верификации

В большинстве случаев генерация кода выполняется не для всей программы в целом, а последовательно для отдельных ее конструкций. При этом связи между фрагментами в полной мере не учитываются. В итоге код результирующей программы может содержать лишние команды и данные. Поэтому большинство современных компиляторов выполняют еще один необязательный этап – оптимизацию результирующей программы. Оптимизация нужна, поскольку результирующая программа строится не сразу, а поэтапно. **Оптимизация** – это обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе с целью получения более эффективной, в некотором смысле, результирующей объектной программы. Как правило оптимизация использует два критерия эффективности – размер и скорость. В большинстве случаев увеличение скорости приводит к увеличению размера и наоборот. Кроме того, невозможно построить код программы, который был бы самым быстрым или самым коротким кодом результирующей программы, эквивалентной исходной. Для современных систем оптимизация может привести к увеличению быстродействия (уменьшению объема) в среднем на 10-30%. Различают два вида оптимизирующих преобразований:
- преобразования исходной программы, не зависящие от результирующего объектного языка
- преобразования результирующей объектной программы.

Первый вид преобразований не зависит от архитектуры системы. Второй – зависит не только от свойств объектного языка, но и от архитектуры системы. Используемые методы оптимизации не должны приводить к изменеиию смысла программы. Для преобразований первого вида это легко соблюдается. Преобразования второго вида могут вызвать проблемы, поскольку не всегда они имеют теоретическое обоснование и доказательство. Оптимизация может выполняться для следующих типов синтаксических конструкций: линейных участков программы, логических выражений, циклов, вызовов процедур и функций, и др.

Оптимизация линейных участков. **Линейный участок программы** – выполняемая по порядку последовательность операций, имеющая один вход и один выход. Ни одна операция линейного участка не может быть пропущена либо выполнена большее число раз, чем остальные операции данного линейного участка. Для линейных участков могут выполняться следующие виды оптимизации: удаление бесполезных присваиваний, исключение лишних вычислений, свертка операций объектного кода, перестановка операций, арифметические преобразования.

Удаление бесполезных присваиваний основано на том, что если в составе линейного участка имеется операция присваивания некоторой переменной A с номером i, и операция присваивания той же переменной A с номером j, j>i, и ни в одной операции между i и j значение переменной A не используется, то операция присваивания с номером i является бесполезной. В общем случае бесполезными могут оказаться не только операции присваивания, но и любые иные операции линейного участка, результат выполнения которых нигде не используется.

Исключение избыточных вычислений опирается на обнаружение и удаление из объектного кода операций, которые повторно обрабатывают одни и те же операнды. Операция с номером i считается лишней, если существует идентичная ей операция с номером j, j<i и никакой операнд, обрабатываемый этой операцией, не изменяется никакой операцией с номером между i и j.

Свертка объектного кода – это выполнение во время компиляции тех операций исходной программы, для которой значения операндов уже известны.. Например, вычисление выражения, все операнды которого являются константами.

Перестановка операций заключается в изменении порядка следования операций, которое может повысить эффективность выполнения, но не повлияет на результат. Например: 2*B*C*3 можно представить как (2*3)*B*C. Например, выражение, (B+C)+(P+A) может потребовать память для хранения промежуточного результата. Перестановка операций в виде B+(C+(P+A)) скорее всего обойдется без этого.

Арифметические преобразования представляют собой выполнение изменения характера и порядка следования операций на основе известных алгебраических и логических тождеств. Например, B*C+B*A = B*(C+A).

Оптимизация вычисления логических выражений основано на том, что не всегда необходимо полностью выполнять вычисление для того, чтобы определить его результат. Операция называется **предопределенной** для некоторого значения операнда, если ее результат зависит только от этого операнда и остается неизменным относительно значений других операндов. Например, операция OR предопределена для значения операнда True, а операция AND – для значения операнда false. Однако иногда такие преобразования не инварианты к смыслу программы. Например A OR F(B) если результат предопределен относительно значения A, F(B) не будет выполняться. Однако если функция помимо возвращения значения (расчета) выполняла побочные действия, например, изменяла значения глобальных переменных и т.д., то результат выполнения программы может измениться. Существуют и арифметические операции, которые предопределены для некоторого значения. Например, умножение на 0. Операция называется **инвариантной** относительно некоторого значения операнда, если ее результат не зависит от этого значения операнда и определяется другими операндами. Оптимизация может включать и исключение вычислений для инвариантных операндов.

Оптимизация передачи параметров в процедуры и функции. Передача параметров через стек является неэффективной, если процедура или функция выполняет несложные вычисления над небольшим количеством параметров. В результате код размещения параметров в стеке и освобождения стека по выходу из процедуры может занимать значительную долю операций такой функции. Используют два подхода: передача параметров через регистры процессора и подстановка кода функции непосредственно в место вызова в объектном коде (inline функции). Передача параметров через регистры имеет тот недостаток, что зависит от архитектуры системы. Кроме того, таким образом соптимизированные функции не могут использоваться в качестве библиотечных. Использование inline функций ускоряет обработку, но увеличивает размер кода.

Оптимизация циклов. Чтобы обнаружить все циклы в исходной программе, используются методы, основанные на построении графа управления программы. При оптимизации циклов используют: вынесение инвариантных вычислений за пределы цикла; замена операций с индуктивными переменными; слияние и развертывание циклов. В первом случае, за пределы цикла

выносятся те операции, операнды которых не изменяются в процессе выполнения цикла. Переменная называется **индуктивной** в цикле, если ее значения в теле цикла образуют арифметическую прогрессию. В простейшем случае, это может быть замена умножения на счетчик цикла сложением. Например: for(S=10, i=1; i<=N; i++) A[i]=i*S; можно заменить на
for(S=10, i=1; i<=N; i++, S+=10) A[i]=S;
Слияние и развертывание циклов предусматривает слияние двух вложенных циклов в один и замену цикла линейной последовательностью операций. Например: for(i=1;i<=N;i++) for(j=1;j<=M;j++) A[i][j]=0; успешно заменяется циклом:
for(X=M*N, i=1; i<=X; i++) A[i]=0; Развертывание циклов можно выполнить для тех из них, кратность выполнения которых известна уже на этапе компиляции.
Машино-зависимые методы оптимизации. Они ориентированы на конкретную архитектуру системы. Например, использование регистров общего назначения для хранения значений операндов и результатов вычислений увеличивает быстродействие.Или, например, не все операции могут быть выполнены с операндом в памяти и требуют предварительной загрузки в регистр, иногда жестко определенный, процессора. Поскольку количество программно доступных регистров ограничено, встает вопрос об их распределении. Здесь может возникать ситуация, аналогичная подкачке страниц в память – например, надо загруить переменную в регистр, а все доступные регистры уже заняты. Какой из них выгрузить в память? Ряд процессоров и систем позволяют параллельное выполнение операций. Можно строить компилятор таким образом, что по соседству будет максимальное количество операций, операнды которых не зависят друг от друга. Конечно, в челом это не решаемая пока задача, однако для конкретного оператора решение заключается в в порядке выполнения операций.

Свойство программы, характеризующее отсутствие в ней ошибок по отношению к целям разработки, называют **корректностью** программы. Корректность программы, как формальную, так и смысловую, нужно доказать. Задача доказательства синтаксической правильности решена благодаря описанию синтаксиса языка на основе теории формальных грамматик. Естественно возникает вопрос: можно ли аналогично решить и задачу семантической корректности? В этом случае системы программирования стали бы качественно новыми системами – системами доказательного программирования. Традиционно семантическая корректность проверяется путем тестирования программы. **Тестирование** – процесс выполнения программы с целью обнаружения ошибок. Выполнение всестороннего тестирования сложной программы практически невозможно. В то же время корректность программы имеет смысл только при четко сформулированной цели разработки. Формализованное описание постановки задачи называется **спецификацией задачи**. **Верификация** – процесс доказательства соответствия между программной реализацией и спецификацией задачи. Верификация фактически представляет собой аналитическое исследование свойств программы по ее тексту, т.е. без выполнения самой программы. Наиболее распространенным методом доказательства частичной корректности программ является **метод индуктивных утверждений**. Он позволяет свести анализ свойств программы к доказательству конечного числа утверждений, записанных в виде формул логического языка спецификации и имеющих интерпретацию в проблемной области решаемой задачи. При выполнении программы с различными вариантами исходных данных возможна реализация различных цепочек операторов. Такие цепочки называются **трассами вычислений**. При наличии в программе повторяющихся вычислений перебор всех трасс обычно оказывается невозможен. Для решения этой проблемы вводят **инварианту цикла** – утверждение, приписанное циклу, которое должно сохранять истинное значение при каждом выполнении тела цикла. Доказательство правильности программы, основанное на применении метода индуктивных утверждений, содержит следующие основные этапы:
- построение схемы алгоритма решения задачи
- формулировка утверждений для входаи выхода программы в виде формул логического языка спецификации
- выявление всех циклов и формулировка контрольного утверждения для каждого из них на логическом языке спецификации
- составление списка путей между контрольными точками алгоритма
- построения условия верификации для каждого пути с использованием семантики операторов, образующих путь
- доказательство истинности всех условий верификации как теорем формальной теории проблемной области решаемой задачи
- доказательство завершения программы.

Исследования по формализации семантики начались с середины 60-х. выработаны три основных подхода: операционный, аксиоматический и денотационный. Наиболее практичным считается аксиоматичный подход, предложенный Хоаром. В настоящее время применение ПК для верификации программ идет преимущественно по пути создания систем верификации, предусматривающих диалог с оператором на некоторых этапах работы. На первом этапе выполняется анализ программы, т.е. контроль лексической и синтаксической правильности аннотированной программы. Программа переводится на промежуточный язык. Далее происходит генерация условий верификации. Она основана на применении аксиоматической системы используемого ЯП и осуществляется без участия оператора. Далее выполняется доказательство условий верификации. При этом могут выполняться некоторые эквивалентные преобразования и упрощения условий верификации. При необходимости возможно применение системы аксиом пользователя. Результаты доказательства исследуются анализатором доказательства. Возможны следующие ситуации:
- все условия верификации истинны, завершение работы
- доказательство отдельных условий не завершено. Эти условия возвращаются на доказательство с применением дополнительной информации, вводимой пользователем
- среди условий верификации обнаружены ложные. Ошибки могут быть как в спецификации программы, так и в операторах самой программы, формально различить эти ситуации невозможно. Пользователь должен выполнить модификацию аннотированной программы и повторить процедуру обработки.

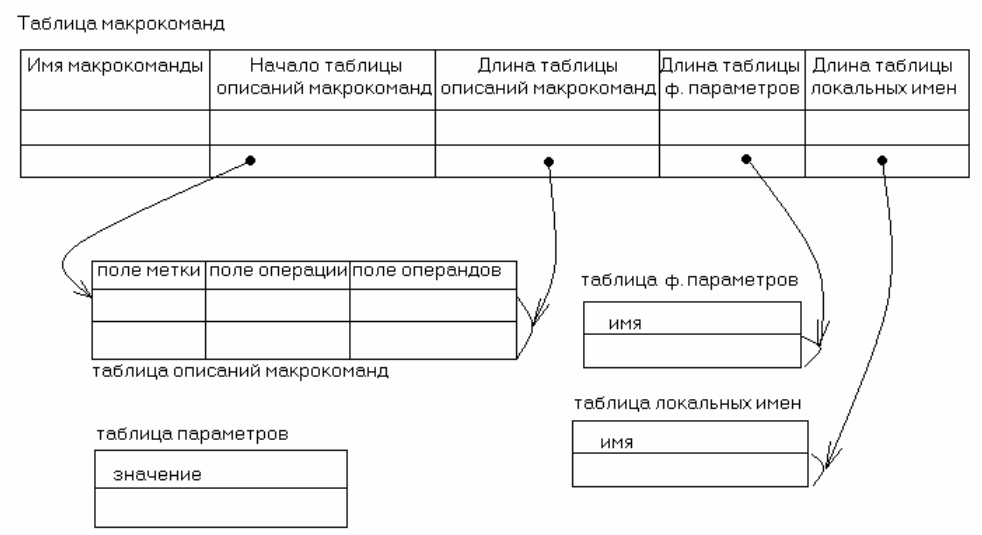
Ассемблеры и компоновщики

Схема макроассемблера. Макроопределения. Таблицы макрогенератора. Обработка макрокоманды. Обработка команд и вложенных макрокоманд. Задачи и схемы ассемблера. Таблицы ассемблера. Формирование команд. Способы адресации. Редактор связей и загрузчик. Объектный модуль. Загрузочный модуль. Коррекция адресов программы

Транслятор языка ассемблер с макросредствами называется макроассемблером. В большинстве случаев он строится по двухступенчатой схеме. Первая ступень переводит программу с макроязыка на язык ассемблера и называется макрогенератором. Вторая, ассемблер, транслирует с языка ассемблер в машинный код. Макроопределения располагают перед сегментами программы, макрокоманды – в том месте, где должны быть выполнены соответствующие действия. **Макроопределения** позволяют указать, какие идентификаторы на какие строки необходимо заменять. **Макрокоманда** представляет собой текстовую подстановку, при выполнении которой идентификатор определенного вида заменяется на заданную цепочку символов. Макроопределение может содержать параметры. Тогда каждая соответствующая ему макрокоманда должна содержать строку символов на месте каждого из параметров. При макроподстановке соответствующий параметр будет заменен этой строкой. Макрокоманды могут образовывать и вложенные вызовы, в том числе с сильными ограничениями и рекурсию. Пример макрокоманды:

```
push0   macro          mulx2   macro k, x      вызов макрокоманды   mulx2   7, 15 приведет к подстановке такого кода:
        xor ax, ax             mov ax, x              mov ax, 15
        push ax                mul x                  mul 15
        endm                   mul k                  mul 7
                               endm
```

Макроопределение может содержать локальные переменные и метки. Для их определения служит ключевое слово local. Макрогенератор строится по однопроходной схеме. Им создается пять временных таблиц, четыре из которых заполняются при обработке макроопределений.



Для вышеприведенного примера таблицы будут заполнены примерно следующей информацией:

ТМК

mulx2	15	3	2	0

ТФП

k
x

ТЛИ

ТП

7
15

ТОМК

…		
	mov	ax, @2
	mul	@2
	mul	@1

ТМК является списком всех макрокоманд исходного модуля. ТОМК хранит тексты тел всех макроопределений программы, формальные параметры заменены в этих текстах ссылками на ТФП, локальные имена – ссылками на ТЛИ. ТФП заполняется при обработке команды macro, строится заново для каждого макроопределения. ТЛИ заполняется при обработке команды local, строится заново для каждого макроопределения. При обработке макрокоманд строится ТП, в которую заносят фактические параметры макрокоманды.

Команды генерации могут располагаться как в теле макроопределения, так и в тексте основной программы. Команды генерации могут использовать особые переменные, которые существуют только в момент макрогенерации. Как правило, обработку команд генерации ведет отдельный компонент макрогенератора. **Вложенной макрокомандой** называются такие макрокоманды, которые расположены внутри макроопределения. Такие макроопределения обрабатываются обычным образом. При обнаружении внутренней макрокоманды, замена внешней макрокоманды приостанавливается, и выполняется замена внутренней макрокоманды, после чего продолжается замена внешней.

Главная задача ассемблера – перевод команд исходной программы в машинный код. Ассемблер решает следующие основные задачи:

- распределяет память для объектов программы
- переводит в машинный код команды и константы программы
- обнаруживает ошибки в исходной программе и выдает диагностическую информацию
- формирует печатный документ (листинг)

- формирует объектный модуль.
Существуют разные схемы построения ассемблера, наиболее известные – однопроходной и двухпроходной. Современные ассемблеры строятся по второй схеме. На первом проходе выявляются все имена, распределяется память и контролируется правильность текста, на втором – генерируются машинные коды, формируется объектный модуль и листинг. Ассемблер использует постоянные и временные таблицы. Основные постоянные таблицы – таблица операций ТОП и таблица стандартных текстов ТСТ. В ТОП хранятся мнемонические коды операций всех команд ассемблера и соответствующие им цифровые коды или шаблоны команд. ТСТ содержит стандартные тексты заголовков листинга и сообщений об ошибках. Временные таблицы могут организовываться различными способами, один из вариантов состоит из 6 таблиц: таблицы сегментов, содержащей имена сегментов, их длины и характеристики; таблицы сегментных регистров, устанавливающей соответствие между именами сегментов и кодом сегментных регистров; таблицы имен, хранящей все имена, обнаруженные в поле метки, за исключением имен сегментов; таблицы внешних имен; таблицы глобальных имен; таблицы ошибок, содержащей номера предложений с ошибками и признаки типа ошибки; таблицы использованных имен, содержащей имена из полей операндов машинных команд и номера предложений с этими командами.
Алгоритм формирования команд на машинном коде процессора x86 не очень сложен, но достаточно громоздок. Для формирования кода необходим мнемонический код операции и характеристики операндов. Построение команды может выполняться по ее шаблону. Команды могут занимать один байт и более. Структура команды следующая: байт-префикс, код операции (КОП), пост-байт, байты данных. Обязательным в этой структуре только КОП, наличие остальных полей зависит от команды и состава операндов. В состав КОП могут входить биты признаков формата: w – признак длины операнда (0 – короткий, 1 – длинный), s – признак расширения непосредственного операнда при выполнении (0 – не расширять, 1- расширять), d – признак порядка размещения операндов в машинной команде. Пост-байт служит для указания способа адресации и структуры команды. Он состоит из 3 полей: mod (биты 6,7), reg (биты 5-3), r/m (биты 0-2). Поля mod и r/m формируются в зависимости от способа адресации операнда. Например, для регистровой адресации mod=11, r/m содержит номер регистра, для абсолютной адресации mod=00, r/m=110. Поле reg предназначено для номера регистра, если он является одним из двух операндов команды. Если в этом поле размещается номер первого операнда, d=1, второго – 0. Абсолютный адрес и непосредственное значение всегда располагаются в поле данных. При работе с однобайтными регистрами, w=0, с двухбайтными – w=1. При работе с 32 разрядной архитектурой, расширенные регистры (4 байта) нумеруются аналогично соответствующим 2 байтовым, а двухбайтовые дополнительно помечаются байтом-префиксом с кодом 66h.

Рассмотрим основные варианты адресации.
Регистровая. Операнд представлен символическим именем регистра. Регистр кодируется трех- либо двухразрядным двоичным номером. Иногда регистровый операнд явно не указывается, используя номер по умолчанию.
Непосредственная. Операндом является выражение, которое при вычислении транслятором получает числовое значение. Это значение размещается непосредственно в команде.
Абсолютная (прямая). Операндом является выражение, являющееся адресом в пределах сегмента. В команде будет представлен этот адрес.
Относительная. Операндом является выражение, значение которого вычисляется как смещение относительно следующей по порядку команды.

Объектный модуль является единицей хранения программ. Трансляторы многих систем программирования строят объектные модули одинаковой структуры, что позволяет объединять модули, созданные различными системами, в один исполняемый файл. Объектный модуль содержит текст программы на машинном языке и вспомогательную информацию, включая данные о сегментах, внешних и глобальных именах, и др. Он может включать записи различных типов:
80 – признак начала ОМ
96 – список имен сегментов, хранящийся в алфавитном порядке, каждому имени предшествует однобайтовый указатель длины
98 – характеристики сегмента (длину, коды характеристик, указатель на соответствующее имя в записи типа 96), порядок следования сегментов соответствует порядку их размещения в исходном модуле
8C – внешнее имя (одно имя и его длина)
A0 – текст сегмента на машинном языке, а также порядковый номер записи типа 98 для данного сегмента
9A – определение группы – ссылка на имя группы в записи типа 96, указатели на записи типа 98 с характеристиками сегментов, принадлежащих группе
9C – таблица настройки. Для каждого адреса программы, подлежащего коррекции, в таблице содержится отдельный элемент с признаком C8 (адрес сегмента), C4 (адрес в сегменте), CC (адрес в другом модуле).
90 – глобальное имя, а также номер сегмента и адрес в сегменте, где расположено определение этого имени
8A – признак конца ОМ, может содержать информацию о расположении точки входа в программу.

Современные компоновщики обеспечивают работу с платформами разной разрядности (32,16,64). Обычно для каждой платформы создается свой редактор связей. Формат ОМ не связан с конкретной ОС. Именно редактор связей готовит ОМ к выполнению в конкретной операционной среде. Наиболее распространены следующие форматы исполняемых модулей:
COM – односегментный абсолютный модуль под ОС MS-DOS
MZ – многосегментный загрузочный модуль под ОС MS-DOS
NE – загрузочный модуль для windows 3.1
PE – загрузочный модуль для Windows 9x/NT
COFF – загрузочный модуль для UNIX
Компоновка загрузочного модуля из объектных состоит в размещении сегментов программы в тексте загрузочного модуля и настройке межсегментных связей с учетом нового положения сегментов. Совокупность объектных модулей, указанных редактору связей, называют **потоком редактирования**. Редакторы связей обычно также организованы по двухпроходной схеме, поскольку модули и сегменты в потоке могут располагаться в произвольном порядке. На первом проходе распределяется

память загрузочного модуля для сегментов, на втором – корректируются адреса в командах программы для учета нового положения сегментов. Компилятор строит код каждого сегмента исходя из нулевого начального адреса. Редактор связей же полагает равным нулю начальный адрес текста загрузочного модуля. Поэтому сегменты программы перемещаются в адресном пространстве на новое место. Недостающие объектные модули редактор связей ищет в доступных ему библиотеках объектных модулей. Редактор связей использует ряд временных таблиц, основные из которых – таблица сегментов, таблица внешних имен и таблица глобальных имен.