1. Основные функции ОС, требования, предъявляемые к ОС.

Функции ОС:

1. управление ресурсами вычислительной системы (процессорное время, оперативная память, периферийные устройства и ресурсы математического обеспечения);
2. упростить общение пользователя с ЭВМ. Создание удобного интерфейса пользователя (с такими элементами системного программирования как трансляторы, отладчики, загрузчики и т.д.).

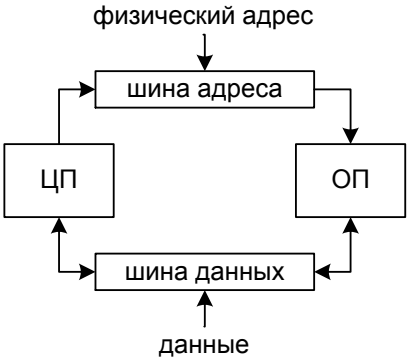
Требования к ОС:

1. ***Надежность*** – система должна быть так же надежна, как и аппаратура, на которой она работает; должна производить определение и диагностирование ошибок, защищать пользователя от его же ошибок.
2. ***Защита*** – система должна защищать себя и выполняющиеся задачи пользователя от влияния друг на друга.
3. ***Предсказуемость*** – система должна отвечать на запросы пользователя предсказуемым образом, результат выполнения команд пользователя должен быть одним и тем же, вне зависимости от последовательности выполнения этих команд.
4. ***Удобство*** – система команд должна освобождать пользователей от решения задач по распределению ресурсов и по управлению этими ресурсами.
5. ***Эффективность*** – система должна максимально повышать использование системных ресурсов пользователем. Сама система не должна использовать большое количество ресурсов.
6. ***Гибкость*** – системные операции должны настраиваться, ресурсы могут быть увеличены или уменьшены для того, чтобы эффективность и доступность.
7. ***Расширяемость*** – к системе могут добавляться новые средства.

2. Адресация памяти в реальном режиме, сегментация памяти, сегментные регистры.

Процессор делит адресное пространство на произвольное количество сегментов, каждый из которых содержит не более 64 Кбайт. Адрес первого байта сегмента всегда кратен 16 и называется адресом сегмента или параграфом сегмента.

Для работы с памятью используются 2 шины – шина адреса и шина данных. Физически память устроена таким образом, что возможна адресация как 16-битовых слов, так и отдельных байтов памяти. Процессоры i80386, i80486 могут адресовать 32-битовые слова памяти.



ФА передается из процессора в память по шине адреса. Ширина шины адреса определяет максимальный объем физической памяти, непосредственно адресуемой процессором. Например, при 20-разрядной ША и 16-разрядной ШД можно адресоваться к 1 Мбайту памяти, причем к байтам и словам размером в 16 бит, 00000h<=ФА<=FFFFFh.

Возникает проблема представления 20-разрядного ФА при помощи содержимого 16-разр. регистров → используется 2-х компонентный ЛА, состоящий из сегмента памяти и смещения внутри сегмента.

Для получения 20-разрядного ФА к сегментной компоненте приписываются справа 4 нуля, затем прибавляется компонента смещения. Формат ЛА: <сегмент: смещение>.

Микропроцессоры имеют 14 специализированных регистров:

1. **регистры общего назначения** - используются для временного хранения промежуточных результатов и операндов арифметических и логических операций: AX (аккумулятор), BX (база), CX (счетчик), DX (данные). Можно адресоваться к старшему (AH, BH, …) или младшему (AL, …) байту.
2. **сегментные регистры** – хранят начальные адреса 4-х сегментов:
   a. DS: сегмент данных,
   b. CS – кода,
   c. ES – дополнительный (расширенный) сегмент данных,
   d. SS - стека).

   Каждый сегментный регистр, обеспечивает адресацию памяти объемом 64 Кбайт, которая называется текущим сегментом. Адрес в сегментном регистре автоматически умножается на 16, чтобы он указывал на одну из 16-байтовых границ мегабайтного адресного пространства микропроцессора. Адрес в CS складывается с IP для получения ссылки на текущую команду. DS плюс смещение указывает на ячейку в сегменте данных. ES используется для адресации данных, находящихся в разных физических сегментах.
3. IP – **указатель команд**,
4. SP – **указатель вершины стека**,
5. BP – **указатель базы (основания) стека** – используется только в момент загрузки для
6. SI – **индекс источника**,
7. DI – **индекс назначения**.
8. **регистр флагов**: по битам: 15-12 – не используются, OF- флаг переполнения, DF – направления, IF – прерывания, TF – трассировки, SF – знака, ZF – нуля, 5 – не используется, AF – вспомогательного переноса, 3 – не используется, PF – четности, 1 – не используется, CF – переноса.

3. Стек. Вектора прерываний.

Стек
SS – адрес сегмента стека.
SP – указатель вершины стека
BP – основание стека. Используются только в момент загрузки программы. Это базовый регистр. Его также можно использовать для адресации.
ES и DS адреса сегментов данных. DS по умолчанию.
DS : SI для DS
ES : DI для ES
SI и DI могут использоваться в качестве индексных регистров.
Стек заполняется со старших разрядов.
Работает по принципу LIFO. В стек можно записывать данные размером 2 байта.

При записи в стек содержимое SP уменьшается на 2 и по этому адресу записывается определенное значение в стек. При чтении данных из стека данные читаются, а затем SP:=SP+2. Система MS-DOS не следит за переполнением стека. Программист должен отслеживать это.
Переполнение происходит следующим образом: SP=0000 и записываем в стек, т.е. SP:=SP-2 и указатель стека будет равен FFFE и косяк!!!

Вектора прерываний.

Операционная система в значительной степени управляется с помощью системы прерываний. В реальном режиме имеются 2 типа прерываний: аппаратные и программные. Программные прерывания инициализируются командой int. Аппаратные – внешними событиями, асинхронными по отношению к выполняемой программе. Обычно аппаратные прерывания инициализируются аппаратурой ввода/вывода после завершения выполнения текущей операции.

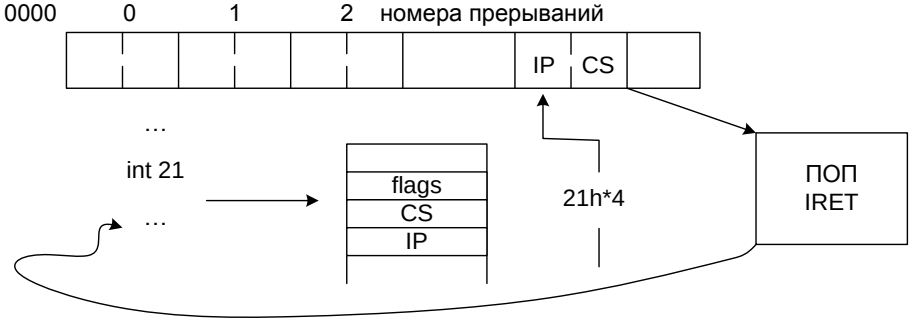
Для обработки прерываний в реальном режиме процессор использует таблицу векторов прерываний. Сегментные адреса, используемые для определения местоположения программ обработки прерываний, называются векторами прерываний.

Таблица векторов прерываний располагается в самом начале ОП, т.е. её физический адрес = 0. Таблица векторов прерываний состоит из 256 элементов по 4 байта. Вектора состоят из 16 битового сегментного адреса и 16 битового смещения (сначала идет смещение, а потом сегмент). Каждый вектор прерываний имеет свой номер, называемый номером прерывания, который указывает на его место в таблице. Этот номер, умноженный на 4, дает абсолютный адрес вектора памяти.

Вектора прерываний получают свои значения при запуске системы. Сначала BIOS выполняя процедуры инициализации, задает значения определенных векторов прерываний. При загрузке DOS задаются значения векторам прерываний DOS.

DOS может переназначить некоторые из векторов BIOS к своим подпрограммам. Пользователь также может изменить значения векторов прерываний. Когда происходит программное или аппаратное прерывание, текущее состояние регистров CS:IP, а также значения регистра флагов записываются в стек программы. Далее из таблицы векторов прерываний выбираются новые значения CS:IP. При этом управление передается на процедуру обработки прерываний.

Перед входом в процедуру обработки прерываний принудительно сьрасывется флаг трассировки TF и флаг разрешения прерываний IF. Завершив обработку прерывания, процедура должна выдать команду IRET, по которой из стека будут извлечены значения регистров CS:IP и регистра флагов. Далее продолжается выполнение прерванной программы.



Процедура обработки прерывания обязательно должна закончится процедурой IRET, после которой считываются значения flags, CS, IP.

4. Клавиатура. Клавиатурный буфер.

Клавиатура – это отдельное символьное устройство с микропроцессорным управлением. При включении питания клавиатура устанавливается в исходное состояние. Затем выполняется автотест, который проверяет схемы и память клавиатуры.

После прохождения теста клавиатура начинает работу. Работой клавиатуры управляет специальная электрическая схема – контроллер клавиатуры. В его функции входит распознавание нажатой клавиши и помещение закрепленного за ней кода во входной регистр (порт).

В ходе работы клавиатура непрерывно проверяет, есть ли изменения в состоянии клавиши. При каждом нажатии клавиши по интерфейсу посылается однобайтовый код, который равен позиционному номеру клавиши. Этот код называется позиционным или скэн кодом.

При освобождении клавиши к ее номеру добавляется 80h и полученный позиционный код также посылается по интерфейсу, т.о. каждой клавише соответствуют 2 позиционных кода: код нажатия и код освобождения.

Если клавиша остается нажатой дольше некоторого времени, её скэн код начинает посылаться по интерфейсу через 0,1 секунду до освобождения клавиши.

Нажатие, а также отпускание одной клавиши вызывает сигнал аппаратного прерывания, заставляющий процессор прервать выполняемую программу и перейти на программу обработки прерывания от клавиатуры. Процессор совместно с сигналом прерывания получает ещё и номер вектора прерывания 09h.

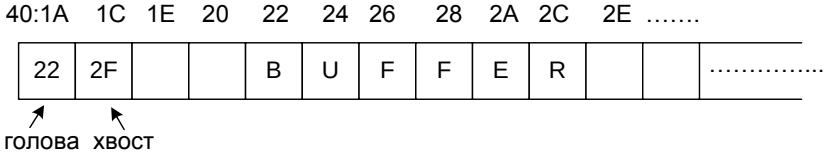
Программа int 09h получив управление в результате прерывания от клавиатуры считывает из порта 60h скэн-код клавиши и анализирует его значение.

Если скэн-код принадлежит одной из управляющих клавиш и к тому же представляет собой код нажатия, в байте статуса клавиатуры устанавливается бит, соответствующий нажатой клавише. При нажатии других клавиш программа int 009h по таблице трансляции скэн-кодов в коды ASCII формирует 2х байтовый код, старший байт которого содержит скэн-код, а младший – код ASCII, т.к. за каждой клавишей закреплено не менее 2х символов, то каждому скэн-коду соответствует не менее 2х кодов ASCII.

Чтобы выбрать соответствующий ASCII код используется байт статуса клавиатуры, в котором хранится информация о нажатии клавиш Alt, Shift, Ctrl и т.п. После подачи одного позиционного кода клавиатура ожидает от компьютера подтверждения его готовности принять следующий код. Если до получения сигнала подтверждения нажаты другие клавиши, клавиатура записывает их коды в буфер. Буфер построен как циклическая очередь, работающая по принципу FIFO. Он занимает непрерывную область адресов памяти, имеет 2 указателя, которые хранят позиции головы и хвоста строки символов, находящихся в буфере в текущий момент.

Указатель на голову установлен на первый введенный символ. Указатель на хвост указывает на позицию за последним введенным символом. Когда оба указателя равны, буфер пуст.

В буфере 16 символов – размер буфера 32 байта.



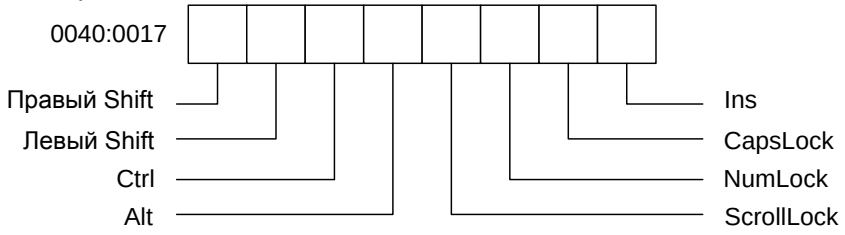
Расширенные коды присвоены клавишам или комбинациям клавиш, которые не имеют представляющего их символа ASCII – это функциональные клавиши или комбинации с клавишей Alt.

Расширенные коды имеют длину 2 байта, причем первый байт всегда = 0 позволяет программе определить, принадлежит ли данный код расширенному набору. 2й байт – это номер расширенного кода.

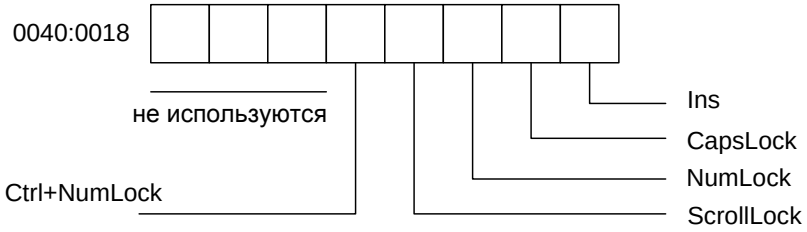
5. Клавиатура. Байты статуса. Основные шаги обработки прерываний от клавиатуры.

Определяют, нажаты ли Shift, Alt, CapsLock …

2 байта, расположенные в ячейках памяти с адресами памяти 0040:0017 и 0040:0018 содержат биты, отражающие статус клавиш-переключателей.



2й байт отображает моменты нажатия клавиши.



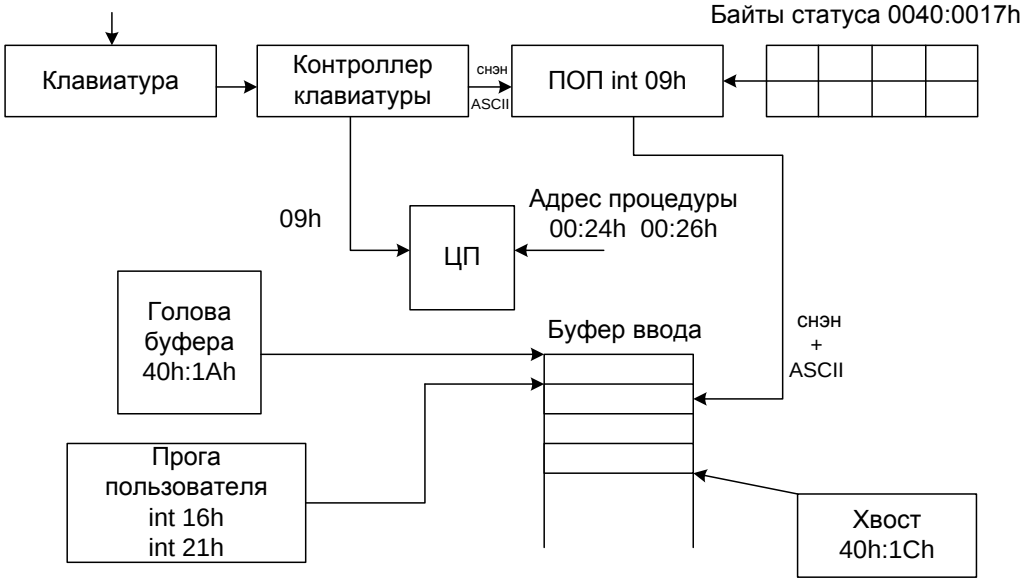
Прерывание клавиатуры обновляет эти биты статуса, как только будет нажата одна из клавиш переключателей даже если не было считано одного символа из буфера клавиатуры. Прерывание клавиатуры проверяет состояние статусных битов каждый раз перед тем как интерпретировать нажатые клавиши.

В прерывании клавиатуры можно выделить 3 основных шага:

1. прочитать скэн-код и послать клавиатуре подтверждающий сигнал
2. преобразовать скэн-код в номер кода или в установку соответствующего байта статуса.
3. поместить код клавиши в буфер клавиатуры

В момент вызова прерывания скэн-код находится в порте. Сначала он анализирует, что клавиша была нажата или отпущена. Все коды освобождения отбрасываются за исключением клавиш переключателей, для которых выполняются соответствующие изменения в байтах статуса. С другой стороны все коды нажатия отбрасываются. При этом могут изменяться байты статуса клавиш переключателей. После того, как введенный символ идентифицирован, процедура ввода с клавиатуры должна найти соответствующий код ASCII или расширенный код. После этого коды символов помещаются в клавиатурный буфер.

Работу клавиатуры можно представить с помощью схемы:



Существует несколько способов проверки, был ли ввод с клавиатуры:

1) когда значения головы и хвоста равны
2) функция 0Bh прерывания 21h возвращает значение FFh в регистр AL когда буфер клавиатуры содержит хотя бы один символ:

Функция 1) прерывания BIOS 16h предоставляет ту же возможность, кроме того, она показывает, какой символ в буфере. Флаг нуля ZF устанавливается, если буфер пуст и сбрасывается, если в буфере есть символ. Копия символа из буфера помещается в регистр AX, но символ из буфера не удаляется. В регистре AL возвращается ASCII код символа, и если он = 0, то это расширенный код и в регистре AH пересылается номер кода.

6. Видеосистемы.

Основным графическим устройством, с которым чаще всего приходится работать, является видеосистема компьютера. Обычно она состоит из видеокарты (адаптера) и подключенного к ней монитора.

Изображение хранится в растровом виде в памяти видеокарты: аппаратура карты обеспечивает регулярное чтение этой памяти и ее отображение на экране монитора. Поэтому вся работа с изображением сводится к тем или иным операциям с видеопамятью.

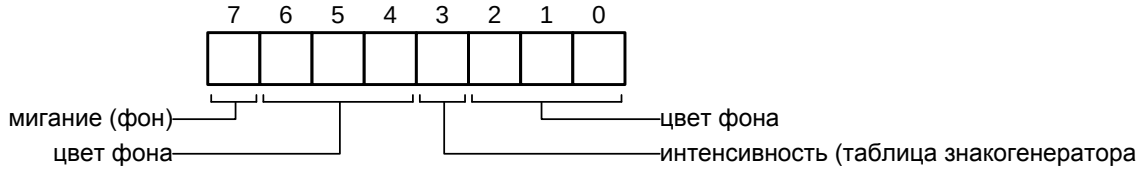
Существует несколько стандартных режимов работы видеоадаптеров, определенных фирмой IBM. Любой из этих режимов можно инициировать конструкцией типа:

```
mov   ah,00h
mov   al,Mode   ;установка номера видеорежима
int   10h
```

В большинстве режимов видеоадаптера видеопамять разделяется на несколько страниц, при этом одна из них является активной и отображается на экране. При помощи функций BIOS можно переключать активные страницы видеопамяти. Вывод информации можно производить как в активные, так и в неактивные страницы. Структура видеопамяти зависит от режима работы адаптера.

В текстовых режимах на экране отображается текстовые символы и символы псевдографики. Для кодирования каждого символа используется 2 байта. Первый содержит ASCII-код отображаемого символа, а второй – его атрибуты.

Коды символов имеют четные адреса, а атрибуты – нечетные. Атрибут определяет цвет символа и цвет фона.



При отображении символа на экране происходит его преобразование из формата ASCII в двумерный массив пикселей. Для этого преобразования используется таблица трансляции символов. BIOS загружает таблицы знакогенератора во второй цветовой слой видеопамяти. Каждая таблица содержит описание 256 символов. Одновременно активными могут быть одна или две таблицы.

Распределение видеопамяти в **графических режимах** отличается от распределения видеопамяти в текстовом, так как необходимо хранить информацию о каждом пикселе изображения.

Отображение видеопамяти на экран не является непрерывным. Первая половина видеопамяти содержит информацию относительно всех нечетных линий экрана, а вторая половина – относительно всех четных. В любом случае изображение хранится в растровом виде в памяти видеокарты. Аппаратура видеокарты обеспечивает регулярное чтение этой памяти и ее отображение на экране монитора.

Вся работа с изображением сводится к тем или иным операциям с видеопамятью.

Для адаптеров SVGA был разработан новый единый стандарт, получивший название VESA от IBM, этот стандарт определен как дополнение к видеофункциям BIOS. Для доступа ЦП к видеопамяти обычно резервируется адресное пространство размером всего 65 Кб. Видеоадаптеры применяют прием, при котором ЦП получает доступ к видеопамяти через перемещаемое окно, чтобы доступ к любому участку видеопамяти. Такой доступ к видеопамяти создает определенные трудности, так как чтобы отобразить на экран монитора пиксель нужно не только вычислить положение соответствующей ячейки видеопамяти, но и также определить смещение для окна доступа.

В большинстве режимов SVGA реализована схема прямого кодирования цвета. Биты, определяющие пиксель группируются в трех основных группах, непосредственно определяющих R, G и B компоненты цвета. Данные из этих трех групп подаются на ЦАП и формируют видеосигнал.

7. Диски. Логическая структура жесткого диска.

Все диски как гибкие, так и жесткие организованы одинаковым образом. Поверхность диска разделена на ряд концентрических окружностей, называемых дорожками. Каждая дорожка радиально разделена на части, называемые секторами. Количество данных, которое можно расположить на каждой из поверхностей диска, зависит от количества дорожек и от размера и количества секторов в них.

Количество используемых поверхностей диска и количество дорожек на каждой поверхности представляют собой аппаратные характеристики, различные для различных дисков и дисковых устройств. Эти характеристики не могут быть изменены программно. В отличие от них, количество и размер секторов на каждой дорожке могут быть изменены программно, и их задание называется физическим форматированием диска.

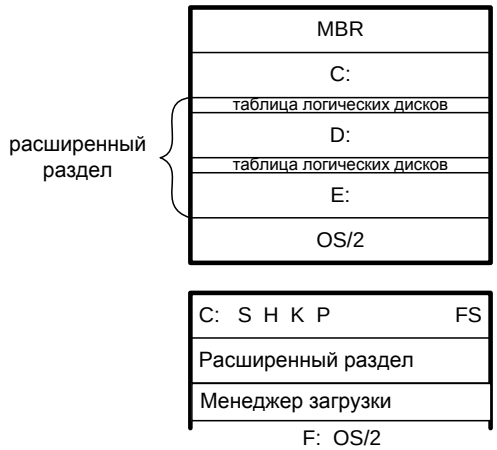
Кроме физического форматирования, осуществляется и логическое форматирование диска, при котором создается логическая структура диска. Под логической структурой подразумевается разделение общего пространства секторов диска на фиксированные области для различных целей. При форматировании в эти области заносится определенная информация, т.е. закладывается основа файловой системы.

Так как сектор – основная физическая единица, участвующая в любой дисковой информации, нужно знать как адресуется каждый отдельный сектор диска.

BIOS служит посредником в работе с аппаратными средствами и использует такой же способ адресации, что и контроллер дисков. Адрес состоит из номера поверхности, номера дорожки и номера сектора. Нумерация дорожек и поверхностей начинается с нуля. Нулевая – самая внешняя дорожка верхней поверхности. Нумерация секторов начинается с единицы. Такая нумерация называется абсолютной и относится к физическому диску в целом, независимо от того разбит ли он на логические диски.

ОС как надстройка BIOS использует более удобный способ адресации: секторы пронумерованы последовательно от периферии к центру и сверху вниз. Нумерация секторов начинается с нуля.

Каждая операционная система имеет средства для создания файловой системы FORMAT и FDISK.



Самый первый сектор жесткого диска содержит MBR, включающую в себя часть программы начальной загрузки и таблицу разделов диска. В таблице разделов указываются адреса и размеры разделов, на которые разбит диск. Всего в таблице разделов зарезервировано место для четырех записей о разделах. DOS предоставляет возможность создания не более двух разделов. Один из них, называемый первичным служит для размещения системных файлов и является загрузочным диском C:\.

Второй раздел называется расширенным, в нем можно создать один или несколько логических дисков. Каждый логический диск занимает целое число цилиндров. Каждому логическому диску, входящему в расширенный раздел, предоставляется/предшествует сектор, содержащий таблицу логических дисков. В этой таблице указывается адреса и размеры данного и следующего логических дисков, т.е. каждый диск будет описан 2 раза.

Таблица логических дисков располагается в самом первом секторе области, выделенной под логический диск. Каждый логический диск имеет свою относительную нумерацию секторов, начинающуюся с нуля. Сектор с таблицей логических дисков, как и сектор главного загрузчика не входит в систему относительной нумерации секторов и в этом смысле не принадлежит логическому диску.

8.Диски. Структура логического диска.

Под логической структурой подразумевается разделение общего пространства (секторов) диска на фиксированные области для различных целей - запись начальной загрузки, таблица распределения дискового пространства, основной каталог и область для данных. При форматировании в эти области записывается определенная информация - закладывается основа файловой структуры диска.

Запись начальной загрузки имеет все диски, даже если на них нет ОС. Здесь содержится программа-загрузчик ОС и параметрическая информация о диске. К параметрической информации относится размер сектора в байтах, размер кластера в секторах, размер FAT в секторах, количество элементов корневого каталога.

Кластер – это группа последовательных секторов, распределяемых как единый блок информации. Основная причина группировки секторов в кластеры – это ограниченная длина номера кластера.

Каждый элемент FAT за исключением первых двух соответствует определенному кластеру диска. Первые два элемента FAT используются идентификатором формата диска. Нумерация фактических элементов начинается с 2х. Нумерацию кластеров также принято начинать с 2.

Каждый элемент FAT содержит число, которое идентифицирует соответствующий кластер или как свободный или включенный в файл или неиспользуемый или зарезервированный для специальных целей.

Поскольку сектор - основная физическая единица, участвующая в любой дисковой операции, нужно прежде всего знать, как идентифицируется (адресуется) каждый отдельный сектор диска. BIOS служит посредником в работе с аппаратными средствами и использует такой же способ адресования, что и контроллер дисков. Адрес состоит из номера дорожки, номера поверхности и номера сектора. Нумерация дорожек и поверхностей начинается с 0 (самая внешняя дорожка, верхняя поверхность). Нумерация секторов начинается с 1. Такая нумерация еще называется абсолютной нумерацией и относится к физическому диску в целом, независимо от того, разбит ли он на логические диски.

DOS, как надстройка над BIOS, использует более удобный способ адресования - секторы пронумерованы последовательно от периферии к центру и сверху вниз. Нумерация начинается с 0 - это сектор 1 на поверхности 0 и дорожке 0 согласно нумерации BIOS. Дальше идут остальные сектора на той же стороне и дорожке, а затем продолжают сектора поверхности 1, дорожки 0 и т.д. до последней поверхности.

9. Метод файлового дескриптора для работы с файлами.

Существуют два метода работы с файлами. Первый метод основывается на использовании блока управления файлом FCB (File Control Block). Второй метод использует файловый манипулятор или дескриптор (File Handle) и иногда называется Handle - ориентированным методом. В методе файлового дескриптора ОС автоматом строит блок управления файлом в рабочей области при его открытии и его адрес неизвестен пользовательской программе. Блок и соответствующий файл идентифицируется 2-байтовым номером, который ОС возвращает программе после открытия файла. Этот номер и есть файловый дескриптор, т.е. при открытии файла пользовательская программа сообщает ОС его имя и получает обратно номер, который служит логическим именем файла при всех дальнейших операциях с файлами.

Открыть файл – это означает выделить для него фиксированную область памяти (блок управления), обнаружить файл и перенести определенную информацию из каталога в блок управления.

После выполнения функции OPEN файл идентифицируется при помощи дескриптора. Неоткрытый файл дескриптора не имеет и система работать с ним не может. Открывая файл, система назначает ему очередной свободный элемент в специальной системной таблице, называемой таблицей открытых файлов.

Информация из каталогов об открываемом файле записывается в этот элемент.

При открытии файла определяются права доступа программы к файлу: «только для чтения», «только для записи», «для чтения и записи». Кроме того определяются права доступа к файлу для других процессов, которые пытаются открыть файл до его закрытия главным процессом.

Для каждого файла устанавливается режим наследования, который указывает, как файл может быть использован порожденным процессом. Порожденный процесс может наследовать все открытые родительским процессом файлы в их текущем состоянии и с тем же режимом доступа.

Порожденный процесс может выполняться как независимая программа и работать с файлом в соответствие с режимом совместного использования.

Закрытие файла означает прекращение взаимодействия между блоком управления и файлом.

При закрытии файла ОС очищает все входные буферы, которые содержат записи этого файла, независимо от того, заполнены они или нет. Таким образом, обеспечивается целостность данных в файле. Файловый дескриптор и соответствующий блок управления освобождаются. При этом информация в каталоге заменяется информацией из блока управления.

В системной области хранятся идентификаторы всех открытых файлов. Таблиц, где хранятся идентификаторы, может быть несколько, тогда в заголовке первой таблицы имеется ссылка на след таблицу. Так же при открытии файла он должен существовать. Режим открытия файла: 1байт. С 0 по 2 биты – определяют режим доступа. Бит 3 зарезервирован. 4-6 режим совместного использования. 7 – режим наследования. Режимы доступа: 000-read, 001- запись, 010 – чтение/запись. Режим совместного использования определяет права доступа к файлу со стороны других процессов, которые пытаются открыть файл до его закрытия главным процессом. Также один и тот же файл можно открыть многократно со стороны одного процесса. При каждом открытии ОС создает новый дескриптор. Режим наследования определяет, как файл будет использоваться порожденным процессом. Если он равен 0 то порожденный процесс наследует все открытые файлы, иначе процесс будет выполнятся как независимая программа. При закрытие файла ДОС очистит все выходные буферы вне зависимости от того, заполнены они или нет, а дескриптор освобождается.

Существует два вида доступа к данным файла: последовательный и прямой. С точки зрения файловой организации файл – непрерывная последовательность байт.

Если файл состоит из записей разной длинны – его необходимо обрабатывать последовательно, при этом записи отделяются друг от друга специальными разделительными символами.

Организовать прямой доступ к файлу позволяет наличие указателя SFT – это есть байт относительно начала файла.

10. Хранение длинных имен.

Требование совместимости, которым должна удовлетворять система Windows означает что невозможно просто изменить существующий формат хранения данных на диске, который применяется в FAT. VFAT поддерживает как длинные так и короткие имена. 32х битный элемент каталога идентичен тому формату, который поддерживают предыдущие версии ОС. Метод работы с длинными именами файлов строится на использовании байта атрибута элемента каталога для коротких имен файлов. Установка младших 4х битов этого байта задает элементу каталога атрибуты: только чтение, скрытый, системный, метка тома. Добавление метки тома дает не имеющее смысла сочетание и защищает элемент каталога от изменения. Windows использует для формирования длинного имени несколько последовательных коротких имен элементов каталога, защищая каждое при помощи байта атрибута 0Fh.

Элементы с длинным именем файла располагаются в каталоге в соответствии с определенным форматом. Длинное имя файла не может существовать без связанного с ним элемента с коротким именем. Если есть такая ситуация, значит, нарушена целостность данных на диске.

Сегмент длинного имени располагается так:

Каждый 32х байтный элемент описывающий длинное имя содержит порядковый номер, защитный байт атрибута, и контрольную сумму. Порядковый номер позволяет виндовозу узнать о непоследовательном или некорректном изменении структуры каталога. Контрольная сумма вычисляется по связанному с данным файлом короткому имени, и если короткое имя изменится вне структуры винды то ОС поймет что элементы длинного имени больше не имеют смысла. Система хранит длинные имена файлов в виде символов таблиц юникод (2байта на символ). Поле тип нигде не используется.

Контрольная сумма вычисляется по связанному с длинным файлом короткому имени. Если короткое имя изменится вне среды Windows, то элементы длинного имени больше не будут иметь смысл.

Основная проблема при формировании короткого имени, связанного с длинным, заключается в создании уникального короткого имени, которое не совпадет с уже каким-нибудь существующим коротким именем.

При формировании коротких имен учитываются следующие правила:

- если длинное имя может быть принято как короткое, оно должно быть уникальным
- если из длинного имени не получается короткое, ОС выполняет ряд операций усечения и преобразования, чтобы получить допустимое короткое имя. Longna~1.doc

Для работы с короткими именами используют функции 56h и 7156h.

!!! Дальше идет инфа, которую нам не давала Караваева.

Для того, чтобы распространить использование длинных имен файлов на все типы приложений, Microsoft расширила набор функций прерывания MS DOS Int 21h для работы с длинными именами файлов, блокировки устройств с заменяемыми носителями и блокировки дисков. Расширение состояло в добавлении новых функций, которые полностью эквивалентны функциям Win32 API, и в совершенствовании существующих функций MS DOS, которые работают с именами. Обращения к новым и к модифицированным функциям Int 21h продолжают использовать стандартные для MS-DOS соглашения о передаче и возвращении параметров через регистры. Эти функции по-прежнему реализованы в виде 16-разрядного кода.

Прежде, чем использовать функции поддержки длинных имен файлов, программа должна выполнить несколько проверок. Во-первых, необходимо проверить версию MS-DOS. Если значение, возвращаемое функцией 30h прерывания Int 21h меньше 7, то функции использовать нельзя. Затем нужно проверить, работает ли программа под управлением Windows - в режиме DOS-приложений эти функции также использовать нельзя. Далее следует проверить, поддерживаются ли длинные имена файлов для данного устройства. Новая функция 71A0h прерывания Int 21h позволяет получить информацию об устройстве. В регистре BX возвращаются флаги, описывающие данное устройство.

11. Функции блокировки диска.

Поскольку Windows – многозадачная ОС, то к одному и тому же диску могут одновременно обращаться несколько приложений. Программы изменяющие структуры файловой системы без учета работы других приложений рискуют повредить данные, хранящиеся на дисках. Чтобы предотвратить потерю данных, операционная система берет на себя управление всеми запросами на прямой доступ к диску.

Дисковые утилиты и другие программы, напрямую изменяющие такие структуры файловой системы как элементы каталогов, должны перед внесением каких-либо изменений файловую систему применять монопольную блокировку тома.

Это предотвращает случайную запись на диск другими приложениями в тот момент, когда модифицируется файловая система.

Существует 4 уровня блокировки:

1) определяется уровень доступа приложений
2) разрешается читать диск
3) приложения не имеют доступа для чтения и записи
0) доступ к форматированию диска

Уровни с 1 по 3й образуют иерархию, которая ограничивает доступ к файловой системе на основе прав доступа, установленных для приложения в момент получения им блокировки уровня 1. Чем выше уровень, тем жестче ограничения в рамках данной иерархии.

Особенность 0го уровня состоит в наличии дополнительного подуровня с более жесткими ограничениями для программ, формирующих тома.

Прямая запись на диск допустима только при блокировке уровня 0 или 3. Для получения этого уровня приложение сначала должно получить блокировки 1 и 2. Особой осторожности требует блокировка системного SWAP-файла. Ядро Windows имеет доступ к этому файлу даже если приложение находится на уровне блокировки 3.

Уровень блокировки 2.

Запрещает другим процессам запись на диск, но разрешает чтение с диска. В зависимости от прав доступа указанных при блокировке уровня 1, система либо блокирует, либо отвергает операции. Вызов функции разблокировки понижает уровень блокировки до 1го и дает возможность системе выполнить ранее заблокированные операции, но разрешенные на более низком уровне блокировки.

Уровень блокировки 3.

Прежде чем его установить, программа вызывает специальную функцию, чтобы определить, не произошло ли на диске каких-либо изменений. Блокировка уровня 3 запрещает другим процессам как чтение, так и запись на диск. Операции чтения блокируется, а операции записи либо блокируются, либо отвергаются в зависимости от установленных прав доступа.

Блокировка уровня 3 устанавливает максимальный набор ограничений, устанавливая блокировку, ограничена в своих действиях.

Поскольку операции чтения на уровнях 3 блокируются, программа не может вызывать никакие функции пользовательского интерфейса или вывода на экран, не имеет права запускать другие приложения, загружать DLL и т.д.

Блокировка уровня 3 предназначена только для записи изменений на диск.

После того, как программа установит блокировку уровня 3, файловая система предпринимает меры для того, чтобы процесс мог вести прямую запись на диск.

Прежде всего система сбрасывает на диск содержимое всех файловых буферов и КЭШей, затем переводит кэш в режим сквозной памяти и закрывает все открытые файлы на уровне драйверов файловой системы.

Кроме того ОС фиксирует размер файла подкачки, сохраняя возможность его чтения и записи.

Перед снятием блокировки уровня 3 процесс обязан вернуть файловую систему в нормальное состояние. Процесс должен корректно модифицировать все данные файловой системы. Открытый перед блокировкой файл нельзя удалить, переименовать или переместить на другой том, иначе система перейдет в нестабильное состояние.

После снятия блокировки уровня 3 система разблокирует все ожидающие обработку операции чтения, вновь открывает закрытые файлы и возвращает кэш режим отложенной записи.

Блокировка 0 уровня.

Невозможна для тома, на котором открыты файлы или описатели. Поскольку ОС всегда открывает какие-либо файлы, приложение никогда не получит блокировку 0 уровня на томе с системными файлами Windows.

12. Правила блокировки.

Приложения, блокирующие или изменяющие тома должны соблюдать следующие требования, чтобы не снизить эффективность системы и предотвратить потерю данных.

1) если на томе нет открытых файлов, операции прямой записи на диск следует выполнять при установленной блокировке уровня 0, а если есть – то использовать иерархию блокировок и выполнять операции прямой записи на уровне блокировки 3.

2) чтобы предельно сократить период блокировки уровня 3, приложения на этом уровне должны выполнять только дисковый ввод/вывод. Снимая блокировку 0 и 3го уровней, приложение обязано привести файловую систему в состояние, согласующееся с тем, что было до блокировки.

3) нельзя вызывать функции 21 прерывания, пока информация на диске находится в промежуточном состоянии.

4) приложение не должны перемещать файл подкачки.

5) приложения должны обращаться к диску исключительно через низкоуровневые функции.

6) приложение, находясь на 3м уровне блокировки не должны отдавать управление, обновлять изображение на экране, запускать другие программы, т.е. все, что может заставить Windows подкачать новый, или ранее выгруженный сегмент.

7) весь код Windows приложения, действующий на 3м уровне блокировки должен находиться в функции, обрабатывающей какое-то одно сообщение.

IFS-менеджер и FSD-драйверы.

Функции блокировки не явл-ся частью ОС, а реализ-ся спец. виртуальным драйвером IFSMgr(IFS диспетчер)

IFS перехватывает прерывание int 21h и проверяет, вызывается ли ф-я группы 71h и обрабатывает данный вызов. Основная роль – получение всех вызовов ф-и, относящихся к файловой системе, преобразование их в обращение к подходящ. IFS интерфейсу, и передаче этих обращений соответствующему драйверу файловой системы. IFS загружается в ходе реализации системы. Он постоянно нах-ся в памяти и должен оказываться там до того, как будет работать один из драйверов файловой системы FSD. IFS позволяет нескольким FSD работать параллельно. При каждом вызове FSD передаёт диспетчеру IFS адрес одной точки входа, опред. ф-ю, котор. будет вызвана диспетчером IFS при первом его обращении к FSD.

3 способа определения, какой FSD вызвать для удовлетвор. конкретного обращения:

1.Если при обращении к ф-и, в кач-ве одного из пар-ров указ-ся путь, диспетчер IFS исп-т обозначающую дисковод букву или сразу всё имя для того, чтобы определить целевой FSD.

Каждый FSD представляет собой отдельный VXD-драйвер – виртуальный драйвер внешнего устройства, ответственный за реализацию семантики его собственной файловой системы. VXD-драйвер – низкоуровневый программный модуль, который управляет отдельным ресурсом. Вся инфо об орг-ции конкретн. файловой системы содержится исключительно в пределах кода FSD, диспетчер IFS работает только с дискрипторами, и только FSD знает, каким д-м на томе файл. сис-мы соотв-ют поступившее от приложения имя. Единственная точка входа, кот. передаёт FSD при своей регистрации у диспетчера IFS, определяет ф-ю монтажирования тома. Эта функция входит в состав набора стандартных директорий, опр-ет для интерфейса диспетчера IFS. Когда IFS обращается к одной из точек входа, FSD возвращает указатель на таблицу дополнительных точек входа. Последующие вызовы IFS при помощи этих нов. точек входа будут адресоваться уже к конкретным функциям.

При первом обращении к устройству или при смене носителя, диспетчер IFS вызывает нач. ф-ю файловой системы. Такой вызов просит FSD попытаться смонтировать том. При этом FSD должен распознавать формат носителя или устройства. Если ему это удаётся, он возвращает дисп. IFS дескриптор тома и указатель на начало таблицы ф-й. Дескриптор будет использоваться при всех последующих обращениях к данному FSD. Для дисков дескриптор тома обозначает лог. раздел HDD, либо конкретную дискету. Диспетчер IFS нах-ся на самом верхнем уровне представляет собой единств. вирт. Драйвер внешнего устройства, кот. обеспечивает интерфейс м\у запросами приложения, и конкретной файловой системой, к которой обращается это приложение. Диспетчер IFS преобразует эти обращения в обращения к след. уровню – уровню файловой системы.

13. Ассемблеры. Режимы адресации

Трудно установить различие между ассемблером и компилятором, но можно сказать, что ассемблеры переводят машинно-ориентированные языки, а компиляторы – проблемно-ориентированные языки в машинные коды. Ассемблер образует программу, команды которой отражают внутреннюю структуру машины.

Существует ряд основных функций, таких как трансляция мнемонических кодов операций в их эквиваленты на машинном языке или присваивание машинных адресов символическим меткам, которые должны выполняться любым ассемблером.

Однако за пределом этого базового уровня возможности, предоставляемые ассемблерами, а также схемы их построения сильно зависят от входного языка, а также языка машины.

Ассемблеры – машинно-ориентированные языки (состоят из мнемокодов-команд процессора).

Одним из аспектов машинной зависимости является имеющиеся различия в форматах машинных команд и кодах операций. С другой стороны некоторые средства языка не имеют прямой связи со структурой компьютера. Их выбор является произвольным и определяется разработчиком языка. Многие Ассемблеры представляют гибкие средства обработки исходной программы и соответствующего ей объектного кода.

Одни из этих средств позволяют располагать сгенерированные машинные команды и данные в объектной программе в порядке, отличном от порядка, в котором расположены соответствующие им исходные предложения.

Другие позволяют создавать несколько независимых частей объектной программы. Каждая из этих частей сохраняет свою индивидуальность и обрабатывается загрузчиком отдельно от других частей.

Для перевода исходной программы в ее объектное преставление необходимо:

1) Преобразовать мнемонические коды операций в их эквиваленты на машинном языке.
2) Преобразовать символические операнды в эквивалентные им машинные адреса.
3) Построить машинные команды в соответствующем формате.
4) Преобразовать константы, заданные в исходной программе во внутренние машинные представления.
5) Записать объектную программу и выдать листинг.

Все указанные действия (кроме 2) могут быть выполнены простой построчной обработкой исходной программы, но трансляция адресов вызывает определенные трудности т. к. неизвестны адреса, которые будут присвоены меткам поэтому большинство Ассемблеров выполняют 2 просмотра исходной программы.

Основной задачей первого просмотра является поиск символических имен и назначение им адресов.

Фрагменты программы, содержащие ссылки “вперед” запоминаются во время первого просмотра. Дополнительный просмотр таких заполненных определений выполняются по мере продвижения этого процесса ассемблирования, по завершению этого процесса выполняется обычный 2-й просмотр. Фактическая трансляция выполняется во время второго просмотра.

Ассемблер наряду с трансляцией команд исходной программы должен так же выполнять и директивы Ассемблера. Эти директивы не переводятся непосредственно в машинные коды, а управляют работой самого Ассемблера. В заключительной фазе своей работы Ассемблер должен записать полученный объектный код на некоторые устройства вывода.

Если рассматривать объектный код отдельно то в общем случае невозможно определить какие значения представляют адреса, зависящие от местонахождения программы, а какие – не изменяемые объекты. Поскольку Ассемблеру не известен фактический адрес начала загрузки он не может выполнить необходимую настройку адресов, используемых программой. Однако Ассемблер может указать загрузчику те части объектной программы, которые нуждаются в настройке при загрузке. Объектная программа содержит информацию необходимую для выполнения подобной модификации называется перемещаемой программой.

Не обязательно внутри объектной программы сгенерированные машинные коды и данные должны располагаться в том же порядке, в котором они были в исходном тексте. Многие Ассемблеры предоставляют средства, позволяющие создавать несколько независимых частей объектной программы. Часть программы, которая сохраняет после Ассемблирования свою индивидуальность и может загружаться и перемещаться независимо от других, называется управляющей секцией (УС)

Имена управляющей в одной УС не может непосредственно используются в другой секции, но поскольку управляющие секции образуют связные части программы, то необходимо предоставить средства для связывания их друг с другом. Команда одной управляющей секции должны иметь возможность ссылаться на команды или области данных, расположенные в другой секции. Такие ссылки нельзя обрабатывать обычным образом, поскольку управляющие секции загружаются и перемещаются независимо друг от друга а Ассемблеру ничего не известно о том где будут расположены другие управляющие секции во время выполнения программы такие ссылки между управляющими. Называются внешними ссылками для каждой внешней ссылки Ассемблер генерирует информацию, которая дает возможность загрузчику выполнить требуемые связывание программы. Ассемблер должен запоминать в какой управляющей секции было определено то или иное имя. Любая попытка использовать имя из другой управляющей секции должен отмечаться как ошибка, если это имя не объявлено в качестве внешнего. Ассемблер должен решать использовать одинаковые имена в разных управляющих секциях. Для занесения внешних имен Ассемблер оставляет место в объектном коде. Кроме того он должен включить в объектную программу информацию, позволяющую загрузчику занести необходимые данные куда требуется.

Для этого надо определить 2 типа записей:

- Запись-определение (содержит информацию о внешних именах, определенных в данной управляющей секции)

- Запись–ссылка (содержит список имен, используемых в качестве внешних ссылок)

Большинство Ассемблеров наряду с одиночными терминами разрешается использовать выражения. Выражения классифицируются как абсолютные или относительные, в зависимости к какому типу относятся их значения.

Наличие нескольких управляющих секций, каждая из которых может перемещаться независимо от других, создает дополнительные трудности при обработке выражений.

Когда в выражении используются внешние ссылки, Ассемблер, в общем случае, не может определить, корректно оно или нет.

Группировка относительных терминов для проверки корректности не может быть сделана, если неизвестно какие термины принадлежат одной из секций. А во время Ассемблирования эта информация отсутствует. В этом случае Ассемблер вычисляет известные ему термины и комбинирует их так, чтобы получить начальное значение выражения. Кроме того, он генерирует записи-модификаторы, позволяющие загрузчику закончить вычисления.

Команды центрального процессора рассматриваются как операционные ресурсы нижнего уровня доступные программисту.

Использование операционных ресурсов планируются в виде последовательности машинных команд процессора, которые в языке ассемблера отличаются названием операций и форматами операндов, что синтаксически определяется режимами адресации.

Главной особенностью языка ассемблера является использование одного названия операций для всех семантически идентичных действий независимо от форматов и размеров данных.

Форматы данных определяются типом именованных областей памяти или явными указателями.

Режимы адресаций можно разделить на 7 групп:

1) регистровая адресация (mov AX,BX) при которой микропроцессор извлекает операнд из регистра и помещает результат в регистр.

2) непосредственная адресация, которая позволяет указывать значение константы в качестве операнда источника mov cx,25h

3) Прямая, при которой исполнительный адрес является составной частью команды. Микропроцессор добавляет этот адрес к содержимому регистра DS и получает 20-битовый физический адрес операнда (mov AX, DAN) DAN – символическое имя, определяемое в сегменте данных.

4) косвенно регистровая, при которой исполнительный адрес содержится в базовом регистре BX, указателе базы BP или индексном регистре SI или DI (mov DI, offset DAN mov AL, [BX]). Косвенные регистровые операнды заключаются в квадратные скобки.

5) адресация по базе (mov AL, [BX]+4) Ассемблер вычисляет исполнительный адрес с помощью сложения содержимого регистров BX или BP, в которые загружается базовый адрес со значением сдвига.

6) прямая адресация с индексированием

(mov DI, 5

mov AL, TABL[DI]) загрузили 6й элемент массива

Исполнительный адрес вычисляется как сумма значений сдвига и содержимого индексного регистра SI или DI.

7) адресация по базе с индексированием - адрес высчитывается как сумма значений базового регистра, индексного регистра и сдвига (mov AX,AL[BX][DX]

mov AX,[BX+2+DI]

mov AX, [BX][DI+2])

14. Структура машинной команды. Префикс замены сегмента в памяти.

Машинная команда должна содержать следующую информацию:

1) код операции, который помещается в первом байте команды.

2) способ адресации и собственно адрес. В команде должно быть закодировано: содержаться ли операнды в регистрах, или это непосредственные данные, или операнд находится в памяти.

Форматы команд ЦП в языке ассемблера строится в соответствие с двоичной внутренней машинной формой представления, по которой обобщенный формат использует базовую 2х байтную структуру для построения основной части команды, имеющей вид.

15 0	15 8	7 0
000000 d/o w	mod reg/000 r/m	d8 / d16 / d32	i8 / i16 / i32
Кодируется КОП и способ адр-ии	Пост-байт	Размер данных	

Размер данных и количество следующих байтов команды определяются конкретным кодом операции и наличием в команде фиксированных байтов, префиксов, разрядностей данных и адресаций.

Буквами 0 обозначены байты КОП.

d - направление пересылки результатов

w – размер данных: 0 – байт; 1 – слово / двойное слово

mod – модификация операндов

r/m – регистр или указатель типа операнда

reg/000 – могут продолжаться биты КОП

В машинной форме после основной части команды с указателями модификации адресов памяти может рассматриваться 2х байтный адрес для прямой адресации, однобайтное или 2х байтное смешение, одно или 2х байтные непосредственные данные, 2х байтные смещения и 2х байтный сегментный адрес для прямой межсегментной адресации.

Длина команды ЦП без префиксов может достичь 6 байт. Некоторые группы команд имеют дополнительные сокращенные форматы, которые обрабатываются быстрее.

Если на код операции и режим адресации выделено 2 байта, то поля mod и r/m а также бит w определяют разные режимы адресации и назначение сегментных регистров по умолчанию.

Второй байт, используемый для адресации почти всех операндов называется пост-байтом.

Mod и r/m используются совместно. Дают 32 комбинации. 8 регистров и 24 режима адресации.

Reg/000 это регистры во втором операнде и код операции.

Примеры:

1) команда mov Si,BX

расширяем побитно структуру команды

100010	D	W	Mod	Reg	r/m
КОП		Длина операнда	Работа с регистрами		BX

2) mov DS,DX используется сегментный регистр

10001110	Mod	Reg	r/m
Коп	Работа с 2мя регистрами		

3) пересылка непосредственных данных в регистр

mov AH,9h

1011	0	100	00001001
КОП	W	AH	

Комплекс префиксов и суффиксов обеспечивает выбор режимов выполнения команд. Команды могут использовать следующие группы префиксов, систематизированных по семантическим признакам:

1) Задаваемые в виде предварительной операции

LOCK-префикс для блокировки шины данных (0F0h)

REP[xx]-префиксы повторения команд обработки строк

2) Задаваемые явно в операнда префиксами задания сегментов

DS:,CS:,SS:,ES:. Или неявно через директиву управления доступом к сегментам памяти через директиву ASSUME.

3) префикс определяется режимом трансляции и форматом операнда
префикс 32-х разрядного адреса
префикс 32-х разрядных данных

4) префикс замены сегментов в памяти. При переназначении сегм. Регистра процессор создает байт следующего вида:001RG110

Если режим адресации использует регистр BP для формирования физического адреса, по умолчанию используется содержимое сегментного регистра SS, а в других режимах адресации – содержимое регистра DS.

Если в команде используется префикс замены сегмента, то длина основной команды увеличивается на 1 байт префикса переключения сегментов, а время выполнения – на 2 цикла.

Аналогичная замена невозможна, при вычислении адреса текущей команды. В этом случае всегда используется регистр CS.

При выполнении основных манипуляций чтения и записи со стеком с использованием регистра SP всегда подключается сегментный регистр ES.

Программист может использовать сегментный префикс и переназначать сегментное назначение. При переназначении сегментного регистра процессор создает байт 001RG110.

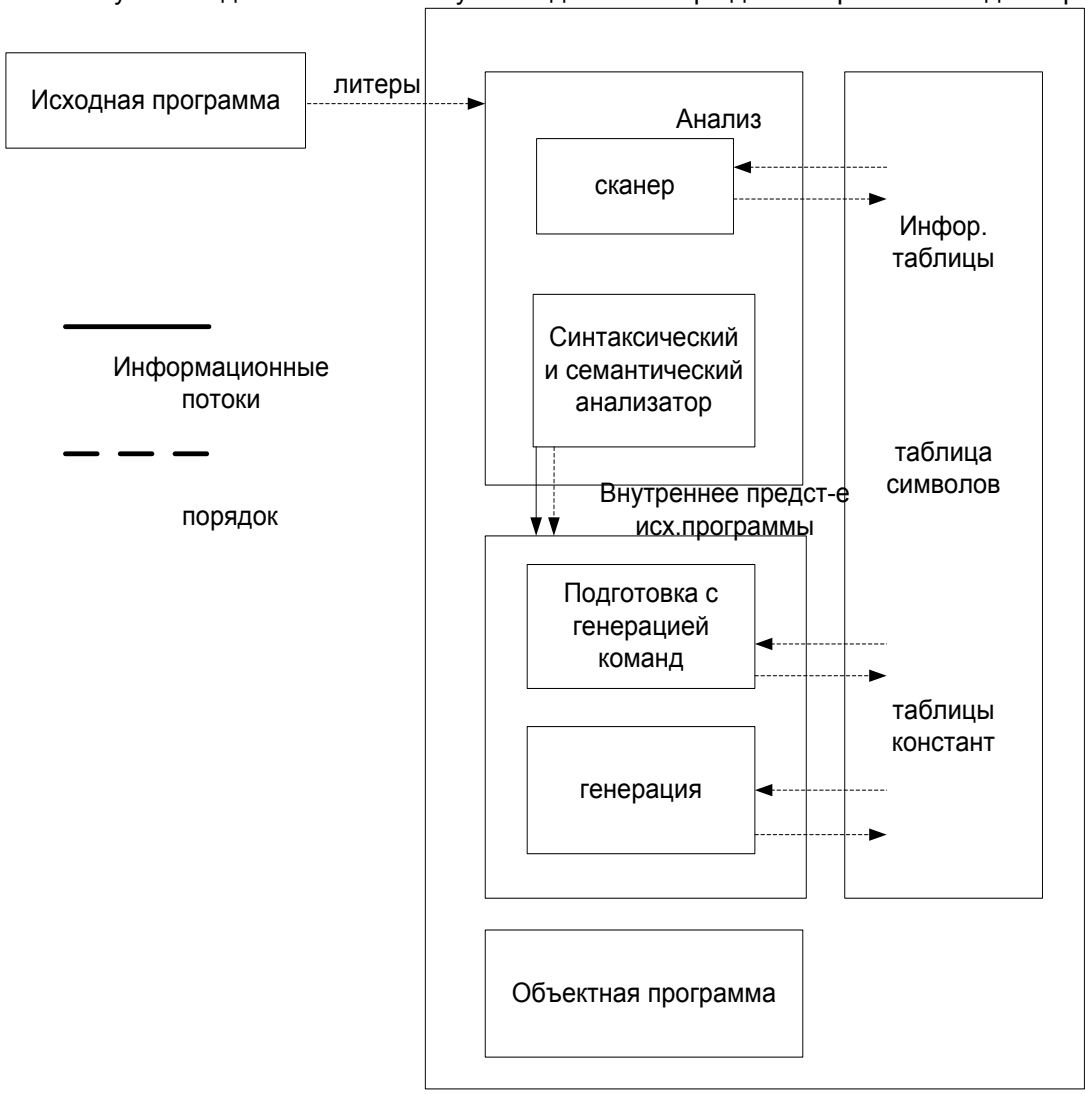
Транслятор при построении объектного кода также встраивает сегментные префиксы в команды в случае необходимости.

15. Компиляторы. Формальные языки и грамматики.

Трансляторы, компиляторы и интерпретаторы.

Интерпретатор-программа , которая не выдает результирующей программы.

Целью компилятора является трансляция программы, написанной на языке высокого уровня в машинные коды некоторого компьютера. Большинство ЯВУ построено так, чтобы быть относительно независимыми от тех машин на которых будут использоваться. Это означает что процесс анализа синтаксиса программ написанных на этих языках д.б. также машинно независимым. Но процесс генерации кода является машинно- зависимым поскольку необходимо знать систему команд компьютера для которого этот код генерируется



Компилятор должен выполнять анализ исходной программы, а затем синтез объектной программы. Сначала исх. программа разлагается на ее составляющие части. Затем из них строятся ее составные части. Затем из них части эквивалентной объектной программы. Для этого на этапе анализа компилятор строит несколько таблиц, которые затем используются как при анализе, так и при синтезе. Для облегчения построения компиляторов ЯВУ обычно описываются в терминах некоторой грамматики. Эта грамматика определяет форму или синтаксис допустимых предложений может быть сформулирована как проблема поиска соответствия, программистом предложений, строкам, определенным грамматикой и генерацией кода для каждого предложения Наиболее существенная часть для любого процесса компиляции это выделение в исходной программе элементарных составляющих, идентификаторов, ограничителей, операторов, чисел и так далее, которые называются лексемами. Предложения исходной программы удобнее представить в виде последовательности таких лексем. Просмотр исходной программы удобнее представить в виде последовательности таких лексем. Просмотр исходной программы распознавания и классификации различных лексем называются лексическим анализом. Часть компилятора, которая выполняет эту функцию называют лексическим анализатором или сканером. Обычно сканер последовательно читает строки исх. программы разбивает их на лексемы и создает несколько таблиц, используемых на дальнейших этапах.

1)Таблица терминальных символов - таблица каждой строке, которой соответствует информация об одном из терминальных символов. Терминальные символы: арифметические операции, ключевые слова и так далее

2)Таблица переменных , т.е таблица каждой строке которой сооответвтвует информация о каждой переменной, имеющихся в программе, либо созданных компилятором в процессе выполнения на этапе синтаксического анализа.

3)Таблица констант.

Как только лексемы выделены , каждое предложение программы м.б. распознано как некоторая конструкция языка. Процесс называемый синтаксическим анализом осуществляется той частью компилятора, которая называется синтаксическим анализатором. Здесь лексемы полученные на стадии лексического анализа используется для идентификации более крупных программных структур. Синтаксический анализ обычно чередуется с семантическим. Сначала синтаксический анализатор идентифицирует последовательность лексем,

формируя синтаксическую единицу, а затем вызывается синтаксическую единицу, а затем вызывается семантический анализатор. Процесс распознавания предложений исходной программы рассматривают как постр-е дерева грамматического разбора для этих предложений. Последним шагом базов. Схемы процесса трансляции является генерация объектного кода.

Алфавит - это конечное множество символов, которое в дальнейшем будем обозначать **V**. Предполагается, что термин "символ" имеет достаточно ясный интуитивный смысл и не нуждается в дальнейшем уточнении.

Цепочкой символов *в алфавите V* называется любая конечная последовательность символов этого алфавита. Далее цепочки символов будем обозначать греческими буквами: □,□□, □, …

Цепочка, которая не содержит ни одного символа, называется ***пустой цепочкой***. Для ее обозначения будем использовать символ □.

Более формально цепочка символов в алфавите **V** определяется следующим образом:

(1) □ - цепочка в алфавите **V**;

(2) если □ - цепочка в алфавите **V** и **a** - символ этого алфавита, то □**a** - цепочка в алфавите **V**;

(3) □ - цепочка в алфавите **V** тогда и только тогда, когда она является таковой в силу (1) и (2).

Длиной цепочки называется число составляющих ее символов. Например, если □ = abcdefg, то длина □ равна 7. Длину цепочки □ будем обозначать | □ |. Длина □ равна 0.

Основной операцией над цепочками символов является операция ***конкатенации*** - это дописывание второй цепочки в конец первой, т.е. если □ и □ - цепочки, то цепочка □□ называется их конкатенацией. Например, если □ = ab и □ = cd, то □□ = abcd. Для любой цепочки □ всегда □□ = □□ = □.

Операция конкатенации не обладает свойством коммутативности, т.е. □□□□□, но обладает ассоциативностью (□□)□□= □(□□).

Обращением *(*или ***реверсом****)* цепочки □ называется цепочка, символы которой записаны в обратном порядке. Обращение цепочки □ будем обозначать □$^{R.}$

Например, если □ = abcdef, то □^{R} = fedcba. Для пустой цепочки: □ = □^{R}.

Для операции обращения справедливо следующее равенство "a,b: $(ab)^R = b^R a^R$.

N-ой степенью цепочки a (будем обозначать □^{n}) называется конкатенация n цепочек □. □^{0} = □; □^{n} = □□$^{n-1}$ = □$^{n-1}$□.

В общем случае ***язык*** - это заданный набор символов и правил, устанавливающих способы комбинации этих символов между собой для записи осмысленных текстов. Основой любого естественного или искусственного языка является алфавит, определяющий набор допустимых символов языка.

Язык в алфавите **V** - это подмножество цепочек конечной длины из множества всех цепочек над алфавитом **V**. Из этого определения следует два вывода: во-первых, множество цепочек языка не обязано быть конечным; во-вторых, хотя каждая цепочка символов, входящих в язык, обязана иметь конечную длину, эта длина может быть сколь угодно большой и формально ничем не ограничена.

Все существующие языки попадают под это определение. Большинство реальных естественных и искусственных языков содержат бесконечное множество цепочек. Часто цепочку символов, принадлежащую заданному языку, называют ***предложением*** языка.

Обозначим через **V*** множество, содержащее все цепочки в алфавите **V**, включая пустую цепочку □. Например, если V = {0,1}, то V* = {□, 0, 1, 00, 11, 01, 10, 000, 001, 011, ...}.

Обозначим через **V**$^{+}$ множество, содержащее все цепочки в алфавите **V**, исключая пустую цепочку □. Следовательно, верно равенство **V* = V**$^{+}$ □ **{□}**.

Известно несколько различных *способов описания языков*:

1. Перечислением всех допустимых цепочек языка.

2. Указанием способа порождения цепочек языка (заданием грамматики языка).

3. Определением метода распознавания языка.

Первый из методов является чисто формальным и на практике не применяется, т.к. большинство языков содержат бесконечное число допустимых цепочек и перечислить их просто невозможно.

Второй способ предусматривает некоторое описание правил, с помощью которых строятся цепочки языка. Тогда любая цепочка, построенная с помощью этих правил из символов алфавита языка, будет принадлежать заданному языку.

Более всего интересующий нас третий способ предусматривает построение некоторого логического устройства (распознавателя) - автомата, который на входе получает цепочку символов, а на выходе выдает ответ: принадлежит или нет эта цепочка заданному языку. Например, читая этот текст, вы сейчас в некотором роде выступаете в роли распознавателя (надеемся, что ответ о принадлежности текста русскому языку будет положительным).

Синтаксис языка - это набор правил, определяющий допустимые конструкции языка. Синтаксис определяет «форму языка» - задает набор цепочек символов, которые принадлежат языку. Чаще всего синтаксис языка можно задать в виде строгого набора правил, но полностью это утверждение справедливо только для чисто формальных языков.

Например, строка «3+2» является арифметическим выражением, «3 2 +» - не является.

Семантика языка - это раздел языка, определяющий значение предложений языка. Семантика определяет «содержание языка» - задает значение для всех допустимых цепочек языка.

Например, используя семантику алгебры мы можем сказать, что строка «3+2» есть сумма чисел 3 и 2, а также то, что «3+2 = 5» - это истинное выражение.

Лексика - это совокупность слов (словарный запас) языка. Слово или лексическая единица (лексема) языка - это конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций.

Грамматика - это описание способа построения предложений некоторого языка. Иными словами, грамматика - это математическая система, определяющая язык.

Формально ***порождающая грамматика*** **G** - это четверка **G (VT, VN, P, S)**, где

VT - множество *терминальных символов (терминалов)*,

VN - множество *нетерминальных символов (нетерминалов)*, не пересекающееся с VT,

P - множество правил (продукций) грамматики вида □ □ □, где □□V^+, □□V^*,

S - *целевой* (*начальный) символ* грамматики, S □ VN.

Для записи правил вывода с одинаковыми левыми частями

□ □ $□_1$ □□□ □ $□_2$ ... □ □ $□_n$

будем пользоваться сокращенной записью

□ □ $□_1$ | $□_2$ |...| $□_n$.

Каждое $□_i$, i = 1, 2, ... ,n , будем называть *альтернативой* правила вывода из цепочки □.

Такую форму записи правил грамматики называют ***формой Бэкуса-Наура***. Она предусматривает также, что все нетерминальные символы берутся в <> скобки.

Пример: <число>::= <цифра>|<цифра><число>

<цифра>::= 0|1|2|3|4|5|6|7|8|9

16. Компиляторы. Формальные языки и грамматики. Классификация Хомского.

По класс. Хомского выделяют 4 грамматики:

1.Тип 0 (ноль). G=(VT,NV,P,S) V=VNUVT

V называется грамматикой типа 0, если на ее правила вывода не оказывается никаких ограничений. Правила имеют вид: α->β где αЄV*, βЄV*.

2.Тип 1. Грамматику типа 1 можно определить как контекстно зависимую, либо как неукорачивающую. Грамматика называется контекстно зависимой, если каждое правило из P имеет вид: α1A α2-> α1βα2, где α1α2βЄV* AЄVN βЄB. Грамматика называется неукорачивающей, если каждое правило из P имеет вид: α1->β, где α,βЄV+ |β|>=| α|

Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых неукорачиваемыми грамматиками совпадает с множеством языков пораждаемых контекстно зависимыми грамматиками.

3. Тип 2. Грамматику типа 2 можно определить как контекстно свободную либо как укорачивающую контекстно свободную. Грамматика называется контекстно свободной если любое правило из P имеет вид: A-> β AЄVN β ЄV+

Грамматика называется, укорачивающей контекстно свободной, если любое правило из P имеет вид: A-> β AЄVN β ЄV+

Возможность выбора обусловлена тем, что для каждой укорачивающей контекстно свободной грамматики сущ-ют почти эквивалентная ей контекстно свободная грамматика.

4. Тип 3. Грамматику типа 3, можно определить либо как праволинейную либо как леволенейную. Грамматика называется праволенейной, если любое правило из P имеет вид: A->γβ A->γ A,BЄVN, γЄVT

Грамматика G называется леволенейной, если каждое из правил имеет вид: A->γβ A->γ A,BЄVN, γЄVT.

Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых праволенейными грамматиками совпадает с множеством языков, порождаемых леволинейными грамматиками.

Язык L(G) является языком типа R если его можно описать грамматикой типа K. Языки классифицируются в соответствии с типами грамматик, с помощью которых они созданы.

*Тип 0 – это языки с фразовой структурой (самые сложные языки, разговорные). *Тип 1 – контекстнозависимые языки. Время на распознавание предолжений языка экспоненциально зависит от длины исходной цепочки символов. Языки и грамматики типа 1 применяются в анализе и переводе текстов на естественных языках. Распознаватели, построенные на их основе, позволяют анализировать тексты с учетом контекстной зависимости в предложениях входного языка. *Тип 2 – контекстно свободные языки. КС языки лежат в основе синтаксических конструкций большинства современных языком программирования. На их основе функционируют некоторые сложные командные процессоры, допускающие управляющие команды цикла и условия. Эти обстоятельства определяют распространенность данного класса языков. В общем случае время на распознавание предложений языка, относящегося к типу 2 полиноминально зависит от исходной цепочки символов, но среди КС языков существует много классно языков, для которых эта зависимость линейна. *Тип 3 – регулярные языки. Время на распознавание предложений всегда линейно зависит от длины исходной цепочки символов. Данные языки лежат в основе простейших конструкций языков программирования, на их основе строятся мнемокоды команд. Для работы с регулярными языками можно использовать регулярные множества и выражения, а так же конечные автоматы.

18. Компиляторы. Формальные языки и грамматики. Цепочки вывода

Выводом называется процесс порождения предложения языка на основе правил, определяющих язык грамматики. Цепочка β=δ1γδ2 называется непосредственно выводимой из цепочки α=δ1ωδ2 в грамматике G(VT,VN,P,S) V=VTUVN δ1γδ2ЄV*; ωЄV+, если в грамматике G существует правило ω->γЄγ

Непосредственная выводимость цепочки β из цепочки α обозначается как α=>β. Иными словами, цепочка β выводима из цепочки α в том случае, если можно взять несколько символов цепочки α, заменить их на другие символы согласно некоторому правилу грамматики можно получить цепочку β. Любая из цепочек δ1 и δ2 может быть пустой. В предельном случае вся цепочка α может быть заменена на цепочку β тогда в грамматике G должно существовать правило: Цепочка β называется выводной из цепочки α в том случае, если выполняется одной из 2-х условий: 1. α=>β 2.сущ γ такая, что α=>* γ и γ=> β

Такое определение выводимости цепочки является рекурсивным. Суть его заключается в том, что цепочка β, выводимая из α, если может построить последовательность непосредственно выводимых цепочек от α к β следующего вида: α=>γ1=> γ2=>….=> γn=>β n>=1

Такая последовательность непосредственно выводимых цепочек называется выводом или цепочкой вывода. Каждый переход от одной непосредственно выводимой цепочки к следующей цепочке вывода называется шагом вывода.

α=>+β Если известно кол-во шагов вывода то можно выводить в такой форме: $\alpha =>^{4} \beta$

Пример: G({0,1,2,3,4,5,6,7,8,9,+,-,},{S,T,F},P,S)

P: S->T |+T |-T T->F|Tf F->0|1|2|3|4|5|6|7|8|9

Построим несколько произвольных цепочек вывода.

1. S=>-T=>-TF=>-TFF=>-FFF=>-4FF=>-4FF=>??
2. S=>TF=>T8=>F8=>18
3. T=>TF=>T0=>TF0=>T50=>F50=>350
4. TFT=>TFTF=>TFFF=>FFFF=>1FFF=>1FF4=>10F4
5. F=>5

1.$S =>^{*} -479$ или $S =>^{+} -479$ или $S =>^{7} -479$

2. $S =>^{*} 18$ или $S =>^{+} 18$ или $S =>^{5} 18$
3. $T =>^{*} 350$ или $T =>^{+} 350$ или $T =>^{6} 350$
4. $TFT =>^{*} 1004$ или $TFT =>^{+} 1004$ или $TFT =>^{7} 1004$
5. $F =>^{*} 5$ или $F =>^{1} 5$

Вывод называется законценым, если на основе цепочки β, полученной в результате вывода нельзя больше сделать ни одного шага вывода т.е. вывод называется законченным, если цепочка β – пустая или содержит только терминальные символы грамматики. Цепочка β, получается в результате законченного вывода, называется конечной цепочкой вывода. Цепочка символов αЄV* называется сентенциальной формой грамматики, если она выводима из целевого символа грамматики S=>*α. Если цепочка αЄVT получена в результате законченного вывода, то она называется конечной сентенциальной формой. Язык L с заданной грамматикой G(VT,VN,P,S) – это множество всех конечных сентенц-х форм грамматики G. Алфавитом такого языка L(G) будет множество терминальных символов грамматики (VT), т.к. все конечные сентец-е формы грамматики это цепочки над алфавитом VT. Вывод может быть левосторонним и правосторонним, если в нем на каждом шаге вывода правила грамматики применяются всегда к крайнему левому нетерминальному символу в цепочке т.е. вывод называется левосторонним, если на любом шаге вывода происходит подстановка цепочки символов на основании правила грамматики вместо крайнего левого нетерминального символа в исх цепочке, аналогично для правостороннего вывода. В грамматике для одной и той же цепочки может быть несколько выводов, эквивалентных в том смысле, что в них в одних и тех же местах применяются одни и те же правила вывода, но в различном порядке.

Для контекстно свободных грамматик. КС грамматика G называется неоднозначной, если сущ хотя бы одна цепочка αЄL(G), для которой може быть построено два или более различных деревьев вывода. Это утверждение эквивалентно тому, что цепочка α имеет два или более разных левосторонних или правосторонних выводов, иначе грамматика называется однозначной. Если грамматика используется для определения языка программирования то она должна быть однозначной.

Класс языка определяется классом самой простой, в смысле иерархии Хомского, из описывающих его грамматик. Для языка, определенного регулярной грамматикой всегда можно написать конекстно свободную грамматику. Одной из наиболее простых и широко используемых форм записи грамматик является нормальная форма Бэкуса-Наура (НФБН). Терминальные символы записываются как обычные символы алфавита, а нетерминальные – как имена в угловых скобках.

<число>:=<цифра>|<цифра>|<число>

<цифра>->0|1|2|3|4|5|6|7|8|9

Грамматика БНФ состоит из подмножества правил вывода, каждая из которых определяет синтаксис некоторой конструкции языка прграммирования.

<read>->READ(<id_list>)

<id_list>->id|<id_list>,id

Результат анализа исх предложения, в терминах грамматич-х конструкций удобно представить в виде дерева, которое называется деревом грамматического разбора или синтаксическим деревом.

Для предложения READ(VALUE) дерево должно выглядеть след образом:

<read>

|

| | <id_list> |

READ ( )

{value}

Характер распознаваемых строк может намного успростить процесс лексического анализа, например любые вещественные числа могут сгенерировать посредством регулярного выражения (+/-) цифра*.цифра*. Реальное числосотавление:

1.Возможно знак

2.Последовательность из 0 или > цифр

3. (**.**)

4. Последовательность из 0 или > цифр

17. Компиляторы. Формальные языки и грамматики. Распознаватели.

Распознаватели могжно классифицировать в зависимости от вида составляющих их компонентов (считывающее у-во, УУ, внешняя память).

По видам считывающего у-ва распознаватели могут быть двусторонними и односторонними. По видам УУ распознаватели могут быть детерминированные и недетерминированные. Распознаватель называется детерминированным в том случае, если для каждой допустимой конфигурации распознавателя, которая возникла на некотром шаге его работы, существуют единственно возможная конфигурация, в которую распознаватель перейдет на следующем шаге работы. Иначе распознаватель называется недетерминированным, т.е. такой распознаватель может иметь допустимую конфигурацию для которой существует некоторое конечное множество конфигураций, возможных иметь на следующем шаге работы. Достаточно иметь хотя бы одну такую конфигурацию, чтобы распознаватель был недетерминированным. По видам внешней памяти распознаватели бывают след типов: 1)Распознаватели без внешней памяти; 2)Распознаватели с ограниченной внешней памятью; 3)Распознаватели с неограниченной внешней памятью.

Классификация распознавателей определяет сложность алгоритма работы распознавателя. Кроме того сложность распознавателя также напрямую связано с типом языка, входные цепочки которого может принимать распознаватель. Для каждого из 4-х осн типов языков сущ свой тип распознавателя. Для языков с фразовой структурой нужен недетерминированный двусторонний автомат, имеющий неограниченную внешнюю память. Для Контекстно Зависимых языков распознавателями являются двусторонние недетерминированные автоматы с линейно ограниченной внешней памятью. Для контекстно свободных языков распознавателями являются односторонние недетерминированные автоматы с магазинной (стековой) внешней памятью (МП-автоматы). Среди всех контекстно свободных языков можно выделить класс детерминированных контекстно свободных языков, распознавателями для которых являются детерминированные автоматы с магазинной внешней памятью (ДМП-автом). Эти языки обладают свойством однозначности. Доказано что для любого детерминированного конт свободного языка всегда можно построить однозначную грамматику. Для регулярных языков распознавателями являются односторонние недетерминированные автоматы без внешней памяти (конечные автоматы). В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы, т.е. для выделения в нем простейших конструкций языка. Задача разбора в общем виде заключается в том, что на основе имеющейся грамматики некоторого языка нужно построить распознаватель для этого языка. Заданная грамматика и распознаватель должны быть эквивалентны, т.е. определять один и тот же язык.

19. Компиляторы. Лексический анализ. Преобразования грамматик.

Для грамматик, например леволинейного типа, существует алгоритм определения того, принадлежит ли анализируемая цепочка языку, порождаемому этой грамматикой. Алгоритм заключается в слудйющем: 1-й символ исходной цепочки a1a2a3a….an# заменяетися нетерминальным символом A, для которого в грамматике есть правило вывода A->ai, т.е. производится свертка терминала a1 к нетерминальному A. Затем многократно до тех пор, пока не считан признак конца цепочка, выполняется следующие шаги. Полученный на предыдущем шаге нетерминал A и расположенный непосредственно справа это него очередной терминал ai исх цеп, заменяется нетерминалом B, для которого в грамматике есть правило вывода вида B->Aai. Это эквивалентно построению дерева разбора методом снизу вверх. При работе алгоритма возможны слуд ситуации:

1. Прочитана вся цепочка, на каждом шаге находилась единственная нужная свертка, на последнем шаге свертка произошла к символу S. Это значит, что цепочка принадлежит языку. a1a2a3…an#ЄL(G).

2. Прочитана вся цепочка. На каждом шаге находилась единственная нужная свертка, на последнем шаге – свертка произошла к символу, отличному от S. Это означает, что исх цепочка не принадлежит языку a.

3.На некотором шаге не нашлаось нужной свертки, т.е. для полученного на предыдущем шаге нетерминала A и расположенного непосредственно справа от него очередного терминала ai исх цепочки, не нашлось нетерминала B, для которого в грамматикае было бы правило вывода B->Aai. Это означает, что исх цепочка не принадлежит языку.

4. На некотором шаге работы алгоритма оказалось, что есть более одной подходящей свертки, т.е. в грамматике разные нетерминалы имеют правила вывода с одинаковыми правыми частями и поэтому не понятно к которому из них производить свертку. Это говорит о недетерминированности разбора. Для того, чтобы при лексическом разборе быстрее находить правила с подходящей правой частью фиксируют все возможные свертки в описании грамматики. Это определяется только грамматикой и не зависит от вида анализируемой цепочки.

Диаграмма определяет конечный автомат, построенный по регулярной грамматике, который допускает множество цепочек составляющих язык, определяемый этой грамматикой. Среди всех состояний выделяются начальные и конечные. По мере считывания каждой литеры строки контроль передается от состояния к состоянию в соответствие с заданным множеством переходов. Если после считывания последней литеры строки автомат будет находится в одном из последних состояний, то строка принадлежит языку, принимаемому автоматом. Формально конечный автомат определяется 5-ю хар-ками:

1) Конечным множеством состояний K. 2) Конечным множеством входным алфавитом. 3)множеством переходом. 4)Начальным состоянием. 5)Множеством последних сотояний.

M=(K,VT,F,H,S)

Один из наиболее важных результатов теории конечных автоматов состоит в том, что класс языков, определяемых недетерминированными конечными автоматами, совпадает с классом языков, определяемых детерминированными конечными автоматами. Это означает, что для любого недетерминированного конечного автомата всегда можно построить детерминированный конечный автомат, определяющий тот же язык. Соответствие лексическому анализу заключается в том, что каждому языку 3-его типа соответствует детерминированный конечный автомат, распознающий строки этого языка. Исходя из этого, можно сказать, что написание лексического анализатора частично сводится к моделированию различных автоматов для распознавания конструкций языка, таких как идентификаторы, числа, ключевые слова и т.д. Лексический анализ вкл в себя просмотр компилируемой программы и распознавание лексем, составляющих предложение исходного текста. Точный перечень лексем, которые нужно распознать, зависит от ЯП и от грамматики, используемой для описания этого языка.

Лексический анализатор преобразовывает исх программу в последовтельность символов. При этом идентификаторы и константы производной длины заменяются символами

фиксированной длины. Слова языка также заменяются какими-нибудь стандартными представлениями. Для повышения эффективности последних действий, каждая лексема обычно определяется кодом. Коды, создаваемые для слов языка, не должны зависеть от программы. В случае же идентификатора дополнительно необходимо конкретное имя распознаваемого идентификатора. Получение кодов-идентификаторов производится последовательно, начиная с начала по тексту проги. Кадый экземпляр определенного идентификатора на любом уровне заменяется одним и тем же кодом. Из этого следует необходимость создания таблицы идентификаторов, где будут храниться соответствия кодов и самих идентификаторов.

Аналогично таблиза необходима и для констант. Некоторые компиляторы проверяют длину констант и вычисляют их в процессе лексического анализа. Однако табл констант используется на более поздних стадиях компиляции, и поэтому ее обычно просто заполняют на этапе лексического анализа.

Сущ проблема с именами идентификаторов в том случае, если переменная с одинаковым именем присутствует в процедуре и в основной программе. В этом случае необходимо создавать столько строк в таблице идентификаторов, сколько раз определена данная переменная, кроме того нужно к таблице добавить поле для занесения информации о том, какой процедуре принадлежит идентификатор. При лексическом анализе базовым терминальным символом является буква, что вытекает их традиционного определения идентификаторов как последовательности букв и цифр, начинающихся с буквы. Еще более важным фактором является необходимое во всех языках наличие разделитилей или пробела между лексемами. Сам этот пробел и недопустимые буквы тяжелее всего распознать традиционными средствами грамматики. В дополнение к своей основной функции лексический анализатор обычно также выполняет чтение строк исходной программы и возможно печать листинга исходной программы. Многие языки имеют специфические особенности, которые должны учитываться при программировании лексического анализатора.

20. Компиляторы. Лексический анализ. Таблица сверток и диаграмма состояний.

Для того, чтобы при лексическом разборе быстрее находить правило с подходящей правой частью, фиксируют все возможные свертки в описании грамматики (это определяется только грамматикой и не зависит от вида анализируемой цепочки).

Это можно сделать в виде таблицы, строки которой помечены нетерминальными символами грамматики, столбцы - терминальными. Значение каждого элемента таблицы - это нетерминальный символ, к которому можно свернуть пару "нетерминал-терминал", которыми помечены соответствующие строка и столбец.

Например, для грамматики G = ({a, b, #}, {S, A, B, C}, P, S), такая таблица будет выглядеть следующим образом:

P: S → C#
C → Ab | Ba
A → a | Ca B → b | Cb

	a	b	#
C	A	B	S
A	-	C	-
B	C	-	-
S	-	-	-

Знак "-" ставится в том случае, если для пары "терминал-нетерминал" свертки нет.

Чаще информацию о возможных свертках представляют в виде ***диаграммы состояний*** - неупорядоченного ориентированного помеченного графа, который строится следующим образом:

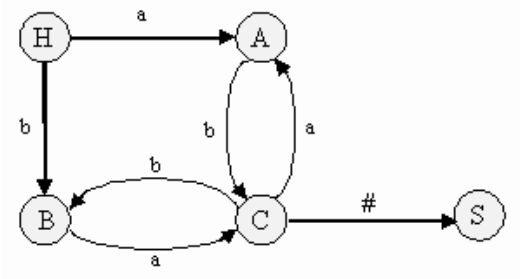
(1) строят вершины графа, помеченные нетерминалами грамматики (для каждого нетерминала - одну вершину), и еще одну вершину, помеченную символом, отличным от нетерминальных, - H. Эти вершины будем называть ***состояниями*.** H - начальное состояние.

(2) соединяем эти состояния дугами по следующим правилам:

а) для каждого правила грамматики вида W → t соединяем дугой состояния H и W (от H к W) и помечаем дугу символом t;

б) для каждого правила W → Vt соединяем дугой состояния V и W (от V к W) и помечаем дугу символом t.

Диаграмма состояний для грамматики G (см. пример выше):



Тогда для диаграммы состояний применим следующий *алгоритм разбора*:

(1) объявляем текущим состояние H;

(2) затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: считываем очередной символ исходной цепочки и переходим из текущего состояния в другое состояние по дуге, помеченной этим символом. Состояние, в которое мы при этом попадаем, становится текущим.

При работе этого алгоритма возможны следующие ситуации (аналогичные ситуациям, которые возникают при разборе непосредственно по регулярной грамматике):

(1) прочитана вся цепочка; на каждом шаге находилась единственная дуга, помеченная очередным символом анализируемой цепочки; в результате последнего перехода оказались в состоянии S. Это означает, что исходная цепочка принадлежит L(G).

(2) прочитана вся цепочка; на каждом шаге находилась единственная "нужная" дуга; в результате последнего шага оказались в состоянии, отличном от S. Это означает, что исходная цепочка не принадлежит L(G).

(3) на некотором шаге не нашлось дуги, выходящей из текущего состояния и помеченной очередным анализируемым символом. Это означает, что исходная цепочка не принадлежит L(G).

(4) на некотором шаге работы алгоритма оказалось, что есть несколько дуг, выходящих из текущего состояния, помеченных очередным анализируемым символом, но ведущих в разные состояния. Это говорит о *недетерминированности разбора*. Анализ этой ситуации будет приведен далее.

21. Компиляторы. Синтаксический анализ. Метод восходящего разбора.

Синтаксический анализ – во время синтаксического анализа предложения исходной программы распознаются как языковые конструкции описываемые используемые грамматикой. Можно рассматривать этот процесс как построение дерева грамматического разбора для транслируемых предложений.

Методы грамматического разбора можно разбить на 2 больших класса: восходящие и нисходящие в соответствии с порядком построения дерева грамматического разбора. Нисходящие методы начинают с правила грамматики, определяющие конечную цель анализа с корня дерева грамматического разбора и пытаются так его наращивать чтобы последующие узлы дерева соответствовали синтаксису анализируемого предложения.

Восходящие методы начинают с конечных узлов дерева грамматического разбора и пытаются объединить их построением узлов все более и более высокого уровня, до тех пор, пока не будет достигнут корень дерева. Восходящий метод грамматического разбора называют методом предшествования. Он основан на анализе пар последовательно расположенных операторов исходной программы и решений вопроса о том который из них должен выполняться первым.

При использовании метода грамматического разбора основанного на отношении операторного предшествования, анализируемое предложение рассматривается слева направо до тех пор, пока не будет найдено подвыражение операторы которого имеют более высокий уровень операторного предшествования, чем соседние операторы. Далее это подвыражение распознается в терминах правил перехода, используемой грамматики. Этот процесс продолжается до тех пор, пока не будет достигнут корень дерева, что и будет означать конец грамматического разбора. Первым шагом при разработке процессора грамматического разбора, основанного на методе предшествования д.б установление отношений предшествования между операторами грамматики.

	begin	read	Id	(	)
Begin		<	<		
Read				=	
Id					>
(			<		=
)					

Знак = означает что обе лексемы имеют одинаковый уровень предшествования и должна рассматривается грамматическим процессором в качестве одной конструкции языка. Если для пар отношение предшествования не существует это означает что они не могут находиться рядом ни в каком грамматически правильном предложении. Если подобные комбинации лексем встречаются в процессе грамматического разбора то она должна рассматриваться как синтаксическая ошибка. Существуют алгоритмы автономного построения матриц предшествования на основе формального описания грамматики. Для применимости метода операторного предшествования необходимо чтобы отношения предшествования были заданы однозначно.

Процесс сканирования слева направо продолжается на каждом шаге грамматического разбора лишь до тех пор пока не определится очередной фрагмент предложения для грамматического распознавания, т.е. первый фрагмент ограниченный угловыми скобками. Как только подобный фрагмент выделен он интерпретируется как некоторый очередной терминальный символ в соответствии с каким либо правилом грамматики. Этот процесс продолжается до тех пор пока предложение не будет распознано целиком, следует обратить внимание на то что каждый фрагмент дерева грамматического разбора строится начиная с конечных узлов вверх в сторону корня дерева. Для метода операторного предшествования имена нетерминальных символов не существенны т.о. вся информация о грамматике и синтаксических правилах языка содержатся в матрице операторного предшествования.

Другой метод грамматического разбора:

На этапе синтаксического анализа нужно учитывать имеет ли цепочка лексем структуру заданного синтаксисом языка и зафиксировать эту структуру. Следовательно снова надо решать эту задачу разбора. Дана цепочка лексем и надо определить выводима ли она в грамматике определяющей синтаксис языка. Однако структура таких конструкций как выражение, опис-е, оп-р более сложнее чем структура идентификаторов и чисел. Поэтому для описания синтаксисов языков программирования нужны более мощные грамматики чем регулярные. Обычно для этого использует укорачивающие КС грамматики. Грамматики этого класса с одной стороны позволяют доступно полно описать синтаксическую структуру реальных языков программирования, с другой стороны для разных укорачивающих КС грамматик существуют достаточно эффективные алгоритмы разбора.

Существует алгоритм который по любой допустимой КС грамматике и данной цепочке символов выясняет принадлежит ли цепочка языку порождаемому этой грамматикой. Но время работы такого алгоритма экспоненциально зависит от длины цепочки что с практической точки зрения совершенно неприемлемо. Алгоритмы анализа расходующего на обработку входной цепочки линейное время приемлемы только некоторым подклассам КС грамматик.

22. Компиляторы. Синтаксический анализ. Метод нисходящего разбора.

Синтаксический анализ – Во время синтаксического анализа предложения исходной программы распознаются как языковые конструкции описываемые используемые грамматикой. Можно рассматривать этот процесс как построение дерева грамматического разбора для транслируемых предложений.

Методы грамматического разбора можно разбить на 2 больших класса: восходящие и нисходящие в соответствии с порядком построения дерева грамматического разбора. Нисходящие методы начинают с правила грамматики, определяющие конечную цель анализа с корня дерева грамматического разбора и пытаются так его наращивать чтобы последующие узлы дерева соответствовали синтаксису анализируемого предложения.

Нисходящий метод(метод рекурсивного спуска)

Другой метод грамматического разбора - нисходящий метод, называемый рекурсивным спуском. Процессор грамматического разбора, основанный на этом методе, состоит из отдельных процедур для каждого нетерминального символа, определенного в грамматике. Каждая такая процедура старается во входном потоке найти подстроку, начинающуюся с текущей лексемы, которая может быть интерпретирована как нетерминальный символ, связанный с данной процедурой. В процессе своей работы она может вызывать другие подобные процедуры или даже рекурсивно саму себя для поиска других нетерминальных символов. Если эта процедура находит соответствующий нетерминальный символ, то она заканчивает свою работу, передает в вызвавшую ее программу признак успешного завершения и устанавливает указатель текущей лексемы на первую лексему после распознанной подстроки. Если же процедура не может найти подстроку, которая могла бы быть интерпретирована как требуемый нетерминальный символ, она заканчивается с признаком ошибки или вызывает процедуру выдачи диагностического сообщения и процедуру восстановления.

Рассмотрим в качестве примера правило <read> ::= READ (<id-list>), записанное в форме Бекуса-Наура (БНФ). Грамматика БНФ состоит из подмножества правил вывода, каждое из которых определяет синтаксис некоторой конструкции языка программирования. Это определение синтаксиса предложения READ языка Паскаль, обозначенное в грамматике как <read>. Символ “::=” можно читать как “является по определению”. Строки символов, заключенные в угловые скобки “<” и ”>”, называются нетерминальными символами, т.е. являются именами конструкций, определенными внутри грамматики. То, что не заключено в угловые скобки, называется терминальными символами грамматики - лексемами. Таким образом, это правило определяет, что конструкция <read> состоит из лексемы READ, за которой следует лексема “(“, за ней следует конструкция языка, называемая <id-list>, за которой следует лексема “)”. Для распознавания нетерминального символа <read> необходимо, чтобы было определение для нетерминального символа <id-list>.

<id-list> ::= id | <id-list>,id

Это правило является рекурсивным, т.е. конструкция <id-list> определяется фактически в терминах себя самой.

Процедура метода рекурсивного спуска, соответствующая нетерминальному символу <read>, прежде всего исследует две последовательные лексемы, сравнивая их с READ и “(“. В случае совпадения эта процедура вызывает другую процедуру, соответствующую нетерминальному символу <id-list>. Если процедура <id-list > завершится успешно, то процедура <read> сравнивает следующую лексему с “)”. Если все эти проверки окажутся успешными, то процедура <read> завершается с признаком успеха и устанавливает указатель текущей лексемы на лексему, следующую за “)”. В противном случае процедура <read> завершится с признаком неудачи. Поэтапный процесс разбора представлен на рисунке 2.

Нисходящий грамматический разбор неприменим непосредственно для грамматик, содержащих левые рекурсии, т.к. один рекурсивный вызов приведет к еще одному рекурсивному вызову и т.д., в результате чего образуется бесконечная цепочка рекурсивных вызовов. Исключая левую рекурсию, получаем правило:

<id-list> ::= id { , id },

которое определяет нетерминальный символ <id-list> как состоящий из единственной лексемы id или же из произвольного числа следующих друг за другом лексем id, разделенных запятыми.

Выводы

В языке программирования отсутствуют какие-либо особенности, заставляющие отдать предпочтение одному из методов грамматического разбора. Возможно одновременное использование обоих методов. Некоторые компиляторы используют метод рекурсивного спуска для распознавания конструкций относительно высокого уровня, например, до уровня отдельных предложений языка, а потом переключаются на метод, аналогичный методу предшествующего спуска для анализа таких конструкций, как, например, арифметические выражения.

23. Компиляторы. Семантический анализ.

Контекстно-свободные грамматики, с помощью которых описывают языки программирования, не позволяют задавать контекстные условия, имеющиеся в любом языке.

К контекстным условиям относятся:

1. каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;
2. при вызове функции число фактических параметров и их типы должны соответствовать числу типов формальных параметров;
3. обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке; на типы левой и правой частей в операторе присваивания, на тип параметра цикла, на тип условия в операторе цикла и условном операторе.

Проверку контекстных условий часто называют семантическим анализом. Его можно выполнить сразу после синтаксического анализа. Некоторые требования можно контролировать во время итерации кода (ограничения на типы операторов в выражении), а можно совместить синтаксический анализ с семантическим.

На этапе семантического анализа обрабатываются структуры, распознанные синтаксическим анализатором. На этой стадии выполняются и такие функции, как ведение таблицы символов, обнаружение ошибок, макрорасширений и выполнение инструкций, относящихся ко времени компиляции.

При простых трансляциях семантический анализатор может породить объектный код. Но обычно выходом этой стадии служит некоторая внутренняя форма выполняемой программы, с которой затем компилятор работает на стадии оптимизации.

Обычно семантический анализатор разделяется на ряд более мелких анализаторов, каждый из которых предназначен для конкретной программной конструкции. Например, описание массива обрабатываются одним анализатором, а арифметические выражения - другим. Соответствующий семантический анализатор вызывается синтаксическим анализатором как только синтаксический анализатор распознает синтаксическую единицу, требующую обработки.

Семантические анализаторы взаимодействуют между собой посредством информации, хранящейся в различных структурах данных, в частности в центральной таблице символов. Например, семантический анализатор, обрабатывающий описание типа для простых переменных не делает ничего, кроме занесения описываемых типов в таблицу символов. Позже семантический анализатор, обрабатывающий арифметические выражения может использовать описанные типы для включения соответствующих специфичных типов арифметических операций в объектный код.

Основные функции семантического анализатора, кроме ведения таблиц символов является:

1. включение неявной информации, т.е. информация, задаваемая в программе неявно, должна быть представлена в явном виде в программе низкого уровня;
2. обнаружение ошибок и макрообработка.

24. Компиляторы. Генерация кода. Триады. Тетрады. Префиксная и постфиксная записи. Три формы объектного кода.

После анализа программы, когда во все таблицы занесена информация, необходимая для генерации кода, компилятор должен переходить к построению соответствующей программы в машинном коде. Код генерируется при обходе дерева разбора, построенного анализатором на предыдущих стадиях. Обычно генерация выполняется параллельно с построением дерева, но может выполняться и за отдельный проход. Фактически для получения машинного кода требуются два отдельных прохода:

- генерация промежуточного кода;
- генерация собственно машинного кода

Промежуточные коды можно проектировать на различных уровнях. Иногда промежуточный код получают, просто разбивая сложные структуры исходного языка на более удобные для обращения элементы. Однако чаще в качестве промежуточного кода используют какой-либо обобщенный машинный код. Существует несколько видов таких обобщенных кодов.

1. Тетрады

Представляет собой запись операций в форме из 4 составляющих:

<операция> (<операнд_1>,<операнд_2>,<результат>)

Тетрады представляют собой линейную последовательность команд, поэтому для них несложно написать алгоритм, который будет преобразовывать последовательность тетрад в последовательность команд результирующей программы, либо в последовательность команд ассемблера. Тетрады не зависят от архитектуры вычислительной системы, на которую ориентирована разрабатываемая программа, поэтому они представляют собой машинно-независимую форму внутреннего представления программы.

(-a+b)*(c+d)

можно представить тетрады следующим образом:

-a = 1
1+b=2
c+d=3
2*3=4

2. Триады

Представляют собой запись операций в форме 3 составляющих

<операция> (<операнд_1>,<операнд_2>

Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду, в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады Поэтому триады при записи последовательно нумеруются для удобства указания ссылок одних триад на другие.

Как и в случае с тетрадами, алгоритм преобразования последовательности триад в команды достаточно прост, но здесь требуется также и алгоритм, отвечающий за расширение памяти, необходимой для промежуточных результатов вычислений, так как временные переменные для этой цели не используются. Триады требуют меньше памяти для своего представления, чем тетрады, кроме того, они явно отражают взаимосвязь операций между собой. Триады ближе к двухадресным машинам, чем тетрады.

a+b+c*d

можно представить, как в виде тетрад:

a+b=1	a+b
c*d=2	c*d
1+2=3	1+2

Префиксная и постфиксная записи

В префиксной нотации каждый знак операции ставится перед своими операндами, а в постфиксной – после. В этом и состоит их основное отличие от обычной (инфиксной) записи, когда операция проставляется между операндами. Например, инфиксное выражение a+b в префиксной нотации примет вид +ab, а в постфиксной ab+.

(a+b)*(c+d)

в префиксной записи будет выглядеть

*+ab+cd

Преимущество записи выражений в префиксной или постфиксной нотации заключается в том, что нет необходимости в скобках и кроме того виден порядок выполнения операций. Кроме того в данных операциях нет приоритета операндов, хотя при преобразовании из обычной записи в префиксную или постфиксную их необходимо учитывать.

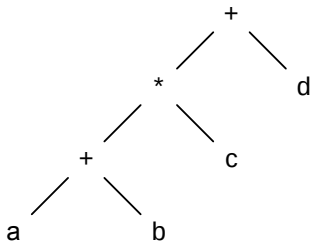
Получение постфиксного или префиксного представления возможно разными путями. В частности используется алгоритм стека, который в простейшем случае сведется к:

1. сохранению идентификатора, когда он встретится при чтении слева направо;
2. помещению в стек знака операции;
3. в момент встречи конца выражения выдача из стека знака операции, который находится на вершине.

Другой метод в получении постфиксного выражения из выражения, представленного в виде бинарного дерева. Например, выражение

(a+b)*c+d

представляется в виде бинарного дерева как на рисунке.



Чтобы получить постфиксное выражение используют порядок обхода, определенный Кнутом:

1. обход левого поддерева снизу:
2. обход правого поддерева снизу:
3. посещение корня.

что дает в результате

ab+c*d+

Генерация машинного кода

Самым простым способом генерации объектного кода является метод, который генерирует объектный код для каждого фрагмента программы как только распознан синтаксис этого фрагмента. Для реализации такого метода необходим набор подпрограмм, соответствующих каждому правилу и каждой альтернативе в правилах грамматики.

Как только в результате грамматического разбора будет распознан фрагмент текста исходной программы, соответствующий некоторому правилу грамматики, вызывается подпрограмма, соответствующая этому правилу.

Исторически различают три формы объектного кода:

1. абсолютные команды, расположены в фиксированных ячейках памяти, после окончания компиляции такая программа немедленно выполняется.
2. программа на языке ассемблера, которую придется потом еще раз транслировать.
3. программа на машинном языке, представленная образами кодов и записанная во внешней памяти в виде двоичного объектного модуля.

25. Компиляторы. Генерация кода. Общая схема распределения памяти.

При генерации собственно машинного кода используются таблицы, полученные на предыдущих этапах и сформированная программа в виде промежуточного кода в одной из его записей. В генерации в данном случае можно выделить этап предварительной подготовки, т.е. выделение памяти, распределение сегментов под процедуры и т.д. Затем идет собственно генерация кода. Каждая строка промежуточного кода читается, обрабатывается и на ее основе генерируется соответствующий ассемблерный или двоичный код. Главная задача этой стадии – чтение и обработка параметров команд, преобразование адресов и генерации кода.

Для формализованного анализа в задаче необходимо построить формализованную модель ресурсов целевого компьютера. В их состав должны входит:

1. модель информационных или запоминающих ресурсов.
2. модель операционных ресурсов, эквивалентных операторам языка программирования в виде таблиц команд, которая включает коды операций и базовых модификаций адресов в качестве ключей и соответствующие машинные коды как характеристики поиска.

Так как схема распределения памяти и форматы областей данных готовой программы определяются управляющими программами ОС, обращение к ним сравнительно просто формируется во время компиляции.

Общая схема распределения памяти может выглядеть следующим образом: исходим из того, что для каждого процесса компьютер отводит одну область данных. В эту область входят все внутренние переменные, фактические параметры, переменные, определяемые программистом и временные переменные, используемые процедурами. В общем случае компьютер имеет таблицу, где перечислены все области данных объектной программы.

Программа, осуществляющая присваивание измененных адресов имеет дело с одной процедурой. Ее первая задача – отмечать в начальной области место для связи с предшественником по трассе вызова, если оно используется и место для формальных параметров. В этой таблице фиксируется относительный адрес в сегменте, который используется стандартными объектами.

Другая задача состоит в том, чтобы обработать элементы таблицы символов, соответствующих формальным параметрам, а также переменных, описанных в этой процедуре и присвоить им адреса.

Для каждого элемента в таблице символов выполняется следующее:

1. элементу таблицы символов присваивается смещение, равное значение текущего предела, выделяемого в памяти, т.е. ему присваивается адрес первой свободной ячейки или области данных.
2. текущая таблица увеличивается на длину элементов, отведенных под формальные параметры.

Требуется, чтобы для таблицы символов можно было узнать размер области памяти, нужной для каждой переменной. В языках с нестрогим условием декларации данных эта информация неизвестна к окончанию процесса просмотра всей исходной программы, поэтому для присваивания адресов после обычного анализа необходим второй просмотр. Во время первого, главного просмотра компилятор генерирует команды машины непосредственно из входящего языка. Ссылки на переменные в машинных командах определяются указанием на соответствующие элементы таблицы символов. Во время второго просмотра переменным присваиваются адреса, которые записываются в элементы таблицы символов. Потом пересматриваются машинные команды и каждая ссылка на таблицу символов заменяется соответствующим значением адреса, взятым из элемента, на который ссылается команда.

В языках, которые требуют описания переменных до их использования, распределение памяти выполняется семантическими программами, предназначенными для обработки деклараций данных. В этом случае возможен однопросмотровый компилятор.

Процесс генерации кодов отдельных команд отражается на каноническое множество адресов и модификаций команд. Во множество адресов включается как минимум:

1. прямая адресация глобальных и динамических данных.
2. относительная адресация аргументов и локальных данных, процедур и функций в стеке.
3. индексная адресация глобальных и локальных данных.

На основании допустимых операций и форматов адресов данных строится таблица, в которой аргументами являются последовательность команд и формы адресации, сопоставимые с выборками в тетрадной форме и характеризуются простыми правилами формирования адресов в объектном коде.

При изучении форматов объектных модулей решить следующие вопросы:

1. как указывается внешняя подпрограмма и данные, требуемые объектному модулю и каким образом определяются точки входа командам либо данных модуля, которые могут использоваться другими программами.
2. как достигается перемещаемость объектных программ, т.е. возможность размещать ее в любом месте ОП.

Словарь внешних символов (ESD) определяет и дает единственное имя и номер таким объектам как кодовые сегменты и внешние ссылки на программные сегменты и другие точки входа, к которым обращаются в этом модуле, но которые не включены в него.

26. Компиляторы. Оптимизация кода

Интерпритатор – программа которая не выдает результат программы

Целью компилятора является трансляция программы, написанной на ЯВУ в машинные коды некоторого компьютера. Большинство ЯВУ построено так чтобы независимыми от тех машин на которых они будут исполняться. Это означает что процесс анализа синтаксиса программ написанных на этих языках должен быть так же машинно-независимым. Но процесс генерации кода является машинно-зависимым поскольку необходимо знать систему команд компьютера для которого этот код генерируется.

Компилятор должен выполнять анализ программы а затем синтез объектной программы. Сначала исходная программа разлагается на ее составные части. Затем из них строятся части эквивалентной объектной программы. Для этого на на этапе анализа компилятор строит несколько т-ц, которые затем исполняются , как при анализе так и при синтезе. Для облегчения построения компиляторов ЯВУ обычно описывается в терминах некоторой грамматики. Эта грамматика определяет форму или синтаксис допустимых предложений языка. Проблема компиляции может быть сформулирована как проблема поиска соответствия написанных программой предложений, структурам определяемым грамматикой и генерацией соответствующего кода для каждого предложения. Наиболее существенная часть любого процесса компиляции это определение в исходной программе элементарных составляющих , идентификаторов, ограничителей, операторов, чисел и т.д. которые называются лексемами. Предложения исходной программы удобнее представлять виде последовательности лексем. Просмотр исходного текста распознавания и классификации различных лексем называется лексическим анализом.

Синтаксический анализ – Во время синтаксического анализа предложения исходной программы распознаются как язык. Конструкции описываемые используемые грамматикой. Можно рассматривать этот процесс как построение дерева грамматического разбора для транслируемых предложений.

Методы грамматического разбора можно разбить на 2 больших класса: восходящие и нисходящие в соответствии с порядком построения дерева грамматического разбора. Нисходящие методы начинают с правила грамматики, определяющие конечную цель анализа с корня дерева грамматического разбора и пытаются так его наращивать чтобы последующие узлы дерева соответствовали синтаксису анализируемого предложения.

Оптимизация

Оптимизацией называется обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе в целях получения более эффективного объектного кода. Обычно выделяют машинно-зависимую и машинно-независимую оптимизацию. Под машинно-зависимой оптимизацией понимается преобразование исходной программы в ее внутреннем представлении, что означает полную независимость от выходного языка, в отличие от машинно-зависимой, выполняемой на уровне объектной программы.

Среди машинно-независимых методов можно выделить самые основные:

- Свертка, т.е. выполнение операций, операнды которых известны во время компиляции..
- Исключение лишних операций за счет однократного программирования общих подвыражений.
- Вынесение из цикла операций, операнды которых не изменяются внутри цикла.

Оптимизация внутри линейных участков

Линейным участком является выполняемая по порядку последовательность операций с одним входом и одним выходом (первая и последняя операции соответственно). Например, последовательность операций представленных ниже образуют линейный участок.

```
I:=1+1;
I:=3;
B:=7+I;
```

Внутри линейного участка обычно проводят две оптимизации: свертку и устранение лишних операций.

Свертка

Свертка – это выполнение во время компиляции операций исходной программы, для которых значения операндов уже известны, и, поэтому, нет нужды их выполнять во время счета. Например, внутреннее представление в виде триад линейного участка программы представленного выше изображено ниже.

```
(1) + 1,1
(2) := I,(1)
(3) := I,3
(4) + 7,(3)
(5) := B,(4)
```

Видно, что первую триаду можно вычислить во время компиляции и заменить результирующей константой. Менее очевидно, что четвертую триаду также можно вычислить, т.к. к моменту ее обработки известно, что I равно 3. Полученный после свертки результат

(1) := I,2
(2) := I,3
(3) := B,10

Главным образом свертка применяется к арифметическим операциям, так как они наиболее часто встречаются в исходной программе. Кроме того, она применяется к операторам преобразования типа. Причем проблема упрощается, если эти преобразования заданы явно, а не подразумеваются по умолчанию.

Процесс свертки операторов, имеющих в качестве операндов константы, понятен и сводится к внутреннему их вычислению. Свертка операторов, значения которых могут быть определены в результате некоторого анализа, несколько сложнее. Обычно свертку осуществляют только в пределах линейного участка при помощи таблицы Т, вначале пустой, содержащей пары (К, А), где А – простая переменная для которой известно текущее значение К. Кроме того, каждая свертываемая триада заменяется новой триадой (С, К, 0), где С (константа) – новый оператор, для которого не надо генерировать команды, а К – результирующее значение свернутой триады. Алгоритм
свертки последовательно просматривает триады линейного участка.Пример работы данного алгоритма приведен в таблице 1.

Таблица 1 - Последовательность шагов алгоритма свертки

	Шаг 1	*Шаг 2*	*Шаг 3*	*Шаг 4*	*Шаг 5*
1	+ 1,1	C 2	C 2	C 2	C 2
2	:= I,(1)	:= I,(1)	:= I,2	:= I,2	:= I,(1)
3	:= I,3	:= I,3	:= I,3	:= I,3	:= I,3
4	+ 7,(3)	+ 7,(3)	+ 7,(3)	C 10	C 10
5	:= B,(4)	:= B,(4)	:= B,(4)	:= B,(4)	:= B,10
T			(I,2)	(I,3)	(I,3)
T					(B,10)

С точки зрения работы компилятора процесс свертки является дополнительным проходом по внутреннему представлению исходной программы представленной триадами. Но свертку можно проводить и с тетрадами. Кроме того, можно оптимизировать программу в семантических программах во время получения внутреннего представления и при этом отпадает необходимость в дополнительном проходе. Например, для выражения

A:=B+C

при восходящем грамматическом разборе программа должна выполнить только одну дополнительную проверку семантик В и С. Если они константы или их значения известны, то программа их складывает, и результат связывает с А. В данном случае можно использовать таблицу переменных с известными значениями, которая должна сбрасываться в местах генерации команд передачи управления.

Исключение лишних операций

i-я операция линейного участка считается лишней, если существует более ранняя идентичная j-я операция и никакая переменная от которой зависит эта операция, не изменяется третьей операцией лежащей между I-й и j-й операциями. Например на линейном участке

D:=D+C*B
A:=D+C*B

можно выделить лишние операции. Если представить линейный участок в виде триад, как это представлено ниже, то видно, что операция С*В во второй раз лишняя.

(1) * C,B
(2) + D,(1)

(3) := D,(2)
(4) * C,B
(5) + D,(4)
(6) := A,(5)

Лишней триада (4) является из-за того, что ни C ни B не изменяются после триады (1). В отличии от этого триада (5) лишней не является, т.к. после первого сложения D с C*B 3-я триада изменяет значение D.

Алгоритм исключения лишних операций просматривает операции в порядке их появления. И если i-я триада лишняя, так как уже имеется идентичная ей j-я триада, то она заменяется триадой (SAME, j, 0), где операция SAME ничего не делает и не порождает никаких команд при генерации.

Для слежения за внутренней зависимостью переменных и триад, можно поставить им в соответствие числа зависимостей (dependency numbers). При этом используются следующие правила.

- Вначале для переменной A число зависимости dep(A) выбирается равным нулю, так как ее значение не зависит ни от одной триады;
- После обработки i-й триады, где переменной A присваивается какое-либо значение число зависимости становиться равным i, так как ее новое значение зависит от i-й триады;
- При обработке i-й триады ее число зависимостей dep(i) становиться равным максимальному из чисел зависимостей ее операндов + 1.

Числа зависимостей используются следующим образом: если i-я триада идентична j-й триаде (j<i), то i-я триада является лишней тогда и только тогда, когда dep(i)=dep(j).

Таблица 3.4 – Пример исключения лишних операций

Обрабатываемая триада I	Dep (переменная) A B C D	Dep(i)	Полученная в результате триада
(1) * C,B	0 0 0 0	1	(1) * C,B
(2) + D,(1)	0 0 0 0	2	(2) + D,(1)
(3) :=D,(2)	0 0 0 0	3	(3) :=D,(2)
(4) * C,B	0 0 0 3	1	(4) SAME 1
(5) + D,(4)	0 0 0 3	4	(5) + D,(4)
(6) :=A,(5)	0 0 0 3	5	(6) :=A,(5)
	6 0 0 3		

Вынесение инвариантных операций за тело цикла

Инвариантными называются такие операции, при которых ни один из операндов не изменяется внутри цикла. Например, если из цикла, выполняемого тысячу раз вынести одно умножение, то на этапе выполнения экономия составляет девятьсот девяносто девять умножений.

В общем случае данная оптимизация выполняется двойным просмотром тела цикла с анализом операндов каждой из операций внутри цикла. В случае если они инвариантны, операция выносится назад за тело цикла, а внутри цикла выбрасывается. Для проверки инвариантности переменных используют специальную таблицу инвариантности.

Во время первого прохода цикла заполняется таблица инвариантных переменных, во время второго – выполняется собственно вынесение операций.

27. Компоновщики – редакторы связей.

Компон-к – это прога, преодоставляемая проектировщиком системы, которая связывает независ. лог. обл-ти кажд. подр-мы и кажд. модуля в одну единств. лог. обл-ть. Компоновка включает по крайней мере следующие этапы:

1.Сбор всех модулей из библиотек пользователей и системных библиотек.

2.Установка всех внешних ссылок м/д модулями.

3.Собщение о неопределенных ссылках.

4.Конструирование структуры оверлея.

5.Определение и построение загрузочного модуля

Главная задача программы компоновщика создать комплекс объект. модулей, полученных от компиляторов с различных языков в какой-нибудь из разновидностей исполняемых модулей и заменить неопределенные ссылки на внешние имена с соответствующими адресами или информацией доступной для загрузчиков. Классификация типов связей разграничивает 2 класса: управляющие и информационные. Для реализации которых используется единый технический подход, опирающийся на глобально-доступные имена элементов программных модулей. Управляющие связи в современных системах программирования определяются способами передачи управления и действиями, выполняемыми для решения частей общей задачи и представленные в одной из шести форм:

1.Вызов подпрограммы того же загрузочного модуля в процессе его выполнения, реализуемого простыми средствами модульного программирования.

2.Макровызов, как подготовка действий по решению части задачи на этапе компиляции или создания программы, которая реализуется с помощью встроенных макросредств системы программирования.

3.Вызов подпрограммы, копия которой имеется в основной программе и который обычно реализуется в форме обращения к общей системной программе ОС или управляющей надстройке пользователя.

4.Вызов подпрограмм с диска можно подзагрузкой их кодов и управляющих данных, при котором иногда различают оверлейную структуру программ с фиксированным взаимным расположение модулей и фрагментов и динамическую подзагрузку в совершенно произвольном порядке.

5.Динамический вызов параллельных процессов решения подзадач включая выполнение exe-модули и dll-модули, которые подключаются на этапе выполнения.

6.Передача сообщения и сигналов автономным параллельным процессам и задачам широко применяемым для управления вычислениями в объектно-ориентированных оболочках.

Кроме управляющих необходимо установить и информационные связи между модулями для передачи в подчиненные модули операт. данных или аргументов подпрограмм и возврата результатов в вызывающие программы, а также для передачи универсальных управляющих данных в другие программные модули. К основным способам установления информационных связей относят 3 следующих способа:

1.Передача аргументов при вызове подпрограмм и функций унифицированная стандартами обращения к подпрограммам, общая для компоновщиков и специализированными для языков программирования.

2.Возрат результатов при вызове функций, при котором соблюдают стандарт. связанный с ЯП.

3.Использование глобальных областей памяти для обмена данными между подпрограммами и задачами.

Комбинирование связи чаще всего возникают, когда управляющая или адресная информация рассматриваются как данные в аргументах функций и процедур или адресов подпрограмм, но с другой стороны точки ветвления могут быть рассмотрены как адреса передачи управления для их реализации достаточно использовать типичные средства связывания с незначительными особенностями во входных языках для определения процедурных типов.

Для реализации компоновщика и его эксплуатации необходимо проанализировать форматы его элементов хотя бы на самом общем уровне. При изучении форматов объектных модулей особый интерес представляют такие вопросы:

1. как определяется? Какие внешние подпрограммы и данные нужны объектному модулю и каким образом определяются адреса обращения или точки входа в подпрограммах или данных, которые могут использоваться программами других модулей?
2. как проверить корректность типов данных при обращении к подпрограммам и задачам?
3. как достигается переместимость объект. программы, т.е. возможность размещать ее в любом месте ОП?
4. как и почему компонуются отдельные логические сегменты в единый физический сегмент?

Объединение логических сегментов требует корректировки относительного адреса в уже сформированный коде путем добавления относительного смещения смещения начала логического сегмента для перемещимых адресов. В случае использования внешних имен условный код указателя заменяется конкретным адресом соответственного объекта. Многообразие объектов переместимости, а иногда и использование разных кодов для внутреннего представления типов переместимости делают несовместимыми объектные коды, получаемые от трансляторов разных фирм или даже разных систем программирования одной фирмы. В модульном программировании различают связи по управлению и по данным, реализуемые с помощью адресных указателей или ссылок на данные и коды. При объедении модулей необходимо определить имена областей памяти и фрагментов программ, используемых в др модулей в списке операндов оператора public, внешние имена из других модулей вместе с их типами описываются в списке операндов оператора extended. Для указателей используются только ключевые слова word и dword, т.е. указатели в ассемблере нетипизированы. При обращении к подпрограммам и функциям по прямым адресам используются типы near и far. Передача параметров в форме непосредственных значений или адресных указателей осуществляются через стек.

Необходимость проверки соответствия типов данных, аргументов, процедур и функций, а также числа аргументов в процедурах, существенно усложняет представления словаря внешних ссылокв объектных файлах. Компоновщики используют программные модули, хранящие информацию об именах исходных модулей программы, типе компилятора породившем этот модуль, а иногда и даже и о модели памяти под которую этот модуль оттранслирован.

28. Загрузчики: абсолютные, перемещающие, связывающие.

Загрузчик копирует готовый к выполнению модуль в ПМ по команде ОС. Во время этого копирования загрузчик может осуществлять адресную трансляцию перемещаемого модуля. Загрузчик может быть либо двоичным (абсолютным), либо перемещаемым. При двоичной загрузке готовому к выполнению модулю присваиваются физические адреса еще во время компоновки и т.о. никакое перемещение не возможно. Перемещаемый загрузчик загружает программу в любую область ОП.

Абсолютный загрузчик выполняет запись объектов программы в ОП и передачу управления на адрес начала ее исполнения. Поскольку от абсолютного загрузчика не требуется выполнения связывания и перемещения программ его работа очень простая. Все выполняется за один просмотр. В начале для того чтобы удостоверится, что программа, переданная для загрузки, корректна и что для нее достаточно места в ОП, просматривается запись-заголовок. Затем последовательно считываются записи тело программы, и содержащийся в них объектный код помещается в ОП по указанному адресу. И как только будет прочитана запись-конец, загрузчик передает управление по адресу, заданному в качестве адреса начала исполнения программы. Одним из очевидных недостатков абсолютных загрузчиков является то, что требуется определить фактически адрес начала загрузки программы до ее ассемблирования. Абсолютные загрузчики благодаря своей простоте используются в качестве начальных загрузчиков ОС.

Перемещаемые загрузчики обеспечивают эффективное разделение ресурсов компьютера при одновременном выполнении нескольких независимых программ, совместно использующих ОП и другие ресурсы. На ряду с простой функцией размещение программы в ОП, данный загрузчик выполняет также перемещение программ и при этом может использовать аппаратные средства.

Сегмент программ или данных может определять символ для возможной ссылки на них из других программ, а также ссылаться на символическую информацию, определяемую другими модулями. Пример: имя программы, точки входа в программу и имена областей данных.

Ссылки на внешние символы реально встречаются при вызове процедур и при использовании глобальных данных. Как внутренне определенные символы, так и внешние используемые символы должны быть явно указаны в перемещаемых объектных модулях для использования их связывающей программой. С каждым перемещаемым сегментом должна быть связана таблица определения символов и таблица использования символов.

Таблица определения символов перечисляет все символы класса, каждая запись таблицы – это пара (DName[i],DVal[i]), где DName содержит символы, а DVal – соответствующие им значения. Значением символа обычно является перемещаемый адрес А команды или данных в пределах сегмента. Когда сегмент перемещается загрузчиком, элементы сегмента с перемещаемым адресом настраиваются единообразно на место в памяти с абсолютным адресом, определяемым символом этого элемента.

Эта таблица затем становится частью глобальной таблицы символов, которая используется, чтобы отыскать адреса для внешне определенных символов.

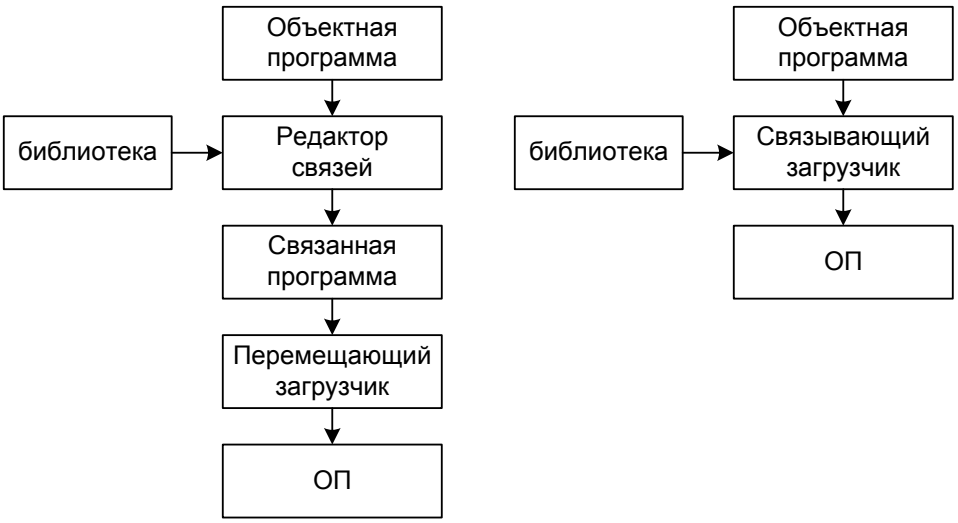
Таблица использования состоит из списка всех внешних символов, на которые имеются ссылки в сегменте.

Важной переменной, используемой в перемещающем загрузчике, является адрес загрузки программы (PROGADDR).Существует несколько алгоритмов построения перемещающих загрузчиков:

1. Перемещаемые с помощью записей-модификаторов – для описания каждого фрагмента объектного кода, требующего изменения при перемещении используется специальная запись-модификатор. Каждая запись-модификатор определяет начальный адрес и длину поля, значение которого необходимо изменить, а также тип требуемой модификации. Записи-модификаторы являются удобным средством для задания информации о перемещаемой программе. Алгоритм работы загрузчика, использующего записи-модификаторы состоит из следующих этапов:
    a. Прочитать запись-заголовок;
    b. Проверить имя и длину программы;
    c. Получить PROGADDR от ОС;
    d. Прочитать первую запись тела программы;
    e. Пока тип записи не равен «E» выполнить следующие шаги:
        i. Если объектный код задан в символическом виде, то преобразовать его во внутреннее представление;
        ii. Если тип записи равен «T», то переписать объектный код в заданное место ОП;
        iii. Если тип записи равен «N», то модифицировать заданное поле, т.е. прибавить к нему значение PROGADDR;
        iv. Прочитать следующую запись объектной программы;
    f. Передать управление по адресу, заданному в записи конец, т.е. тип записи «E» плюс PROGADDR.
2. Перемещение с помощью маски – эффективен на компьютерах, где используется прямая адресация и фиксированный командный формат. Формат записей тела программы тот же, что и ранее, за исключением того, что теперь с каждым словом объектного кода связан разряд перемещения. Эти разряды собираются вместе и образуют маску, которая записывается после указателя длины каждой записи тела программы. Если разряд перемещения установлен в единицу, то при перемещении программы начальный адрес программы добавляется к слову, соответствующему этому разряду. Значение разряда перемещения равное нулю показывает, что никаких преобразований при перемещении делать не нужно.

Связывающие загрузчики

Выполняют связывание и перемещение во время загрузки.



Несмотря на то, что операции связывания и загрузки выполняются загрузчиком, работа загрузчика разделяется на две части. Первая часть вырабатывает загрузочный модуль, состоящий из всех объектных сегментов, связанных и перемещаемых вместе относительно стандартного базового адреса. Другая часть – операция загрузки – загружает модуль в основную память, настраивая адреса в соответствии с распределением памяти для модуля.

Редактор связей выполняет часть работы по распределению памяти, но основную работу по распределению памяти выполняет загрузчик. Статический принцип настройки адресов выполняет статический алгоритм распределения памяти. Вся необходимая основная память для пользовательской программы и данных назначается до начала выполнения программы, а все адреса настраиваются так, чтобы отразить это назначение. Если настройка происходит во время выполнения, непосредственно предшествуя каждому обращению к памяти, то адреса настраиваются динамически. При динамической настройке адресов связывающий загрузчик должен быть построен по более сложным алгоритмам.

Входящая информация для связывающего загрузчика состоит из набора объектных программ, т.е. управляющих секций, которые должны быть связаны друг с другом. В управляющих секциях могут использоваться внешние ссылки на имена, значения которых во входном потоке еще не были определены. В этом случае требуемое связывание не может быть выполнено до тех пор, пока не будут назначены адреса для всех требующих имен, т.е. до тех пор, пока не будет прочитана требуемая управляющая секция, поэтому связывающий загрузчик обычно выполняет два просмотра входного потока точно так же, как это делает ассемблер. Т.е. во время первого просмотра назначаются адреса для всех внешних ссылок, а во время второго – выполняет фактическая перемещение, связывание и загрузка.

Основная структура данных, необходимая для связывающего загрузчика – это таблица внешних имен (ESTAB). Данная таблица используется для хранения имен и адресов всех внешних ссылок для всего набора управляющих секций, загружаемых совместно. Очень часто в этой таблице также запоминается информация о том, какая управляющая секция содержит определение имени.

Обычно ESTAB организуется в виде КЭШ-таблицы. Двумя другими важными переменными являются PROGADDR и CSADDR – начальный адрес той управляющей секции, которая обрабатывается загрузчиком в данный момент. Этот адрес добавляется к всем относительным адресам данной управляющей секции для того, чтобы преобразовать их в фактические адреса.

Во время первого просмотра загрузчик обрабатывает только запись-заголовок и записи определения управляющих секций. PROGADDDR становится начальным адресом первой управляющей секции входного потока. Имя управляющей секции, полученное из записи-заголовка, записывается в ESTAB и ему присваивается текущее значение CSADDR.

Все внешние имена из записей-определений также заносятся в ESTAB. Значения их адресов получается путем сложения значения из записи-определения с CSADDR.

После того, как прочитана запись-конец к CSADDR добавляется длина управляющей секции и таким образом получается адрес начала следующей управляющей секции.

После завершения первого просмотра ESTAB содержит все внешние имена, определенные в данном наборе управляющих секций вместе с назначенными им адресами.

Далее осуществляется связывание, перемещение и загрузка. Переменная CSADDR используется также, как и во время первого просмотра. Она всегда содержит фактический начальный адрес загруженной в данный момент секции. Завершает работу загрузчик обычно передачей управления на загружаемую программу.

Запись-конец каждой управляющей секции может содержать адрес первой команды данной секции, с которой должно начинаться ее исполнение. Если адрес передачи управления задан более чем в одной управляющей секции, то загрузчик использует последний встретившийся. Если ни одна из управляющих секций не содержит адрес передачи управления, то используется PROGADDR.

Адрес передачи управления должен помещаться только в записи-конце главной программы, но не в подпрограммах, чтобы стартовый адрес не зависел от порядка следования управляющих секций.

29. Защита информации в персональных ЭВМ. Средства, позволяющие контролировать доступ.

При загрузке с винта система BIOS читает главную загрузочную запись MBR. Обычно программа записанная в MBR загружает загрузочный сектор активного раздела, однако, можно использовать MBR для организации контроля входа в ПЭВМ, так как размер программы MBR невелик, то эффективно использовать MBR для загрузки большей по объему программы. При применении стандартной разбивки жесткого диска на разделы с помощью программы FDISK сектора, начиная с третьего на нулевой дорожке нулевого цилиндра, не используются, поэтому их можно применить для хранения программы, выполняющей функции контроля. Помимо того в записи MBR хранится первичная таблица описателей логических дисков, изменение этой таблицы позволяет предотвратить доступ к логическому диску при загрузке с дискеты.

Первый сектор активного раздела, осуществляющий загрузку ОС можно также использовать для загрузки программы, реализующей разграничение доступа. Модификация таблицы параметров, которые содержатся в загрузочной записи логического диска, может использоваться для предотвращения доступа к этому логическому диску при загрузке с дискеты. Функции разграничения доступа можно реализовать с помощью драйвера устройства. Этот драйвер может контролировать доступ к файлам и нестандартно определенным логическим дискам. На драйвер можно возложить запрос пароля на вход в систему. Перехват прерываний на различных этапах загрузки системы позволяет организовать дополнительные разграничения доступа. Перехват прерывания 13h дает возможность реализовать режим прозрачного шифрования данных на жестком диске, а также запрещать доступ к гибким дискам и к отдельным частям жесткого. Этот режим предохраняет от копирования данных с жесткого диска при загрузке с дискеты. Перехват прерывания int 13h целесообразно осуществлять до загрузки ОС. Перехват прерываний int 21h и других прерываний позволяет организовать разграничение доступа к файлам и каталогам.

Система разграничения доступа должна обеспечивать выполнение следующих функций:

1. Аутентификация пользователя по паролю и возможно по ключевой дискете.
2. Разграничение доступа к логическим дискам.
3. Прозрачное шифрование логических дисков, что имеет смысл только при наличии соответствующей аппаратуры, так как любые программные способы подразумевают хранение ключей в ОП во время выполнения шифрования.
4. Шифрование выбранных файлов, которое может осуществляться в прозрачном режиме или по требованию.
5. Разграничение доступа к каталогам и файлам, включая посекторную защиту от чтения-записи данных, защиту от переименования и перемещения и запрет модификации областей FAT и каталогов для выбранных файлов. Для реализации этой функции необходимо хранить таблицы большого объема, описывающие полномочия по доступу к каждому сектору и теневые таблицы FAT и каталогов для проверки корректности при их перезаписи.
6. Разрешение запуска строго определенных для пользователя программ, загрузка и выполнение программ реализуется средствами прерывания int 13h BIOS, поэтому фактически для всех программ, запуск которых запрещен для пользователя, должен устанавливаться режим недоступности соответствующих файлов.
7. На всех этапах работы должна реализовываться защита от отладчиков.

Наиболее важным является этап аутентификации пользователя. При ее выполнении до загрузки ОС желательно проверить что ни одна программа не загружена в память, а если загружена, то не получит управление. Для этого необходимо убедится в выполнении следующих условий:

1. система не запущена в режиме виртуального процессора 8086;
2. закрыта двадцатая адресная линия;
3. размер памяти в области данных системы BIOS соответствует размеру памяти ЭВМ;
4. Все используемые программой или аппаратурой вектора прерываний указаны на область постоянной памяти.

30. Защита информации в персональных ЭВМ. Защита от копирования.

Будем считать программу защищенной от нелегального копирования, если в ней есть встроенные средства, позволяющие проверить саму программу или характерные признаки ПК на котором она выполняется с целью определить была ли копия программы сделана с соблюдением всей необходимой технологией. Если технология создания копии была нарушена, программа перестает работать нормальным образом. Любая копия защищенной программы должна содержать в себе или во внешнем файле ключ. В момент проверки программа сравнивает некоторые специальные признаки рабочей среды с заранее закодированными в ключе и по результатам сравнения формирует соответствующий признак. Таким образом, чтобы копия программы стала работоспособной ей необходимо передать ключ, настроенный на работу с определенным компьютером.

Система защиты от копирования должна выполнять следующие функции:

1. Установка программы на жесткий диск. Функция состоит в привязке программы к конкретной ПЭВМ и обеспечивает защиту от копирования. В этом случае при установке программы на жесткий диск сохраняются характеристики ПЭВМ специфические, только для данного компьютера.
2. Проверка ключевой дискеты. Функция состоит в чтении записанной ранее информации и сравнения ее с эталонной. Наряду с функцией проверки ключевой дискеты необходимо предусмотреть функции проверки и модификации определенных счетчиков, содержащихся в данных и скрытых на ключевой дискете. Данные могут быть расположены либо в секторах, невидимых для ОС, либо в межсекторных промежутках.
3. Создание дискет защищенных от копирования. Эта функция предполагает запись на дискету информации, не копируемой обычными средствами. Наиболее эффективно данную функцию можно реализовать, использовав приемы нестандартного программирования контроллера гибких дисков.
4. Защита программ от отладчика и модификаций. Функция служит для предотвращения анализа алгоритмов программ, как входящих в состав системы защиты от копирования, так и защищаемых этой системой. Защита программ от модификации препятствует несанкционированному изменению кода программ и обычно реализуется путем подсчета контрольных сумм по определенному алгоритму. В программе защищаемой от отладчика нежелательно использовать стандартные средства DOS и BIOS. Защита от отладчиков заключается в том, что программа распознает факт пошаговой трассировки и пытается тем или иным способом противодействовать этому процессу.
5. Использование нестандартного форматирования дискет - например, запись нестандартного количества секрторов на дорожку, пропуск одного или нескольких секторов, запись сектора нестандартной длинны.

Программа может определить факт трассировки:

1. с помощью контроля отладочных прерываний (int1 и int3);
2. с помощью замера времени выполнения некоторого эталонного участка программы;
3. с помощью перехвата прерывания от клавиатуры;
4. с помощью перехвата прерывания от таймера;
5. использование нестандартного форматирования дискет:
   - форматирование с непоследовательными номерами секторов;
   - запись нестандартного количества секторов на дорожку;
   - пропуск одного или нескольких секторов;
   - запись секторов нестандартной длины;
   - запись секторов с особым номером;
   - запись неверных значений для байтов, описывающих номер цилиндра и номер поверхности в адресном поле.