Билет №1

1. Архітектура пристрою з плаваючою точкою. - 3
2. Машинно-незалежна оптимізація. - 4
3. Способи організації драйверів в ОС. - 5

Билет №2

1. Форми даних пристрою з плаваючою точкою. - 7
2. Методи висхідного синтаксичного аналізу. - 7
3. Роль переривань в побудові драйверів. – 9

Білет №3

1. Команди для пересилання даних з пристроєм з плаваючою точкою. - 10
2. Алгоритм синтаксичного аналізу з використанням матриць передування. – 11
3. Основні стани виконання задач в ОС. – 11

Білет №4

1. Особливості іменування та використання регістрів з плаваючою точкою. - 12
2. Поняття граматик та їх використання для розв'язування задач. – 17
3. Організація роботи планування задач і процесів. – 18

Білет 5

1. Базові команди арифметичних операцій пристрою з плаваючою точкою. – 19
2. Граматики, що використовуються для синтаксичного аналізу – 20
3. Організація захисту пам'яті в ОС – 21

Білет 6

1. Перевірка умов за результатами пристрою з плаваючою точкою – 22
2. Граматики, що використовуються для лексичного аналізу – 23
3. Ієрархічна організація програм введення-виведення – 24

Білет 7

1. Особливості архітектури розширених ММХ. - 25
2. Машинно-залежна оптимізація. - 26
3. Необхідність синхронізації даних в задачах ввод-вивід - 27

Білет №8

1. Команди для обчислення часткових математичних функцій. - 28
2. Грамматики классифицируются по виду их правил вывода (по Хомскому) – 31
3. Збереження стану задач в реальному режимі - 31

Білет №9

1. Особливості арифметичних операцій розширення ММХ - 32
2. Організація інтерпретації вхідної мови. - 34
3. Організація захисту пам'яті в процесорах. – 37

Білет №10

1. Класифікація системних програм – 38
2. Оганізація семантичного аналізу в компіляторах - 39
3. Типовий склад програм ОС – 44

Екзаменаційний білет №11.

1. Узагальнена структура системної програми – 44
2. Формування графів синтаксичного розбору при синтаксичному аналізі. – 44
3. Збереження стану задач в захищеному режимі. – 46

Білет №12

1. Оранізація таблиць у вигляді масивів записів – 47
2. Загальний підхід до ораганізації семантичної обробки в трансляторах – 48
3. Способи організації драйверів в ОС -52

Білет №13

1. Лінійний пошук в різних видах таблиць. - 55
2. Способи організації трансляторів з мов програмування. - 56
3. Особливості визначення пріоритетів задач в ОС. – 56

Білет №14

1. Пошук за прямою адресою та хеш-пошук. - 58
2. Типові об’єкти системних програм. - 59
3. Ієрархічна організація програм введення-виведення. – 60

Білет №15

1. Основні способи організаціїї таблиць та індексів - 61
2. Системні оброблюючі програми - 63
3. Механізми переключення задач в архітектурі процесора – 64

Білет № 16

1. Організація таблиць у вигляді структур з посиланнями – 66
2. Задачі лексичного аналізу – 66
3. Графи та їх обробка – 66

№17

1. Двійковий пошук - 67
2. . Задачі семантичної обробки – 69
3. Способи організації переключення задач – 70

№18

1. Задачі лексичного аналізу – 69
2. Поняття граматик та їх застосування для розв'язання задач – 70
3. Типи ОС та режими їх роботи – 71

БИЛЕТ 19

1. Системні управляючі програми – 73
2. Алгоритм Низхідного синтаксичного розбору – 73
3. . Способи переключення задач – 76

Білет 20

1. Основні способи організації пошуку в таблицях – 76
2. Метод синтаксичних графів для синтаксичного аналізу - 79
3. Організація генерації кодів – 79

БИЛЕТ 22

1. Класифікація граматик за Хомським – 80
2. Особливості математичних операцій MMX-розширення - 80
3. Використання команд введеня-виведення для управління зовнішніми пристроями – 82

Білет №24

1. Методи низхідного розбору. – 84
2. Особливості пересилок і перетворень в розширенні ММХ. - 87
3. Побудова таблиць у вигляді списків. - 88

Білет 25

1. Метод рекурсивного спуску – 89
2. Розширення ММХ – 90
3. Організація підсистем введення – виведення – 92

Білет №26

1. Метод синтаксичних графів. – 93
2. Організація циклів на базі процесора з плаваючою точкою – 93
3. Особливості роботи з БПП – 94

Билет №27

1. Построение алгоритмов анализа с использованием функций предшествования. – 96
2. .Особенности команд логических операций и сдвигов в расширениях ММХ – 96
3. Программно-аппаратные взаимодействия при обработке прерываний в машинах IBM PC. - 96

Билет №1

1. ***Архітектура пристрою з плаваючою точкою.***

Архитектурой называют комплекс програмно достигнутых аппаратных средств. Архитектура устройства с плав.зап. включает 8 регистров , каждый и з которых включает 80 бит. В архитектуру входят 3 двухбайтных регистра , которые имеют название CW – управл., SW- состояния, TW – рег.тэгов . Слово состояния сохраняет 3-х битовый указатель на вершину вх. стека , который образовывается в этих 8 реги страх . В этом рег. записываются признаки результатов. Они похожи на признаки результатов флагов F центр. процессора .В мл. разрядах исп. 6 бит признаков исключительных ситуаций , кот. возникают при вычислении . CW включает управления режимами процессора . В мл. разрядах есть 6 бит маскирования исключительных ситуаций , кот. фиксируятся в SW . TW использует по 2 бита определения состояния для каждого из 8-ми регистров . Состояние может указывать на неопределенное значение регистра , на обычное или специальное значение регистра.

Представление для dd
|7 mmmmmmmm 0|15 mmmmmmmm 8| 23 c.mmmmmmm 16|31 s ccccccc 24|
Представление для dq
|7 mmmmmmmm 0|15… 41| 55 cccc.mmmm 48|63 s ccccccc 56|
.data
X dd (dq) 23.23
X=41 b9 d7 0a (40 37 3a e1 47 ae 14 7b)

Командой **обмена** является команда FXCH она осуществляет обмен двух регистров сопроцессора (лише в стеку, один з регістрів st[0]). Команда применяется в двух форматах: без операндов и с одним операндом. Команда без операндов осуществляет обмен st(0) и st(1), если задан операнд [FXCH ST(i)], обмен осуществляется между st(0) и st(i).

Для ***загрузки*** операндов в стек можно использовать следующие команды: ***FLD*** источник – загрузка в st(0) вещественного числа из области памяти; FLD1, FLDL2T, FLDL2E, FLDLG2, FLDLN2, FLDPI, FLDZ – загрузка в вершину стека констант: вещ. единица (1.0), log2(10), log2(e), log10(2), ln(2), π, нуль соответственно.

Аргументы стандартных математических функций языка передаются в подпрограмму через верхушку стека процессора в виде числа с плавающей точкой в формате длинного вещественного представления (float double), для чего в вызывающей последовательности для одноместной функции можно использовать следующие команды, включая команды сопроцессора:

```
fld  x;     Загрузка аргумента в стек сопроцессора
sub  sp,8;  Подготовка места в стеке главного процессора
mov  bp,sp; Подготовка указателя
fstp qword ptr[bp]; Пересылка аргумента в стек из сопроцессора
call  1имя входной точки функции в библитечном модуле
add  sp,8;  Освобождение стека
```

Для снятия результата используются команды ***FST (FSTP)*** периемник – команды **сохранения вещ. число** из st(0) **в память или др. регистр** стека. **Различие** команд только в том, что если есть P, то происходит выталкивание значения из вершины стека сопроцессора(+1 к вказівнику стеку).

Приклад:

```
.data
X dd 0.5
.code
FLD X           ; ST[0] ← X
FLDL2E; ST[1] ← ST[0]; ST[0] ← LOG2(e)
FXCH            ; обмін ST[0] і ST[1]
```

2. ***Машинно-незалежна оптимізація.***

Целью данной фазы обработки программы является уменьшение затрат программы по памяти и по времени.

- изъятие повторных вычислений, кот-е не использовались, фрагменты программ на которые не возможно попасть с помощью дополнительных управляющих операторов.
- преобразование блок-схемы программы, т.е. фрагменты программы , что повторяются в циклах, выходят за границы цикла, что бы избежать повторений.

Наиболее хорошо проработаны алгоритмы для некоторых частных случаев избыточности, однако в общем случае оптимизация связана с анализом смысла и поиском решения задачи. Фазы оптимизации не всегда присутствуют в составе компил яторов - все зависит от целей проектирования компилятора. Если целью проектирования является скорость работы компилятора, то фаза оптимизации не включается в его состав. Если же целью является минимизация затрат памяти и максимизация скорости работы программы, то модуль оптимизации включается в состав компилятора и время его работы может быть соизмеримо с суммарным временем работы остальных модулей компилятора. Ниже рассматриваются несколько простых алгоритмов оптимизации программы в промежуточной форме:

1) исключение общих подвыражений;

2) вычислений во время трансляции;

3) вынесение инвариантных выражений за цикл;

4) оптимизация булевых выражений.

Для виконання такого виду оптимізації важливо використовувати комбінації команд, яка використовують спеціальні особливості регістрів загального призначення. Машинно незалежна оптимізація передбачає вилучення окремих груп вузлів з внутрішнього подання і їх заміну більш ефективними елементами, або раніше виконуваними елементами.

Вона включає і інші задачі серед яких є усунення повторних обчислень та обчислень, результат яких не використовується. Щоб усунути повторні обчислення в програмі необхідно проаналізувати граф програми на наявність однакових під графів, для яких використовуються однакові значення змінних підграфа. Однакові підграфи можна замінити посиланнями на перше використання підграфа. Для цього треба вміти організовувати пошук чергового підграфа серед підграфів програми. Для того щоб реалізувати такий пошук відносно швидким доцільно побудувати індекс над вершинами підграфа за визначеним відношенням підгарафа. Однак аналіз областей існування значень змінних потребує додаткових інформаційних структур, які використовувались в тому чи іншому іншому піддереві.

3. ***Способи організації драйверів в ОС.***

Драйвер периферийного устройства служит логическим *интерфейсом между подсистемой ввода/вывода операционной системы и* обслуживаемыми ею *аппаратными средствами*. В некоторых случаях драйвер заменяет или расширяет определенные аспекты BIOS'a и таким образом несет ответственность за обработку характеристик обслуживаемых аппаратных средств. В других драйверах физический ввод/вывод осуществляется исключительно через BIOS. В ранних версиях операционной системы *MS-DOS* не была предусмотрена процедура установки драйверов. Однако, *начиная с версии 2.0*, программисты получили возможность дополнять операционную систему. Несмотря на отсутствие общих стандартов для резидентных программ, *интерфейс между операционной системой и драйвером периферийного устройства жестко определен*, поэтому большинство драйверов может работать друг с другом совместно.

Драйверы бывают двух типов: *ориентированные на символьный либо блоковый обмен данными*. Драйверы, обслуживающие хранение и/или доступ к данным на дисках либо других устройствах прямого доступа, обычно ориентированы на блоковый обмен данными. Блоковые драйверы передают данные фиксированными порциями - отсюда и происходит их название.

Драйвер состоит из трех основных частей: ***заголовка, программы стратегий и программы прерываний***. *Заголовок* описывает возможности и атрибуты драйвера, присваивает имя драйверу символьного устройства и содержит NEAR (только смещение, одно слово) указатели на программы стратегий и прерываний, а также FAR (смещение и сегмент, двойное слово) указатель на следующий драйвер в цепочке драйверов (цепочка является однонаправленной, что затрудняет поиск ее начала). Указатель на следующий драйвер устанавливает MS-DOS сразу после завершения процедуры инициализации, а в самом драйвере ему должно быть присвоено начальное значение -1

(FFFFFFFFH). *Драйвер должен помещаться в 64К памяти, т.к. программы стратегий и прерываний содержат лишь смещения внутри выделенного драйверу сегмента*.

Программа стратегий. Программа стратегий вызывается, когда драйвер устанавливается при загрузке операционной системы, а также при каждом сгенерированном операционной системой запросе на ввод/вывод. Единичный запрос на ввод/вывод от прикладной программы может породить несколько запросов к драйверу. *Задача данной* программы заключается лишь в том, чтобы *сохранить* где-нибудь некоторый *адрес для дальнейшей обработки*. Этот адрес, *передаваемый в паре регистров ES:BX*, *указывает на структуру, называемую* ***заголовком запроса,*** которая содержит информацию, сообщающую драйверу, какую операцию он должен выполнить.

```
rlength db 0 ; 0 - длина существенных данных в заголовке
unit db 0 ; 1 - номер элемента
command db 0 ; 2 - фактическая команда
status dw 0 ; 3 - состояние после возврата
reserve db 8 dup (0) ; 5 - зарезервировано для DOS
media db 0 ; 13 - дескриптор среды
address dd 0 ; 14 - длинный указатель для ввода/вывода
count dw 0 ; 18 - число символов для ввода/вывода (целое без знака)
sector dw 0 ; 20 - начальный сектор
```

Существенно то, что ***программа стратегий*** *не осуществляет никаких операций ввода/вывода, а лишь сохраняет адрес заголовка запроса для последующей обработки программой прерываний*. В мультизадачной системе этот адрес должен был бы храниться в некотором массиве, который при последующем вызове процедуры прерываний подвергался бы сортировке с целью оптимального использования устройства. Под управлением MS-DOS программа прерываний вызывается сразу после программы стратегий. Следует обратить внимание на то, что *между вызовами программ стратегий и прерываний допустимы прерывания*, а это может создать трудности, если драйвер написан в предположении, что между этими двумя вызовами проходит "нулевое время".

Программа прерываний. В программе прерываний *выполняется вся фактическая работа драйвера*, поэтому она является наиболее сложной его частью. *При вызове этой программы исследуется командный байт (третий байт) ранее сохраненного заголовка запроса* и в зависимости от его значения выполняются те или иные действия. Программа прерываний *обычно использует командный байт в качестве индекса для некоторой управляющей таблицы*, вызывая таким образом нужную процедуру для каждой команды. Конечно, при желании можно использовать таблицу переходов.

Если вместо цикла ожидания использовать систему апаратных прерываний по готовности, то нет необходимости менять структуру драйвера, но цикл ожидания надо заменить на обращения к системной программе, обеспечивающей взаимодействие с обработчиком прерываний. Для реализации системы прерываний(в частности апаратных прерываний по готовности внешних устройств). В интерфейсные схемы включен блок програмируемых прерываний. Этот блок, как и ВУ, подключается через порты ввода/вывода, т.е. в нем есть порты программирования и порты настройки. В машинах типа IBM PC настройка этих устройств стандартизована, т.е. она выполняется так, что обращение к аппаратным прерываниям осуществляется по адресам(номерам векторов прерываний с 08h по 0Fh и с 0C0h...0C7h). Таким образом, блок программируемого прерывания имеет 16 входов, к которым можно подключать сигналы готовности ВУ. При поступлении сигнала готовности блок ПП генерирует команду int<Nвектора>.

Структура підпрограм драйверів пристроїв включає наступні блоки:

1. видача команди на пристрій для його підготовки до обміну
2. очікування готовності пристрою до обміну
3. виконання власне обміну
4. видача на пристрій команди для закінчення операції обміну
5. організація обміну драйвера даними з програмою, яка його використовує.

При створенні мікропроцесорів фактично було повторено процес створення програм обміну для зовнішніх пристроїв, але вже з деякою систематизацією. В структурі процесорів Pentium використали системи обміну через порт введення/виведення.

Порт- фізична адреса пристрою, яка розпознається чеерз інтерфейсні схеми in al(ax,eax),[dx(edx)]

Підключення зовнішніх пристроїв до мікропроцесорів виконувалось шляхом визначення вихідного командного порту, вхідного порту стану, які мали однаковий номер, та вхідного та вихідного порту для введення/виведення даних, які також мали однакову адресу. Тому щоб написати узагальнений простий драйвер треба було визначити адреси портів

```
CMPRT EQU 41h
CTPRT EQU CMPRT
INPRT EQU 42h
OUTPRT EQU INPRT
drIn Proc
 mov al,CmOn
 out CMPRT,al
 in al,STPRT
kew:     test al,RdyBit
 jnz kew
 in al,INPRT
 push ax
 mov al,CmOff
 out CMPRT,dl
 pop ax
 ret
drIn endp
```

Билет №2

1. ***Форми даних пристрою з плаваючою точкою.***

Устройство с плавающей точкой, а именно, микросхема 8087 работает с 7-ю типами данных, 6-ть из которых присущи только этой микросхеме. четыре формата представляют целые числа, а это слово – 16 бит (единственный формат данных общий для 8088 и 8087) – оператор dw, короткое целое – 32 бита – оператор dd и длинное целое – 64 бита (эти три формата представлены в двоичном дополнительном коде) – оператор dq и четвертый - упакованные десятичные числа (10 байт, 1 байт - знаковый) – оператор dt. три формата представляют вещественные числа или числа с плавающей точкой, а это - короткий формат (32 бита), длинный (64 бита), и специальный (80 бит) – формат промежуточного действительного числа для повышенной точности.

Представление для dd
|7 mmmmmmmm 0|15 mmmmmmmm 8| 23 c.mmmmmmm 16|31 s ccccccc 24|
Представление для dq
|7 mmmmmmmm 0|15… 41| 55 cccc.mmmm 48|63 s ccccccc 56|

```
.data
X1 dw  2
X2 dd  5
X3 dq  6
```

Устройство с плавающей точкой, а именно, микросхема 8087 работает с 7-ю **типами данных**, 6-ть из которых присущи только этой микросхеме. четыре формата представляют целые числа, а это слово – 16 бит (единственный формат данных общий для 8088 и 8087) – оператор dw, короткое целое – 32 бита – оператор dd и длинное целое – 64 бита (эти три формата представлены в двоичном дополнительном коде) – оператор dq и четвертый - упакованные десятичные числа (10 байт, 1 байт - знаковый) – оператор dt. три формата представляют вещественные числа или числа с плавающей точкой, а это - короткий формат (32 бита), длинный (64 бита), и специальный расширеный (80 бит) – формат промежуточного действительного числа для повышенной точности.

Итак, 0,625 = 0,101b. При записи вещественных чисел всегда выполняют нормализацию — умножают число на такую степень двойки, чтобы перед десятичной точкой стояла единица, в нашем случае $0,625 = 0,101b = 1,01b * 2^{-1}$

Говорят, что число имеет мантиссу 1,01 и экспоненту -1. Как можно заметить, при использовании этого алгоритма первая цифра мантиссы всегда равна 1, так что ее можно не писать, увеличивая тем самым точность представления числа дополнительно на 1 бит. Значение экспоненты хранят не в виде целого со знаком, а в виде суммы с некоторым числом так, чтобы хранить всегда только положительное число и чтобы было легко сравнивать вещественные числа — в большинстве случаев достаточно сравнить экспоненту. c = 2^(n-1) --1 + p, где n- кол. разрядов для предоставления порядка, p – знач. двоичного порядка для представления мантиссы числа с единицей в целой части.

Вещественные форматы, используемые в процессорах Intel:

короткое вещественное: бит 31 — знак мантиссы, биты 30 – 23 — 8-битная экспонента + 127, биты 22 – 0 — 23-битная мантисса без первой цифры;

длинное вещественное: бит 63 — знак мантиссы, биты 62 – 52 — 11-битная экспонента + 1024, биты 51 – 0 — 52-битная мантисса без первой цифры;

расширенное вещественное: бит 79 — знак мантиссы, биты 78 – 64 — 15-битная экспонента + 16 383, биты 63 – 0 — 64-битная мантисса с первой цифрой (то есть бит 63 равен 1).

FPU выполняет все вычисления в 80-битном расширенном формате, 32- и 64-битные числа используются для обмена данными

2. ***Методи висхідного синтаксичного аналізу.***

Алгоритм должен содержать в себе блоки обращения к лексическому анализу для получения лексем, и возвращения к семантической обработке.

Для того чтобы алгоритм имел более общую форму, в начальных установках алгоритма целесообразно сформировать фиксированный элемент с низким приоритетом фиктивной операции, который будет служить признаком конца обработки выражения. Ображения к процедуре, в результате можно получить или одну лексему или пару (операнд и операция). Сравнение приоритетов текущей операции с предыдущей. Если новой выше, то ее рядом с операндом следует

запомнить в стэке и снова обратится к *выражению,* если нет, то к программе семантической обработки с тем, чтобы получить со стэка предыдущую операцию и выполнить необходимые действия.

Программы с таким алгоритмом можно записать в трех вариантах :

1) программа которая использует специальный стэк в виде массива;

2) рекурсивная процедура на языке высшего уровня, которая использует системный стэк для сохранения лексем как аргументов рекурсивных вызовов и адресов возврата в программу;

3) циклическую программу на языке асм., которая будет использовать в середине процедуры только для сохранения промежуточных значений.

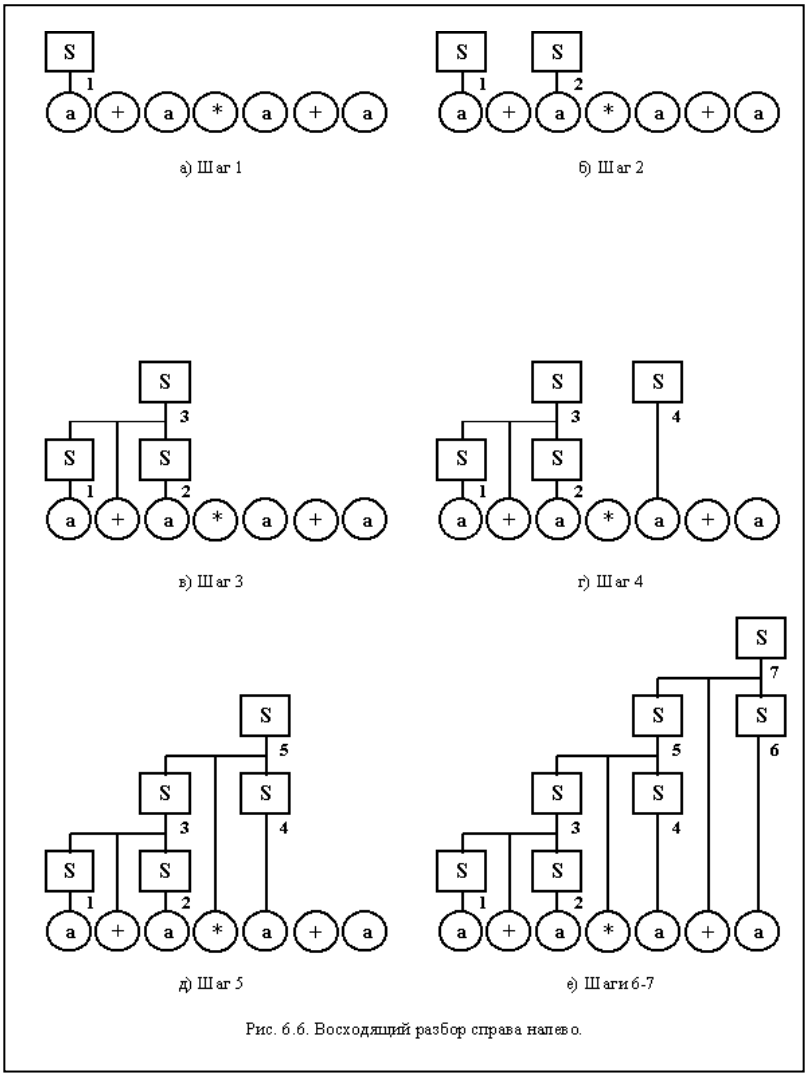


Рис. 6.6. Восходящий разбор справа налево.

При **восходящем разборе** дерево начинает строиться от терминальных листьев путем подстановки правил, применимых к входной цепочке, опять таки, в общем случае, в произвольном порядке. На следующем шаге новые узлы полученных поддеревьев используются как листья во вновь применяемых правилах. Процесс построения дерева разбора завершается, когда все символы входной цепочки будут являться листьями дерева, корнем которого окажется начальный нетерминал. *Если, в результате полного перебора всех возможных правил, мы не сможем построить требуемое дерево разбора, то рассматриваемая входная цепочка не принадлежит данному языку*.

Восходящий разбор также непосредственно связан с любым возможным выводом цепочки из начального нетерминала. Однако, эта связь, по сравнению с нисходящим разбором, реализуется с точностью до "наоборот".

На рис. приведены примеры построения деревьев разбора для грамматики $\mathbf{G_6}$ и процессов порождения цепочек, представленных выражениями.

В общем случае *для восходящего разбора* строится так называемые *грамматики предшествования*. В грамматике предшествования строится матрица попарных отношений всех терминальных и не терминальных символов. При этом определяется три вида отношений: R предшествует S (R<•S); S предшествует R (R•>S); и операция с одинаковым предшествованием (R•=S); четвертый вариант отношение предшествия отсутствует (это говорит о недопустимости использования R и S рядом в тексте записи на этом языке)

(з конспекту)

Найпростіший алгоритм висхідного розбору для аналізу математичних виразів. На основі пріоритетів окремих операцій. Порівнюємо пріоритет наступної з пріоритетом попередньої. Незалежно від результатів порівняння цього пріоритету треба обрати варіант подальшої роботи - якшо має низький пріоритет - відкласти обробку операції записуючи її в стек. Якщо пріоритет більш високий - виконати інтерпретацію чи саму операцію. Результат такого розбору буде дерево розбору (дерево підлеглості)

3. ***Роль переривань в побудові драйверів.***

Драйвер состоит из трех основных частей: ***заголовка, программы стратегий и программы прерываний***.

Программа прерываний. В программе прерываний *выполняется вся фактическая работа драйвера*, поэтому она является наиболее сложной его частью. *При вызове этой программы исследуется командный байт (третий байт) ранее сохраненного заголовка запроса* и в зависимости от его значения выполняются те или иные действия. Программа прерываний *обычно использует командный байт в качестве индекса для некоторой управляющей таблицы*, вызывая таким образом нужную процедуру для каждой команды. Конечно, при желании можно использовать таблицу переходов.

Заголовок запроса содержит всю необходимую информацию для корректной обработки каждой команды и сообщает вызывающей программе (в большинстве случаев таковой является MS-DOS) о состоянии запроса после завершения соответствующей процедуры. *Слово, хранящее состояние после возврата, разбито на несколько полей. Оно содержит бит ошибки*, указывающий на то, что в оставшейся части содержится специфический код ошибки, *бит выполнения*, сигнализирующий о том, что требуемая операция была завершена, и *бит занятости*, призванный в первую очередь сигнализировать о текущем состоянии устройства.

Обязательно должны присутствовать три раздела драйвера: **ЗАГОЛОВОК ДРАЙВЕРА, ПРОГРАММА СТРАТЕГИЙ и ПРОГРАММА ПРЕРЫВАНИЙ.** Программа ПРЕРЫВАНИЙ это не тоже самое, что программа обработки прерываний, которая может присутствовать в качестве необязательной части работающего по прерываниям драйвера. На самом деле, программа ПРЕРЫВАНИЙ - это точка входа в драйвер для обработки получаемых от MS-DOS команд.

Выше представлен скелет драйвера устройства. Хотя структура драйвера похожа на структуру .COM программы, важно отметить следующие отличия:

- *Программа начинается с нулевого смещения, а не 100H*.
- Образ программы начинается с директив определения данных для заголовка драйвера.
- Программа не содержит директивы ASSUME для стекового сегмента.
- Программа не содержит директивы END START.

Заголовок драйвера, программы СТРАТЕГИЙ и ПРЕРЫВАНИЙ

```
DRIVER SEGMENT PARA
ASSUME CS:DRIVER,DS:NOTHING,ES:NOTHING
ORG 0
START EQU $ ; Начало драйвера
;******* ЗАГОЛОВОК  ДРАЙВЕРА************************
dw -1,-1 ; Указатель на следующий драйвер
dw ATTRIBUTE ; Слово атрибутов
dw offset STRATEGY ; Точка входа в программу STRATEGY
dw offset INTERRUPT ; Точка входа в программу INTERRUPT
db 8 dup (?) ; Количество устройств/поле имени
;******* РЕЗИДЕНТНАЯ ЧАСТЬ ДРАЙВЕРА
                        req_ptr dd ? ; Указатель на заголовок запроса
;******* ПРОГРАММА СТРАТЕГИИ
; Сохранить адрес заголовка запроса для программы СТРАТЕГИЙ
; в REQ_PTR.
; На входе адрес заголовка запроса находится в регистрах ES:BX.
STRATEGY PROC FAR
mov cs:word ptr [req_ptr],bx
mov cs:word ptr [req_ptr + 2],bx
ret
STRATEGY ENDP
;******* ПРОГРАММА ПРЕРЫВАНИЙ
; Обработать команду, находящуюся в заголовке запроса.
; Адрес заголовка запроса содержится в REQ_PTR в форме
; СМЕЩЕНИЕ:СЕГМЕНТ.
INTERRUPT PROC FAR
pusha ; Сохранить все регистры
lds bx,cs:[req_ptr] ; Получить адрес заголовка запроса
…
INTERRUPT ENDP
…
DRIVER ENDS
END
```

Білет №3

1. **Команди для пересилання даних з пристроєм з плаваючою точкою.**

Командой обмена является команда FXCH она осуществляет обмен двух регистров сопроцессора (лише в стеку, один з регістрів st[0]). Команда применяется в двух форматах: без операндов и с одним операндом. Команда без операндов осуществляет обмен st(0) и st(1), если задан операнд [FXCH ST(i)], обмен осуществляется между st(0) и st(i).

Для *загрузки* операндов в стек можно использовать следующие команды: *FLD* источник – загрузка в st(0) вещественного числа из области памяти; FLD1, FLDL2T, FLDL2E, FLDLG2, FLDLN2, FLDPI, FLDZ – загрузка в вершину стека констант: вещ. единица (1.0), log2(10), log2(e), log10(2), ln(2), π, нуль соответственно.

Для снятия результата используются команды *FST (FSTP)* периемник – команды сохранения вещ. число из st(0) в память или др. регистр стека. Различие команд только в том, что если есть P, то происходит выталкивание значения из вершины стека сопроцессора(+1 к вказівнику стеку).

Fild, fist-ціле значення

FILD источник – загрузить целое число в стек

Преобразовывает целое число со знаком из источника (16-, 32- или 64-битная переменная) в вещественный формат, помещает в вершину стека и уменьшает TOP на 1.

FIST приемник – скопировать целое число из стека

FISTP приемник – считать целое число из стека

Преобразовывает число из вершины стека в целое со знаком и записывает его в приемник (16- или 32-битная переменная для FIST; 16-, 32- или 64-битная переменная для FISTP). FISTP после этого выталкивает число из стека (помечает ST(0) как пустой и увеличивает TOP на один). Попытка записи слишком большого числа, бесконечности или не-числа приводит к исключению «недопустимая операция» (и записи целой неопределенности, если IM = 1).
FBLD источник – загрузить десятичное число в стек
Преобразовывает BCD число из источника (80-битная переменная в памяти), помещает в вершину стека и уменьшает TOP на 1.
FBSTP приемник – считать десятичное число из стека
Преобразовывает число из вершины стека в 80-битное упакованное десятичное, записывает его в приемник (80-битная переменная) и выталкивает это число из стека (помечает ST(0) как пустой и увеличивает TOP на один). Попытка записи слишком большого числа, бесконечности или не-числа приводит к исключению «недопустимая операция» (и записи десятичной неопределенности, если IM = 1).
Приклад:

```
.data
X dd 0.5
.code
FLD X          ; ST[0] ← X
FLDL2E         ; ST[1] ← ST[0]; ST[0] ← LOG2(e)
FXCH           ; обмін ST[0] і ST[1]
```

2. Алгоритм синтаксичного аналізу з використанням матриць передування.

В матрицях передування визначаються 3 можливі відношення передування:

- * =
- R <* L
- L *> R

Вони визначають різні співвідношення, бо елементи беруться за напрямком.
R $\cup$ L – відсутність передування – забороняє передування двох елементів синтаксичної конструкції.
Матриця передування – квадратна матриця, координатами якої по X і Y є повна сукупність термінальних та не термінальних позначень. На перетині координат матриці записується одне з чотирьох відношень передування. Таким чином для висхідного розбору можна визначити граматику будь-якої контекстно-залежної мови, але в таблиці буде багато надлишкових елементів, що визначають можливі помилкові варіанти конструкцій. Тобто проектування висхідного синтаксичного аналізу методом матриць передування призводить до необхідності перетворення правил опису стандартної мови на відповідну матрицю.
Звичайно кількість термінальних та нетермінальних позначень включає кілька десятків таких елементів, що дозволяю отримати відносно невеликі таблиці
Потужні комп'ютерні мови включають декілька сотень граматичних правил, а розміри матриць дещо збільшуються. В тих випадках коли матрицю передування можна перетворити на функцію передування кажуть, що такі граматики можна лінізувати, або зробити лінійними.
Операторне передування – коли нема дужок і спеціальних символів

3. Основні стани виконання задач в ОС.

Задачі в процесі свого виконання можуть перебувати в таких станах:
- створення задачі при якому завантажувач ОС переносить код програми з диску до оперативної памяті і формує сегмент TSS і включає відновлення даних до таблиці задач.
- стан активної задачі. Використовуються всі регістри центрального процесора, а TSS відновлює момент попереднього запуску при переводі в активний стан
- стан очікування готової задачі для подальшого виконання задачі. Якщо задача має низький пріоритет вона запускається коли всі задачі з більшим пріоритетом не знаходяться в активному стані
- очікування результату вводу-виводу

Білет №4

1. Особливості іменування та використання регістрів з плаваючою точкою.

При написании программы удобно распределить регистры и в верхушке стека вычислять следующий член ряда, а сумму накапливать в глубине стека,чтобы потом перенести в память командами fst и fstp.

Процессор 8087 может вычислить любую элементарную трансцендентную функцию с аргументом, точность представления которого определяется содержимым регистра управления, и последующей выдачей результата такой же точности. В названиях команд и их описаниях принято обозначать аргументы, содержащиеся в st[0] как X, а в st[1] - как Y. Результаты, формирующиеся в st[0], будем обозначать как x, а в st[1] - как y.

Мнемоника команды	Название операции, Вычислительная формула	Ограничения на аргументы
FPTAN	Частичный тангенс --st, y/x = tg(X)	0<= X <=1 pi/4
FPATAN	Частичный арктангенс ++st, x = arctg(Y/X)	Y > X >0
FYL2X	Вычисление логарифма ++st, Y*log2(X)	X > 0
FYL2XP1	Вычисление логарифма ++st, Y*log2(X+1)	abs(X) < 1-sqrt(2)/2
F2XM1	Вычисление X=2^X-1	

Вычисление тригонометрических функций основано на выполнении команды FPTAN - нахождения частичного тангенса, которая в качестве результата дает два таких числа x и y, что y/x = tg(X). Число y заменяет старое содержимое st[0], а число x включается в стек дополнительно.

Диапазон изменения аргумента можно свести к допустимому командой FPREM или проверить с помощью команды FXAM, так как он должен быть нормализован и находиться в диапазоне 0 < st[0] < п/4. Если аргумент X 0лежит вне этого диапазона, то в начале программы необходимо выполнить преобразования по сведению аргумента к требуемому

диапазону и запомнить данные для обратного пребразования. Для функции tg(X) можно использовать 4 преобразования:

X < 0; S = sign(X); Y = abs(X); tg(X) = S*tg(Y); (1)

в этом перечне сначала указано условие сведения для отрицательных аргументов, затем две формулы для получения сведенных аргументов и, наконец, формула для восстановления итогового результата.

Y > п; Z = Y - n*п; tg(Y) = tg(Z); (2)

Здесь устраняется многократное повторение периода и дополнительных действий по восстановлению результата не требуется. Следующие два преобразования могут быть проведены в безусловной форме, так как модифицированный аргумент лежит в диапазоне 0 <= Y <= п.

U = Z/2; tg(Z) = 2*tg(U)/(1-tg(U)*tg(U)); (3)
V = U/2; tg(U) = 2*tg(V)/(1-tg(V)*tg(V)); (4)

Эти формулы, основанные на двухкратном применении известной в математике универсальной тригонометрической подстановки через тангенс половинного угла, легко реализуются, так как деление на 2 быстро осуществляется командой FSCALE, и могут быть преобразованы с учетом того, что результат вычисления функции FPTAN(V) сформирован в виде чисел x и y. Тогда

tg(V) = 2*x*y/(x*x-y*y) и u = 2*x*y; v = x*x-y*y, (5)

где u/v = tg(V) Эти формулы можно рассматривать как базовые для расчета тангенса и других тригонометрических функций с помощью команд FPTAN и FPREM, используя таблицу формул приведения и следующие формулы, выраженные через u и v.

sin(Y) = 2*(u/v)/(1+(u/v)^2);
cos(Y) = (1-(u/v)^2) / (1+(u/v)^2);
cosec(Y) = (1+(u/v)^2) / 2(u/v);
sec(Y) = (1+(u/v)^2) /(1-(u/v)^2).

Пример процедуры вычисления тангенса, построенной по рассмотренным формулам:

```
_tg PROC
    PUSH BP     ; Стандартное сохранение базового указателя стека.
    MOV  BP,SP  ; Установка нового значения базового указателя.
    FLDPI       ; Загрузка числа  1п
    FLD  QWORD PTR[BP+4] ; Загрузка начального значения суммы  x.
    FTST
    FSTSW stsw
    PUSH  stsw
rm: FPREM           ; Исключение периода
    fj     p,rm     ; Циклическое исключение остатка
;JP PF = 1 если количество единичных битов результата четно (четный паритет)
    FLD1
    FCHS
    FADD   st,st    ; Формирование - 2
    FXCH   st[1]
    FSCALE          ; Деление на 4
    FPTAN           ; Вычисление составляющих tg
    FLD    st[1]    ; Дублирование числителя
    FMUL   st,st    ; Квадрат числителя  y*y
    FXCH   st[1]    ; Обмен на знаменатель
    FMUL   st[2],st ; Вычисление  x*y
    FMUL   st,st    ; Вычисление  x*x
    FSUBP  st[1],st ; Получение -v
    FMUL   st,st[2] ; Получение -u
    FLD    st       ; Дублирование числителя u
    FMUL   st,st    ; Квадрат числителя  u*u
    FXCH   st[2]    ; Обмен на числитель
    FMUL   st[1],st ; Вычисление  u*v
    FMUL   st,st    ; Вычисление  v*v
    FSUBP  st[2],st ; Получение - знаменателя tg
    FMULP  st[2],st ; Получение - числителя tg
; Для этой команды нужно предусмотреть защиту от  особых ситуаций.
    FDIVP  st[1],st ; Получение значения tg
    FSTP   result   ; Сохранение результата
    MOV    AX,offset DGROUP:result
    POP  BP ; Стандартное восстановление базового указателя стека.
    RET     ; Выход из подпрограммы.
_tg ENDP
```

Кроме рассмотренного способа тригонометрические функции когут вычисляться либо через тангенс половинного угла по формуле (3), либо через тангенс полного угла по формулам:

sin(x) = tg(x) / sqrt(1+(tg(x)^2);
cos(x) = 1 / sqrt(1+(tg(x)^2);
ctg(x) = 1 / tg(x).

Однако эти формулы усложняют программу в части сведения углов и восстановления результатов по сравнению с формулами, использующими тангенс половинного угла.

Команда FPATAN вычисляет arctg(st[1]/st[0]) = arctg(Y/X). Два верхних элемента извлекаются из стека, а результат включается в стек. Операнды этой команды должны удовлетворять условию 0 < Y << X, иначе необходимо использовать формулы приведения:

arctg(x) = -arctg(-x), arctg(x) = п/2 - arctg(1/x).

Остальные обратные тригонометрические функции находятся с помощью команд FPATAN, FSQRT и таких формул:

arcsin(z) = arctg(z/(sqrt((1-z)*(1+z))) = arctg(Y/X),
Y = z, X = sqrt((1-z)*(1+z));
arccos(z) = 2*arctg(sqrt((1-z)/(1+z)))= 2*arctg(Y/X),
Y = sqrt(1-z), X = sqrt(1+z);
arctg(z) = arctg(Y/X);
arcctg(z) = arctg(1/z) = arctg(X/Y);
arccosec(z) = arctg(sign(z)/sqrt((z-1)*(z+1)))=arctg(Y/X),
Y = sign(z); X = sqrt((z-1)*(z+1));
arcsec(z) = 2*arctg(sqrt((z-1)*(z+1))) = 2*arctg(Y/X),
Y = sqrt((z-1)*(z+1)), X = 1.

В этих формулах z - аргумент вычисляемой функции, X, Y - значения составного аргумента в регистрах st[0], st[1] перед выполнением команды FPATAN, результат которой помещается в вершину стека.

Формулы для вычисления логарифмических функций:

log2 (x) => FYL2X при Y => FLD1, X => FLD x;
ln(x) => FYL2X при Y => FLDLN2, X => FLD x;
lg(x) => FYL2X при Y => FLDLG2, X => FLD x.

Команда FYL2X вычисляет st(1)*log2 (st[0]) при st[0] > 0. Оба операнда извлекаются из стека, а затем результат помещается в стек. Команды FLDL2E, FLDL2T загружают константы со значением двоичных логарифмов числа e и десятки.

Еще одна логарифмическая команда FYL2XP1 вычисляет st[1]*(st[0]+1) при st[0] < 1-1/sqrt(2) и используется для вычисления обратных гиперболических функций. Команда F2XM1 вычисляет 2^st[0]-1, причем должно выполняться условие: 0 < st[0] < 0.5.

Формулы для вычисления обратных гиперболических функций ориентированы на использование команд вычисления логарифмов, квадратного корня и загрузки констант:

arsh(x) = sign(x)*ln(2)*log2 (1+z), где z = │x│+│x│/(1/│x│+sqrt(1+(1/x)^2)));
arch(x) = ln(2)*log2 (1+z), где z = x-1+sqrt((x-1)*(x+1)) и x > 1;
arth(x) = sign(x)*ln(2)log2 (1+z), где z = 2*│x│/(1-│x│) и -1 < x < 1.

Для интервалов вне допустимых значений аргументов используются формулы приведения для обратных величин аргументов: arcth(x) = arth(1/x); arcsh(x) = arsh(1/x); arsch(x) = arch(1/x).

Вычисление гиперболических и показательных функций организуется с использованием команды F2XM1.

sh(x)=[(e^│x│-1)+(e^│x│-1)/e^│x│*sign(x)/2;
ch(x)=0.5*(e^│x│+1/e^│x│);
th(x)=sign(x)-(e^(2*│x│)-1)/(e^(2*│x│)+1);
cth(x) = 1/th(x); csh(x) = 1/sh(x); sch(x) = 1/ch(x).

Формулы для вычисления показательных функций:

2^x = (2^x-1)+1 => F2XM1(x)+1;
e^x = 1+(2^(x*log2(e))-1) => 1+F2XM1(x*FLDL2E);
10^1x = 1+(2^(x*log2(10)-1) => 1+F2XM1(x*FLDL2T);
x^y = 1+(2^(y*log2(x))-1) => 1+F2XM1(FYL2X(y,x)).

Таким образом, для реализации последней функции достаточно выполнить такую последовательность команд:

```
FLD1     ; Загрузка единицы
```

```
      FLD  y   ; Загузка показателя
      FLD  x   ; Загрузка основания
      FYL2X    ; Вычисление логарифма результата
      F2XM1    ; Вычисление показательной функции
      FADD     ; Коррекция
; ---------- Вычисление COS(x) командами 8087 --------
        mov dx,offset dgroup:tit2; Вывод заголовка 2

        push dx
        call _printf
        add sp,2
        fldz; ------ Очистка стека
        fmul
        fmul
        fmul
        fmul
       ;-------- Приведение к стандартному диапазону-------------
        xor bx,bx      ; Сброс флага обратной величины
        fld1
        fld1
        fld1
        fadd           ; Получение в стеке: 2,1
        fld    st
        fadd   st,st[1]; Получение в стеке: 4,2,1
        fldPI          ; Загрузка PI
        ; В стеке PI,4,2,1
        fdivr st[2],st  ; Вычисление PI/2
        fdivr st[1],st  ; Вычисление PI/4
        fxch  st[1]
        ; В стеке:PI/4, PI, PI/2, 1
        ;----- Определение знака COS(X)
        fld st[1]
        fadd st,st[2]   ; вычисление 2*PI
        fld argm        ; загрузка аргумента X
        fadd st,st[1]
        fadd st,st[4]   ; добавление PI/2
        fprem           ; нахождение остатка
        xor bp,bp       ; обнуление флага знака
        fcomp st[3]
        Fj b,BO
        inc bp
   BO:  fldz
        fmul
        fadd
        ;--------------------------------------
        fld argm       ; Загрузка в стек аргумента X
        fabs           ; Взятие модуля аргумента X
        fcom st[1]     ; Сравнение X с PI/4
        Fj   b,B8
        ; В стеке X, PI/4, PI, PI/2, 1
        fxch st[1]
        fmulp st[4],st
        ; В стеке X, PI, PI/2, PI/4
        ;------------- Приведение X в диапазон от 0 до PI
   B5:  fcom st[1]     ; Сравнение X с PI
        Fj    b,B6
        fprem          ; Исключение периода
        jmp B5
   B6:  fxch st[1]
        fldz
        fmul
        fadd
        ; В стеке X, PI/2
        ;------ Приведение аргумента в диапазон от 0 до PI/2
        fcom st[1]    ; Сравнение X с PI/2
        Fj   b,B7
        fsub st,st[1] ; X:=X-PI/2
        xor  bx,1     ; установка в 1 флага обратной величины
        ; Приведение аргумента в диапазон от 0 до PI/4
```

```
B7:  fxch st[1]
     ; В стеке X, PI/2, PI/4,
     fcom st[2]
     Fj   b,B8
     fsubr st,st[1]
     xor bx,1
    ;-------- Вычисление Tg(X) и потом Cos(X)-----------------
B8:  fptan          ; Вычисление Tg(X) = M/G
     and bx,bx      ; Проверка флага обратной величины
     jz  B9
     fxch st[1]
B9:  fdivp st[1],st
     fld st         ; Копирование Tg(X) в вершину стека
     fmul           ; В вершину стека Tg(X)^2
     fld1
     fadd           ; В вершине стека - Tg(X)^2+1
     fld1
     fdiv st,st[1]  ; Вычисление 1/(1 + Tg(X)^2)
     fsqrt
     and bp,bp      ; Проверка флага знака результата
     jz B10
     fchs           ; Изменение знака результата
B10: sub sp,8         ;Загрузка результата в стек процессора
     mov bp,sp
     fstp qword ptr[bp]
     mov bp,ax
     mov dx,offset dgroup:fat ;Загрузка формата
     push dx
     call _printf             ;Печать результата
     add sp,10
```

```
     ;------------ Вычисление 2^X ----------------------------
     fld argm         ;Поставить аргумент в вершину стека сопроц.
     f2xm1            ;Вычисление 2^X-1
     fld1
     fadd
     sub sp,8         ;Вывод 2^x
     mov bp,sp
     fstp  qword ptr[bp]
     mov dx,offset dgroup:fs10
     push dx
     call _printf
     add sp,10
```

```
;------------ Вычисление e^X ----------------------------
     fld argm         ;Поставить аргумент в вершину стека сопроц.
     fLDL2E            ;Вычисление e^X-1
     fmul st[0],st[1]
   f2xm1
   fld1
     fadd
     sub sp,8         ;Вывод e^x
     mov bp,sp
     fstp  qword ptr[bp]
     mov dx,offset dgroup:fs10
     push dx
     call _printf
     add sp,10
```

2. Поняття граматик та їх використання для розв'язування задач.

Грамматикой называется четверка G = (N, T, P, S), где N - конечное множество нетерминальных символов (нетерминалов), T - множество терминалов (не пересекающихся с N), S - символ из N, называемый начальным, P - конечное подмножество множества: $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$, называемое множеством правил. Множество правил P описывает процесс порождения цепочек языка. Элемент $p_i = (\alpha, \beta)$ множества P называется правилом (продукцией) и записывается в виде $\alpha \Rightarrow \beta$. Здесь α и β - цепочки, состоящие из терминалов и нетерминалов. Данная запись может читаться одним из следующих способов:

- цепочка α порождает цепочку β;
- из цепочки α выводится цепочка β.

Таким образом, правило P имеет две части: левую, определяемую, и правую, подставляемую. То есть правило p_i - это двойка (p_{i1}, p_{i2}), где $p_{i1} = (N \cup T)\, N^* (N \cup T)^*$ - цепочка, содержащая хотя бы один нетерминал, $p_{i2} = (N \cup T)^*$ - произвольная, возможно пустая цепочка (ε - цепочка). Если цепочка α содержит p_{i1}, то, в соответствии с правилом p_i, можно образовать новую цепочку β, заменив одно вхождение p_{i1} на p_{i2}. Говорят также, что цепочка β выводится из α в данной грамматике. Для описания абстрактных языков в определениях и примерах будем пользоваться следующими обозначениями:

- терминалы обозначим буквами a, b, c, d или цифрами 0, 1, ..., 9;
- нетерминалы будем обозначать буквами A, B, C, D, S (причем нетерминал S - начальный символ грамматики);
- буквы U, V, ..., Z используем для обозначения отдельных терминалов или нетерминалов;
- через α, β, γ... обозначим цепочки терминалов и нетерминалов;
- u, v, w, x, y, z - цепочки терминалов;
- для обозначения пустой цепочки (не содержащей ни одного символа) будем использовать знак ε;
- знак "→" будет отделять левую часть правила от правой и читаться как "порождает" или "есть по определению". Например, A→cd, читается как "A порождает cd".

Эти обозначения определяют некоторый язык, предназначенный для описания правил построения цепочек, а значит, для описания других языков. Язык, предназначенный для описания другого языка, называется метаязыком. Пример грамматики G1: G1 = ({A, S}, {0, 1}, P, S), где P: 1) S → 0A1; 1) 0A → 00A1; 2) A → ε.

Выводимая цепочка грамматики G, не содержащая нетерминалов, называется терминальной цепочкой, порождаемой грамматикой G. Язык L(G), порождаемый грамматикой G, - это множество терминальных цепочек, порождаемых грамматикой G. Введем отношение $\Rightarrow_G$ непосредственного вывода на множестве $(N \cup T)^*$, которое будем записывать следующим образом: $\varphi \Rightarrow_G \psi$.

Данная запись читается: ψ непосредственно выводима из φ для грамматики G = (N, T, P, S) и означает: если αβγ - цепочка из множества $(N \cup T)^*$ и β → δ - правило из P то $\alpha\beta\gamma \Rightarrow_G \alpha\delta\gamma$.

Через $\Rightarrow_G^+$ обозначим транзитивное замыкание (нетривиальный вывод за один и более шагов). Тогда $\varphi \Rightarrow_G^+ \psi$ читается как: ψ выводима из φ нетривиальным образом.

Через $\Rightarrow_G^*$ - обозначим рефлексивное и транзитивное замыкание (вывод за ноль и более шагов). Тогда $\varphi \Rightarrow_G^* \psi$ означает: ψ выводима из φ. Пусть $\Rightarrow^k$ k-я степень отношения ⇒. То есть, если $\alpha \Rightarrow^k \beta$, то существует последовательность $\alpha_0\alpha_1\alpha_2\alpha_3 \ldots \alpha_k$ из к+1 цепочек $\alpha = \alpha_0, \alpha_1, \ldots \alpha_{i-1} \Rightarrow \alpha_i$, $1 \le i \le k$ и $\alpha_k = \beta$.

Пример выводов для грамматики G1:

$S \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 0011$;

$S \Rightarrow^1 0A1$; $S \Rightarrow^2 00A11$; $S \Rightarrow^3 0011$;

По виду правил выделяют несколько классов грамматик. В соответствии с классификацией Хомского грамматика **G** называется:

- **праволинейной**, если каждое правило из **P** имеет вид: **A→xB** или **A→x**, где **A**, **B** - нетерминалы, **x** - цепочка, состоящая из терминалов;
- **контекстно-свободной** (КС) или **бесконтекстной**, если каждое правило из **P** имеет вид: A→α, где $A \in N$, а $\alpha \in (N \cup T)^*$, то есть является цепочкой, состоящей из множества терминалов и нетерминалов, возможно пустой;
- **контекстно-зависимой** или **неукорачивающей**, если каждое правило из **P** имеет вид: $\alpha \to \beta$, где $|\alpha| \le |\beta|$. То есть, вновь порождаемые цепочки не могут быть короче, чем исходные, а, значит, и пустыми (другие ограничения отсутствуют);
- **грамматикой свободного вида**, если в ней отсутствуют выше упомянутые ограничения.

Пример праволинейной грамматики: G2 = ({S,}, {0,1}, P, S), где P:

1) S → 0S; 2) S → 1S; 3) S → ε, определяет язык {0, 1}*.

Пример КС-грамматики: G3 = ({E, T, F}, {a, +, *, (,)}, P, E) где P:

1) E →T; 2) E → E + T; 3) T → F; 4) T → T * F; 5) F → (E); 6) F → a.

Данная грамматика порождает простейшие арифметические выражения.

Пример КЗ-грамматики: G4 = ({B, C, S}, {a, b, c}, P, S) где P:

1) S → aSBC; 2) S → abc; 3) CB → BC; 4) bB → bb; 5) bC → bc; 6) cC → cc, порождает язык $\{ a^n b^n c^n \}$, $n \ge 1$.

Примечание 1. Согласно определению каждая праволинейная грамматика является контекстно-свободной.

Примечание 2. По определению КЗ-грамматика не допускает правил: A → ε, где ε - пустая цепочка. Т.е. КС-грамматика с пустыми цепочками в правой части правил не является контекстно-зависимой. Наличие пустых цепочек ведет к грамматике без ограничений. Соглашение. Если язык L порождается грамматикой типа G, то L называется языком типа G. Пример: L(G3) - КС язык типа G3. Наиболее широкое применение при разработке трансляторов нашли КС-грамматики и порождаемые ими КС языки.

3. Організація роботи планування задач і процесів.

Супервизор – программа многозадачной ОС, обеспечивающая наилучшее использование ресурсов ЭВМ при одновременном выполнении нескольких задач.

Звичайно задача супервізора запускається вкінці обробки різних переривань, які змінюють стан системи. Переривання можуть свідчити:

−Про звертання до обслуговування супервізора від активної задачі

−Виникнення апаратного переривання, що свідчить по готовність даних інших ресурсів, які потрібні для задач, що очікують ресурси.

Звичайно вкінці обробки потрібних програм відбувається звернення до супервізора, який вибирає наступну активну задачу серед найбільш пріоритетних задач, які використовуються в ОС.

Програми планувальники планують розподіл ресурсів комп.системи. Основним ресурсом комп сист.- вважається процесорний час головних проц. Інші ресурси с-ми - головна або віртуальна пам”ять, зовн пристрої або пристрої обміну даними розпад. За задачами або в монопольному режимі або в режимі сумісного використання. Щоб забезпечити гнучку роботу із зовн. Пристроями для них будують ієрархію віртуальних машин обміну. На нижчих рівнях ієрархії створюють машини обміну безпосередньо з фіз. Пристроями. На більш високих рівнях будуються більш загальні віртуальні пристрої до яких відносять файлові системи, с-ми відобр. Викон. Задачі або сист граф інтерфейсу користувача, як проміжні рівні можна використ пристрої обміну для управління вірт памятю, а також вірт пристрої шифрування для захисту файлу від несанкц доступу. Режими роботу ОС визн насамперед планувальниками задач процесів або потоків. Більшість ранніх ОС була орієнт на пакетну обробку задач, при цьому пакет задач обробл за принципом FIFO.

Для реаліз планув задач треба визначити послідовність станів які проход задача під час свого вик-ня і побудувати табл задач. Основними станами задач є

- створення задач, відокремлення ресурсів, запуск обчисл процесу
- активний стан(обробки задачі ЦП). В цьому стані повинен завантажити контекст задачі, який включає в себе набір всіх регістрів задач
- стан готовності до подальшої роботи в ЦП, але при відсутності проц часу
- очікування результатів
- закінчення задачі

5. Білет

1. Базові команди арифметичних операцій пристрою з плаваючою точкою.

арифметические команды сопроцессора аналогичны оным у микросхемы 8086 – сложение (fadd), вычитание (fsub), умножение (fmul) и деление (fdiv), а также – реверсивные деление (divr) и вычитание (subr). форматы команд следующие (на примере add) – fadd операнд (операнд 1 – верхушка стека, операнд 2 – память), fiadd операнд (операнд 1 – верхушка стека, операнд 2 – целый операнд из памяти), fadd st[i],st (операнд 1 – i-ый регистр стека, операнд 2 – верхушка стека), faddp st[i],st (операнд 1 – i-ый регистр стека, операнд 2 – верхушка стека) – сложение с удалением верхушки стека.

Fdiv**rp**/fsub**rp** st[n],st[0]

```
fld arg2

fadd arg2       ; st[0] = st[1] + arg2

fadd            ; st[0] = st[0] + st[1]

fadd st[1],st   ; st[1] -> st[0] = st[1] + st[0]
```

Приклад:

FLD ARG1 ; ST[0] ← ARG1

FLD ARG2 ; ST[1] ←ST[0]; ST[0] ←ARG2

FADD ARG2 ; ST[0]:= ST[0] + ARG2 = 2ARG2

FADD ; ST[0]:= ST[0] + ST[1] = 2ARG2 + ARG1

FADD ST[1],ST ; ST[1]:= ST[1]+ST[0] = ARG1 + 2 ARG2 + ARG1 = 2ARG1 + 2ARG2

Для організації умовних переходів, при розрахунку з плаваючою точкою, можна використовувати тільки прапорці базових операцій:

Приклад (процедура обчислення експоненти)

```
Exp Proc
        Fld1 – перший член ряду
        Fld x -  другий член ряду
        Fld 1
        Fadd st[2], st[0]
        Ьщи тб2
      L0: fmul st[0],st[1]
        Fidiv n
        Inc n
        Cmp n,10
        Je L0
        Fst y – звільнення стеку
        Fst y
        Fst y
ret
```

2. Граматики, що використовуються для синтаксичного аналізу

Синтаксический разбор (распознавание) является первым этапом синтаксического анализа. Именно при его выполнении осуществляется подтверждение того, что входная цепочка символов является программой, а отдельные подцепочки составляют синтаксически правильные программные объекты. Вслед за распознаванием отдельных подцепочек осуществляется анализ их семантической корректности на основе накопленной информации. Затем проводится добавление новых объектов в объектную модель программы или в промежуточное представление.

Разбор предназначен для доказательства того, что анализируемая входная цепочка, записанная на входной ленте, принадлежит или не принадлежит множеству цепочек порождаемых грамматикой данного языка. Выполнение синтаксического разбора осуществляется распознавателями, являющимися автоматами.

Поэтому данный процесс также называется *распознаванием входной цепочки*. Цель доказательства в том, чтобы ответить на вопрос: принадлежит ли анализируемая цепочка множеству правильных цепочек заданного языка. Ответ "да" дается, если такая принадлежность установлена. В противном случае дается ответ "нет". Получение ответа "нет" связано с понятиям отказа. *Единственный отказ на любом уровне ведет к общему отказу*. Чтобы получить ответ "да" относительно всей цепочки, надо его получить для каждого правила, обеспечивающего разбор отдельной подцепочки. *Так как множество правил образуют иерархическую структуру, возможно с рекурсиями, то процесс получения общего положительного ответа можно интерпретировать как сбор по определенному принципу ответов для листьев, лежащих в основе дерева разбора, что дает положительный ответ для узла, содержащего эти листья*. Далее анализируются обработанные узлы, и уже в них полученные ответы складываются в общий ответ нового узла. И так далее до самой вершины.

На этапе синтаксического анализа нужно установить, имеет ли цепочка лексем структуру, заданную синтаксисом языка, и зафиксировать эту структуру. Следовательно, снова надо решать задачу разбора: дана цепочка лексем, и надо определить, выводима ли она в грамматике, определяющей синтаксис языка. Однако структура таких конструкций как выражение, описание, оператор и т.п., более сложная, чем структура идентификаторов и чисел. Поэтому *для описания синтаксиса языков программирования нужны более мощные грамматики, чем регулярные*. Обычно для этого используют укорачивающие контекстно-свободные грамматики (УКС-грамматики), правила которых имеют вид $A \to \alpha$, где $A \in VN$, $\alpha \in (VT \cup VN)^*$. Грамматики этого класса, с одной стороны, позволяют достаточно полно описать синтаксическую структуру реальных языков программирования; с другой стороны, для разных подклассов УКС-грамматик существуют достаточно эффективные алгоритмы разбора.

С теоретической точки зрения существует алгоритм, который по любой данной КС-грамматике и данной цепочке выясняет, принадлежит ли цепочка языку, порождаемому этой грамматикой. Но время работы такого алгоритма (синтаксического анализа с возвратами) экспоненциально зависит от длины цепочки, что с практической точки зрения совершенно неприемлемо.

Существуют табличные методы анализа ([3]), применимые ко всему классу КС-грамматик и требующие для разбора цепочек длины n времени cn^3 (алгоритм Кока-Янгера-Касами) либо cn^2 (алгоритм Эрли). Их разумно применять только в том случае, если для интересующего нас языка не существует грамматики, по которой можно построить анализатор с линейной временной зависимостью.

3. Організація захисту пам’яті в ОС

Для гарантированного разделения данных и кодов необходимо использовать гораздо больше информации о сегменте, чем это возможно в реальном режиме. В защищенном режиме загружается так называемый селектор, в котором хранится указатель на *восьмибайтный блок памяти*, называемый *дескриптором*, в котором хранится вся нужная информация о сегменте. При выполнении команд загрузки сегментных регистров содержимое дескриптора для получения нормальной скорости работы процессора переносится в теневой регистр, связанный с сегментным, но непосредственно недоступный программисту .*Эти блоки хранятся в сегментах памяти,* называемых *таблицами дескрипторов*: глобальной – GDT, локальной – LDT, определяемой для каждой задачи и прерываний – IDT, замещающей таблицу векторов прерываний реального режима. Селектор содержит относительный адрес в таблице бит типа таблицы: bl = 0 – для LDT, bl = 1 – для GDT, а также двухбитовое поле запрашиваемого уровня привилегий для контроля правильности доступа в механизме защиты. Соотношение запрашиваемого и предоставляемого уровня привилегий определяют 4 кольца защиты : 00 – кольцо ядра ОС, 01 – кольцо обслуживания аппаратуры, 10 – кольцо системы программирования базами данных и расширениями ОС, 11 – кольцо прикладных программ пользователя. Поля дескриптора используются со следующим назначением:

- Базовый адрес сегмента определяет начальный линейный адрес и занимает байты 2,3,4 и 7 дескриптора.
- 20-битовое поле предела определяет границу сегмента и занимает байты 0 и1, а также младшие 4 бита байта 6 дескриптора. Поле предела позволяет аппаратно контролировать выход используемого адреса за пределы сегмента.

В 5 дескриптора закодированы права доступа к сегменту. Бит присутствия p дает возможность управления виртуальной памятью: p=1, когда описаный сегмент присутствует в физической памяти, p=0, когда он перемещен на диск.

6. Білет

1. Перевірка умов за результатами пристрою з плаваючою точкою

Чтобы использовать более сложные условия завершения цикла необходимо анализировать признаки результата, которые формируются в блоке результата. Большинство таких команд не меняет признак результата.

FCOM источник – сравнить вещественные числа

FCOMP источник – сравнить и вытолкнуть из стека

FCOMPP источник – сравнить и вытолкнуть из стека два числа

Команды выполняют сравнение содержимого регистра ST(0) с источником (32- или 64-битная переменная или регистр ST(n), если операнд не указан — ST(1)) и устанавливают флаги C0, C2 и C3 в соответствии с таблицей 14.

Таблица 14. *Флаги сравнения FPU*

Условие	**C3**	**C2**	**C0**
ST(0) > источник	0	0	0
ST(0) < источник	0	0	1
ST(0) = источник	1	0	0
Не сравнимы	1	1	1

Если один из операндов - не-число или неподдерживаемое число, происходит исключение «недопустимая операция», а если оно замаскировано (флаг IM = 1), все три флага устанавливаются в 1. После команд сравнения с помощью команд FSTSW и SAHF можно перевести флаги C3, C2 и C0 в соответственно ZF, PF и CF, после чего все условные команды (Jcc, CMOVcc, FCMOVcc, SETcc) могут использовать результат сравнения, как после команды CMP

FTST **–** проверить, не содержит ли SP(0) ноль

Організація умовних переходів с учетом осбенностей структуры регистра F центрального процессора и регистра состояния математического сопроцессора, а также наличием команды сопроцессора:

```
FSTSW wrd  ; Запись в память регистра состояния сопроцессора
```

и команды центрального процесора:

```
SAHF       ; Сохранение содержимого регистра ah в регистре F.
```

Условные переходы по результатам сравнений в сопроцессоре можно организовать макроопределением следующего вида:

```
fj   macro  cd,lb   ; Прототип макровызова.
```

```
fstsw  stsw    ; Сохранение регистра состояния
fwait          ; Ожидание окончания пересылки
mov    ah,byte ptr stsw+1 ; Копирование регистра состояния
sahf   ; Пересылка старшего байта регистра состояния в F.
j&cd   lb      ; Условный переход
endm
```

Для программирования условного перехода по результату сравнения программисту достаточно использовать макровызов вида:

[Метка:] fj Условие перехода, Метка перехода

Первый операнд макрокоманды, определяет условие перехода теми же буквами, которые используются в командах условных переходов по результатам беззнаковой арифметики, то есть a - больше, b - меньше, n - отрицание и e - равенство. Команда FXAM позволяет получить гораздо больше информации о содержимом st[0], но дает особое кодирование битов признака результатов, и может использоваться также для инициализации кодов условия. Для забезпечення переходу по умові краще використовувати команди JA, JB. Інколи необхідно дочекатися результатів перевірки – це можна зробити командами FWAIT, WAIT.

Приклад:

```
L:      FCOM ST[0], Y
        FSTSW AX
        SAHF
        JNE L
```

При выполнении команд FPU могут возникать шесть типов особых ситуаций, называемых исключениями. При возникновении исключения соответствующий флаг в регистре SR устанавливается в 1 и, если маска этого исключения в регистре CR не установлена, вызывается обычное прерывание INT 10h (если бит NE в регистре центрального процессора CR0 установлен в 1) или IRQ13 (INT 75h), обработчик которого может прочитать регистр SR, чтобы определить тип исключения и команду, которая его вызвала, а затем попытаться исправить ситуацию. Если бит маски наступившего исключения в регистре CR установлен в 1, выполняются следующие действия по умолчанию:

неточный результат: результат округляется в соответствии с битами RC (на самом деле это исключение происходит очень часто. Например: дробь 1/6 не может быть представлена точно десятичным вещественным числом любой точности и округляется). При этом флаг C1 показывает, в какую сторону произошло округление: 0 — вниз, 1 — вверх; ***антипереполнение***: результат слишком мал, чтобы быть представленным обычным числом, — он преобразуется в денормализованное число;

переполнение: результат преобразуется в бесконечность соответствующего знака;

деление на ноль: результат преобразуется в бесконечность соответствующего знака (учитывается и знак нуля);

денормализованный операнд: вычисление продолжается, как обычно;

недействительная операція

2. Граматики, що використовуються для лексичного аналізу

Під регулярною граматикою будемо розуміти якусь ліволінійну граматику. Це така граматика, що її P правила мають вигляд A → Bt або A → t, де A є N, B є N, t є T. Для прикладу будемо розуміти якийсь символ «@» кінцем ланцюжка, що аналізується лексичним аналізатором. Для граматик такого типу передбачений свій алгоритм розбору для отримання лексеми:

- Перший символ вихідного ланцюжка замінюємо не терміналом A, для якого в граматиці є правило A → a
- Далі проводимо ітерації до кінця ланцюжка за наступною схемою: отриманий раніше не термінал A і розташований правіше від нього термінал a вихідного ланцюжка замінюємо не терміналом B, для якого в граматиці є правило B → Aa

Стан автомату повинен відповідати типу розпізнаної лексеми або типу помилки лексичного аналізу. Вхідними сигналами автомату повинні бути літери вхідної послідовності, а точніше класифікаційні ознаки цієї літери. Таким чином для побудови лексичного аналізатора доцільно мати класифікаційну таблицю літер мови, що обробляє лексичний аналізатор,- двомірну матрицю переходів, що визначає код наступного стану автомату, ондим з яких є поточний стан автомату а другим – класифікатор вхідної літери. Тобто, щоб визначити такий автомат необхідно визначити перенумеровані типи даних для кодів стану і кодів специфікаторів. Власне програма повинна виділяти чергову лексему шляхом циклічного прогряду літер. Цикл закінчується в тому випадку, коли знайдена помилка або лексема закінчується роздільником. Для того, щоб використовувати лексичний аналіз методом теорії автоматів або автоматним методом необхідно використовувати таблицю класифікаторів, складних роздільників та таблицю ключових слів. Результати роботи лексичного аналізу треба рознести по таблицях імен, по таблицях констант і, можливо, по таблицях модулів. Результати лексичного аналізу доцільно розміщувати в спеціалізованих структурах, в яких зберігається інформація про код вхідної лексеми та її внутрішнє подання.

3. Ієрархічна організація програм введення-виведення

Початковим поштовхом до розробки ОС були проблеми автоматизації завантаження програм та використання узагальнених механізмів введення-виведення. На початковому етапі розроблення ОС найбільш коштовною частиною був ЦП, тому головною вважалася задача ефективного використання процесору Для цього основною проблемою була організація ефективного введення-виведення з одночасною роботою ЦП над розв’язанням інших задач. Для цього необхідно механізм апаратного переривання, який замінював цикл ЦП з очікуванням готовності даних.

Структура підпрограми драйверів пристроїв включала наступні блоки:

- Видача команди на пристрій для його підготовки до обміну
- Очікування готовності пристрою до обміну
- Виконання власне обміну
- Видача на пристрій команди для закінчення операції
- Організація обміну драйвера даними з програмою, яка його використовує.

При створенні мікропроцесорів фактично було повторено процес створення програм обміну для зовнішніх пристроїв, але з деякою систематизацією.

Підключення зовнішніх пристроїв до мікропроцесору виконувалось шляхом визначення вихідного командного порту, вхідного порту стану, які мали однакові номер та вхідного і вихідного порту для введення і виведення даних, які мали однакову адресу. Щоб написати узагальнений драйвер введення-виведення треба визначити адреси портів за допомогою

```
CMPRT EQU    41H

STPRT  EQU    CMPRT

INPRT  EQU    42H

OUTPRT        EQU    INPRT
```

Возвращает в ax 1 байт данных введенных с некоторого устройства

```
drIn    Proc
        mov     al,cmOn
        out     CMPRT,al
lwr:    in      al,STPRT
        test    al,RdyBit
        jnz     lwr
        in      al,INPRT
        push    ax
        mov     al,cmOff
        out     CMPRT
        pop     ax
        ret
drIn    endp
```

Білет 7

1. **Особливості архітектури розширених MMX.**

Основа аппаратной компоненты **расширения mmx** – *восемь* новых *регистров*, которые на самом деле являются регистрами *сопроцессора*, только вместо 80-ти разрядов *используется 64 младших разряда* (мантисса). при работе со стеком сопроцессора в режиме mmx он рассматривается как обычный массив регистров с произвольным доступом. использовать стек сопроцессора по его прямому назначению и как регистры mmx-расширения одновременно невозможно. Основным принципом работы команд mmx является одновременная обработка нескольких единиц однотипных данных одной командой.

MMX-расширение предназначено для поддержки приложений, ориентированных на *работу с большими массивами данных целого и вещественного типов, над которыми выполняются одинаковые операции*. С данными такого типа обычно работают мультимедийные, графические, коммуникационные программы – от этого и название MultiMedia eXtensions. Важное отличие MMX-команд от обычных команд процессора в том, как они реагируют на ситуации переполнения и заема. Возникают ситуации, когда результат арифм. операции выходит за размер разрядной сетки исходных операндов. В этом случае производится усечение старших бит результата и возвращаются только те биты, которые умещаются в пределах исходного операнда (арифметика с циклическим переносом). Некоторые MMX-команды в такой ситуации действуют иначе. В случае выхода значения результата за пределы операнда, в нем фиксируется максимальное или минимальное значение (арифметика с насыщением). MMX_расширение имеет команды, которые выполняют арифметические операции с использование обоих принципов. При этом среди них есть команды, учитывающие знаки элементов операндов. ПРИМЕР (см. рис. слева).

include mmx16.inc

.data

mem dw 4444h

df 111122223333h

.code

movd rmmx0, mem ; rmmx0=0000 0000 3333 4444

movq rmmx0, mem ; rmmx0=1111 2222 3333 4444

(2-источник)

Розширення архітектури MMX

основа аппаратной компоненты расширения MMX – восемь новых регистров mm0..mm7, которые на самом деле являются регистрами сопроцессора, только вместо 80-ти разрядов используется 64 младших разряда (мантисса). При работе со стеком сопроцессора в режиме mmx он рассматривается как обычный массив регистров с произвольным доступом. Нельзя одновременно пользоваться командами для работы с числами с плавающей запятой и командами MMX, а если это необходимо — следует пользоваться командами FSAVE/FRSTOR,каждый раз перед переходом от использования FPU к MMX и обратно (эти команды сохраняют состояние регистров MMX точно так же, как и FPU). Основным принципом работы команд mmx является одновременна обработка нескольких единиц однотипных данных одной командой.

MOVQ MMREG1, A

MOVQ MMREG2, B

PADDB MMREG2, MMREG1

MOVQ B, MMREQ2

2. **Машинно-залежна оптимізація.**

Маш.-завис-я оптимизация заключ. в эффективном исп-ии ресурсов целевого компьютера (процесса), т.е надо эф-но исп-ть след-е ресурсы:

- регистры спец. и общего назначения(с плав-ей запятой, MMX)
- память сверх оперативного обращения к данным, кот-е в главной памяти
- главная память
- память накопителя

Надо эф-но исп-ть систему команд отдавая предпочтение командам которые быстрее организовать поиск. Для оптимизации компилятора здесь происходит разделение RG или общего или регистра спец-ого назначения. Простое распределение-регистра с плавающей запятой. При обработке выражений промежуточные значения сохраняем в стеке рег-ра с плавающей запятой. При переполнении стека возникает прерывание, где сохраняем состояние рег-ра сопроцессора в главном стеке задачи и продолжать работу с оновленым стеком. Такая организация накладывает ограничения. *Для эф-ой генерации команд надо построить таблицу,которая покроет все операции языка программирования*. Для оптимизации м.б.лучше построить неоднозначную таблицу с несколькими вариантами комбинаций операций, где *ключевая часть таблицы - коды операций и коды типов данных в внутренней форме*. Как функцтонал-я часть, в таблицу заносятся коды машинных или ассемблеровских команд, которые генерируются на выходе компилятора. *Генератор кодов должен формировать маш-й команды с неявной информацией про операции и данные*. Генератор кодов формирует команды с мелких фрагментов ком-д , операций , имен или адресов данных,

индексных и базовых регистров. Откомпелированый код с пом-ю компилятора объединяется с стандартными функциями в библиотеки. *Сейчас исп-ют **формальную верификацию** на этапе **семант-ой обработки***:

- формальное доказательство соответствия программы или модели в ранее заданной спецификации программы

нахождение в программе фрагментов , что соответствуют корректным или ошибочным примерам.

3. Необхідність синхронізації даних в задачах ввод-вивід

Общая схема процедуры обмена (ввода или вывода) одним физическим элементом драйвера ввода-вывода включает такую последовательность действий.

- *Выдача подготовительной команды*, включающих исполнительные механизмы или электронные устройства.
- Проверка готовности устройства к обмену.
- Собственно обмен: ввод или вывод данных взависимости от типа устройства и нужной функции.
- Сохранение введенных данных и подготовка информации о завершении ввода-вывода.
- Выдача заключительной команды, освобождающей устройство для возможного использования в других задачах.
- Выход из драйвера.

```
Drin PROC

MOV AL, cmOn ; загрузка управляющего кода включения устройства.

OUT cmPtr, AL ; Пересылка кода включения в порт управления

l:IN AL, stPrt ; ввод содержимого порта состояний

TEST AL,avMask ; контроль по маске байтов аварийного состояния

JNZ lErr ; на обработку аварийного состояния устройства

TEST AL,rdyIn ; контроль готовности данных для ввода

JZ l ; на начало цикла ожидания готовности
```

Эту последовательность действий для драйвера ввода устройства, содержащего команду, порт управляющего действиями устройства, порт состояния, контролирующей работоспособности и получение данных с устройства, и порт данных для обмена данными, можно записать для однобайтного канала обмена таким образом (См. рис.).

Номера управляющих портов (stPrt и cmPtr) и порта данных (dtPrt), а также коды команд внешнего устройства (cmOnn и cmOff) и маски контроля разрядов порта состояний (avMask и rdyIn) должны быть определены в начале программы. Если номера портов имеют значения больше 0ffh, то команды IN и OUT следует заменить парами операций:

```
MOV DX,cmPrt ; подготовка адреса порта

mov DX,AX ; выдача команды на порт управления.
```

(2-источник)

Організація системи ввода-виводу.

Початковим поштовхом до розробки ОС були проблеми автоматизації завантаження програм та використання узагальнених механізмів введення-виведення. На початковому етапі розроблення ОС найбільш коштовною частиною був ЦП, тому головною вважалася задача ефективного використання процесору Для цього основною проблемою була організація ефективного введення-виведення з одночасною роботою ЦП над розв'язанням інших задач. Для цього необхідно механізм апаратного переривання, який замінював цикл ЦП з очікуванням готовності даних.

Структура підпрограми драйверів пристроїв включала наступні блоки:

- Видача команди на пристрій для його підготовки до обміну
- Очікування готовності пристрою до обміну
- Виконання власне обміну
- Видача на пристрій команди для закінчення операції
- Організація обміну драйвера даними з програмою, яка його використовує.

При створенні мікропроцесорів фактично було повторено процес створення програм обміну для зовнішніх пристроїв, але з деякою систематизацією.

Підключення зовнішніх пристроїв до мікропроцесору виконувалось шляхом визначення вихідного командного порту, вхідного порту стану, які мали однакові номер та вхідного і вихідного порту для введення і виведення даних, які мали однакову адресу. Щоб написати узагальнений драйвер введення-виведення треба визначити адреси портів за допомогою

```
        CMPRT EQU     41H
        STPRT   EQU     CMPRT
        INPRT   EQU     42H
        OUTPRT          EQU     INPRT
```

Возвращает в ax 1 байт данных введенных с некоторого устройства

```
drIn    Proc
        mov     al,cmOn
        out     CMPRT,al
lwr:    in      al,STPRT
        test    al,RdyBit
        jnz     lwr
        in      al,INPRT
        push    ax
        mov     al,cmOff
        out     CMPRT
        pop     ax
        ret
drIn    endp
```

Білет №8

1. ***Команди для обчислення часткових математичних функцій.***

Процессор 8087 может вычислить любую элементарную трансцендентную функцию с аргументом, точность представления которого определяется содержимым регистра управления, и последующей выдачей результата такой же точности. В названиях команд и их описаниях принято обозначать аргументы, содержащиеся в st[0] как X, а в st[1] - как Y. Результаты, формирующиеся в st[0], будем обозначать как x, а в st[1] - как y.
Используются такие команды: FPTAN – частичный тангенс (st/x = tg(X), 0<= X <=1 pi/4); FPATAN – частичный арктангенс (++st, arctg(Y/X), Y > X >0); FYL2X - Вычисление логарифма (++st, Y*log2(X), X > 0); FYL2XP1 - Вычисление логарифма (++st, Y*log2(X+1), abs(X) < 1-sqrt(2)/2); F2XM1 - Вычисление (X=2^X-1). Вычисление тригонометрических функций основано на выполнении команды FPTAN - нахождения частичного тангенса, которая в качестве результата дает два таких числа x и y, что y/x = tg(X). Число y заменяет старое содержимое st[0], а число включается в стек до полнительно. Диапазон изменения аргумента можно свести к допустимому командой FPREM или проверить с помощью команды FXAM, так как он должен быть нормализован и находиться в диапазоне 0 < st[0] < п/4. Если аргумент Хлежит вне этого диапазона, то в начале программы необходимо выполнить преобразования по сведению аргумента к требуемому диапазону и запомнить данные для обратного пребразования. Для функции tg(X) можно использовать 4 преобразования:

X< 0; S = sign(X); = abs(X); tg(X) = S*tg(Y); (1)

в этом перечне сначала указано условие сведения для отрицательныхаргументов, затем две формулы для получения сведенных аргументов и, наконец, формула для восстановления итогового результата.

Y> п; Z= Y - n*п; tg(Y) = tg(Z); (2)

Здесь устраняется многократное повторение периода и дополнительных действий по восстановлению результата не требуется. Следующие два преобразования могут быть проведены в безусловной форме, так как модифицированный аргумент лежит в диапазоне 0 <= Y <= п.

U= Z/2; tg(Z) = 2*tg(U)/(1-tg(U)*tg(U)); (3)

V= U/2; tg(U) = 2*tg(V)/(1-tg(V)*tg(V)); (4)

Эти формулы, основанные на двухкратном применении известной в математике универсальной тригонометрической подстановки через тангенс половинного угла, легко реализуются, так как деление на 2 быстро осуществляется командой FSCALE, и могут быть преобразованы с учетом того, что результат вычисления функции FPTAN(V) сформирован в виде чисел и . Тогда

tg(V) = 2*x*y/(x*x-y*y) и u = 2*x*y; v=x*x-y*y, (5)

где u/v = tg(V) Эти формулы можно рассматривать как базовые для расчета тангенса и других тригонометрических функций с помощью команд FPTAN и FPREM, используя таблицу формул приведения и следующие формулы, выраженные через и .

sin(Y) = 2*(u/v)/(1+(u/v)^2);

cos(Y) = (1-(u/v)^2) / (1+(u/v)^2);

cosec(Y) = (1+(u/v)^2) / 2(u/v);

sec(Y) = (1+(u/v)^2) /(1-(u/v)^2).

Кроме рассмотренного способа тригонометрические функции могут вычисляться либо через тангенс половинного угла по формуле (3), либо через тангенс полного угла по формулам:

sin(x) = tg(x) / sqrt(1+(tg(x)^2);

cos(x) = 1 / sqrt(1+(tg(x)^2);

ctg(x) = 1 / tg(x).

ПРИМЕР:

```
_tg PROC

PUSH BP ; Стандартное сохранение базового указателя стека.
MOV BP,SP ; Установка нового значения базового указателя. FLDPI ; Загрузка числа п

FLD QWORD PTR[BP+4] ; Загрузка начального значения суммы . FTST

FSTSW stsw

PUSH  stsw

rm: FPREM; Исключение периода

fj p,rm ; Циклическое исключение остатка

FLD1

FCHS

FADD   st,st ; Формирование - 2

FXCH   st[1]

      FSCALE ; Деление на 4

      FPTAN ;Вычисление составляющих tg

      FLD    st[1] ; Дублирование числителя

      FMUL   st,st ; Квадрат числителя *y

      FXCH   st[1] ; Обмен на знаменатель

      FMUL   st[2],st ; Вычисление *y

      FMUL   st,st ; Вычисление *x

      FSUBP  st[1],st ; Получение -v

      FMUL   st,st[2] ; Получение -u

      FLD    st ; Дублирование числителя u

      FMUL   st,st ; Квадрат числителя *u

      FXCH   st[2] ; Обмен на числитель

      FMUL   st[1],st ; Вычисление *v

      FMUL   st,st ; Вычисление *v

      FSUBP  st[2],st ; Получение - знаменателя tg

      FMULP  st[2],st ; Получение - числителя tg

; Для этой команды нужно предусмотреть защиту от особых ситуаций.

          FDIVP st[1],st ; Получение значения tg
```

FSTP result ; Сохранение результата

MOV AX,offset DGROUP:result

POP BP ; Стандартное восстановление базового указателя стека. RET ; Выход из подпрограммы.

_tg ENDP

2::: *Грамматики классифицируются по виду их правил вывода (по Хомскому)*

ТИП 0: Грамматика G = (VT, VN, P, S) называется *грамматикой типа 0*, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики).

ТИП 1: Грамматика G = (VT, VN, P, S) называется *неукорачивающей грамматикой*, если каждое правило из P имеет вид □□□, где □□ (VT □ VN)$^+$, □□ (VT □ VN)$^+$ и | □ | ≤ | □ | ≈ *контекстно-зависимой (КЗ)*, если каждое правило из P имеет вид □□□, где □ = ξ_1Aξ_2; □ = ξ_1□ξ_2; A □ VN; □□ (VT □ VN)$^+$; ξ_1,ξ_2 □ (VT □ VN)*.

Множество языков, порождаемых неукорачивающими грамматиками, *совпадает с множеством языков*, порождаемых *КЗ*-грамматиками.

ТИП 2: Грамматика G = (VT, VN, P, S) называется *контекстно-свободной (КС)*, если каждое правило из P имеет вид A □ □, где A □ VN, □ □ (VT □ VN)$^+$ ≈ *укорачивающей контекстно-свободной (УКС)*, если каждое правило из P имеет вид A □ □, где A □ VN, □ □ (VT □ VN)*.

Для каждой УКС-грамматики существует почти эквивалентная КС-грамматика.

ТИП 3: Грамматика G = (VT, VN, P, S) называется *праволинейной*, если каждое правило из P имеет вид A □ tB либо A □ t, где A □ VN, B □ VN, t □ VT.

Соотношения между типами грамматик:

(1) любая регулярная грамматика является КС-грамматикой;

(2) любая регулярная грамматика является УКС-грамматикой;

(3) любая КС-грамматика является КЗ-грамматикой;

??? [(4) любая КС-грамматика является неукорачивающей грамматикой;

Грамматика G = (VT, VN, P, S) называется *леволинейной*, если каждое правило из P имеет вид A □ Bt либо A □ t, где A □ VN, B □ VN, t □ VT.

Грамматику типа 3 (регулярную, P-грамматику) можно определить как праволинейную либо как леволинейную.

Множество языков, порождаемых праволинейными грамматиками, совпадает с множеством языков, порождаемых леволинейными грамматиками

3) Збереження стану задач в реальному режимі

В реальном режиме имеются программные и аппаратные прерывания. ПП инициируются командой INT, АП - внешними событиями, по отношению к выполняемой программе. Обычно АП инициируются аппаратурой ввода/вывода после завершения выполнения текущей операции.

Кроме того, некоторые прерывания зарезервированы для использования самим процессором - прерывания по ошибке деления, прерывания для пошаговой работы, немаскируемое прерывание и т.д.

Для обработки прерываний в реальном режиме процессор использует Таблицу Векторов Прерываний. Эта таблица располагается в самом начале оперативной памяти, т.е. её физический адрес - 00000. ТВП

реального режима состоит из 256 элементов по 4 байта, таким образом её размер составляет 1 килобайт. Элементы таблицы - дальние указатели на процедуры обработки прерываний. Указатели состоят из 16-битового сегментного адреса процедуры обработки прерывания и 16-битового смещения. Причём смещение хранится по младшему адресу, а сегментный адрес - по старшему.

Когда происходит программное или аппаратное прерывание, текущее содержимое регистров CS, IP а также регистра флагов FLAGS записывается в стек программы (который, в свою очередь, адресуется регистровой парой SS:SP). Далее из таблицы векторов прерываний выбираются новые значения для CS и IP, при этом управление передаётся на процедуру обработки прерывания.

Перед входом в процедуру обработки прерывания принудительно сбрасываются флажки трассировки TF и разрешения прерываний IF. Поэтому если процедура прерывания сама должна быть прерываемой, то необходимо разрешить прерывания командой STI. В противном случае, до завершения процедуры обработки прерывания все прерывания будут запрещены.

Завершив обработку прерывания, процедура должна выдать команду IRET, по которой из стека будут извлечены значения для CS, IP, FLAGS и загружены в соответствующие регистры. Далее выполнение прерванной программы будет продолжено.

Что же касается аппаратных маскируемых прерываний, то в компьютере IBM AT и совместимых с ним существует всего шестнадцать таких прерываний, обозначаемых IRQ0-IRQ15. В реальном режиме для обработки прерываний IRQ0-IRQ7 используются вектора прерываний от 08h до 0Fh, а для IRQ8-IRQ15 - от 70h до 77h.

Сохранения состояния в реальном режиме

По идее (поскольку все организовано на прерываниях) сохраняются только регист указателя инструкции (на x86 это ip) и дальше уже обработчик прерываний сохраняет все регистрый которые собирается использовать

Білет №9

1) Особливості арифметичних операцій розширення MMX

Архітектура розширення MMX.

Основа аппаратной компоненты расширения mmx – восемь новых регистров, которые на самом деле являются регистрами сопроцессора, только вместо 80-ти разрядов используется 64 младших разряда (мантисса). при работе со стеком сопроцессора в режиме mmx он рассматривается как обычный массив регистров с произвольным доступом. использовать стек сопроцессора по его прямому назначению и как регистры mmx-расширения одновременно невозможно. Основным принципом работы команд mmx является одновременная обработка нескольких единиц однотипных данных одной командой.

Особливості операцій розширення MMX.

MMX-расширение предназначено для поддержки приложений, ориентированных на работу с большими массивами данных целого и вещественного типов, над которыми выполняются одинаковые операции. С данными такого типа обычно работают мультимедийные, графические, коммуникационные программы – от этого и название MultiMedia eXtensions. Важное отличие MMX-команд от обычных команд процессора в том, как они реагируют на ситуации переполнения и заема. Возникают ситуации, когда результат арифм. операции выходит за размер разрядной сетки исходных операндов. В этом случае производится усечение старших бит результата и возвращаются только те биты, которые умещаются в пределах исходного операнда (арифметика с циклическим переносом). Некоторые MMX-команды в такой ситуации действуют иначе. В случае выхода значения результата за пределы операнда, в нем фиксируется максимальное или минимальное значение (арифметика с насыщением).

MMX_расширение имеет команды, которые выполняют арифметические операции с использование обоих принципов. При этом среди них есть команды, учитывающие знаки элементов операндов. ПРИМЕР:

include mmx16.inc

.data

mem dw 4444h

df 111122223333h

.code

movd rmmx0, mem ; rmmx0=0000 0000 3333 4444

movq rmmx0, mem ; rmmx0=1111 2222 3333 4444

Особливості команд арифметичних операцій

PADDB приемник,источник – сложение байт
PADDW приемник,источник – сложение слов
PADDD приемник,источник – сложение двойных слов

Команды выполняют сложение отдельных элементов данных (байт — для PADDB, слов — для PADDW, двойных слов — для PADDD) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при сложении возникает перенос, он не влияет ни на следующие элементы, ни на флаг переноса, а просто игнорируется (так что, например, для PADDB 255 + 1 = 0, если это числа без знака, или -128 + -1 = +127, если со знаком).

PADDSB приемник,источник – сложение байт с насыщением
PADDSW приемник,источник – сложение слов с насыщением

Команды выполняют сложение отдельных элементов данных (байт — для PADDSB и слов — для PADDSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если результат сложения выходит за пределы байта со знаком для PADDSB (больше +127 или меньше -128) или слова со знаком для PADDSW (больше +32 767 или меньше -32 768), в качестве результата используется соответствующее максимальное или минимальное число, так что, например, для PADDSB -128 + -1 = -128.

PADDUSB приемник,источник – беззнаковое сложение байт с насыщением

PADDUSW приемник,источник – беззнаковое сложение слов с насыщением

Команды выполняют сложение отдельных элементов данных (байт — для PADDUSB и слов — для PADDUSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если результат сложения выходит за пределы байта без знака для PADDUSB (больше 255 или меньше 0) или слова без знака для PADDUSW (больше 65 535 или меньше 0), в качестве результата используется соответствующее максимальное или минимальное число, так что, например, для PADDUSB 255 + 1 = 255.

PSUBB приемник,источник – вычитанние байт

PSUBW приемник,источник – вычитанние слов
PSUBD приемник,источник – вычитанние двойных слов

Команды выполняют вычитание отдельных элементов данных (байт — для PSUBB, слов — для PSUBW, двойных слов — для PSUBD) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при вычитании возникает заем, он игнорируется (так что, например, для PSUBB -128 - 1 = +127 — для чисел со знаком или 0 - 1 = 255 — для чисел без знака).

PSUBSB приемник,источник – вычитание байт с насыщением
PSUBSW приемник,источник – вычитание слов с насыщением

Команды выполняют вычитание отдельных элементов даннях (байт — для PSUBSB и слов — для PSUBSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если результат вычитания выходит за пределы байта или слова со знаком, в качестве результата используется соответствующее максимальное или минимальное число, так что, например, для PSUBSB -128 - 1 = -128.

PSUBUSB приемник,источник – беззнаковое вычитание байт с насыщением
PSUBUSW приемник,источник – беззнаковое вычитание слов с насыщением

Команды выполняют вычитание отдельных элементов даннях (байт — для PSUBUSB и слов — для PSUBUSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если результат вычитания выходит за пределы байта или слова без знака, в качестве результата используется соответствующее максимальное или минимальное число, так что, например, для PSUBUSB 0 - 1 = 0.

PMULHW приемник,источник – старшее умножение

Команда умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Старшее слово каждого из результатов записывается в соответствующую позицию приемника.

PMULLW приемник,источник – младшее умножение

Умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Младшее слово каждого из результатов записывается в соответствующую позицию приемника.

PMADDWD приемник,источник – умножение и сложение

Умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Произведения двух старших пар слов складываются между собой, и их сумма записывается в старшее двойное слово приемника. Сумма произведений двух младших пар слов записывается в младшее двойное слово.

2) Організація інтерпретації вхідної мови.

Все, что было найдено в интернете по этому вопросу. Четкого ответа в конспекте и в шпорах не было.

(первый источник)

ЭВМ непосредственно выполняет программы на ***машинном языке*** программирования данной ЭВМ. При этом программа представляет собой набор отдельных команд компьютера. Эти команды являются достаточно «простыми», например, сложение, умножение, сравнение или пересылка отдельных данных. Каждая команда содержит в себе сведения о том, какая операция должна быть

выполнена (код операции), с какими операндами (адреса данных или непосредственно сами данные) выполняются вычисления и куда (адрес) должен быть помещен результат.
Машинные языки были первыми языками программирования. Программирование на них затруднительно ввиду того, что, во-первых, эти языки различны для каждого типа ЭВМ, во-вторых, являются трудоемкими для большинства пользователей по причине необходимости знания особенностей конкретной ЭВМ и большого количества реализуемых ею операций (команд). Данные языки обычно используются для разработки системных программ, при этом чаще всего применяются специальные символические языки — Ассемблеры, близкие к соответствующим машинным языкам.
Человеку свойственно формулировать и решать задачи в выражениях более общего характера, чем команды ЭВМ. Поэтому с развитием программирования появились языки, ориентированные на более высокий уровень абстракции при описании решаемой на ЭВМ задачи. Эти языки получили название языков высокого уровня. Их теоретическую основу составляют алгоритмические языки, например, Паскаль, Си, Бейсик, Фортран, PL/1.
Для перевода программы, написанной на языке высокого уровня, в соответствующую машинную программу используются ***языковые процессоры.*** Различают два вида языковых процессоров: интерпретаторы и трансляторы.
Интерпретатор — это программа, которая получает исходную программу и по мере распознавания конструкций входного языка реализует действия, описываемые этими конструкциями.
Транслятор — это программа, которая принимает исходную программу и порождает на своем выходе программу, записываемую на объектном языке программирования (объектную программу). В частном случае объектным может служит машинный язык, и в этом случае полученную на выходе транслятора программу можно сразу же выполнить на ЭВМ. В общем случае объектный язык необязательно должен быть машинным или близким к нему (автокодом). В качестве объектного языка может служить и некоторый промежуточный язык.
Для промежуточного языка может быть использован другой транслятор или интерпретатор — с промежуточного языка на машинный. Транслятор, использующий в качестве входного язык, близкий к машинному (автокод или язык Ассемблера) традиционно называют Ассемблером.
Транслятор с языка высокого уровня называют ***компилятором*.**

(второй источник)

Любую программу, которая переводит произвольный текст на некотором входном языке в текст на другом языке, называют транслятором. В частности, исходным текстом может быть входная программа. Транслятор переводит её в выходную или объектную программу.

В смысле этого определения простейшим транслятором можно считать, загрузчик, который переводит программу в условных адресах, оформленную в виде модуля загрузки, в объектную программу в абсолютных адресах. В этом случае входной язык (язык загрузчика) и объектный язык (язык ЭВМ) являются языками одного уровня. Однако чаще входной и объектный языки относятся к разным уровням. Обычно уровень входного языка выше уровня объектного языка.

По уровню входного языка трансляторы принято делить на ассемблеры, макроассемблеры, компиляторы, генераторы.

Входным языком ассемблера является мнемокод, макроассемблера - макроязык, компилятора - процедурно-ориентированный язык, а генератора - проблемно – ориентированный язык. В связи с этим входной язык называют по типу транслятора: язык ассемблера, язык макроассемблера и т.д.

Программа, полученная после обработки транслятором, либо непосредственно исполняется на ЭВМ, либо подвергается обработке другим транслятором.

Так, для человека привычно задавать данные в виде текстов и рисунков. Эффективность обработки требует представления тех же данных в виде массивов и списковых структур. Отсюда и возникает необходимость двух уровней представления данных: внешний (уровень человека - входной язык компилятора) и внутренний (уровень компьютера - выходной язык компилятора).

По сути дела, прикладная задача компиляции предполагает существование функционального однозначного **преобразования трансляции**: **TRANS: Li → Lj, (1)** которое каждому предложению **l**i входного языка **Li** ставит в соответствие вполне определенное предложение **lj** выходного языка **Lj**.

Мы определяем **вычислительный процесс компиляции** как процесс решения прикладной задачи компиляции. **Реализация вычислительного процесса компиляции** - вычисление значения **TRANS(li)** преобразования трансляции (2) в заданной точке.

Вычислительный процесс компиляции определяется следующим образом:

• распознавание принадлежности заданной цепочки α входному языку **Li**;

• в случае удачи распознавания - вычисление преобразования трансляции (1) в

точке α;

• в случае неудачи распознавания - сообщение об ошибке и диагностика ошибки.

Для решения прикладной задачи компиляции конструируется прикладная программа **компилятор**, которая может быть как специальной, так и универсальной.

Вообще говоря, проблема интерпретации предложения некоторого языка является далеко не тривиальной. Она предполагает не только распознавание принадлежности входной цепочки некоторому профессиональному языку, но также распознавание заложенного в предложении смысла (семантики) и дальнейшей реакции на этот смысл. Проще всего обстоит дело с императивными языками. Каждое предложение **li** такого языка предписывает произвести вполне определенное преобразование набора исходных данных **d** в набор результирующих данных **r**. Таким образом, можно говорить, что смысл этого предложения выражается в виде специфики преобразования данных. (Смысл определяется через действие).

3) Організація захисту пам’яті в процесорах.

Защита программ и данных в процессоре i286.

Взаимная защита программ и данных в MS DOS основана на "джентельменском" соглашении о неиспользовании чужих областей. Но проблема заключается в том, что соглашение может быть нарушенно непреднамеренно всвязи с ошибкой в программе или из-за того что ОС не всегда может учесть требования всех загруженных потребителей адресного пространства.

Для организации защиты информации необходимо обеспечить:

1. Запретить пользовательским программам обращаться к областям, содержащим системную информацию (таблици устройств, системные переменные);
2. Закрыть доступ пользовательских программ к памяти и устройствам, распределенным для других программых модулей (пользовательских и ОС);
3. Исключить влияние программ друг на друга через систему комманд и исполняющую систему (зависание или программный останов одной задачи не должен влиять на ход выполнения другой);
4. Определить привила и очередность доступа к общим ресусам (устройству печати, дисплею и дискам).

Для решения этих задач используется:

1 Схема "супервизор-пользователь":

- ОС работает в привелигированном режиме "супервизор". Ей доступна вся оперативная память и все внешние устройства;
- Все остальные программы работают в "пользовательском" режиме и им доступен ограниченный размер ОП (ограничение может быть аппаратным, например принудительной установкой старших битов адреса в определенное значение). Для выполнения операций ввода/вывода пользовательские программы обращаются к супервизору, имеющему систему буферов обмена с физическими устройствами;

2 Система ключей:

- Каждая выполняемая программа имеет числовой идентификатор - "ключ";
- Каждая выделенная область памяти имеет аналогичный идентификатор;
- При обрашении ключи программы и памяти сравниваются и делается вывод о возможности доступа;

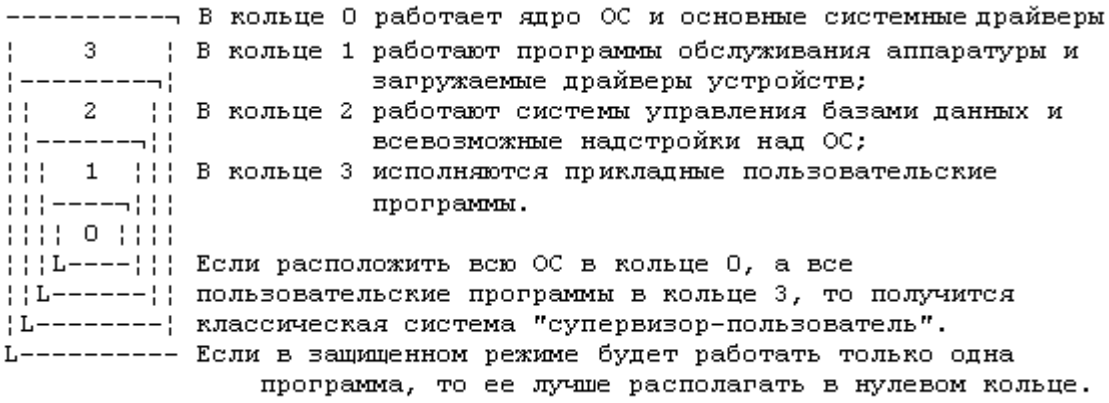
3 Система привелигированных и чувствительных команд;

- Все команды делятся на три группы по возможности разрушения системы;
- Привелигированные команды могут выполняться только программами, образующими ОС. Попытка выполнить такую команду в пользовательской программе приводит к ее игнорированию или обращению к ОС через "отказ";
- Чувствительные команды меняют свой алгоритм в зависимости от статуса выполняемой программы (могут не выполняться отдельные варианты команд или не выбираться отдельные операнды);
- Нечувствительные команды, их выполнение не зависит от внешних причин.

ПРИМЕЧАНИЕ: В реальных системах используются комбинации описанных схем защиты.

Кольца защиты.

Процессор i286 имеет четырехуровневую систему привилегий, и связанную с ней систему защиты, называемую КОЛЬЦАМИ ЗАЩИТЫ.



Когда программа загружается в память ей присваивается текущий уровень привилегий CPL (Current Privilege Level). Это значение записывается в поля DPL записей, образующих ее таблицу дескрипторов и в поле RPL регистра сегмента кода CS. Программа может проанализировать предоставленный уровень привилегий, но не может его изменить. Однако возможно изменение полей RPL сегментных регистров данных.

Структура дескриптора типа "шлюз вызова":

1. СМЕЩЕНИЕ (16 бит) - смешение точки входа от сегмента памяти;
2. СЕЛЕКТОР (16 бит) - значение селектора для вызова;

3. СЧЕТЧИК СЛОВ (8 бит) - количество 16-ти битовых слов (для i386+ слова 32-х битовые), подлежащих передаче в вызываемый модуль. *ВНИМАНИЕ: используются только пять младших битов этого поля, старшие три бита должны быть равны нулю;*
4. ДОСТУП (8 бит) - поле доступа должно соответствовать шлюзу вызова, а его поле DPL указывать минимальный уровень привилегий необходимых для прохода через вентиль;
5. РЕЗЕРВ (16 бит).

При вызове указанное количество параметров копируется из стека вызывающей программы в стек вызываемой (который специально создается вновь). При возврате сохраненнная информация восстанавливается из стека вызывающей программы. Место для создатия стека определяется при помощи TSS вызывающей задачи.
Режим ВИРТУАЛЬНОЙ ПАМЯТИ в процессоре i286 реализован недостаточно эффективно, основтой недостаток - отсутствие аппаратной борьбы с фрагментацией памяти. Дефрагментация осуществляется перемещением информации с последующей перестройкой таблиц дескрипторов.

Білет №10

1) Класифікація системних програм

Системная программа – программа общего пользования, выполняемая вместе с прикладными программами и служащая для управления ресурсами компьютера: центральным процессором, памятью, вводом-выводом. Системная программа - программа, предназначенная:
- для поддержания работоспособности системы обработки информации;
- для повышения эффективности ее использования.

Различают системные управляющие и системные обслуживающие программы.
Системні управляючі програми призначені для виконання та управління в ОС. Ці програми були призначені для введення завантаження програм в комп'ютер; введення/виведення інформації на зовнішні пристрої; управління подіями, що виникають в обчислювальних пристроях; розподілу ресурсами; управління доступом і захистом інформації в комп'ютері. Упр. Сис. Прогр. Автоматизують виконання задач на комп'ютері. Їх ще називають «прозорими».
Управляющие системные программы организуют корректное функционирование всех устройств системы. Системные управляющие программы составили основу операционных систем и в таком виде и в таком виде используются и в настоящее время. Основные системные функции управляющих программ:

- управление вычислительными процессами и вычислительными комплексами и
- работа с внутренними данными ОС.

До системних управляючих програм відносять всі програми ОС:

- Програми початкового завантаження
- Програми попереднього програмування та контролю обладнання
- Програми управляння файловою системою
- Програми переключення задач

Как правило, они находятся в основной памяти. Это резидентные программы, составляющие ядро ОС. Управляющие программы, которые загружаются в память непосредственно перед выполнением, называю транзитными (transitive). В настоящее время системные управляющие программы поставляются фирмами-разработчиками и фирмами-дистрибьюторами в виде инсталляционных пакетов операционных систем и драйверов специальных устройств.

Системні оброблюючі програми включають в себе програми, які підготовлюють задачі для розв'язання на комп'ютері: компілятори, компонувальники, текстові редактори; автоматизують підготовку машиних кодів для виконання програм. Системні оброблюючи програми часто постачаються окремо під ОС і служать для розробки програмного забезпечення. Загальна структура системних програм передбачає введення вхідних даних на деякій вхідній мові з наступним лексичним та синтаксичним аналізом, а також наступну симантичну чи змістовну обробку. Програми, що обробляють вхідні мови звичайно перетворюють їх на деяку внутрішню форму.
Обрабатывающие системные программы выполняются как специальные прикладные задачи, или приложения. Их пользователь вызывает при создании новых и модификации имеющихся программ.

Системные обрабатывающие программы предназначены для автоматизации подготовки программ для компьютера. К этой группе относят:

- Трансляторы (компиляторы и интерпритаторы)
- Компоновщики – которые объединяют оттранслированные прогр. модули с прогр. модулями библиотек, реализующих стандартные функции и порождают выполняемые коды.
- Текстовые редакторы и текстовые процессоры и программы – программы обеспечивающие связи между разными шагами разработки и отладки программ.
- Оболонки, що дозволяють автоматизувати роботу по створенню програмних проектів
- Програми налагодження розроблених програм

Більшість видів аналізу системних оброблюючих програм базується на використанні таблиць та правил обробки.

Різниця таблиць системних оброблюючих програм від таблиць реляційних баз даних полягає в тому, що для зберігання таблиць баз даних використовується носії інформації, а для системних оброблюючих програм – пам'ять. В процесі виконання частина бази даних переноситься до оперативної памяті, а частина системної обробляючої програми, якщо їй не вистачає памяті, зберігається на накопичувачі. Таблиці в системних оброблюючих програмах використовують, щоб повертати дані, значення полів як аргументи пошуку. Для генерації кодів, до наступного виконання, використовують таблиці відповідності внутрішнього подання.

2) Оганізація семантичного аналізу в компіляторах

Для того щоб створити і використовувати узагальнену програму семантичної обробки необхідно побудувати набори функцій семантичної обробки для кожного з вузлів графа розбору. При семантичному аналізі програма семантичного аналізу повинна перевірити аргументи за таблицею відповідностей аргументів та типів результатів.

При аналізі задач оптимізації можна виділити семантичний аналіз у вигляді термінальних вузлів, або у вигляді обробки нетерм. Вузлів доцільно використю табл. Відповідності аргументів чи операндів разом з полями визначених вихідних типів. Аргументами пошуку такої табл. Буде: внутр. Код(подання операції), тип і довжина 1-го та 2-го операнду. Функціональними полями в табл. Повинні бути тип і спосіб визначення довжини результату. Спосіб обчислення довжини може обчислюватись за допомогою якоїсь процедури обчислення довжини. Кожен алгоритм обробки викристовується і для інших видів семант. Обробки, тобто використ. Та сама рекурсія, але використ. Інші проц. Семант. Обробки.

Для семантичного аналізу програм може бути використана 1 табл. Відповідностей, в якій серед функц. Полів будують тип результату і функції визначення довжини результатута ф-ції інтерпретацій визначення операцій. Для реалізації повного інтерпретатора мови необх. Визначити в табл. Відповідності рядки для кожної з операцій, або фрагмента оператора мови. Найчастіше як внутр. Подання програми використовувались формат польського інверсного запису для виразу або постфіксна форма подання виразу. Деревовидна форма подання , в якій кореневі вузли піддерев визначають обчислення, що виконуються в підпідлеглих структурах. Різновидом таких дерев можна вважати спрямовані ациклічні графи(DAG). Звичайно такі графи мінімізують деревовидні подання, замінюючи повторювані піддерева відповідними посиланнями. В більш ранніх компіляторах часто використовували тетрадні та тріадні подання, які складалися з команд(послідовності команд) віртуальної машини реалізації мови. Тріадні подання включали код операції та адреси або вказівники на 2 операнди.В тетрадних поданнях використовувались 3 окремі посилання: 2-на операнди, а 3-й на результат, тобто вони відповідали 2-х та 3-х адресним командам комп'ютерів попередніх поколінь. Такі форми є залежними від обраної архітектури віртуальних машин. Внутрішні подання Паскалю у формі Р-кодів ближче до 1-адресної машини, а початкова форма більш спрямована до циклічної є взагалі архітектуронезал., а тому більш загальною.

//

Приклад реалізації кодування внутрішнього подання типів для занять з комп'ютерного практикуму включає базовий перенумерований тип, який фактично задає розбиття на три поля кодування:

```
enum datType//кодування типів даних в семантичному аналізі
{_v,                                    // порожній тип даних
 _uc=4,_us,_ui,_ui64,// стандартні цілі без знака
 _sc=8,_ss,_si,_si64,     // стандартні цілі зі знаком
 _f,_d,_ld,_rel,          // дані з плаваючою точкою
 _lbl,                          // мітки
// інші стандартні типи
 _geq = 0x0ffe,           // загальний тип для рівності
 _gen = 0x0fff,           // загальний (довільний) тип
 _enm = 0x1000,                 // перенумеровані типи enum
 _str = 0x2000,           // структурні типи /*_record*/,
 _unn = 0x3000,                 // типи об'єднань union
 _cls = 0x4000,           // типи класів
 _obj = 0x5000,           // типи об'єктів
 _fun = 0x6000,           // функціональні типи
 _ctp = 0x7000,           // умовні типи мови Pascal
 _fl, _tp, _vl, _vr,      //
};
// модифікатори кодів типів мови C/C++
#define cdPtr 0x100000          // код покажчика 1-го рівня
#define cdCns 0x080000          // код константного типу даних
#define cdArr 0x108000          // код даних типу масиву
#define cdCna 0x188000          // код константного масиву
#define cdReg 0x010000          // код регістрового типу даних
#define cdExt 0x020000          // код зовнішнього типу даних
#define cdStt 0x030000// код статичного типу даних
#define cdAut 0x040000          // код автоматичного типу даних
#define cdVlt 0x070000// код примусового типу даних
```

Таблиці семантичного аналізу включають механізми визначення кодів типів даних за допомогою перенумерованого типу **enum** datType та констант модифікаторів типів. Коди типів даних визначаються в

операторах обробки декларацій і використовують такі основні таблиці: визначення типів з модифікаціями; управління семантичною обробкою операцій. …

Елементи основних таблиць визначаються наступними структурами:

```
// Елемент таблиці модифікованих типів
struct recrdTPD// структура рядка таблиці модифікованих
                                        // типів
{enum tokType kTp[3];// примірник структури ключа
 unsigned       dTp;// примірник функціональної частини
 unsigned       ln;     // базова або гранична довжина даних типу
};
struct recrdSMA         // структура рядка таблиці припустимості
                                        // типів для операцій
{enum tokType oprtn;// код операції
 int oprd1, ln1;  // код типу та довжина першого аргументу
 int oprd2, ln2;  // код типу та довжина другого аргументу
 int res, lnRes;   // код типу та довжина результату
 _fop    *pintf;  // покажчик на функцію інтерпретації
 char    *assCd;
};
```

Базові типи, що визначаються ключовими словами, показані в таблиці характеристик типів відносно коду першого ключового слова, що визначає тип. Елементи цієї таблиці мають структуру.

```
struct recrdTMD         // структура рядка таблиці базових типів
{enum datType tpLx;// примірник структури ключа
 unsigned       md;     // модифікатор
 unsigned       ln;     // базова або гранична довжина даних типу
};
```

А сама таблиця для мови C/C++ має вигляд.

```
struct recrdTMD tpLxMd[]=
                        // масив кодів та ознак ключових слів типів
{{_v, 0, 0},            //0 _void
 {_v, 0, 0},            //1 _extern
 {_v, 0, 0},            //2 _var
 {_v,cdCns,0},  //3 _const
```

```
 {_enm, 0,32},   //4 _enum
 {_str, 0, 0},       //5 _struct/*_record*/
 {_unn, 0, 0},     //6 _union
 {_v,cdReg,0},   //7 _register
 {_ui,0,32},                 //8 _unsigned
 {_si,0,32},                 //9 _signed
 {_si,0,8},          //10 _char
 {_si,0,16},         //11 _short
 {_si,0,32},         //12 _int
 {_si,0,32},         //13 _long
 {_si,0,64},                 //14 _sint64
 {_ui,0,64},                 //15 _uint64
 {_f,0,32},          //16 _float
 {_d,0,64},          //17 _double
};
```

Таблиця типів, що визначаються декількома словами для мови C/C++ має наступний вигляд.

```
struct recrdTPDtpTbl[]=// таблиця модифікованих типів
{{{_void,_void,_void},_v,0},
 {{_enum,_void,_void},_enm,32},
 {{_struct,_void,_void},_str,0},
 {{_union,_void,_void},_unn,0},
 {{_unsigned,_void,_void},_ui,32},
 {{_signed,_void,_void},_si,32},
 {{_char,_unsigned,_void},_uc,8},
 {{_char,_signed,_void},_sc,8},//4
 {{_char,_void,_void},_sc,8},
 {{_short,_void,_void},_si,16},
 {{_short,_unsigned,_void},_ui,16},
 {{_short,_signed,_void},_si,16},
 {{_int,_void,_void},_si,32},//9
 {{_int,_unsigned,_void},_ui,32},
 {{_int,_signed,_void},_si,32},
```

{{_int,_long,_void},_si,32},

{{_long,_void,_void},_si,32},

{{_float,_void,_void},_f,32},//14

{{_double,_void,_void},_d,64},

{{_double,_long,_void},_ld,80},

{{_class,_void,_void},_cls,0},

};

Якщо таблицю типів розширити константними, регістровими та зовнішніми типами, то в кожному з їх елементів в останньому елементі ключа додається відображення першого ключового слова-модифікатора з потроєнням загального обсягу таблиці.

(з нашого конспекту)

Семантичний аналіз звичайно виконується як перевірка відповідності операндів (аргументів операндів) з формуванням типів результатів певних операндів

Для семантичного аналізу необхідно створити таблицю семантичної відповідності операндів та операцій. Для будь-якої припустимої пари операндів треба визначити тип результату.

До ключоваї частини таблиці повиння входити операція, або її тип та результати(типи) аргументів.

Тип результату розміщується в функціональній частині таблиці. При такій схемі сполучення операцій з операндами, для яких тема запису в таблиці вважається помилковими.

Алгорим обходу дерева графа скаладається так, що за винятком « := » включає 2 рекурсивних виклики тієї ж функції для піддерев. У випадку досягнення термінальних позначень виклики не виконуються.

Такий саме алгоритм можна використовувати для інших видів семантичної обробки.

3) Типовий склад програм ОС

ОС состоит из: ядра, базовой системы ввода-вывода, командного интерпретатора (необязательно), сервисных программ.

Ядро операционной системы — часть ОС, выполняющееся при максимальном уровне привилегий. Как правило, в ядро помещаются процедуры, выполняющие манипуляции с основными ресурсами системы и уровнями привилегий процессов, а также критичные процедуры. **Базовая система ввода-вывода -** (БСВВ, BIOS) — набор программных средств, обеспечивающих взаимодействие ОС и приложений с аппаратными средствами. Обычно БСВВ представляет набор компонент — драйверов. Также в БСВВ входит уровень аппаратных абстракций, минимальный набор аппартно-зависимых процедур ввода-вывода, необходимый для запуска и функционирования ОС. **Командный интерпретатор** — необязательная, но существующая в подавляющем большинстве ОС часть, обеспечивающая управление системой посредством ввода текстовых команд (с клавиатуры, через порт или сеть). Операционные системы, не предназначенные для интерактивной работы часто его не имеют. Также его могут не иметь некоторые ОС для рабочих станций

(MacOS 7, 8 (?)). **Сервисные программы** — пакет приложений, включённый в состав ОС, реализующий множество полезных фун

Екзаменаційний білет №11.

1.Узагальнена структура системної програми

На вхід системної програми звичайно вводяться дані, подані за синтаксичними правилами деякої мови на виході системної програми формуються результати або в головній пам'яті системи або у вигляді файлів на дискових накопичувачах. Через це звичайно системна програма виконується у декілька етапів , які у найбільш загальному випадку включають:

1. лексичний аналіз – розбиває на лексеми дані або лексеми
2. синтаксичний аналіз - перевірка відповідності вхідних даних синтаксичним правилам вхідної мови та побудова дерев розбору , які точніше є графами підлеглості виконуваних операторів та операцій.
3. включає деякі види семантичної або змістовної обробки. В трансляторах або при реалізації мов насамперед використовуються програми семантичного або змістовного аналізу, який перевіряє відповідність типів даних в операціях та формує типи даних результату. Другий тип семантичної обробки це інтерпретація або виконання.
4. (тільки в компіляторах)машино – незалежна оптимізація – вилучення зайвих або повторюваних фрагментів
5. Генерація кодів

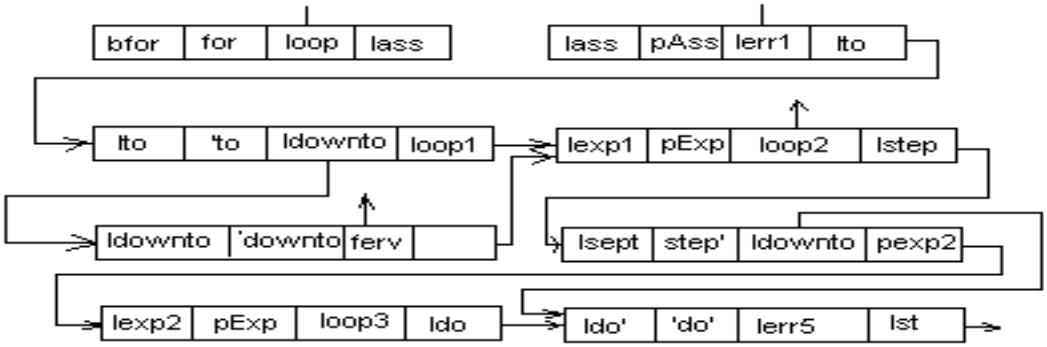
2. Формування графів синтаксичного розбору при синтаксичному аналізі.

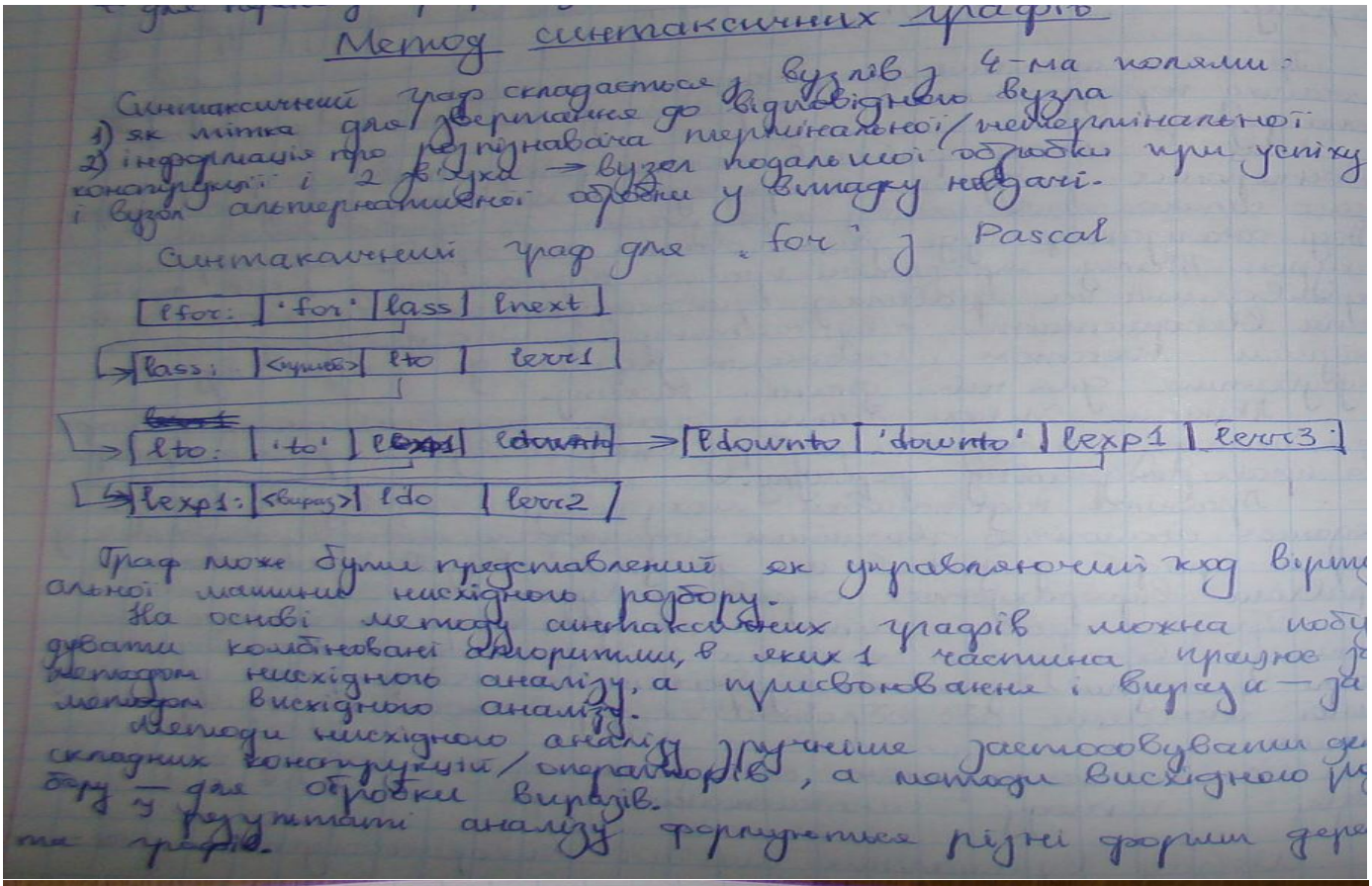
В цьому методі послідовність правил синтаксичного розбору організована у формі графа, кожен вузол якого має 4 складових :

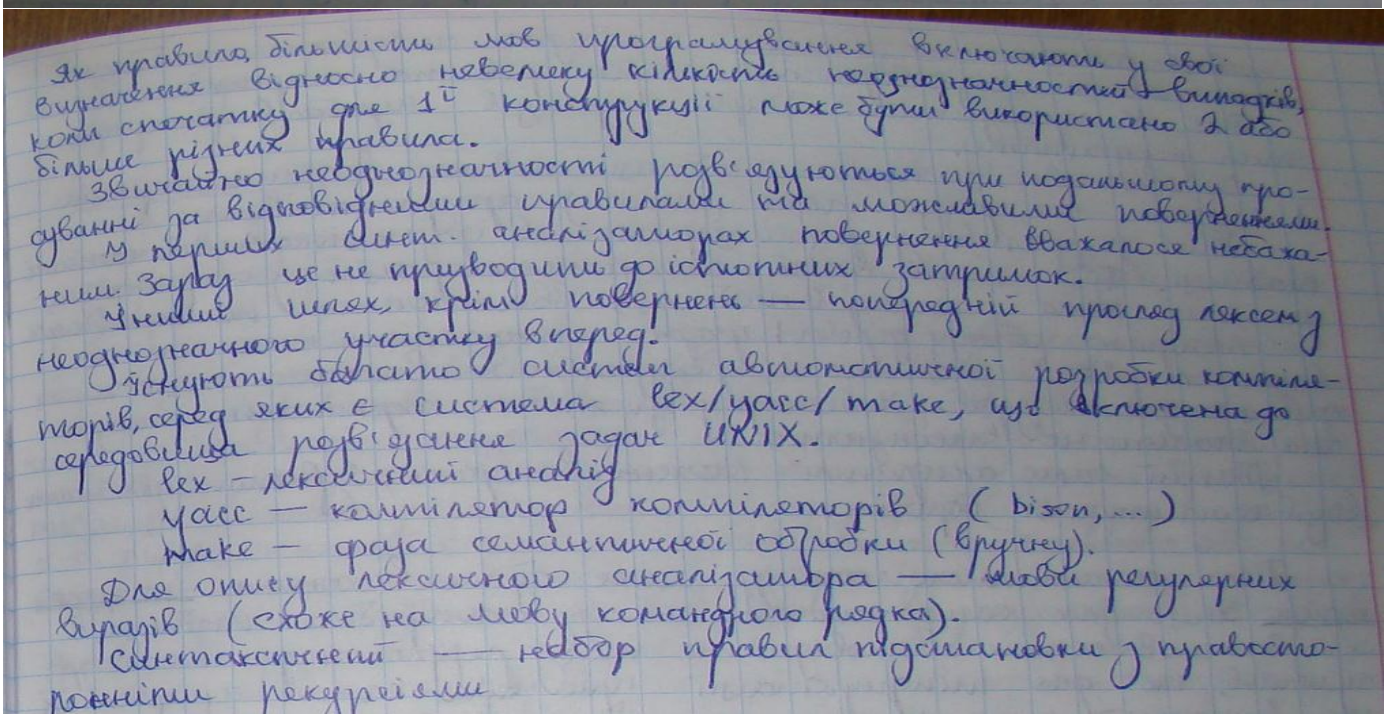
1. мітка, що призначена для зв'язування вузлів графа
2. прототип, який може розпізнати термінальні або не термінальні елементи синтаксичної конструкції для синтаксичного аналізу в компіляторах. За термінальні позначення обирають імена , константи, ключові слова, тощо.За не термінальні – вирази, списки, оператори , тощо.
3. покажчик на наступний вузол графу, до якого відбувається перехід у випадку успішної роботи прототипу
4. показник на альтернативне відгалуження синтаксичного графу, коли робота прототипу не буде успішною. По формі структур даних синтаксичний граф може бути представлений як сукупність вузлів з двійковим розг. У випадку коли синтаксична конструкція виявиться неправильною необхідно передбачити нейтралізацію помилок, тобто видачу діагностики з пошуком конструкції після якої можна продовжити синтаксичний аналіз.

Синтаксично буває зручним використовувати змішані алгоритми аналізу. Як раз для обробки операторів мов програмування краще використовувати синтаксичний граф, а для обернених виразів – висхідний розбір.

Також існують зв'язки передачі управління в операторах розгалуження та блоках циклів та варіантному блоці. Таким чином будь-який закінчений фрагмент програми має головний вузол, в якому як підлеглі вузли різних рівнів зберігаються лексеми та синтаксичні структури, які входять до цього блоку. Такий граф розбору є основою для всіх видів подальшої семантичної обробки. Методи низхідного розбору ефективніші при обробці структурованих операторів(for)







3. Збереження стану задач в захищеному режимі.

На сколько мне известно, происходит следующее:

сохраняются

- сохраняются все регистры задачи
- каталог таблиц страниц процесса

защищённый режим:

Перед тем, как переключить процессор в защищённый режим, надо выполнить некоторые подготовительные действия, а именно:

- Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим. Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, разумеется, она работает в нулевом кольце защиты). Программа может модифицировать имеющиеся дескрипторы или добавить новые, загрузив заново регистр GDTR.
- Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по адресу 0040h:0067h, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по адресу 0040h:0067h.
- Запретить все маскируемые и немаскируемые прерывания.
- Открыть адресную линию A20.
- Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
- Загрузить регистр GDTR.

Для перевода процессора i80286 из реального режима в защищённый можно использовать специальную команду LMSW, загружающую регистр состояния процессора (Mashine Status Word). Младший бит этого регистра указывает режим работы процессора. Значение, равное 0, соответствует реальному режиму работы, а значение 1 - защищённому. (Для возврата в RM– сбросить проц)

Если установить младший бит регистра состояния процессора в 1, процессор переключится в защищённый режим: mov ax, 1 lmsw ax.

2. После выполнения сброса (или после отключения) процессор переходит в реальный режим и управление передаётся в BIOS. BIOS анализирует содержимое ячейки CMOS-памяти с адресом 0Fh - байта состояния отключения (используется BIOS для определения способа возврата после аппаратного сброса).

Для обеспечения возврата в реальный режим после сброса по адресу, записанному в области данных BIOS 0040h:0067h можно использовать байты 5 и 0Ah.

Из за особенностей обработки прерываний в ЗР, перед переключением в ЗР необходимо перепрограммировать контроллер прерываний. Восстановить состояние контроллера после возврата в РР можно автоматически, если использовать значение 5 для байта состояния отключения.

Если же не использовать прерывания и не перепрограммировать контроллер прерываний, можно использовать значение 0Ah, при этом после сброса управление будет сразу передано по адресу, взятому из области данных BIOS 0040h:0067h. В этом случае затраченное на возврат в реальный режим время будет меньше.

В следующем фрагменте программы в ячейку CMOS-памяти с адресом 0Fh записываем значение 5.

Для записи числа в ячейку CMOS-памяти необходимо вначале в порт с адресом 70h записать номер нужной ячейки, а затем в порт 71h - записываемые данные.

```
cli
mov    al,8f
out    CMOS_PORT,al
nop    ; ? небольшая задержка
```

```
PROC   enable_a20    NEAR
       mov    al,A20_PORT
       out    STATUS_PORT,al
       mov    al,A20_ON
       out    KBD_PORT_A,al
       ret
ENDP   enable_a20
```

3. (В этом фрагменте программы вместо ячейки 0Fh указано значение 8Fh - это не ошибка.) Единственный способ замаскировать немаскируемые прерывания в компьютере IBM AT - это записать в порт 70h байт, в котором старший бит установлен в 1. Поэтому наш фрагмент программы не только записывает байт состояния отключения, но и маскирует немаскируемые прерывания (!). Нам необходимо также замаскировать обычные прерывания, поэтому мы выдаём команду CLI.

4. Следующий шаг - открытие адресной линии A20 - необходим, если программа будет обращаться к оперативной памяти, лежащей за

пределами первого мегабайта (после начального сброса она закрыта). Для этого: (рис. слева, ниже).

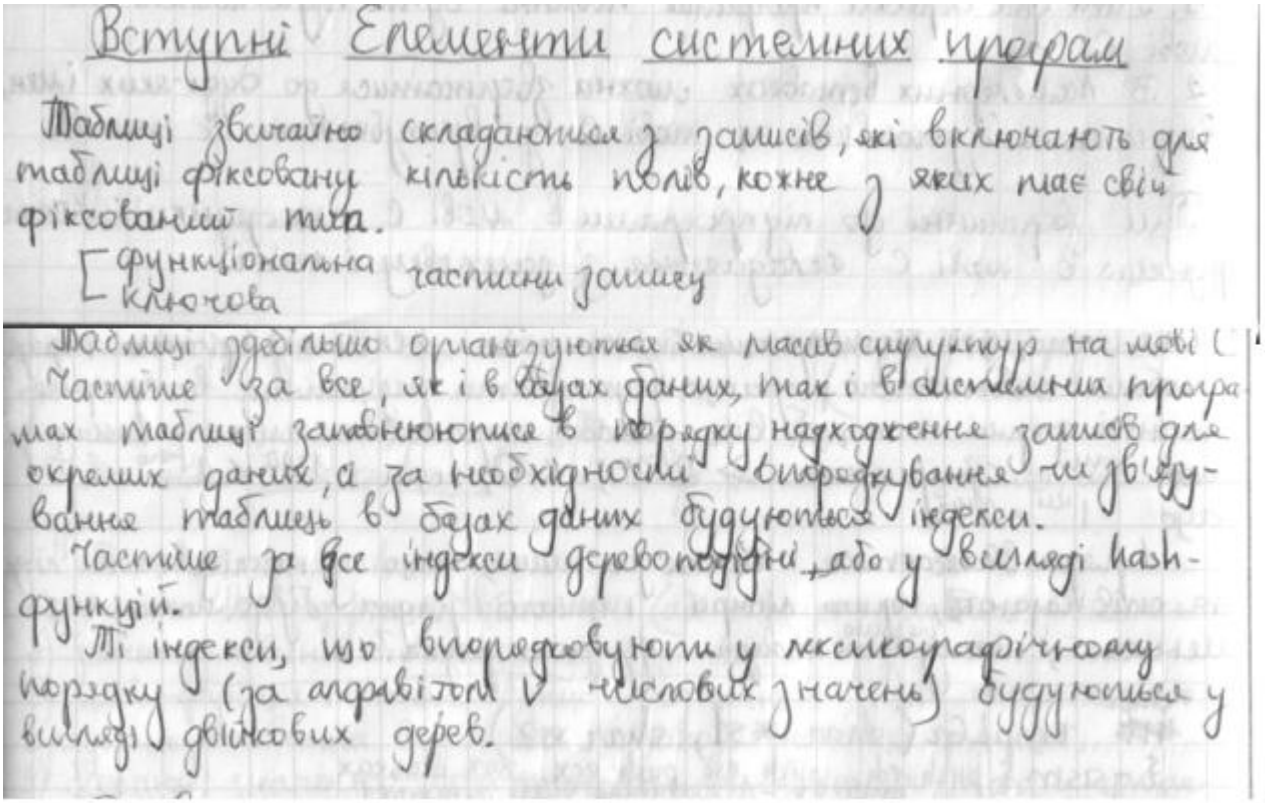
5. Следующий этап - запоминание содержимого сегментных регистров, которые будут нужны при возврате в реальный режим. Это сегментные регистры SS и ES:

```
mov     [real_ss],ss     ; запоминаем указатель стека
mov     [real_es],es     ; для реального режима
```

6. На последнем перед переключением в защищённый режим этапе мы загружаем регистр GDTR адресом подготовленной заранее GDT: lgdt [QWORD gdt_gdt]

Білет №12

1) Оранізація таблиць у вигляді масивів записів


Вступні Елементи системних програм

Таблиці звичайно складаються з записів, які включають для таблиці фіксовану кількість полів, кожне з яких має свій фіксований тип.

[функціональна частини запису
ключова

Таблиці здебільшого організуються як масив структур на мові (
Частіше за все, як і в базах даних, так і в системних програмах таблиці заповнюються в порядку надходження записів для окремих даних, а за необхідності впорядкування чи збільшування таблиць в базах даних будуються індекси.
Частіше за все індекси деревоподібні, або у вигляді hash-функцій.
Ті індекси, що впорядковують у лексикографічному порядку (за алфавітом у числових значень) будуються у вигляді двійкових дерев.

2) Загальний підхід до ораганізації семантичної обробки в трансляторах

Організація семантичного аналізу

Для того щоб створити і використовувати узагальнену програму семантичної обробки необхідно побудувати набори функцій семантичної обробки для кожного з вузлів графа розбору. При семантичному аналізі програма семантичного аналізу повинна перевірити аргументи за таблицею відповідностей аргументів та типів результатів.

При аналізі задач оптимізації можна виділити семантичний аналіз у вигляді термінальних вузлів, або у вигляді обробки нетерм. Вузлів доцільно використю табл. Відповідності аргументів чи операндів разом з полями визначених вихідних типів. Аргументами пошукутакої табл. Буде: внутр. Код(подання операції), тип і довжина 1-гота 2-го операнду. Функціональними полями в табл. Повинні бути тип і спосіб визначення довжини результату. Спосіб обчислення довжини може обчислюватись за допомогою якоїсь процедури обчислення довжини. Кожен алгоритм обробки використовується і для інших видів семант. Обробки, тобто використ. Та сама рекурсія, але використ. Інші проц. Семант. Обробки.

Для семантичного аналізу програм може бути використана 1 табл. Відповідностей,в якій серед функц. Полів будують тип результату і функції визначення довжини результатута ф-ції інтерпретацій визначення операцій. Для реалізації повного інтерпретатора мови необх. Визначити в табл. Відповідності рядки для кожної з операцій, або фрагмента оператора мови. Найчастіше як внутр. Подання програми використовувались формат польського інверсного запису для виразу або постфіксна форма подання виразу. Деревовидна форма подання , в якій кореневі вузли піддерев визначають обчислення, що виконуються в підпідлеглих структурах. Різновидом таких дерев можна вважати спрямовані ациклічні графи(DAG). Звичайно такі графи мінімізують деревовидні подання, замінюючи повторювані піддерева відповідними

посиланнями. В більш ранніх компіляторах часто використовували тетрадні та тріадні подання, які складалися з команд(послідовності команд) віртуальної машини реалізації мови. Тріадні подання включали код операції та адреси або вказівники на 2 операнди.В тетрадних поданнях використовувались 3 окремі посилання: 2-на операнди, а 3-й на результат, тобто вони відповідали 2-х та 3-х адресним командам комп'ютерів попередніх поколінь. Такі форми є залежними від обраної архітектури віртуальних машин. Внутрішні подання Паскалю у формі Р-кодів ближче до 1-адресної машини, а початкова форма більш спрямована до циклічної є взагалі архітектуронезал., атому більш загальною.

На цьомі етапі перевіряється відповідність типів, операндів або аргументів функцій та тип результату. Таким чином щоб реалізувати семантичний аналіз необхідно створити таблицю сумісності типів для всіх операцій та ключових слів В цій таблиці ключовими повинні бути операція або тип операції а потім типи операндів. З них ми повинні витягнути тип результату або код, що визначає несумісність або семантичну помилку.

Яким чином виконується семантичний аналіз. Обробка дерев розбору або циклу графів повинна виконватись в такому ж порядку , що й операції у виразах . Щоб виконати повний семантичний аналіз ми повинні виконати повний обхід дерева починаючи з з руху вліво і вниз . Найпростіший спосіб обходу бінарного ациклічного графу або дерева полягає в організації рекурсивного звертання до того ж самого аналізатора для підлеглих лівого і правого вузлів дерева. За цим рекурсивним алгоритмом може бути виконана будь-яка семантична обробка.

Можна уникнути рекурсії але для цього треба записати посилання на попередні вузли і дерева структури. Якщо програма представлена як дерево або граф, коли ми доходимо до терміналів або не терміналів змін їх треба відобразити щоб реконструювати вхідний текст.

Для виразів необхідно визначити пріорітетністьь розташування дужок. Якщо це індексні дужки то вони відображаються в окремому вузлі дерева.Так само в окремому вузлі функціональні та операторні дужки.

Трансля́тор — программа, которая принимает на вход программу на одном языке (он в этом случае называется *исходный язык*, а программа — *исходный код*), и преобразует её в программу, написанную на другом языке (соответственно, *целевой язык* и *объектный код*).

В качестве целевого языка наиболее часто выступают машинный код, Ассемблер и байт-код, так как они наиболее удобны (с точки зрения производительности) для последующего исполнения.

Наиболее часто встречаются две разновидности трансляторов:

- **Компиляторы** — выдают результат в виде исполняемого файла (в данном случае считаем, что компоновка входит в компиляцию). Этот файл:
  - транслируется один раз — может быть запущен самостоятельно
  - не требует для работы наличия на машине создавшего его транслятора

- **Интерпретаторы** — исполняют программу после разбора (в этом случае в роли объектного кода выступает внутреннее представление программы интерпретатором). Исполняется она построчно. В данном случае программа
  - транслируется (интерпретируется) при каждом запуске (если объектный код кешируется, возможны варианты)
  - требует для исполнения наличия на машине интерпретатора и исходного кода

Помимо «чисто» трансляторов и интерпретаторов, существует множество промежуточных вариантов. Так, большинство современных интерпретаторов перед исполнением переводят программу в байт-код (так как его покомандно выполнять гораздо проще, а значит, быстрее) или даже прямо в машинный код (в последнем варианте от интерпретатора остался только автоматический запуск, поэтому такой «интерпретатор» называется JIT-*компилятором*).

Термінальними називають позначення, які не можуть бути розділені на частини (не підлягають подальшому розбору).
Основа форм Бекуса – правило підстановки.

При створенні транслюторів з комп'ютерних мов часто використовують принцип розщарування граматик
Для лексичного аналізу – 1 граматику
Для синтаксичного аналізу – іншу.

Робота транслятора звичайно включає декілька етапів:

1) Лексичний аналіз (перетворює вхідний текст програми на масив лексем)

В граматиках лексичного аналізатора термінальними символами є літери. Вони можуть класифікуватися: букви, цифри, розділювачі, знаки мат. операцій.
Цільове (об'єкт) позначення граматики — лексема
Послідовно оптимізуємо послідовність лексем.
В результаті лексичного аналізу одержуємо масив вузлів графу підлеглості, але дуги не встановлені.

2) Синтаксичний аналіз проводиться так, що його термінальними позначеннями є лексеми, а цільовим (кінцевим) нетермінальним позначенням — найчастіше програмний модуль.
В результаті синт. аналізу встановлюються дуги між вузлами графа підлеглості, а також при відсутності необхідних правил виявляються синтаксичні помилки.

3) Семантичний аналіз (змістовний аналіз)
У мовах програмування перевіряє працездатність типів аргументів в операторах і виразах.

```
float f, **pff, *pf=&f, &fD=f;
```
(ссылка указатель адрес переменной f; ссылка ↑ обязательно инициализация; переименовали переменную)

В C++ ссылка может быть результатом функции, аргументом. При передаче аргументов по ссылке фактически передается адрес, но ссылка — обычная переменная, а не указатель.
– формує тип результату
– перевіряє адекватність типів аргументів
?? помиловий тип ??

4) Машинно-незалежна оптимізація перетворює попередній граф підлеглості операцій на оптимізований: всі однакові підграфи/піддерева замінюються [покажчиком на перший зв'язком

примірник підтерева
+ ще дещо можливо.

5) інтерпретація константних виразів
розрахунок констант

6) генерація об'єктних кодів
+ машинно-залежна оптимізація

Семантична обробка = $\sum_{i=3}^{6+} ?$

Вхідні дані для семантичної обробки є внутрішня форма. (у вигляді дерев чи графів підлеглості частин за все).

Звичайно семантична обробка виконується шляхом обходу графа / дерева у відповідності з правилами обходу окремих операцій.

Для арифметичних операцій обхід дерева у глибину, потім зліва направо. Для оператору присвоювання — вираз у правій частині, потім результат — для розміщення в призначені результату.

Так організуються всі види семантичної обробки.

При лексичному аналізі символи треба визначати за класами літер → буква, цифра, розділник, арифметична операція, літери кирилиці, лапки та апострофи для обмежених рядків

Лексеми чи типи лексем — стани автомату.

Для определения автомата надо определить все типы корректных и по возможности ошибочных (чтоб попытаться корекцию).

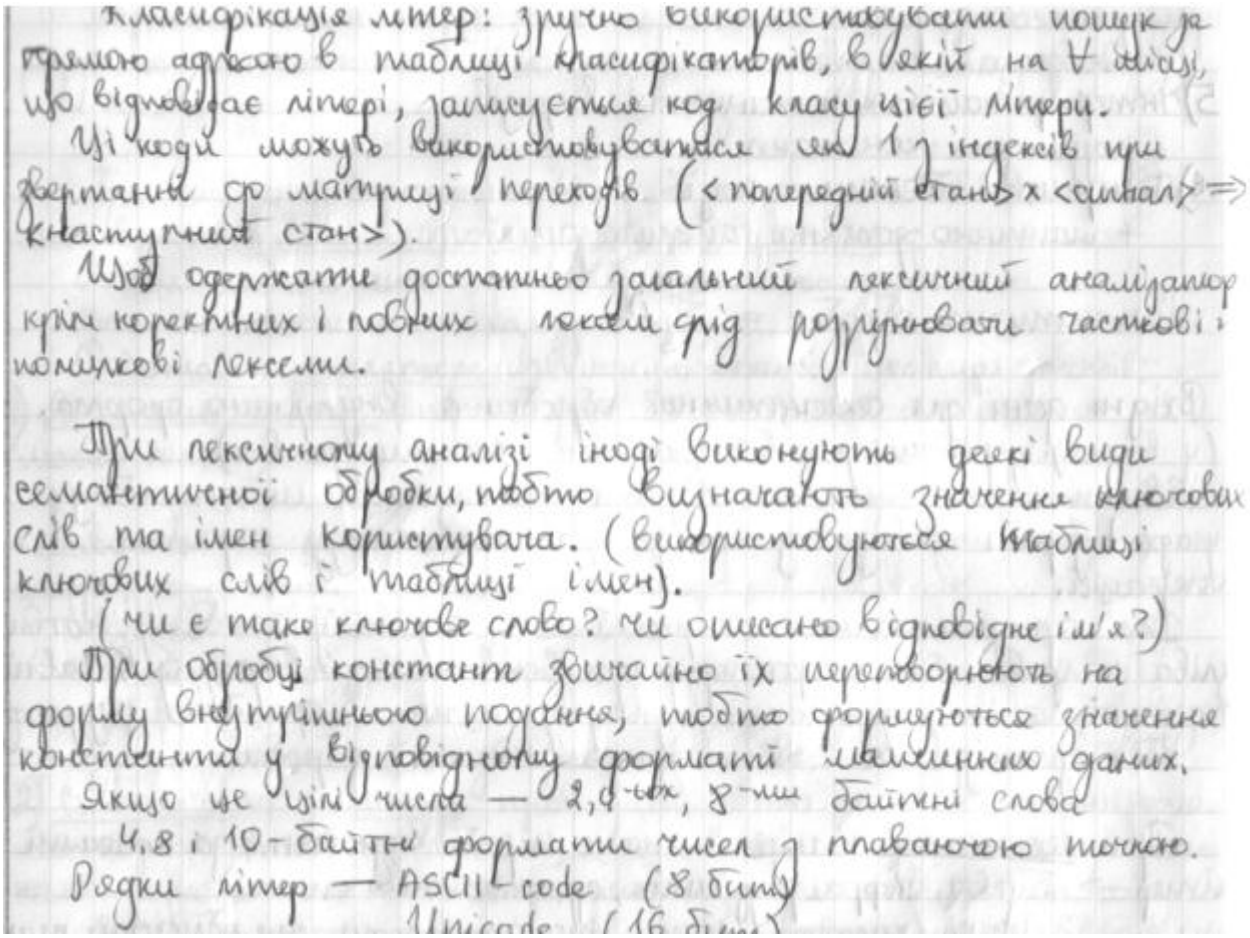
— ключові слова & імена
— числові константи (фіксована / плавоюча точка)
— символьні рядки
— матем. операції & розділники
— матем. операції, що включають декілька символів.

При обробці констант з плавоючою точкою фізично виділити декілька форм прим'ітивних команд.

Стани автомата зручно визначати перенумерованим типом, у якому задаються імена для # повних і частинкових лексем;

Символи — можна використовувати клас. черлової літери чи знаку:

— літера — цифра


Класифікація літер: зручно використовувати пошук за прямою адресою в таблиці класифікаторів, в якій на ∀ символ, що відповідає літері, записується код класу цієї літери.
Ці коди можуть використовуватися як 1 з індексів при звертанні до матриці переходів (<попередній стан> x <сигнал> => <наступний стан>).
Щоб одержати достатньо загальний лексичний аналізатор крім коректних і повних лексем слід розрізнювати частково і помилкові лексеми.

При лексичному аналізі іноді виконують деякі види семантичної обробки, тобто визначають значення ключових слів та імен користувача. (використовуються таблиці ключових слів і таблиці імен).
(чи є таке ключове слово? чи описано відповідне ім/я?)
При обробці констант звичайно їх перетворюють на форму внутрішнього подання, тобто формується значення константи у відповідному форматі машинних даних.
Якщо це цілі числа — 2,4-byte, 8-мі байтні слова
4, 8 і 10-байтні формати чисел з плаваючою точкою.
Рядки літер — ASCII code (8 бит)
Unicode (16 бит)

3) Способи організації драйверів в ОС

Драйвер периферийного устройства служит логическим *интерфейсом между подсистемой ввода/вывода операционной системы и* обслуживаемыми ею *аппаратными средствами*. В некоторых случаях драйвер заменяет или расширяет определенные аспекты BIOS'а и таким образом несет ответственность за обработку характеристик обслуживаемых аппаратных средств. В других драйверах физический ввод/вывод осуществляется исключительно через BIOS. В ранних версиях операционной системы *MS-DOS* не была предусмотрена процедура установки драйверов. Однако, *начиная с версии 2.0*, программисты получили возможность дополнять операционную систему. Несмотря на отсутствие общих стандартов для резидентных программ, *интерфейс между операционной системой и драйвером периферийного устройства жестко определен*, поэтому большинство драйверов может работать друг с другом совместно.

Драйверы бывают двух типов: *ориентированные на символьный либо блоковый обмен данными*. Драйверы, обслуживающие хранение и/или доступ к данным на дисках либо других устройствах прямого доступа, обычно ориентированы на блоковый обмен данными. Блоковые драйверы передают данные фиксированными порциями - отсюда и происходит их название.

Драйвер состоит из трех основных частей: ***заголовка, программы стратегий и программы прерываний***. *Заголовок* описывает возможности и атрибуты драйвера, присваивает имя драйверу символьного устройства и содержит NEAR (только смещение, одно слово) указатели на программы стратегий и прерываний, а также FAR (смещение и сегмент, двойное слово) указатель на следующий драйвер в цепочке драйверов (цепочка является однонаправленной, что затрудняет поиск ее начала). Указатель на следующий драйвер устанавливает MS-DOS сразу после завершения процедуры инициализации, а в самом драйвере ему должно быть присвоено начальное значение -1 (FFFFFFFFH). *Драйвер должен помещаться в 64K памяти, т.к. программы стратегий и прерываний содержат лишь смещения внутри выделенного драйверу сегмента*.

Программа стратегий. Программа стратегий вызывается, когда драйвер устанавливается при загрузке операционной системы, а также при каждом сгенерированном операционной системой запросе на ввод/вывод. Единичный запрос на ввод/вывод от прикладной программы может породить несколько запросов к драйверу. *Задача данной* программы заключается лишь в том, чтобы *сохранить* где-нибудь некоторый *адрес для дальнейшей обработки*. Этот адрес, *передаваемый в паре регистров ES:BX, указывает на структуру, называемую* ***заголовком запроса,*** которая содержит информацию, сообщающую драйверу, какую операцию он должен выполнить.

```
rlength db 0 ; 0 - длина существенных данных в заголовке
unit db 0 ; 1 - номер элемента
command db 0 ; 2 - фактическая команда
status dw 0 ; 3 - состояние после возврата
reserve db 8 dup (0) ; 5 - зарезервировано для DOS
```

Существенно то, что ***программа стратегий*** *не осуществляет никаких операций ввода/вывода, а лишь сохраняет адрес заголовка запроса для последующей обработки программой прерываний*. В мультизадачной системе этот адрес должен был бы храниться в некотором массиве, который при последующем вызове процедуры прерываний подвергался бы сортировке с целью оптимального использования устройства. Под управлением MS-DOS программа прерываний вызывается сразу после программы стратегий. Следует обратить внимание на то, что *между вызовами программ стратегий и прерываний допустимы прерывания*, а это может создать трудности, если драйвер написан в предположении, что между этими двумя вызовами проходит "нулевое время".

```
DRIVER SEGMENT PARA
ASSUME CS:DRIVER,DS:NOTHING,ES:NOTHING
ORG 0
START EQU $ ; Начало драйвера
;******* ЗАГОЛОВОК ДРАЙВЕРА************************
dw -1,-1 ; Указатель на следующий драйвер
dw ATTRIBUTE ; Слово атрибутов
dw offset STRATEGY ; Точка входа в программу STRATEGY
dw offset INTERRUPT ; Точка входа в программу INTERRUPT
db 8 dup (?) ; Количество устройств/поле имени
;******* РЕЗИДЕНТНАЯ ЧАСТЬ ДРАЙВЕРА
req_ptr dd ? ; Указатель на заголовок запроса
;******* ПРОГРАММА СТРАТЕГИИ
; Сохранить адрес заголовка запроса для программы СТРАТЕГИЙ
; в REQ_PTR.
; На входе адрес заголовка запроса находится в регистрах ES:BX.
STRATEGY PROC FAR
mov cs:word ptr [req_ptr],bx
mov cs:word ptr [req_ptr + 2],bx
```

Программа прерываний. В программе прерываний *выполняется вся фактическая работа драйвера*, поэтому она является наиболее сложной его частью. *При вызове этой программы исследуется командный байт (третий байт) ранее сохраненного заголовка запроса* и в зависимости от его значения выполняются те или иные действия. Программа прерываний *обычно использует командный байт в качестве индекса для некоторой управляющей таблицы*, вызывая таким образом нужную процедуру для каждой команды. Конечно, при желании можно использовать таблицу переходов. *Заголовок запроса содержит всю необходимую информацию для корректной обработки каждой команды* и сообщает вызывающей программе (в большинстве случаев таковой является MS-DOS) о состоянии запроса после завершения соответствующей процедуры. *Слово, хранящее состояние после возврата, разбито на несколько полей. Оно содержит бит ошибки*, указывающий на то, что в оставшейся части содержится специфический код ошибки,

бит выполнения, сигнализирующий о том, что требуемая операция была завершена, и *бит занятости*, призванный в первую очередь сигнализировать о текущем состоянии устройства. Обязательно должны присутствовать три раздела драйвера: **ЗАГОЛОВОК ДРАЙВЕРА, ПРОГРАММА СТРАТЕГИЙ и ПРОГРАММА ПРЕРЫВАНИЙ.** Программа ПРЕРЫВАНИЙ это не тоже самое, что программа обработки прерываний, которая может присутствовать в качестве необязательной части работающего по прерываниям драйвера. На самом деле, программа ПРЕРЫВАНИЙ - это точка входа в драйвер для обработки получаемых от MS-DOS команд.

Выше представлен скелет драйвера устройства. Хотя структура драйвера похожа на структуру .COM программы, важно отметить следующие отличия:

- *Программа начинается с нулевого смещения, а не 100H*.
- Образ программы начинается с директив определения данных для заголовка драйвера.
- Программа не содержит директивы ASSUME для стекового сегмента.
- Программа не содержит директивы END START.

Заголовок драйвера, программы СТРАТЕГИЙ и ПРЕРЫВАНИЙ

Команды языка ассемблера один к одному соответствуют командам процессора, фактически, они представляют собой более удобную для человека символьную форму записи (мнемокод) команд и их аргументов. Кроме того, язык ассемблера обеспечивает использование символических меток вместо адресов ячеек памяти, которые при ассемблировании заменяются на автоматически рассчитываемые абсолютные или относительные адреса, а также так называемых директив

Директивы ассемблера позволяют, в частности, включать блоки данных, задать ассемблирование фрагмента программы по условию, задать значения меток, использовать макроопределения с параметрами.

Каждая модель (или семейство) процессоров имеет свой набор команд и соответствующий ему язык ассемблера (автокод).

Существуют ЭВМ, реализующие в качестве машинного язык программирования высокого уровня (Forth, Lisp,Эль-76), фактически в них он является "ассемблером".

Достоинства языка ассемблера

Искусный программист, как правило, способен написать более эффективную программу на ассемблере, чем те, что генерируются трансляторами с языков программирования высокого уровня, то есть для программ на ассемблере характерно использование меньшего количества команд и обращений в память, что позволяет увеличить скорость и уменьшить размер программы.

Обеспечение максимального использования специфических возможностей конкретной платформы, что также позволяет создавать более эффективные программы с меньшими затратами ресурсов.

При программировании на ассемблере возможен непосредственный доступ к аппаратуре, в том числе портам ввода-вывода, регистрам процессора, и др.

Недостатки языка ассемблера

В силу своей машинной ориентации («низкого» уровня) человеку по сравнению с языками программирования высокого уровня сложнее читать и понимать программу, она состоит из слишком «мелких» элементов — машинных команд, соответственно усложняются программирование и отладка, растет трудоемкость, велика вероятность внесения ошибок. В значительной степени возрастает сложность совместной разработки.

Как правило, меньшее количество доступных библиотек по сравнению с современными индустриальными языками программирования.

Отсутствует переносимость программ на ЭВМ с другой архитектурой и системой команд (кроме двоично совместимых).

Происхождение и критика термина «язык ассемблера»

Данный тип языков получил свое название от названия транслятора (компилятора) с этих языков — ассемблера (англ. assembler — сборщик). Название последнего обусловлено тем, что программа "автоматически собиралась", а не вводилась вручную покомандно непосредственно в кодах. Следует иметь в виду возможную путаницу терминов: ассемблером в современном русском языке называют как язык программирования, так и транслятор с него.

Программа на ассемблере может содержать директивы: инструкции, не переводящиеся непосредственно в машинные команды, а управляющие работой компилятора. Набор и синтаксис их значительно разнятся и зависят не от аппаратной платформы, а от используемого транслятора (порождая диалекты языков в пределах одного семейства архитектур). В качестве «джентельменского набора» директив можно выделить следующие:

определение данных (констант и переменных)

управление организацией программы в памяти и параметрами выходного файла

задание режима работы компилятора

всевозможные абстракции (то есть элементы языков высокого уровня) — от оформления процедур и функций (для упрощения реализации парадигмы процедурного программирования) до условных конструкций и циклов (для парадигмы структурного программирования)

макросы

Язык ассемблера в русском языке также иногда называют «автокод».

Использование термина «язык ассемблера» также может вызвать ошибочное мнение о существовании некого единого языка низкого уровня, или хотя бы стандарта на такие языки. При именовании языка ассемблера желательно уточнять, ассемблер для какой архитектуры имеется в виду.

Білет №13

1. Лінійний пошук в різних видах таблиць.
2. Способи організації трансляторів з мов програмування.
3. Особливості визначення пріоритетів задач в ОС.

Відповіді:

1. ::: Линейный поиск можно выполнять как в упорядоченных так и в неупорядоченных таблицах. Метод заключается в последовательном сравнении ключа искомого элемента с ключами элементов таблицы до совпадения или до достижения конца таблицы. При работе с упорядоченной таблицей факт отсутствия элемента может быть установлен без просмотра всей таблицы.

Алгоритм линейного поиска в неупорядоченной таблице

1. Установка индекса первого элемента таблицы.
2. Сравнение ключа искомого элемента с ключом элемента таблицы.
3. Если ключи равны,перейти на обработку ситуации "поиск удачный",иначе на 4.
4. Инкремент индекса элемента таблицы.
5. Проверка конца таблицы.

Если исчерпаны все элементы таблицы, перейти к обработке ситуации "поиск неудачный", иначе на блок 2.

2. Трансля́тор — программа, которая принимает на вход программу на одном языке (он в этом случае называется *исходный язык*, а программа — *исходный код*), и преобразует её в программу, написанную на другом языке (соответственно, *целевой язык* и *объектный код*).

В качестве целевого языка наиболее часто выступают машинный код, Ассемблер и байт-код, так как они наиболее удобны (с точки зрения производительности) для последующего исполнения.

Наиболее часто встречаются две разновидности трансляторов:

- **Компиляторы** — выдают результат в виде исполняемого файла (в данном случае считаем, что компоновка входит в компиляцию). Этот файл:
  - транслируется один раз — может быть запущен самостоятельно
  - не требует для работы наличия на машине создавшего его транслятора

- **Интерпретаторы** — исполняют программу после разбора (в этом случае в роли объектного кода выступает внутреннее представление программы интерпретатором). Исполняется она построчно. В данном случае программа
  - транслируется (интерпретируется) при каждом запуске (если объектный код кешируется, возможны варианты)
  - требует для исполнения наличия на машине интерпретатора и исходного кода

Помимо «чисто» трансляторов и интерпретаторов, существует множество промежуточных вариантов. Так, большинство современных интерпретаторов перед исполнением переводят программу в байт-код (так как его покомандно выполнять гораздо проще, а значит, быстрее) или даже прямо в машинный код (в последнем варианте от интерпретатора остался только автоматический запуск, поэтому такой «интерпретатор» называется JIT-*компилятором*).

3. (Нечто из инета)

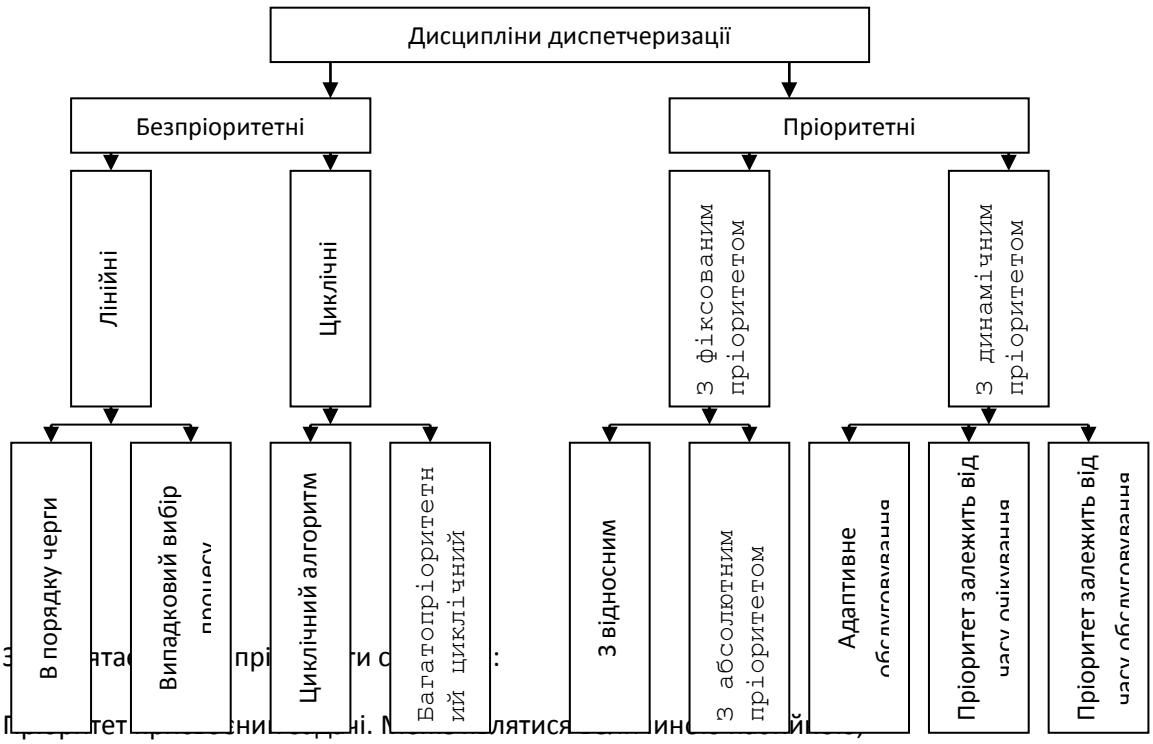
Дисципліни диспетчеризації

Коли говорять про диспетчеризацію, то завжди мають на увазі задачі на поняття процесу. Так як ці терміни часто використовуються іменно у такому смислі ми змушені будемо використовувати термін “процес” як синонім терміну “задача”.

Розрізняють дві дисципліни диспетчеризації – без пріоритетні і пріоритетні.

При без приорітетному обслуговуванні вибір задачі приводиться в деякому наперед установленому порядку без урахування їх відносної важливості і часу обслуговування.

При реалізації приоритетних дисциплін обслуговування окремим задачам надає переважаюче право попасти в стан виконання. Перерахунок дисциплін обслуговування і їх класифікація зображені на мал. 3.1.



Пріоритет задачі може змінюватись в процесі її розв’язання.

Диспетчеризація з динамічним пріоритетом потребує додаткових витрат на вираховування значень пріоритетів виконуючих задач тому у багатьох ОС реального часу використовуються методи диспетчеризації

на основі статистичних пріоритетів. Дисципліна обслуговування SLN потребує щоб для кожного завдання була відома оцінка в потребах машинного часу. Необхідність повідомляти ОС характеристики задач, в котрих описувались би потреби в ресурсах враховуючої системи., привела до тог, що були розроблені відповідні мовні засоби. Мова JCL була однією із найбільш відомих.

Користувачі змушені були вказувати передбачуваний час виконання, і для того, щоб вони не зловживали можливістю вказати менший час виконання ввели підрахунок реальних потрібностей. Диспетчер задач порівнював заказаний час і час виконання в у випадку перевищення вказаної оцінки у даному ресурсі ставив дане завдання не на початок а на кінець черги.

Дисциплінарне обслуговування SJN припускає, що є тільки одна черга завдань, готових до виконання. І завдання котрі в процесі свого виконання були тимчасово заблоковані (наприклад, чекали завершення операції вводу-виводу). Знову попадають в кінець черги готових до виконання на рівні з знову поступаючим. Дисципліна диспетчеризації RR –одна із найпоширеніших дисциплін. Однаково бувають ситуації, коли ОС не підтримує у явному вигляді дисципліну карусельної диспетчеризації. Наприклад в деяких ОС реального часу використовується диспетчер задач, працюючий за принципом абсолютних пріоритетів. Іншими словами, зняти задачу з виконання може тільки поява задачі з більш високим пріоритетом.

Тому, якщо потрібно організувати обслуговування задач таким чином, щоб всі вони почали процесорний час рівномірно і рівноправно, то системний оператор може сам організовувати цю дисципліну. Для цього одночасно всі користувацьким задачам присвоїти однакові пріоритети і створити одну високо пріоритетну задачу, котра не повинна нічого робити, але контра буде по таймеру плануватися на виконання. Ця задача зніме з виконання текуче прикладання, вона буде поставлена в кінець очереді і по-скільки цій високо пріоритетній задачі насправді нічого робити не треба, то вона зразу звільнить процесор і з черги готовності буде взята слідуюча задача. В своїй найпростішій реалізації дисципліна карусельної диспетчеризації передбачає, що всі задачі мають однаковий пріоритет.

Якщо необхідно ввести лаконізм пріоритетного обслуговування то це як правило робиться за рахунок організації декількох черг. Процесорний час буде представлений в першу чергу тим задачам, котрі знаходяться в найпривілейованій черзі.

Витісняючі і невітіснюючі алгоритми диспетчеризації. Диспетчеризація без пере розподілення процесорного часу це є не витісняюча багатозадачність – це такий спосіб диспетчеризації процесі при якому активний процес виконується до тих під поки від сам не віддасть управління диспетчеру задач для вибору із черги іншого готового до виконання процесу. Дисципліна SJN відноситься до невитісняючих.

Диспетчеризація з перерозподілом процесорного часу між задачами є витісняючою багатозадачнішою. Це такий спосіб при якому рішення про переключання процесора з виконання одного процесу на виконання іншого приймається диспетчером задач, а не активною задачою. При витісняючій багатозначності механізм диспетчеризації задач цілком зосереджений в операційній системі і програміст може писати своє прикладання, не турбуючись про те, як воно буде виконуватись правильно з іншими задачами. При цьому ос виконує слідуючі функції визначає момент зняття з виконання текучої задачі, зберігає її конте5кст у дескрипторі задач: вибирає у черги готових задач наступну і запускає її на виконання наперед завантажив її контекст.Дисциплін RR і інші побудовані на її основі відносять до витісняючи.

При не витісняючій багатозначності механізм розподілу процесорно часу розподілений між системою і прикладними програмами.

Білет №14

1. Пошук за прямою адресою та хеш-пошук.
2. Типові об’єкти системних програм.
3. Ієрархічна організація програм введення-виведення.

Відповіді:

1. Поиск в древовидных структурах осуществляется рекурсивно. Необходимо составить процедуру, в которую, как параметры передаются ссылка на элемент дерева и ключ поиска. Процедура при начальной инициализации получает ссылку на вершину дерева. В теле процедуры проверяется равенство ключа

поиска с ключом элемента дерева, если равенство выполняется, то считываются значения всех полей данного элемента дерева. За тем процедура вызывает сама себя с указанием ссылок из текущего элемента. Таким образом осуществляется столько вызовов процедуры, сколько ветвей у данного элемента дерева. Такой алгоритм позволяет пройти по всем элементам неоднородного дерева.

Самым быстрым методом поиска в больших таблицах является прямой, основанный на обращении к элементу с ключевой частью Ki по прямому вычисленному адресу - хеш-адресу (hash - крошить). Прямой поиск выполняется в хеш-таблице с начальным адресом Ан, в которой каждый элемент находится по хеш-адресу = Ан+ H(Ki), где хеш-функция H(Ki) - это такая алгоритмическая функция, которая должна давать неповторяющиеся значения для разных элементов таблицы и как можно плотнее размещать элементы в памяти. Хеш-функция определяет метод хеширования. Часто используются следующие методы:

- метод деления, при котором H(Ki)=Ki mod M, где M- достаточно большое простое число , например 1009;
- мультипликативный метод, при котором H(Ki)= C*Ki mod 1, где C- константа в интервале [0,1];
- метод извлечения битов, при котором H(Ki) образуется путем сцепления нужного количества битов, извлекаемых из определенных позиций внутри указанной строки;
- метод сегментации, при котором битовая строка, соответствующая ключу Ki, делится на сегменты, равные по длине хеш-адресу. Объединение сегментов можно выполнять разными операциями: сумма по модулю 2; сумма по модулю 16 и др; произведение всех сегментов.
- метод перехода к новому основанию, при котором ключ Ki преобразуется в код по правилам системы счисления с другим основанием. Полученное число усекается до размера адреса.

Алгоритм:

1. Формирование хеш-таблицы
2. Выбор искомого ключа
3. Вычисление хеш-функции ключа H(Ki)
4. Вычисление хеш-адреса ключа
5. Сравнение ключевой части элемента таблицы по вычисленному хеш-адресу с искомым ключом. При равенстве обработка ситуации "поиск удачный "и переход на 6; при неравенстве - обработка ситуации "поиск неудачный" и переход на 6.
6. Проверка все ли ключи выбраны; если да, то конец, а если нет ,то переход на 2.

При прямом поиске ситуация "поиск неудачный " может также иметь место при коллизии, то есть когда при Ki # Kj H(Ki) = H(Kj) . Самый простой метод разрешения коллизий - метод внутренней адресации, при котором под коллизирующие элементы используются резервные ячейки самой хеш-таблицы (пробинг).

Так при линейном пробинге к хеш-адресу прибавляется длина элемента таблицы, пока не обнаружится резервная ячейка. Для различия занятых ячеек памяти от резервных один бит в них выделяется под флаг занятости.

2. Класи системних програм

Особливості системних обробляючих програм; Особливості управляючих програм; Типова структура системної програми; Основні об’єкти системних програм

Системная программа – программа общего пользования, выполняемая вместе с прикладными программами и служащая для управления ресурсами компьютера: центральным процессором, памятью, вводом-выводом. Системная программа - программа, предназначенная:
- для поддержания работоспособности системы обработки информации;
- для повышения эффективности ее использования.

Различают системные управляющие и системные обслуживающие программы.
Системні управляючі програми призначені для виконання та управління в ОС. Ці програми були призначені для введення завантаження програм в комп’ютер; введення/виведення інформації на

зовнішні пристрої; управління подіями, що виникають в обчислювальних пристроях; розподілу ресурсами; управління доступом і захистом інформації в комп'ютері.

Управляющие системные программы организуют корректное функционирование всех устройств системы. Системные управляющие программы составили основу операционных систем и в таком виде и в таком виде используются и в настоящее время. Основные системные функции управляющих программ:

- управление вычислительными процессами и вычислительными комплексами и
- работа с внутренними данными ОС.

До системних управляючих програм відносять всі програми ОС:

- Програми початкового завантаження
- Програми попереднього програмування та контролю обладнання
- Програми управляння файловою системою
- Програми переключення задач

Как правило, они находятся в основной памяти. Это резидентные программы, составляющие ядро ОС. Управляющие программы, которые загружаются в память непосредственно перед выполнением, называю транзитными (transitive). В настоящее время системные управляющие программы поставляются фирмами-разработчиками и фирмами-дистрибьюторами в виде инсталляционных пакетов операционных систем и драйверов специальных устройств.

Системні оброблюючі програми включають в себе програми, які підготовлюють задачі для розв'язання на комп'ютері: компілятори, компонувальники, текстові редактори. Системні оброблюючи програми часто постачаються окремо під ОС і служать для розробки програмного забезпечення. Загальна структура системних програм передбачає введення вхідних даних на деякій вхідній мові з наступним лексичним та синтаксичним аналізом, а також наступну симантичну чи змістовну обробку. Програми, що обробляють вхідні мови звичайно перетворюють їх на деяку внутрішню форму.

Обрабатывающие системные программы выполняются как специальные прикладные задачи, или приложения. Их пользователь вызывает при создании новых и модификации имеющихся программ. Системные обрабатывающие программы предназначены для автоматизации подготовки программ для компьютера. К этой группе относят:

- Трансляторы (компиляторы и интерпритаторы)
- Компоновщики – которые объединяют оттранслированные прогр. модули с прогр. модулями библиотек, реализующих стандартные функции и порождают выполняемые коды.
- Текстовые редакторы и текстовые процессоры и программы – программы обеспечивающие связи между разными шагами разработки и отладки программ.
- Оболонки, що дозволяють автоматизувати роботу по створенню програмних проектів
- Програми налагодження розроблених програм

Більшість видів аналізу системних оброблюючих програм базується на використанні таблиць та правил обробки.

Різниця таблиць системних оброблюючих програм від таблиць реляційних баз даних полягає в тому, що для зберігання таблиць баз даних використовується носії інформації, а для системних оброблюючих програм – пам'ять. В процесі виконання частина бази даних переноситься до оперативної памяті, а частина системної обробляючої програми, якщо їй не вистачає памяті, зберігається на накопичувачі. Таблиці в системних оброблюючих програмах використовують, щоб повертати дані, значення полів як аргументи пошуку. Для генерації кодів, до наступного виконання, використовують таблиці відповідності внутрішнього подання.

::: *Узагальнююча структура системної програми:*

На вхід системної програми звичайно вводяться дані, подані за синтаксичними правилами деякої мови на виході сис проги формуються результати або в головній памяті системи або у вигляді файлів на дискових накопичувачах. Через це звичайно системна програма виконується у декілька етапів, які у найбільш загальному випадку включають:

1. лексичний аналіз – розбиває дані на лексеми або суперлекскми
2. синтаксичний аналіз – перевірка відповідності вхідних даних синтаксичним правилам вхідної мови та побудова дерев розбору, які точніше є графами підлеглості виконуваних операторів та операцій.
3. включає деякі види семантичної або змістовної обробки. В трансляторах або при реалізації мов насамперед використовуються програми семантичного або змістовного аналізу, який перевіряє відповідність типів даних в операціях та формує типи даних результату. Другий тип семантичної обробки це інтерпретація або виконання.
4. (тільки в компіляторах)машинно – незалежна оптимізація – вилучення зайвих або повторюваних фрагментів
5. Генерація кодів

3. Початковим поштовхом до розробки ОС були проблеми автоматизації завантаження програм та використання узагальнених механізмів введення-виведення. На початковому етапі розроблення ОС найбільш коштовною частиною був ЦП, тому головною вважалася задача ефективного використання процесору Для цього основною проблемою була організація ефективного введення-виведення з одночасною роботою ЦП над розв'язанням інших задач. Для цього необхідно механізм апаратного переривання, який замінював цикл ЦП з очікуванням готовності даних.

Структура підпрограми драйверів пристроїв включала наступні блоки:

- Видача команди на пристрій для його підготовки до обміну
- Очікування готовності пристрою до обміну
- Виконання власне обміну
- Видача на пристрій команди для закінчення операції
- Організація обміну драйвера даними з програмою, яка його використовує.

При створенні мікропроцесорів фактично було повторено процес створення програм обміну для зовнішніх пристроїв, але з деякою систематизацією.

Підключення зовнішніх пристроїв до мікропроцесору виконувалось шляхом визначення вихідного командного порту, вхідного порту стану, які мали однакові номер та вхідного і вихідного порту для введення і виведення даних, які мали однакову адресу. Щоб написати узагальнений драйвер введення-виведення треба визначити адреси портів за допомогою

```
CMPRT EQU   41H
STPRT EQU   CMPRT
INPRT EQU   42H
OUTPRT      EQU   INPRT
```

Возвращает в ax 1 байт данных введенных с некоторого устройства

```
drIn    Proc
        mov     al,cmOn
        out     CMPRT,al
lwr:    in      al,STPRT
        test    al,RdyBit
        jnz     lwr
        in      al,INPRT
```

```
        push    ax
        mov     al,cmOff
        out     CMPRT
        pop     ax
        ret
drIn    endp
```

Білет №15

1. Основні способи організаціїї таблиць та індексів
2. Системні оброблюючі програми
3. Механізми переключення задач в архітектурі процесора

1. Основні способи організаціїї таблиць та індексів

Організація роботи з таблицями в системних програмах

Таблиці – складні структури даних, завдяки яким можна підвищити ефективність програми. Їх основним призначенням є пошук в них інформації по заданому аргументу. Тому вони мають схожу структуру з базами даних. Таблиці мають задану кількість полів з даними, що є ключовими і функціональними. Таблиці системних програм зберігаються в ОП. Пошук виконується схожим чином з командою вибірки SELECT. Аргумент пошуку визначається значенням, що відповідає умові WHERE. До ключового поля висувається умова однозначності. Для додавання – INSERT. UPDATE і DELETE майже не використовуються в системних програмах.

Основные операции:

- оздание или получение доступа к таблице (конструктор)
- удаление таблицы
- включение нового эл-та
- поиск эл-та (эл-тов)
- упорядочение
- удаление эл-та (эл-тов)
- коррекция эл-та

Приведен пример фрагмента программы поиска по простейшемму линейному алгоритму с последовательным сравнением аргумента поиска с соответствующим полем. Будем считать, что аргумент загружен в аккумулятор AX, начальный адрес таблицы в - ES:DI, а колличество элементов таблицы в - CX

```
MOV     BX, DI   ; сохранение начального адреса таблицы
REPNZ SCASW   ; сканирование
JNZ     NotFnd ; переход если не найден
SUB     DI, BX   ; определение индекса найденного эл-та
SUB     DI, 2    ; компенсация технологического пропуска
SRL     DI, 1    ; получения индекса из разницы адресов
```

Таблиці в системних програмах створюються на період виконання. На кожному етапі роботи системної програми використовуються свої набори таблиць. Наприклад на етапі лексичного аналізу: таблиця імен, таблиця констант, таблиця сегментних регістрів. На відміну від баз даних початковий стан таблиць може бути визначений ініціалізацією початкових елементів масиву записів. В системних програмах висувається вимога унікальності вмісту ключових полів. Для цього в таблицях всі образи цих слів повинні бути унікальними. Як правило не дозволяється використовувати ключ слово, як імя об'єкта користувача. При створенні таблиці імен з врахуванням можливості різної видимості ключ поля склад з образу імені та частіше за все номеру блоку та його визначення.

В системных программах используются таблицы имен и констант в транслирующих программах, которые предназначены для синтаксического анализа и семантической обработки. Структурно таблицы обычно организуются как массивы и ссылочные структуры. Структуры данных табличного типа широко используются в системных программах, так как обеспечивает простое и быстрое обращение к данным. Таблицы состоят из элементов, каждый из которых представляется несколькими полями. Так при трансляции программы создаются, например, таблицы, содержащие элементы в виде: Ключ (переменная /метка) и характеристики: сегмент(данных / кода), смещение (XXXX), тип (байт, слово или двойное слово / близкий или дальний).

Особливості лінійного пошуку в різних видах таблиць

Линейный поиск можно выполнять как в упорядоченных так и в неупорядоченных таблицах. Метод заключается в последовательном сравнении ключа искомого элемента с ключами элементов таблицы до совпадения или до достижения конца таблицы. При работе с упорядоченной таблицей факт отсутствия элемента может быть установлен без просмотра всей таблицы.

Алгоритм линейного поиска в неупорядоченной таблице

6. Установка индекса первого элемента таблицы.
7. Сравнение ключа искомого элемента с ключом элемента таблицы.
8. Если ключи равны,перейти на обработку ситуации "поиск удачный",иначе на 4.
9. Инкремент индекса элемента таблицы.
10. Проверка конца таблицы.

Если исчерпаны все элементы таблицы, перейти к обработке ситуации "поиск неудачный", иначе на блок 2.

Приведен пример фрагмента программы поиска по простейшемму линейному алгоритму с последовательным сравнением аргумента поиска с соответствующим полем. Будем считать, что аргумент загружен в аккумулятор AX, начальный адрес таблицы в - ES:DI, а колличество элементов таблицы в - CX

```
MOV     BX, DI    ; сохранение начального адреса таблицы

REPNZ SCASW   ; сканирование

JNZ     NotFnd ; переход если не найден

SUB     DI, BX    ; определение индекса найденного эл-та

SUB     DI, 2     ; компенсация технологического пропуска

SRL     DI, 1     ; получения индекса из разницы адресов
```

2. Системні оброблюючі програми

Призначені для автоматизації процесу розв'язання задач на комп'ютері. Для цього вони використовують системні програми обміну даними та програми управління обчисленнями. Крім того для розв'язання кожної окремої задачі необхідно виділити системні ресурси: пам'ять, необхідну для зберігання даних і текстів програм; процесорний час для обробки даних; пристрої джерела і пристрої

споживачі інформації. Для виконання задачі розподілу ресурсів використовують планувальники різних типів та

супервізорні програми, які дозволяють визначити поточний розподіл ресурсів та визначити план наступного використання ресурсів. Сучасні системні управляючі програми об'єднуються в комплекси, які називаються операційними системами.

Системная программа - программа общего пользования, выполняемая вместе с прикладными программами и служащая для управления ресурсами компьютера: центральным процессором, памятью, вводом-выводом. Системная программа 90 - программа, предназначенная: - для поддержания работоспособности системы обработки информации; или - для повышения эффективности ее использования. Различают системные управляющие и системные обслуживающие программы. Системні оброблюючи програми включають в себе програми, які

підготовляють задачі для розв'язання на комп'ютері: компілятори, компонувальники, текстові редактори. Системні оброблюючи програми часто постачаються окремо під ОС і служать для розробки програмного забезпечення. Загальна структура системних программ передбачає введення вхідних даних на деякій вхідній мові з наступним лексичним та синтаксичним аналізом, а також наступну симантичну чи змістовну обробку. Програми, що обробляють вхідні мови звичайно перетворюють їх на деяку внутрішню форму. Більшість видів аналізу

системних оброблюючих програм базується на використанні таблиць та правил обробки

Эти программы поставляются чаще в виде дистрибутивных пакетов, включающих _____ПО

Більшість видів аналізу системних оброблюючих програм базується на використанні таблиць та правил обробки

Обрабатывающие системные программы выполняются как специальные прикладные задачи, или приложения. Их пользователь вызывает при создании новых и модификации имеющихся программ. Системные обрабатывающие программы предназначены для автоматизации подготовки программ для компьютера. К этой группе относят:

- Трансляторы (компиляторы и интерпритаторы)
- Компоновщики – которые объединяют оттранслированные прогр. модули с прогр. модулями библиотек, реализующих стандартные функции и порождают выполняемые коды.
- Текстовые редакторы и текстовые процессоры и программы – программы обеспечивающие связи между разными шагами разработки и отладки программ.
- Оболонки, що дозволяють автоматизувати роботу по створенню програмних проектів
- Програми налагодження розроблених програм

Різниця таблиць системних оброблюючих програм від таблиць реляційних баз даних полягає в тому, що для зберігання таблиць баз даних використовується носії інформації, а для системних оброблюючих програм – пам'ять. В процесі виконання частина бази даних переноситься до оперативної памяті, а частина системної обробляючої програми, якщо їй не вистачає памяті, зберігається на накопичувач. Таблиці в системних оброблюючих програмах використовують, щоб повертати дані, значення полів як аргументи пошуку. Для генерації кодів, до наступного виконання, використовують таблиці відповідності внутрішнього подання.

3. Механізми переключення задач в архітектурі процесора

Організація переключення задач

В процесорі Пентіум передбачено спеціальні механізми переключення задач. В реальному режимі або режимі ДОС при створенні нової задачі або процесу (потоку) для нього створюється окремий стек в якому зберігаються зміст регістрів переривань задачі, які часто називають надтекстом задачі, однак в реальному режимі при обробці переривань. програми можуть використовувати для своєї роботи фрагменти стеку

перерваної задачі, що призводить до проблем захисту цілісності кодів виконуваних задач. Щоб усунути проблеми захисту і узагальнити процеси переключення задач важливо використовувати той самий сегмент стану TSS. Крім регістрів в сегменті TSS записується інформація про стан об'єктів введення-виведення у спеціальному розширенні сегмента. Якщо в задачі використовується регістр з плаваючою точкою, то їх збереження покладається на відповідальність програміста, залежно від того як використовуватиме регістр з плаваючою точкою задача що переривається і задача що надходить на виконання.

Для того щоб перейти до кодів, які пов'язані з сегментом перерваної задачі необхідно спочатку підготувати в спеціальному регістрі задач номер відповідної задачі, а потім необхідно виконати команду переходу jmp, call на відповідний сегмент задачі. При цьому в старому сегменті задачі будуть збережені сегментні регістри та регістри загального користування, а з нового сегменту TSS будуть відновлені регістри нової задачі на яку відбувається переключення.

Для організації переключення задач за такою схемою для кожної задачі (в тому числі і для системних) потрібно побудувати принаймі один сегмент статусу задач TSS. Якщо для виконання якоїсь задачі передбачено виконувати підзадачі, які називаються в більшості ОС потоками (thread), які на відміну від процесів (process) виконуються в тому ж адресному просторі, що і головні процеси задач. Таким чином для потоків треба також створювати сегмент TSS, але в ни буде зберігатися інформація про однакові таблиці сегментів і сегментні регістрів, але різна інформація про уточнюючі значення регістрів.

Організація переключення задач в реальному режимі.

При використанні синхронізації примітивів програми обробки переривань звертаються до функції зміни стану примітиву, щоб далі звернутися до супервізора і змінити стан виконуваної задачі. В більшості ОС розрізняють 4 стани програми:

- активна
- готова – має всі ресурси для виконання, але чекає звільнення процесора
- очікує дані
- призупинена – тимчасово-вилучена з процесу виконання

Задача супервізора полягає у виборі найбільш пріоритетного з числа готових і переведення його в стан готового. Переходи на задачі супервізора можна виконати командами JMP або CALL.

Організація переключення задач у захищенному режимі

Для перехода в защищенный режим можно воспользоваться средствами того же BIOS и протокола DPHI, предварительно подготовив таблицы и базовую конфигурацию задач защищенного режима. Для организации переключения задач применен метод логических машин управления. Основу его аппаратно-программной реализации в процессорах ix86 составляем команды IMBCALL и IRET, бит NT регистра флагов, а также прерывания.

Для перехода от задачи к задаче при управлении мультизадачностью используются команды межсегментной передачи управления – переходы и вызовы. Задача также может активизироваться прерыванием. При реализации одной из этих форм управления назначение определяется элементом в одной из дескрипторных таблиц.

Тип дескриптора может быть таким, который инициирует выполнение новой задачи после сохранение состояния текущей. Имеется 2 кода типов, определяющих дескрипторы сегментов состояния задачи (TSS) и шлюза задачи. Когда управление передается любому из дескрипторов этих типов, происходит переключение задачи.

Дескрипторы шлюзов хранят только заполненные байты прав доступа и селектор соответствующего обьекта в глобальной таблице дескрипторов, помещенный на место 2-х младших байтов базового адреса. При каждом переключении задачи процессор может перейти с другой локальной дескрипторной таблицы, что позволяет назначить каждой задаче свое отображение логических адресов на физические.

Переключение задачи состоит из действий выполняемых одной из команд JMPPAR, CALLTAR или RET при NT=1:

1. проверка, разрешено ли уходящей задачи переключиться на новую
2. проверка файла, что дескриптор TSS приходящей задачи отмечен как присутствующий и имеет правильный предел (не меньше 67H)
3. сокращение состояния уходящей задачи
4. загрузка в регистр TR селектора TSS входящей задачи
5. загрузка состояния входящей задачи из ее сегмента TSS и продолжения выполнения.

При переключении задачи всегда сохраняется состояние уходящей задачи

Перед тем, как переключить процессор в защищённый режим, надо выполнить некоторые подготовительные действия, а именно:

- Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT.
- Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по адресу 0040h:0067h, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по адресу 0040h:0067h.
- Запретить все маскируемые и немаскируемые прерывания.
- Открыть адресную линию A20.
- Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
- Загрузить регистр GDTR.

Это самый простой этап. Для перевода процессора i80286 из реального режима в защищённый можно использовать специальную команду LMSW, загружающую регистр состояния процессора (Mashine Status Word). Младший бит этого регистра указывает режим работы процессора. Значение, равное 0, соответствует реальному режиму работы, а значение 1 - защищённому.

Білет № 16

1. Організація таблиць у вигляді структур з посиланнями

Крім простих в асемблері використовуються сегментовано таблиці та таблиці з посиланням. Таблиці повинні включати заголовки. В таблицях баз даних заголовок, як правило, включає імена і типи полів. В таблицях системних програм типи звичайно фіксовані. Тому заголовок вміщає лише допоміжну інформацію.

Набір методів для табличного об’єкту: конструктор і деструктор; методи select, insert, delete.

Методи select повинен визначати алгоритм пошуку елементів таблиці.

Таблиці системних програм будуються на базі масивів або послідовностей записів, що зберігаються в ОП. Для того, щоб використовувати масиви необхідно, щоб розміри всіх записів або рядків таблиці були однакові. Для цього символьну інформацію в таблиці доводиться подавати або у вигляді рядків максимальної довжини або через вказівними. Доцільно текстовий образ зберігати в окремому сегменті, а вказівник в таблиці (звичайно 4 байти). Функціональна частина зберігає тип, адресу, довжину.

```
Computer STRUC
     Frequency  dw ?
     HDD        dw ?
     Monitor    db ?
     CDROM      db ?
     FDD        db ?
Computer Ends

Computers Computer 6 dup(<?,?,?,?,?>)

List1     Computer <1100, 8400,_15inc,32,_35inc>
          Computer <1250, 9000,_17inc,40,_NOFDD>
          Computer <1250, 9000,_17inc,40,_7inc>
```

2. Задачі лексичного аналізу

ЛА предназначен для преобразования текста на входном языке во внутреннюю форму, при этом текст разбивается на лексемы. Для решения задачи лексического анализа могут использоваться разные подходы, один из них основан

на теории грамматик. К этой задаче можно подойти через классификацию лексем как элементов или типов входных данных. В качестве лексем входного языка обычно объявляют разделители, ключевые слова, имена пользователя, операторы, константы и спец. лексемы. Простейший алгоритм лекс. анализа должен содержать действие продвижения до разделителя, классификацию предшествующей лексемы и разделителя. Определения внутренней формы представления лексемы. Значение констант может отличаться или немного не соответствовать исходным константам из-за точности представления. Обычно устанавливается связь между внутренним представлением лексемы и ее местом во входном тексте. Имена, заданные пользователем, обычно сохраняются в таблицах, а во внутренней форме используется указатель на элемент таблицы и тип. Таблица иногда включает длину и некоторые дополнительные характеристики. Ключевые слова и разделители могут иметь стандартизированную внутреннюю форму представления. Чаще всего удобно построить спец. классификационную таблицу. Входной текст может быть в ASCII (каждый символ 1 байт), в UNICODE (2 байта) и др. Основные классы символов:
1) Вспомогательные разделители (пробел, таб., enter, ...);
2) Одно-символьные операции (+, -, *, /, ..., (,));
3) Много символьные операции (>=, <=, <>...);
4) Буквы, которые можно использовать в именах (латиница):
5) Не классифицируемые символы (для представления чисел или констант необходимо определить цифры и признаки основных систем счисления). При формировании внутреннего представления, кроме кода желательно формировать информацию о приоритете или значении предшествующих операторов.

3. Графи та їх обробка

Результати синтаксичного розбору подаються у вигляді зв'язаних вузлів графа розбору, в якому кожний вузол відповідає окремій лексемі, а зв'язки визначають підлеглість операндів до операцій. Вузол об'єднує 4 поля:

- Ідентифікація або мітка вузла
- Прототип співставлення (термінал або нетермінал)
- Мітка продовження обробки (при успішному результаті синтаксичного аналізу)
- Вказівник на альтернативну вітку, яку можна перевірити (якщо аналіз був невдачним)

Синтаксично буває зручним використовувати змішані алгоритми аналізу. Як раз для обробки операторів мов програмування краще використовувати синтаксичний граф, а для обернених виразів – висхідний розбір.

Також існують зв'язки передачі управління в операторах розгалуження та блоках циклів та варіантному блоці. Таким чином будь-який закінчений фрагмент програми має головний вузол, в якому як підлеглі вузли різних рівнів зберігаються лексеми та синтаксичні структури, які входять до цього блоку. Такий граф розбору є основою для всіх видів подальшої семантичної обробки.

№17

1. Двійковий пошук

При больших объемах таблиц (более 50 элементов) эффективнее использовать двоичный поиск, при котором определяется адрес среднего элемента таблицы, с ключевой частью которого сравнивается ключ искомого элемента. При совпадении ключей поиск удачный, а при несовпадении из дальнейшего поиска исключается та половина элементов, в которой искомый элемент находиться не может. Такая процедура повторяется, пока длина оставшейся части таблицы не станет равной 0.

алгоритм:

1. Загрузка нач. (Ан) и кон. (Ак) адресов таблицы
2. Определение адреса ср. элемента таблицы (Аср)
3. Сравнение искомого ключа с ключевой частью ср. элемента таблицы
4. При равенстве ключей поиск удачный. Если искомый ключ меньше ключа среднего элемента, то Ак=Аср, переход на 5. Если искомый ключ больше ключа среднего элемента, то Ан=Аср+длина элемента, переход на 5
5. Сравнение Ан и Ак. Если Ан=Ак, поиск неудачный, иначе переход на 2

При определении адреса среднего элемента следует выполнить следующие действия:

- вычислить длину таблицы Ак-Ан
- определить число элементов таблицы (Ак-Ан)/Lэ, где Lэ- длина элемента
- определить длину половины таблицы ((Ак-Ан) /Lэ)/2*Lэ
- определить адрес среднего элемента таблицы Аср= Ан + ((Ак-Ан)/Lэ)/2*Lэ.

При цьому таблиці кожного разу діляться навпіл при проході. З ключовою частиною елемента по середині злому знов і знов при кожній ітерації порівнюється ключова частина шуканого елементу. При незнаходженні з

пошуку виключається половина, в якій елемент бути не може (змінюється нижня або верхня границі пошуку). Процедура продовжується поки в залишковій частині таблиці нічого не залишиться.

Якщо таблиця складається з елементів довжиною h, то умова закінчення є I = I * h. Адреса елементу на зломі знаходиться як I = (IL + Ih)/2. Ділення на 2 для оптимізації проводиться командою SHR. Процес визначення адреси елементу за схемою:

АДРЕСА = ІНДЕКС * ДОВЖИНА + ПОЧАТКОВА_АДРЕСА

Алгоритм:

6. Завантаження початкової адреси таблиці
7. Визначення адреси злому по середині
8. Перевірка I = IL. Якщо кінець – вихід, або далі
9. Перевірка K = K[i]. Якщо виконується – вихід, якщо менше – IL:=I, більше – IH:=I;
10. Повторення ітерації

```
BINSRCHARG STRUC                ; структура для передачі параметрів
                dd      ?       ; проміжок для BP та адреси повернення
ProtPtr dd      ?       ; вказівник елементу, що перевіряється
ArgLn           dw      ?       ; довжина елементу, що перевіряється
TabPtr dd       ?       ; вказівник початку таблиці
TabLn           dw      ?       ; кількість елементів таблиці
ElmLn           dw      ?       ; довжина елементу таблиці
BINSRCHARG ENDS

public BinSrch
BinSrch         PROC
        push BP                 ; збереження старого BP
        mov BP, SP              ; Визначення бази зони параметрів
        push DS                 ; збереження сегменту даних
        les DI, ProtPtr[BP]     ; підготовка адреси аргументу
        LD6 SI, TabPtr[BP]      ; підготовка адреси таблиці
        xor BX, BX              ; встановлення початкової нижньої межі

lb:
        add AX, TabLn[BP]       ; Обчислення
        shr AX, 1               ; номеру середнього аргументу
        cmp AX, BX              ; Контроль границі розбивання
        jz NotFound             ; Перехід, якщо таблиця переглянута
        push AX
        mul ElmLn[BP]           ; Визначення зміщення елемента
        push SI                 ; Збереження адреси наступного елемента
        push DI                 ; Збереження адреси аргументу пошуку
        add SI, AX              ; Обчислення адреси середнього елемента
        mov CX, ArgLn[BP]       ; Підготовка довжини аргумента
        repz CMPSB              ; Використання групового ресурсу порівняння
        pop DI          ; Відновлення адреси аргументу пошуку
        jz Found
        pop SI          ; Відновлення адреси наступного елементу
        ja LCorrUp
        pop BX                  ; корекція нижньої межі індексу
        jmp lb
LCorrUp:
        pop TabLn[BP]           ; корекція верхньої межі індексу
```

```
        jmp lb
NotFound:
        mul ElmLn[BP]       ; визначення зміщення елементу
        add AX, SI          ; визначення зміщення в сегменті
        xor DI, DI          ; формування признаку відсутності
        jmp SHORT Lret
Found:
        add SP, 4           ; компенсація записів у стек
Lret:
        mov DX, DS          ; формування сегментної частини вказівника
        pop DS              ; відновлення сегменту даних
        pop BP

        RET
BinSrch ENDP
```

Приклад реалізації двійкового пошуку на Java:

```
public static int binarySearch(int[] a, int key) {
        int low = 0;
        int high = a.length - 1;

        while (low <= high) {
          int mid = (low + high) >>> 1;
          int midVal = a[mid];
          if (midVal < key)
            low = mid + 1;
           else if (midVal > key)
             high = mid - 1;
          else
             return mid; // key found
        }
       return -(low + 1);  // key not found.
    }
```

2. Задачі семантичної обробки

Види семантичної обробки

- Семантичний аналіз
- Генерація машинних кодів
- Інтерпретація внутрішнього подання
- Машиннозалежна оптимізація
- Машинно незалежна оптимізація

Задачі семантичного аналізу-аналіз сумісності операндів окремих операцій, формування типів результатів виконання операцій та формування довжин результатів. Генератор кодів формує команди з дрібних фрагментів команд, операцій, імен або адрес даних, індексних і базових регістрів. Генерація кодів полягає у формуванні окремих машинних команд і зібрання з них об'єктного і загрузочного модулів. В об'єктному коді необов'язково знаходиться весь код програми- в ньому можуть бути зовнішні посилання на бібліотечні процедури або процедури у інших модулях. Генератор кодів формує окремі команди з дрібних фрагментів у формі напрямленого ациклічного графу, для семантично коректної програми, де є достатньо інф. Для інтерпретації(ген. кодів). Для реалізації інтерпретаторів будують віртуальну машину, яка може виконувати усі вузли графу по одному. Для роботи інтерпретатора необхідно додатково виділити пам'ять для зберігання даних, завантажити необхідні константи. При генерації об'єктних кодів виконується напрямлений перегляд ациклічного графу, де генератор породжує команди обробки цих даних. Оптимізація внутр. Подання 2-ма

способами: машинно-залежна(ефективне використання ресурсів цільового компєютера-РОН,ММХ,st[i],головна пам'ять, та накопичувач), машинно-незал.(вилучення повторних обчислень, перероблення блок-схеми програми, оптимізація шляхом еквівалентних переворень, вилучення ділянок програми, які не використ. В кінц. результатах).

Для того щоб створити і використовувати узагальнену програму семантичної обробки необхідно побудувати набори функцій семантичної обробки для кожного з вузлів графа розбору. При семантичному аналізі програма семантичного аналізу повинна перевірити аргументи за таблицею відповідностей аргументів та типів результатів.

При аналізі задач оптимізації можна виділити семантичний аналіз у вигляді термінальних вузлів, або у вигляді обробки нетерм. Вузлів доцільно використю табл. Відповідності аргументів чи операндів разом з полями визначених вихідних типів. Аргументами пошукутакої табл. Буде: внутр. Код(подання операції), тип і довжина 1-гота 2-го операнду. Функціональними полями в табл. Повинні бути тип і спосіб визначення довжини результату. Спосіб обчислення довжини може обчислюватись за допомогою якоїсь процедури обчислення довжини. Кожен алгоритм обробки викристовується і для інших видів семант. Обробки, тобто використ. Та сама рекурсія, але використ. Інші проц. Семант. Обробки.

Для семантичного аналізу програм може бути використана 1 табл. Відповідностей,в якій серед функц. Полів будують тип результату і функції визначення довжини результатута ф-ції інтерпретацій визначення операцій. Для реалізації повного інтерпретатора мови необх. Визначити в табл. Відповідності рядки для кожної з операцій, або фрагмента оператора мови. Найчастіше як внутр. Подання програми використовувались формат польського інверсного запису для виразу або постфіксна форма подання виразу. Деревовидна форма подання , в якій кореневі вузли піддерев визначають обчислення, що виконуються в підпідлеглих структурах. Різновидом таких дерев можна вважати спрямовані ациклічні графи(DAG). Звичайно такі графи мінімізують деревовидні подання, замінюючи повторювані піддерева відповідними посиланнями. В більш ранніх компіляторах часто використовували тетрадні та тріадні подання, які складалися з команд(послідовності команд) віртуальної машини реалізації мови. Тріадні подання включали код операції та адреси або вказівники на 2 операнди.В тетрадних поданнях використовувались 3 окремі посилання: 2-на операнди, а 3-й на результат, тобто вони відповідали 2-х та 3-х адресним командам комп'ютерів попередніх поколінь. Такі форми є залежними від обраної архітектури віртуальних машин. Внутрішні подання Паскалю у формі Р-кодів ближче до 1-адресної машини, а початкова форма більш спрямована до циклічної є взагалі архітектуронезал., атому більш загальною.

3. Способи організації переключення задач

в реальному режимі.

При використанні синхронізації примітивів програми обробки переривань звертаються до функції зміни стану примітиву, щоб далі звернутися до супервізора і змінити стан виконуваної задачі. В більшості ОС розрізняють 4 стани програми:

- активна
- готова – має всі ресурси для виконання, але чекає звільнення процесора
- очікує дані
- призупинена – тимчасово-вилучена з процесу виконання

Задача супервізора полягає у виборі найбільш пріоритетного з числа готових і переведення його в стан готового. Переходи на задачі супервізора можна виконати командами JMP або CALL.

у захищенному режимі

Под *многозадачностью* понимается поочередное выполнение нескольких программ (или фрагментов программ - подпрограмм). Переключение задач осуществляется или программно (командами CALL или JMP), или по прерываниям (например, от таймера). В последнем случае каждой задаче отводится определенный квант времени, после чего управление передается следующей задаче (и так циклически), в результате чего возникает иллюзия того, что все задачи выполняются враз. Задачей может быть как целая программа, так и ее часть (например, некоторая процедура), или программа обработки прерывания. При переключении на выполнение новой задачи процессор сохраняет состояние текущей, с тем, что бы потом возобновить ее выполнение. Основное отличие задач от процедур лишь в том, что о каждой задаче процессор «знает» намного больше чем о процедуре. Для хранения информации о задачи существует специальная структура – *TSS*. В то время как при переходе к выполнению процедуры, процессор запоминает (в стеке) лишь точку возврата (CS:IP).

Поддержка многозадачности обеспечивается за счет следующих аппаратно поддерживаемых структур и элементов:

· Сегмент состояния задачи (TSS).

· Дескриптор сегмента состояния задачи.

· Регистр задачи (TR).

· Дескриптор шлюза задачи.

Для сохранения состояния задачи определена такая структура, как сегмент состояния задачи (TSS - Task State Segment), может располагаться как в отдельном сегменте данных, так и внутри сегмента данных задачи TSS описывается системным дескриптором, который может находиться только в GDT .

Программное переключение

Переключение задач выполняется по инструкции межсегментного перехода *(JMP)* или вызова *(CALL)*. Команды *JMP*, *CALL*, представляют собой инструкции, которые могут быть использованы и при обстоятельствах, не приводящих к переключению задачи. Для того чтобы произошло переключение задачи, команда *JMP* или *CALL* может передать управление либо дескриптору TSS, либо шлюзу задачи. Эффект в обоих случаях одинаковый: процессор передает управление требуемой задаче.

```
JMP  dword ptr adr_sel_TSS(/adr_task_gate)
CALL dword ptr adr_sel_TSS(/adr_task_gate)
```

Операция переключения задач сохраняет состояние процессора (в TSS текущей задачи), и связь с предыдущей задачей (в TSS новой задачи), загружает состояние новой задачи и начинает ее выполнение. Задача, вызываемая по команде JMP, должна заканчиваться аналогичной командой обратного перехода. В случае использования команды CALL - возврат должен происходить по команде IRET, которая сохраняет контекст завершаемой задачи и загружает контекст прерванной.

Алгоритм работы команды IRET в случае возврата из прерывания и в случае обратного переключения задач различен. И определяется значением флага NT (Nested Task).

Если флаг сброшен, то выполняется обычный возврат из прерывания (через стек). Если флаг установлен, то команда IRET инициирует обратное переключение задач.

После загрузки компьютера флаг NT находится в установленном состоянии. Однако любое аппаратное прерывание или исключение сбрасывает этот флаг, в результате чего команда IRET, завершающая обработчик, выполняется в «облегченном» варианте (возврат через стек). То же происходит при выполнении процессором команды программного прерывания INT. Поскольку команда IRET восстанавливает исходное состояние регистра флагов, после завершения обработчика флаг NT снова оказывается установленным (если, конечно, он не был явно сброшен выполняемой программой).

При выполнении процедуры переключения на новую задачу через шлюз или непосредственно через TSS, процессор сохраняет в TSS текущей задачи слово флагов и устанавливает в регистре флагов бит NT. Команда IRET, завершающая задачу, обнаруживает NT=1 и, вместо осуществления возврата через стек, инициирует механизм обратного переключения задач.

Если вложенная задача, в свою очередь, выполняет переключение на следующую задачу, текущее слово флагов с установленным битом NT сохраняется в TSS текущей задачи. После завершения новой задачи это слово будет возвращено в регистр флагов и, таким образом, задача будет продолжаться с NT=1, что обеспечит ее правильное завершение.

Задачи могут иметь произвольную вложенность, можно провести аналогию с процедурам в языках высокого уровня. Да и логика программно переключаемых задач не сильно отличается от процедур. Есть всего одно существенное отличие. После выхода из процедуры точка входа по-прежнему указывает на ее начало. В случае с переключением задач - на следующую, после команды обратного переключения, команду (то есть, возможно, что фактически никуда). Поэтому при повторном вызове задачи необходимо откорректировать ее TSS на предмет корректности точки входа.

При переключении задачи процессор выполняет следующую последовательность действий:

1. Проверяет право на переключение по уровню привилегий CPL < DPL;
2. Сохраняет в TSS исходной задачи ее контекст;
3. Загружает в регистр TR селектор TSS новой задачи;
4. В поле связи TSS новой задачи сохраняется селектор TSS исходной задачи, что обеспечивает возможность будущего обратного переключения. Считывает контекст из TSS новой задачи (в CS:IP появляется точка входа в новую задачу). Флаг NT ¬ 1. Переключение произошло.
5. Когда в новой исполняемой задаче встретится команда IRET, она будет выполняться как обратное переключение задач (NT=1);
6. Контекст текущей задачи сохранится в ее TSS ;
7. В регистр TR загрузится селектор TSS исходной задачи (из поля связи TSS текущей задачи);
8. Регистры восстановится контекст исходной задачи.

Переключение по прерываниям.

Переключение задач программным способом, в большинстве случаев практического применения не находит, в том числе из-за жестко и заранее определенной последовательности выполнения задач. Более гибким является метод переключения задач по прерываниям.

Переключение задач может происходить как по аппаратным, так и программным прерываниям и исключениям. Для этого соответствующий элемент в IDT должен являться *дескриптором шлюза задачи*. *Шлюз задачи* содержит селектор, указывающий на дескриптор TSS (В отличие от дескриптора TSS, который указывает на сегмент, содержащий полное состояние процессора)

Как и при обращении к любому другому дескриптору, при обращении к шлюзу проверяется условие *CPL < DPL*. Стоит также отметить, что шлюз задачи может находиться в любой дескрипторной таблице, а, следовательно, обращение к нему может быть чисто-программное (командами CALL и JMP).

№18

1. Задачі лексичного аналізу

ЛА предназначен для преобразования текста на входном языке во внутреннюю форму, при этом текст разбивается на лексемы. Для решения задачи лексического анализа могут использоваться разные подходы, один из них основан на теории грамматик. К этой задаче можно подойти через классификацию лексем как элементов или типов входных данных. В качестве лексем входного языка обычно объявляют разделители, ключевые слова, имена пользователя, операторы, константы и спец. лексемы. Простейший алгоритм лекс. анализа должен содержать действие продвижения до разделителя, классификацию предшествующей лексемы и разделителя. Определения внутренней формы представления лексемы. Значение констант может отличаться или немного не соответствовать исходным константам из-за точности представления. Обычно устанавливается связь между внутренним представлением лексемы и ее местом во входном тексте. Имена, заданные пользователем, обычно сохраняются в таблицах, а во внутренней форме используется указатель на элемент таблицы и тип. Таблица иногда включает длину и некоторые дополнительные характеристики. Ключевые слова и разделители могут иметь стандартизированную внутреннюю форму представления. Чаще всего удобно построить спец. классификационную таблицу. Входной текст может быть в ASCII (каждый символ 1 байт), в UNICODE (2 байта) и др. Основные классы символов:

1) Вспомогательные разделители (пробел, таб., enter, ...);

2) Одно-символьные операции (+, -, *, /, ..., (,));

3) Много символьные операции (>=, <=, <>...);

4) Буквы, которые можно использовать в именах (латиница):

5) Не классифицируемые символы (для представления чисел или констант необходимо определить цифры и признаки основных систем счисления). При формировании внутреннего представления, кроме кода желательно формировать информацию о приоритете или значении предшествующих операторов.

2. Поняття граматик та їх застосування для розв'язання задач

Языком называется совокупность всех конструкций, которые отвечают грамматике этого языка.

Грамматикой называется четверка G = (N, T, P, S), где N - конечное множество нетерминальных символов (нетерминалов), T - множество терминалов (не пересекающихся с N), S - символ из N, называемый начальным, P - конечное подмножество множества: $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$, называемое множеством правил. Множество правил P описывает процесс порождения цепочек языка. Элемент $p_i = (\square, \square)$ множества P называется правилом (продукцией) и записывается в виде □□□□□. Здесь □□и □ - цепочки, состоящие из терминалов и нетерминалов. Данная запись может читаться одним из следующих способов:

- цепочка □ порождает цепочку □;
- из цепочки □ выводится цепочка□□.

Таким образом, правило P имеет две части: левую, определяемую, и правую, подставляемую. То есть правило p_i - это двойка (p_{i1}, p_{i2}), где $p_{i1} = (N \cup T) N^* (N \cup T)^*$ - цепочка, содержащая хотя бы один нетерминал, $p_{i2} = (N \cup T)^*$ - произвольная, возможно пустая цепочка (□ - цепочка). Если цепочка □ содержит p_{i1}, то, в соответствии с правилом p_i, можно образовать новую цепочку □□ заменив одно вхождение p_{i1} на p_{i2}. Говорят также, что цепочка □ выводится из □ в данной грамматике. Для описания абстрактных языков в определениях и примерах будем пользоваться следующими обозначениями:

- терминалы обозначим буквами a, b, c, d или цифрами 0, 1, ..., 9;
- нетерминалы будем обозначать буквами A, B, C, D, S (причем нетерминал S - начальный символ грамматики);

- буквы U, V, ..., Z используем для обозначения отдельных терминалов или нетерминалов;
- через □, □, □... обозначим цепочки терминалов и нетерминалов;
- u, v, w, x, y, z - цепочки терминалов;
- для обозначения пустой цепочки (не содержащей ни одного символа) будем использовать знак □;
- знак "□" будет отделять левую часть правила от правой и читаться как "порождает" или "есть по определению". Например, A□cd, читается как "A порождает cd".

Эти обозначения определяют некоторый язык, предназначенный для описания правил построения цепочек, а значит, для описания других языков. Язык, предназначенный для описания другого языка, называется метаязыком. Пример грамматики G1: G1 = ({A, S}, {0, 1}, P, S), где P: 1) S □ 0A1; 1) 0A □ 00A1; 2) A □□□.

Выводимая цепочка грамматики G, не содержащая нетерминалов, называется терминальной цепочкой, порождаемой грамматикой G. Язык L(G), порождаемый грамматикой G, - это множество терминальных цепочек, порождаемых грамматикой G.

Несмотря на большое разнообразие грамматик, при построении трансляторов нашли широкое применение только ряд из них, имеющих некоторые ограничения. Это связано с практической целесообразностью использования определенных типов правил, так как сложность их построения непосредственно влияет на сложность построения трансляторов. По виду правил выделяют несколько классов грамматик.

В соответствии с классификацией Хомского грамматика **G** называется:

- **праволинейной**, если каждое правило из **P** имеет вид: **A□xB** или **A□x**, где **A**, **B** - нетерминалы, **x** - цепочка, состоящая из терминалов;
- **контекстно-свободной** (КС) или **бесконтекстной**, если каждое правило из **P** имеет вид: A□□, где A □ N, a □ □□(N □ T)*, то есть является цепочкой, состоящей из множества терминалов и нетерминалов, возможно пустой;
- **контекстно-зависимой** или **неукорачивающей**, если каждое правило из **P** имеет вид: □ □ □, где □□□□□□□□□□ То есть, вновь порождаемые цепочки не могут быть короче, чем исходные, а, значит, и пустыми (другие ограничения отсутствуют);
- **грамматикой свободного вида**, если в ней отсутствуют выше упомянутые ограничения.

Пример праволинейной грамматики: G2 = ({S,}, {0,1}, P, S), где P:
1) S □ 0S; 2) S □ 1S; 3) S □□□, определяет язык {0, 1}*.
Пример КС-грамматики: G3 = ({E, T, F}, {a, +, *, (,)}, P, E) где P:
1) E □T; 2) E □ E + T; 3) T □ F; 4) T □ T * F; 5) F □ (E); 6) F □ a.
Данная грамматика порождает простейшие арифметические выражения.
Пример КЗ-грамматики: G4 = ({B, C, S}, {a, b, c}, P, S) где P:
1) S □ aSBC; 2) S □ abc; 3) CB □ BC; 4) bB □ bb; 5) bC □ bc; 6) cC □ cc, порождает язык { $a^n b^n c^n$ }, $n \geq 1$.
Примечание 1. Согласно определению каждая праволинейная грамматика является контекстно-свободной.
Примечание 2. По определению КЗ-грамматика не допускает правил: A □ □□ где □ - пустая цепочка. Т.е. КС-грамматика с пустыми цепочками в правой части правил не является контекстно-зависимой. Наличие пустых цепочек ведет к грамматике без ограничений. Соглашение. Если язык L порождается грамматикой типа G, то L называется языком типа G. Пример: L(G3) - КС язык типа G3. Наиболее широкое применение при разработке трансляторов нашли КС-грамматики и порождаемые ими КС языки.

3. Типи ОС та режими їх роботи

Більшість ранніх ОС були орієнтовані на так звану пакетну обробку задач за принципом FIFO.

Потім виникли багатозадачні, або потокові системи, але вони були організовані так, що в системі можна було виконувати лише обмежену кількість задач. Для кожного класу або типу окрема черга. Тобто задачі могли витісняти вже готові до виконання задачі.

При створенні ОС реального часу та багатопоточнних почали використовувати так звану «витісняючи багатозадачність». В системах реального часу найбільші пріоритети надавалися найбільш важливим задачам

У системах розділення часу пропонується рівноправне обслуговування всіх процесів по черзі, щоб користувач бачив просування своєї задачі. Для цього кожному користувачу по черзі надається квант процесорного часу, за який відбувається виконання якоїсь частки задачи. І маємо комбінацію різних способів управляння задачами.

В 1950-60-х годах сформировались и были реализованы основные идеи, определяющие функциональность ОС: пакетный режим, разделение времени и многозадачность, разделение полномочий, реальный масштаб времени, файловые структуры и файловые системы.

Разделение полномочий

Распространение многопользовательских систем потребовало решения задачи разделения полномочий, позволяющей избежать возможности модификации исполняемой программы или данных одной программы в памяти компьютера другой (содержащей ошибку или злонамеренно подготовленной) программы, а также модификации самой ОС прикладной программой.
Реализация разделения полномочий в ОС была поддержана разработчиками процессоров, предложивших архитектуры с двумя режимами работы процессора — «реальным» (в котором исполняемой программе доступно всё адресное пространство компьютера) и «защищённым» (в котором доступность адресного пространства ограничена диапазоном, выделенном при запуске программы на исполнение).
Режимы:

Защищённый режим

Основная мысль сводится к формированию таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п. При нехватке памяти операционная система может выгрузить часть данных из оперативной памяти на диск, а в таблицу описаний внести указание на отсутствие этих данных в памяти. При попытке обращения к отсутствующим данным процессор сформирует исключение (разновидность прерывания) и отдаст управление операционной системе, которая вернёт данные в память, а затем вернёт управление программе. Таким образом для программ процесс подкачки данных с дисков происходит незаметно.

Реальный режим

В этом режиме процессоры работали только в старых версиях DOS. Адресовать в реальном режиме дополнительную память за пределами 1 Мб было нельзя.

До відома: впоследствии, для полного отказа от реального режима, в защищённый режим был введён ещё один специальный режим виртуальных адресов V86. При этом программы получают возможность использовать прежний способ вычисления линейного адреса, не выходя из защищённого режима процессора.

Реальный масштаб времени

Применение универсальных компьютеров для управления производственными процессами потребовало реализации «реального масштаба времени» («реального времени») — синхронизации исполнения программ с внешними физическими процессами.
Включение функции реального масштаба времени в ОС позволило создавать системы, одновременно обслуживающие производственные процессы и решающие другие задачи (в пакетном режиме и (или) в режиме разделения времени).

БИЛЕТ 19

1.Системні управляючі програми

Системная программа - программа общего пользования, выполняемая вместе с прикладными программами и служащая для управления ресурсами компьютера: центральным процессором, памятью, вводом-выводом.
Системная программа - согласно ГОСТ 197881-90 - программа, предназначенная:
- для поддержания работоспособности системы обработки информации; или
- для повышения эффективности ее использования.
Различают системные управляющие и системные обслуживающие программы.

Системні управляючі програми призначені для виконання та управління в ОС. Ці програми були призначені для введення завантаження програм в комп'ютер; введення/виведення інформації на зовнішні пристрої; управління подіями, що виникають в обчислювальних пристроях; розподілу ресурсами; управління доступом і захистом інформації в комп'ютері.
Управляющие системные программы организуют корректное функционирование всех устройств системы. Системные управляющие программы составили основу операционных систем и в таком виде и в таком виде используются и в настоящее время. Основные системные функции управляющих программ:

- управление вычислительными процессами и вычислительными комплексами и
- работа с внутренними данными ОС.

До системних управляючих програм відносять всі програми ОС:

- Програми початкового завантаження
- Програми попереднього програмування та контролю обладнання
- Програми управляння файловою системою
- Програми переключення задач

Как правило, они находятся в основной памяти. Это резидентные программы, составляющие ядро ОС. Управляющие программы, которые загружаются в память непосредственно перед выполнением, называю транзитными (transitive). В настоящее время системные управляющие программы поставляются фирмами-разработчиками и фирмами-дистрибьюторами в виде инсталляционных пакетов операционных систем и драйверов специальных устройств.

2. Алгоритм Низхідного синтаксичного розбору

Нисходящий разбор заключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в заполнении промежутка между начальным нетерминалом и символами входной цепочки правилами, выводимыми из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом. Подставляемое правило в общем случае выбирается произвольно. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил. В последнем случае входная цепочка отвергается. Построение дерева разбора подтверждает принадлежность входной цепочки данному языку. При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что грамматика данного языка является недетерминированной. Эти рассуждения иллюстрируются следующим примером. Пусть будет дана грамматика G: $\mathbf{G_6 = (\{S\}, \{a, +, *\}, P, S)}$, где P определяется как: 1) S **(ρ)** a; 2) S **(ρ)** S + S; 3) S **(ρ)** S * S
Цепочки, порождаемые данной грамматикой можно интерпретировать как выражения, состоящие из операндов **"a"**, а также операций **"+"** и **"*"**. Недетерминированность грамматики позволяет порождать одну и ту же терминальную цепочки с использованием различных выводов. Например, выражение **"a+a*a+a"** можно получить следующими способами:

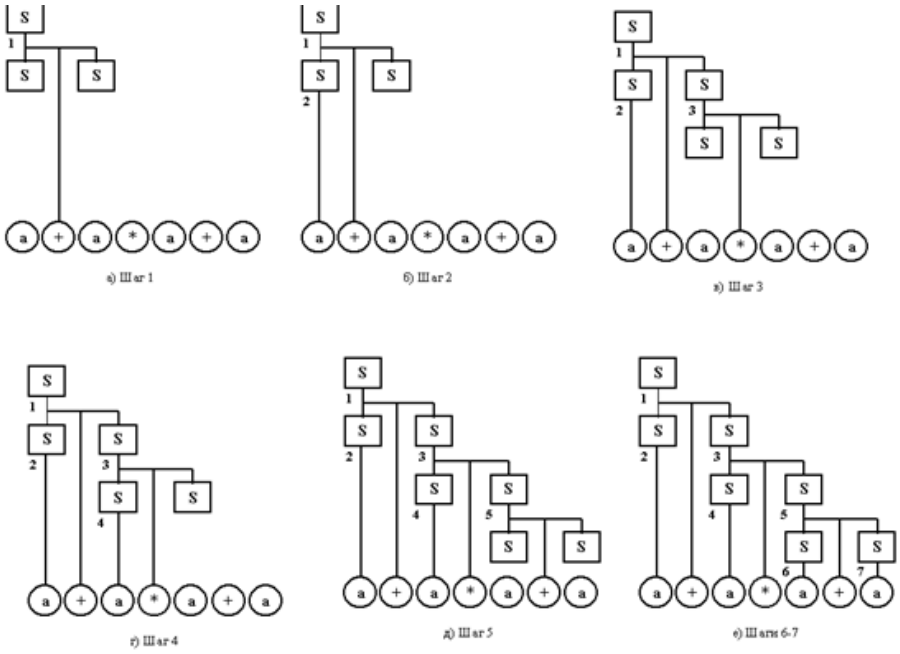


Рис. 6.2. Нисходящий разбор слева направо.

- S ⇒ S+S ⇒ a+S ⇒ a+S*S ⇒ a+ a*S ⇒ a+a*S+S ⇒ a+a*a+S ⇒ a+a*a+a
- S ⇒ S+S ⇒ S+a ⇒ S*S+a ⇒ S*a+a ⇒ S+S*a+a ⇒ S+a*a+a ⇒ a+a*a+a (6.1)
- S ⇒ S*S ⇒ S+S*S ⇒ S+S*S+S ⇒ a+ S*S+S ⇒ a+a*S+S ⇒ a+a*S+a ⇒ a+a*a+a

И так далее. В этом пример число вариантов одной и той же произвольной цепочки вывода настолько велико, что не имеет и смысла говорить о практическом применении данной грамматики. Но в данном случае она позволяет показать, каким образом могут порождаться различные деревья при нисходящем разборе. Пошаговое построение различных деревьев показано на рис. Можно отметить, что процесс построения дерева совпадает с последовательностью шагов вывода входной цепочки.

(по конспекту)

Метод низхідного розбору в якому аналіз конструкції викнується починаючи від найбільш складних конструкції доо термінальних позначень. В цьому випадку за базу можуть бути взяті правила підстановки. Один з найпоширеніших – метод рекурсивного спуску. В цьому методі розбір починається з кінцевого позначення граматики. При виконанні такого розбору аналізатор звертається до підлеглого ресурсу, щоб розібрати спочатку перший а потім наступні позначення правої частини правила підстановки.Рекурсивні правила у формі Бекуса прийнято, так що рекурсивні звертання записуються з правого боку, що утворює ліворекурсивні правила. Однак при такому розборі ми будмо просуватись в глибину рекурсії , не просуваючись вздовж вхідного потоку даних, що фактично призводить до за циклювання аналізатора. Тому для використованого методу рекурсивного спуску необхідно перетворити правила на право рекурсивну форму. Альтернативним методом є метод синтаксичних графів.

```
Number>::=[<Sign>]<digit>{<digit>}[<Separator><digit>{<digit>}]
          [<Exponent>[<Sign>]<digit>{<digit>}]
<digit>::='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<Sign>::='+'|'-'
<Separator>::='.'
<Exponent>::='E'|'e'
```

Теперь на основе этих правил напишем функцию IsNumber, которая в качестве параметра принимает строку и возвращает True, если эта строка удовлетворяет правилам записи числа, и False, если не удовлетворяет.

```
// Проверка символа на соответствие <digit>
function IsDigit(Ch:Char):Boolean;
 begin
  Result:=Ch in ['0'..'9']
 end;

// Проверка символа на соответствие <Sign>
function IsSign(Ch:Char):Boolean;
 begin
  Result:=(Ch='+') or (Ch='-')
 end;

// Проверка символа на соответствие <Separator>
function IsSeparator(Ch:Char):Boolean;
 begin
  Result:=Ch='.'
 end;

// Проверка символа на соответствие <Exponent>
function IsExponent(Ch:Char):Boolean;
 begin
  Result:=(Ch='E') or (Ch='e')
 end;

function IsNumber(const S:string):Boolean;
 // Номер символа выражения, который сейчас проверяется
 var P:Integer;
  begin
   Result:=False;
   // Проверка, что выражение содержит хотя бы один символ.
   // пустая строка не является числом
   if Length(S)=0 then
    Exit;
   // Начинаем проверку с первого символа
   P:=1;
   // Если первый символ - <Sign>, переходим к следующему
   if IsSign(S[P]) then
    Inc(P);
```

```
  // Проверяем, что в данной позиции стоит хотя бы одна цифра
  if (P>Length(S)) or not IsDigit(S[P]) then
   Exit;
  // Переходим к следующей позиции, пока не достигнем
  // конца строки или не встретим не цифру
  repeat
   Inc(P)
  until (P>Length(S)) or not IsDigit(S[P]);
  // Если достигли конца строки, выражение корректно - число,
  // не имеющее дробной части и экспоненты
  if P>Length(S) then
   begin
    Result:=True;
    Exit
   end;
  // Если следующий символ - <Separator>, проверяем,
  // что после него стоит хотя бы одна цифра
  if IsSeparator(S[P]) then
   begin
    Inc(P);
    if (P>Length(S)) or not IsDigit(S[P]) then
     Exit;
    repeat
     Inc(P)
    until (P>Length(S)) or not IsDigit(S[P]);
    // Если достигли конца строки, выражение корректно - число
    // без экспоненты
    if P>Length(S) then
     begin
      Result:=True;
      Exit
     end
   end;
  // Если следующий символ - <Exponent>, проверяем,
  // что после него стоит всё то, что требуется правилами
  if IsExponent(S[P]) then
   begin
    Inc(P);
    if P>Length(S) then
     Exit;
    if IsSign(S[P]) then
     Inc(P);
    if (P>Length(S)) or not IsDigit(S[P]) then
     Exit;
    repeat
     Inc(P)
    until (P>Length(S)) or not IsDigit(S[P]);
    if P>Length(S) then
     begin
      Result:=True;
      Exit
     end
   end
  // Если выполнение дошло до этого места, значит,
  // в выражении остались ещё какие-то символы. Т.к. никакие
  // дополнительные символы синтаксисом не предусмотрены,
  // такое выражение не считается корректным числом.
 end;
```

Для каждого нетерминального символа мы ввели отдельную функцию, разбор начинается с символа самого верхнего уровня - <Number> - и следует правилам, записанным для этого символа. Такой способ синтаксического анализа называется левосторонним рекурсивным нисходящим анализом. Левосторонним потому, что символы в выражении перебираются слева направо, нисходящим - потому, что сначала анализируются символы верхнего уровня, а потом - символы нижнего. Рекурсивность метода на данном примере не видна, т.к. наша грамматика не содержит рекурсивных определений, но мы с этим столкнёмся в последующих примерах.

3. Способи переключення задач

Організація переключення задач

В процесорі Пентіум передбачено спеціальні механізми переключення задач. В реальному режимі або режимі ДОС при створенні нової задачі або процесу (потоку) для нього створюється окремий стек в якому зберігаються зміст регістрів переривань задачі, які часто називають надтекстом задачі, однак в реальному режимі при обробці переривань. програми можуть використовувати для своєї роботи фрагменти стеку перерваної задачі, що призводить до проблем захисту цілісності кодів виконуваних задач. Щоб усунути проблеми захисту і узагальнити процеси переключення задач важливо використовувати той самий сегмент стану TSS. Крім регістрів в сегменті TSS записується інформація про стан об'єктів введення-виведення у спеціальному розширенні сегмента. Якщо в задачі використовується регістр з плаваючою точкою, то їх збереження покладається на відповідальність програміста, залежно від того як використовуватиме регістр з плаваючою точкою задача що переривається і задача що надходить на виконання.

Для того щоб перейти до кодів, які пов'язані з сегментом перерваної задачі необхідно спочатку підготувати в спеціальному регістрі задач номер відповідної задачі, а потім необхідно виконати команду переходу jmp, call на відповідний сегмент задачі. При цьому в старому сегменті задачі будуть збережені сегментні регістри та регістри загального користування, а з нового сегменту TSS будуть відновлені регістри нової задачі на яку відбувається переключення.

Для організації переключення задач за такою схемою для кожної задачі (в тому числі і для системних) потрібно побудувати принаймі один сегмент статусу задач TSS. Якщо для виконання якоїсь задачі передбачено виконувати підзадачі, які називаються в більшості ОС потоками (thread), які на відміну від процесів (process) виконуються в тому ж адресному просторі, що і головні процеси задач. Таким чином для потоків треба також створювати сегмент TSS, але в ни буде зберігатися інформація про однакові таблиці сегментів і сегментні регістрів, але різна інформація про уточнюючі значення регістрів.

В захищеному режимі selector включає 13 бітовий номер дескриптора в локальну або глобальну таблицю дескрипторів в молодшому біті ознака локальної(1) або глобальної (0) таблиці дескрипторів і ще 2 біта необхідні при встановленні доступу. Сам дескрипторний сегмент задачі включає початкову адресу сегментам (32), довжину сегмента (20) а всі інші біти для управління інформацією. Крім того у дескрипторі зберігається біт зайтяності сегмента

Білет 20

1) Основні способи організації пошуку в таблицях

Один из наиболее распространенных линейный по ключу в неупорядоченных структурах данных. При реализации такого алгоритма ключевой аргумент поиска должен поочередно сравниваться со значениями ключей в элементах или рядах таблиц. В компиляторах, как правило, используется поиск по однозначным ключам, поэтому результат поиска можно получить при первом совпадении аргумента с ключом. Для ускорения поиска выполняется два дополнительных подхода – упорядочивание и поиск по прямому адресу, который может быть выполнен в условиях небольшого диапазона значений аргументов поиска. Например, для классификатора букв в лексическом анализаторе, если буквы заданны в Аски-коде достаточно значений 0..255. Чтоб засечь символ в AL необходимо

```
Mov al, tab[eax]

Xlat
```

Алгоритм линейного поиска в неупорядоченной таблице

11. Установка индекса первого элемента таблицы.
12. Сравнение ключа искомого элемента с ключом элемента таблицы.
13. Если ключи равны,перейти на обработку ситуации "поиск удачный",иначе на 4.
14. Инкремент индекса элемента таблицы.
15. Проверка конца таблицы.

```
BINSRCHARG STRUC            ; структура для передачі параметрів
           dd    ?          ; проміжок для BP та адреси повернення
ProtPtr    dd    ?          ; вказівник елементу, що перевіряється
ArgLn      dw    ?          ; довжина елементу, що перевіряється
```

```
TabPtr dd       ?       ; вказівник початку таблиці
TabLn           dw      ?       ; кількість елементів таблиці
ElmLn           dw      ?       ; довжина елементу таблиці
BINSRCHARG ENDS

public BinSrch
BinSrch         PROC
        push BP                 ; збереження старого BP
        mov BP, SP              ; Визначення бази зони параметрів
        push DS                 ; збереження сегменту даних
        les DI, ProtPtr[BP]     ; підготовка адреси аргументу
        LD6 SI, TabPtr[BP]      ; підготовка адреси таблиці
        xor BX, BX              ; встановлення початкової нижньої межі

lb:
        add AX, TabLn[BP]       ; Обчислення
        shr AX, 1               ; номеру середнього аргументу
        cmp AX, BX              ; Контроль границі розбивання
        jz NotFound             ; Перехід, якщо таблиця переглянута
        push AX
        mul ElmLn[BP]           ; Визначення зміщення елемента
        push SI                 ; Збереження адреси наступного елемента
        push DI                 ; Збереження адреси аргументу пошуку
        add SI, AX              ; Обчислення адреси середнього елемента
        mov CX, ArgLn[BP]       ; Підготовка довжини аргумента
        repz CMPSB              ; Використання групового ресурсу порівняння
        pop DI          ; Відновлення адреси аргументу пошуку
        jz Found
        pop SI          ; Відновлення адреси наступного елементу
        ja LCorrUp
        pop BX                  ; корекція нижньої межі індексу
        jmp lb
LCorrUp:
        pop TabLn[BP]           ; корекція верхньої межі індексу
        jmp lb
NotFound:
        mul ElmLn[BP]           ; визначення зміщення елементу
        add AX, SI              ; визначення зміщення в сегменті
        xor DI, DI              ; формування признаку відсутності
        jmp SHORT Lret
Found:
        add SP, 4               ; компенсація записів у стек
Lret:
        mov DX, DS              ; формування сегментної частини вказівника
        pop DS                  ; відновлення сегменту даних
        pop BP

        RET
BinSrch ENDP
```

Приклад реалізації двійкового пошуку на Java:

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;
```

```
    while (low <= high) {
      int mid = (low + high) >>> 1;
      int midVal = a[mid];
      if (midVal < key)
        low = mid + 1;
       else if (midVal > key)
         high = mid - 1;
      else
         return mid; // key found
    }
   return -(low + 1);  // key not found.
  }
```

Вопрядкування таблиць і методи двійного пошуку

1. Загрузка нач. (Ан) и кон. (Ак) адресов таблицы
2. Определение адреса ср. элемента таблицы (Аср)
3. Сравнение искомого ключа с ключевой частью ср. элемента таблицы
4. При равенстве ключей поиск удачный. Если искомый ключ меньше ключа среднего элемента, то Ак=Аср, переход на 5.Если искомый ключ больше ключа среднего элемента,то Ан=Аср+длина элемента,переход на 5
5. Сравнение Ан и Ак. Если Ан=Ак, поиск неудачный, иначе переход на 2

При определении адреса среднего элемента следует выполнить следующие действия:

- вычислить длину таблицы Ак-Ан
- определить число элементов таблицы (Ак-Ан)/Lэ, где Lэ- длина элемента
- определить длину половины таблицы ((Ак-Ан) /Lэ)/2*Lэ
- определить адрес среднего элемента таблицы Аср= Ан + ((Ак-Ан)/Lэ)/2*Lэ

Пошук за прямою адресою.

Поиск в древовидных структурах осуществляется рекурсивно. Необходимо составить процедуру, в которую, как параметры передаются ссылка на элемент дерева и ключ поиска. Процедура при начальной инициализации получает ссылку на вершину дерева. В теле процедуры проверяется равенство ключа поиска с ключом элемента дерева, если равенство выполняется, то считываются значения всех полей данного элемента дерева. За тем процедура вызывает сама себя с указанием ссылок из текущего элемента. Таким образом осуществляется столько вызовов процедуры, сколько ветвей у данного элемента дерева. Такой алгоритм позволяет пройти по всем элементам неоднородного дерева.

Хеш пошуку.

Алгоритм:

7. Формирование хеш-таблицы
8. Выбор искомого ключа
9. Вычисление хеш-функции ключа H(Ki)
10. Вычисление хеш-адреса ключа
11. Сравнение ключевой части элемента таблицы по вычисленному хеш-адресу с искомым ключом. При равенстве обработка ситуации "поиск удачный "и переход на 6; при неравенстве - обработка ситуации "поиск неудачный" и переход на 6.

12. Проверка все ли ключи выбраны; если да, то конец, а если нет ,то переход на 2.

2) Метод синтаксичних графів для синтаксичного аналізу

В цьому методі послідовність правил синтаксичного розбору організована у формі графа, кожен вузол якого має 4 складових :

5. **мітка, що призначена для зв'язування вузлів графа**
6. **розпізнавач, який може розпізнати термінальні або не термінальні елементи синтаксичної конструкції для синтаксичного аналізу в компіляторах. За термінальні позначення обирають імена, константи, ключові слова, тощо. За нетермнальні – вирази, списки, оператори, тощо.**
7. **покажчик на наступний вузол графу, до якого відбувається перехід у випадку успішної роботи розпізновача**
8. **показник на альтернативне відгалуження синтаксичного графу, коли робота розпізнувача не буде успішною. По формі структур даних синтаксичний граф може бути представлений як сукупність вузлів з двійковим розг. У випадку коли синтаксична конструкція виявиться неправильною необхідно передбачити нейтралізацію помилок, тобто видачу діагностики з пошуком конструкції після якої можна продовжити синтаксичний аналіз.**

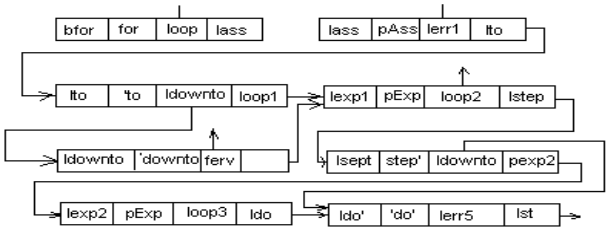
Результати синтаксичного розбору подаються у вигляді зв'язаних вузлів графа розбору, в якому кожний вузол відповідає окремій лексемі, а зв'язки визначають підлеглість операндів до операцій. Вузол об'єднує 4 поля:

- Ідентифікація або мітка вузла
- Прототип співставлення (термінал або нетермінал)
- Мітка продовження обробки (при успішному результаті синтаксичного аналізу)
- Вказівник на альтернативну вітку, яку можна перевірити (якщо аналіз був невдачним)

Синтаксично буває зручним використовувати змішані алгоритми аналізу. Як раз для обробки операторів мов програмування краще використовувати синтаксичний граф, а для обернених виразів – висхідний розбір.

Також існують зв'язки передачі управління в операторах розгалуження та блоках циклів та варіантному блоці. Таким чином будь-який закінчений фрагмент програми має головний вузол, в якому як підлеглі вузли різних рівнів зберігаються лексеми та синтаксичні структури, які входять до цього блоку. Такий граф розбору є основою для всіх видів подальшої семантичної обробки.

Методи низхідного розбору ефективніші при обробці структурованих операторів(for)



3) Організація генерації кодів

Для генерації машинних кодів слід використовувати машинні команди або підпрограми з відповідними аргументами. В результаті генерації кодів формуються машинні команди або послідовності виклику підпрограм або функцій, які повертають потрібний результат. Більшість компіляторів системних програм включає такі коди в об'єктні файли з розширенням OBJ. Такі файли включають 4 групи записів:

- Записи двійкового коду – двійкові коди констант, машинні коди, відносні адреси відносно початку відповідного сегменту
- Записи (елементи) переміщуванності – спосіб настроювання відповідної відносної адреси
- Елементи словника зовнішніх посилань – фіксуються імена, які заявлені як зовнішні або доступні для зовнішніх посилань
- Кінцевий запис модуля – для розділення модулю

Генерація кодів в цілому являє собою формування машинних команд і збірку об'єктного і загрузочного модулів. При генерації об'єктних кодів виконується направлений прохід по графу. Для реалізації такого генератора потрібно визначити повні таблиці співставлення усіх можливих вузлів графа з потрібним набором машинних команд.

Більш ранні компілятори одразу формувалимашинні коди. Компілятори з мови Паскаль формували свої результати у так званих Р-кодах. На етапі виконання цей код оброблювався середовищем Паскаль, яке включало підпрограми для обробки всіх операндів та операторів. Ця схема є проміжною між копіюючою та інтерпретуючою програмами. Тобто за такою схемою для даних виділяють як правило фіксовану пам'ять, а до кодів звертається виконуюча програма, яка дозволяє формувати результат виконання порграми. Але щоб раціонально організовувати модульне програмування необхідно, щоб поєднувані модулі подавалися в однаковому форматі, тому в традиційн. Реалізаціях Паскаля вони добре поєднувалися з модулями на цій же мові, але погано з іншими мовами. Результати трансляції формуються у вигляді obj файлів. В них використ. 4 загальні типи записів або елементів. Генератор кодів формує команди з дрібних фрагментів команд, операцій, імен або адрес даних, індексних і базових регістрів. У формі напрямленого ациклічного графу для семантично коректної роботи програми є достатньо інф. Для її інтерпретації(ген. кодів). Генерація кодів складається з формування окремих машинних команд і зборки з них об'єктного і завантажувального модулів. Для виклику стандартних підпрограм, що використовуються у мові використовують бібліотеки періоду виконання, яка може співпадати з бібліотекою інтерпретатора. При генерації кодів виконується напрямлений перегляд ациклічного графу, де генератор породжує команди обробки цих даних. Для реалізації генераторів необхідно визначити повні табл. Відповідності усіх можливих вузлів напрямленого ациклічного графу у необх. Набор машинних команд

БИЛЕТ 22

1. Класифікація граматик за Хомським

. По виду правил выделяют несколько классов грамматик. В соответствии с классификацией Хомского грамматика **G** называется:

- **праволинейной**, если каждое правило из **P** имеет вид: **A→xB** или **A→x**, где **A**, **B** - нетерминалы, **x** - цепочка, состоящая из терминалов;
- **контекстно-свободной** (КС) или **бесконтекстной**, если каждое правило из **P** имеет вид: $A \to \alpha$, где $A \in N$, а $\alpha \in (N \cup T)^*$, то есть является цепочкой, состоящей из множества терминалов и нетерминалов, возможно пустой;
- **контекстно-зависимой** или **неукорачивающей**, если каждое правило из **P** имеет вид: $\alpha \to \beta$, где $|\alpha| \le |\beta|$. То есть, вновь порождаемые цепочки не могут быть короче, чем исходные, а, значит, и пустыми (другие ограничения отсутствуют);
- **грамматикой свободного вида**, если в ней отсутствуют выше упомянутые ограничения.

Пример праволинейной грамматики: G2 = ({S,}, {0,1}, P, S), где P:
1) $S \to 0S$; 2) $S \to 1S$; 3) $S \to \varepsilon$, определяет язык {0, 1}*.
Пример КС-грамматики: G3 = ({E, T, F}, {a, +, *, (,)}, P, E) где P:
1) $E \to T$; 2) $E \to E + T$; 3) $T \to F$; 4) $T \to T * F$; 5) $F \to (E)$; 6) $F \to a$.
Данная грамматика порождает простейшие арифметические выражения.
Пример КЗ-грамматики: G4 = ({B, C, S}, {a, b, c}, P, S) где P:
1) $S \to aSBC$; 2) $S \to abc$; 3) $CB \to BC$; 4) $bB \to bb$; 5) $bC \to bc$; 6) $cC \to cc$, порождает язык $\{ a^n b^n c^n \}$, $n \ge 1$.
Примечание 1. Согласно определению каждая праволинейная грамматика является контекстно- свободной.
Примечание 2. По определению КЗ-грамматика не допускает правил: $A \to \varepsilon$, где ε - пустая цепочка. Т.е. КС-грамматика с пустыми цепочками в правой части правил не является контекстно-зависимой. Наличие пустых цепочек ведет к грамматике без ограничений. Соглашение. Если язык L порождается грамматикой типа G, то L называется языком типа G. Пример: L(G3) - КС язык типа G3. Наиболее широкое применение при разработке трансляторов нашли КС-грамматики и порождаемые ими КС языки.

2. Особливості математичних операцій MMX-розширення

Розширення архітектури MMX - основа аппаратной компоненты расширения mmx – восемь новых регистров mm0..mm7, которые на самом деле являются регистрами сопроцессора, только вместо 80-ти разрядов используется 64 младших разряда (мантисса). При работе со стеком сопроцессора в режиме mmx он рассматривается как обычный массив регистров с произвольным доступом. Нельзя одновременно пользоваться командами для работы с числами с плавающей запятой и командами MMX, а если это необходимо — следует пользоваться командами FSAVE/FRSTOR, каждый раз перед переходом от использования FPU к MMX и обратно (эти команды сохраняют состояние регистров MMX точно так же, как и FPU). Основным принципом работы **команд mmx** является одновременна обработка нескольких единиц однотипных данных одной командой.

Из инета

Для быстрого снабжения конвейеров командами и данными из памяти шина данных этих процессоров сделана 64-разрядной, из-за чего их первое время иногда ошибочно называли 64-разрядными процессорами. «На закате» этого поколения появилось расширение MMX (Matrics Math Extensions {instruction set} – набор команд для расширения матричных математических операций (первоначально Multimedia Extension {instruction set} – набор команд для мультимедиа-расширения)). Традиционные 32-разрядные процессоры способны выполнять сложение двух 8-разрядных чисел, размещая каждое из них в младших разрядах 32-разрядных регистров. При этом 24 старших разряда регистров не

употребляются, и потому, например, при одной операции сложения ADD осуществляется просто сложение двух 8-разрядных чисел. Команды MMX оперируют сразу с 64 разрядами, где могут храниться восемь 8-разрядных чисел, причем имеется возможность выполнить их сложение с другими 8-разрядными числами в процессе одной операции ADD. Регистры MMX могут употребляться также для одновременного сложения четырех 16-разрядных слов или двух 32-разряных длинных слов. Такой принцип получил название SIMD (Single Instruction/Multiple Data - «один поток команд/много потоков данных») (2-4). Новые команды были предназначены в первую очередь для ускорения выполнения мультимедиа программ, но применять их можно не только к задачам, прямо связанным с технологией мультимедиа. В MMX появился и новый тип арифметики - с насыщением: если результат операции не помещается в разрядной сетке, то переполнения (или «антипереполнения») не происходит, а устанавливается максимально (или минимально) возможное значение числа.

PADDB приемник,источник – сложение байт
PADDW приемник,источник – сложение слов
PADDD приемник,источник – сложение двойных слов

Команды выполняют сложение отдельных элементов данных (байт — для PADDB, слов — для PADDW, двойных слов — для PADDD) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при сложении возникает перенос, он не влияет ни на следующие элементы, ни на флаг переноса, а просто игнорируется (так что, например, для PADDB 255 + 1 = 0, если это числа без знака, или -128 + -1 = +127, если со знаком).

PADDSB приемник,источник – сложение байт с насыщением
PADDSW приемник,источник – сложение слов с насыщением

Команды выполняют сложение отдельных элементов данных (байт — для PADDSB и слов — для PADDSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если результат сложения выходит за пределы байта со знаком для PADDSB (больше +127 или меньше -128) или слова со знаком для PADDSW (больше +32 767 или меньше -32 768), в качестве результата используется соответствующее максимальное или минимальное число, так что, например, для PADDSB -128 + -1 = -128.

PADDUSB приемник, источник – беззнаковое сложение байт с насыщением
PADDUSW приемник, источник – беззнаковое сложение слов с насыщением

Команды выполняют сложение отдельных элементов данных (байт — для PADDUSB и слов — для PADDUSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если результат сложения выходит за пределы байта без знака для PADDUSB (больше 255 или меньше 0) или слова без знака для PADDUSW (больше 65 535 или меньше 0), в качестве результата используется соответствующее максимальное или минимальное число, так что, например, для PADDUSB 255 + 1 = 255.

PSUBB приемник, источник – вычитанние байт
PSUBW приемник, источник – вычитанние слов
PSUBD приемник, источник – вычитанние двойных слов

Команды выполняют вычитание отдельных элементов данных (байт — для PSUBB, слов — для PSUBW, двойных слов — для PSUBD) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если при вычитании возникает заем, он игнорируется (так что, например, для PSUBB -128 - 1 = +127 — для чисел со знаком или 0 - 1 = 255 — для чисел без знака).

PSUBSB приемник, источник – вычитание байт с насыщением
PSUBSW приемник, источник – вычитание слов с насыщением

Команды выполняют вычитание отдельных элементов даннях (байт — для PSUBSB и слов — для PSUBSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если результат вычитания выходит за пределы байта или слова со знаком, в качестве результата используется соответствующее максимальное или минимальное число, так что, например, для PSUBSB -128 - 1 = -128.

PSUBUSB приемник, источник – беззнаковое вычитание байт с насыщением
PSUBUSW приемник, источник – беззнаковое вычитание слов с насыщением

Команды выполняют вычитание отдельных элементов даннях (байт — для PSUBUSB и слов — для PSUBUSW) источника (регистр MMX или переменная) и соответствующих элементов приемника (регистр MMX). Если результат вычитания выходит за пределы байта или слова без знака, в качестве результата используется соответствующее максимальное или минимальное число, так что, например, для PSUBUSB 0 - 1 = 0.

PMULHW приемник, источник – старшее умножение

Команда умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Старшее слово каждого из результатов записывается в соответствующую позицию приемника.

PMULLW приемник, источник – младшее умножение

Умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Младшее слово каждого из результатов записывается в соответствующую позицию приемника.

PMADDWD приемник, источник – умножение и сложение

Умножает каждое из четырех слов со знаком из источника (регистр MMX или переменная) на соответствующее слово со знаком из приемника (регистр MMX). Произведения двух старших пар слов складываются между собой, и их сумма записывается в старшее двойное слово приемника. Сумма произведений двух младших пар слов записывается в младшее двойное слово.

10 ::: Особливості команд логічних операцій та зсувів

PAND приемник, источник – Логическое И

Команда выполняет побитовое «логическое И» над источником (регистр MMX или переменная) и приемником (регистр MMX) и сохраняет результат в приемнике. Каждый бит результата устанавливается в 1, если соответствующие биты в обоих операндах равны 1, иначе бит сбрасывается в 0.

PANDN приемник, источник – логическое НЕ-И

Выполняет побитовое «логическое НЕ» (то есть инверсию бит) над приемником (регистр MMX) и затем побитовое «логическое И» над приемником и источником (регистр MMX или переменная). Результат сохраняется в приемнике. Каждый бит результата устанавливается в 1, только если соответствующий бит источника был равен 1, а приемника — 0, иначе бит сбрасывается в 0. Эта логическая операция носит также название «штрих Шеффера».

POR приемник, источник – логическое ИЛИ

Выполняет побитовое «логическое ИЛИ» над источником (регистр MMX или переменная) и приемником (регистр MMX) и сохраняет результат в приемнике. Каждый бит результата сбрасывается в 0, если соответствующие биты в обоих операндах равны 0, иначе бит устанавливается в 1.

PXOR приемник, источник – Логическое исключающее ИЛИ

Выполняет побитовое «логическое исключающее ИЛИ» над источником (регистр MMX или переменная) и приемником, (регистр MMX) и сохраняет результат в приемнике. Каждый бит результата устанавливается в 1, если соответствующие биты в обоих операндах равны, иначе бит сбрасывается в 0.

3. Використання команд введеня-виведення для управління зовнішніми пристроями

Початковим поштовхом до розробки ОС були проблеми автоматизації завантаження програм та використання узагальнених механізмів введення-виведення. На початковому етапі розроблення ОС найбільш коштовною частиною був ЦП, тому головною вважалася задача ефективного використання процесору Для цього основною проблемою була організація ефективного введення-виведення з одночасною роботою ЦП над розв'язанням інших задач. Для цього необхідно механізм апаратного переривання, який замінював цикл ЦП з очікуванням готовності даних.

Структура підпрограми драйверів пристроїв включала наступні блоки:

- Видача команди на пристрій для його підготовки до обміну
- Очікування готовності пристрою до обміну
- Виконання власне обміну
- Видача на пристрій команди для закінчення операції
- Організація обміну драйвера даними з програмою, яка його використовує.

При створенні мікропроцесорів фактично було повторено процес створення програм обміну для зовнішніх пристроїв, але з деякою систематизацією.

Підключення зовнішніх пристроїв до мікропроцесору виконувалось шляхом визначення вихідного командного порту, вхідного порту стану, які мали однакові номер та вхідного і вихідного порту для введення і виведення даних, які мали однакову адресу. Щоб написати узагальнений драйвер введення-виведення треба визначити адреси портів за допомогою

```
CMPRT EQU   41H
STPRT EQU   CMPRT
INPRT EQU   42H
OUTPRT      EQU   INPRT
```

```
Возвращает в ax 1 байт данных введенных с некоторого
устройства
drIn    Proc
        mov     al,cmOn
        out     CMPRT,al
lwr:    in      al,STPRT
        test    al,RdyBit
        jnz     lwr
        in      al,INPRT
        push    ax
        mov     al,cmOff
        out     CMPRT
        pop     ax
        ret
drIn    endp
```

Білет №23

1. Використання регулярних (автоматичних) граматик для лексичного аналізу

Під регулярною граматикою будемо розуміти якусь ліволінійну граматику. Це така граматика, що її P правила мають вигляд A → Bt або A → t, де A є N, B є N, t є T. Для прикладу будемо розуміти якийсь символ «@» кінцем ланцюжка, що аналізується лексичним аналізатором. Для граматик такого типу передбачений свій алгоритм розбору для отримання лексеми:

- Перший символ вихідного ланцюжка замінюємо не терміналом A, для якого в граматиці є правило A → a
- Далі проводимо ітерації до кінця ланцюжка за наступною схемою: отриманий раніше не термінал A і розташований правіше від нього термінал a вихідного ланцюжка замінюємо не терміналом B, для якого в граматиці є правило B → Aa

Стан автомату повинен відповідати типу розпізнаної лексеми або типу помилки лексичного аналізу. Вхідними сигналами автомату повинні бути літери вхідної послідовності, а точніше класифікаційні ознаки цієї літери. Таким чином для побудови лексичного аналізатора доцільно мати класифікаційну таблицю літер мови, що обробляє лексичний аналізатор,- двомірну матрицю переходів, що визначає код наступного стану автомату, ондим з яких є поточний стан автомату а другим – класифікатор вхідної літери. Тобто, щоб визначити такий автомат необхідно визначити перенумеровані типи даних для кодів стану і кодів специфікаторів. Власне програма повинна виділяти чергову лексему шляхом циклічного прогряду літер. Цикл закінчується в тому випадку, коли знайдена помилка або лексема закінчується роздільником. Для того, щоб використовувати лексичний аналіз методом теорії автоматів або автоматним методом необхідно використовувати таблицю класифікаторів, складних роздільників та таблицю ключових слів. Результати роботи лексичного аналізу треба рознести по таблицях імен, по таблицях констант і, можливо, по таблицях модулів. Результати лексичного аналізу доцільно розміщувати в спеціалізованих структурах, в яких зберігається інформація про код вхідної лексеми та її внутрішнє подання.

2. Організація роботи з таблицями в системних програмах

(по конспекту)

В системному програмуванні звичайно висувається вимога унікальності (змісту ключових слів). Для цього в табл. ключ. слів всі образи цих слів повинні бути унікальними, що не дозволяє виконувати кожне слово як об'єкт користувача. При створенні табл. імен ключові поля складають образ імені і номер блоку його визначення.

Таблиці – складні структури даних, завдяки яким можна підвищити ефективність програми. Їх основним призначенням є пошук в них інформації по заданому аргументу. Тому вони мають схожу структуру з базами даних. Таблиці мають задану кількість полів з даними, що є ключовими і функціональними. Таблиці системних програм зберігаються в ОП. Пошук виконується схожим чином з командою вибірки SELECT. Аргумент пошуку визначається значенням, що відповідає умові WHERE. До ключового поля висувається умова однозначності. Для додавання – INSERT. UPDATE і DELETE майже не використовуються в системних програмах.

Основные операции:

- оздание или получение доступа к таблице (конструктор)
- удаление таблицы
- включение нового эл-та
- поиск эл-та (эл-тов)
- упорядочение
- удаление эл-та (эл-тов)
- коррекция эл-та

Приведен пример фрагмента программы поиска по простейшемму линейному алгоритму с последовательным сравнением аргумента поиска с соответствующим полем. Будем считать, что аргумент загружен в аккумулятор AX, начальный адрес таблицы в - ES:DI, а колличество элементов таблицы в - CX

```
MOV       BX, DI     ; сохранение начального адреса таблицы
REPNZ SCASW          ; сканирование
JNZ       NotFnd     ; переход если не найден
SUB       DI, BX     ; определение индекса найденного эл-та
SUB       DI, 2      ; компенсация технологического пропуска
SRL       DI, 1      ; получения индекса из разницы адресов
```

В системных программах используются таблицы имен и констант в транслирующих программах, которые предназначены для синтаксического анализа и семантической обработки . Структурно таблицы обычно организуются как массивы и ссылочные структуры. Структуры данных табличного типа широко используются в системных программах, так как обеспечивает простое и быстрое обращение к данным. Таблицы состоят из элементов, каждый из которых представляется несколькими полями. Так при трансляции программы создаются, например, таблицы, содержащие элементы в виде: Ключ (переменная /метка) и характеристики: сегмент(данных / кода), смещение (XXXX), тип (байт, слово или двойное слово / близкий или дальний).

3. Основні архітектурні елементи захищеного режиму та їх призначення

Організація переключення задач у захищенному режимі

Для перехода в защищенный режим можно воспользоваться средствами того же BIOS и протокола DPHI, предварительно подготовив таблицы и базовую конфигурацию задач защищенного режима. Для организации переключения задач применен метод логических машин управления. Основу его аппаратно-программной реализации в процессорах ix86 составляем команды IMBCALL и IRET, бит NT регистра флагов, а также прерывания.

Для перехода от задачи к задаче при управлении мультизадачностью используются команды межсегментной передачи управления – переходы и вызовы. Задача также может активизироваться прерыванием. При реализации одной из этих форм управления назначение определяется элементом в одной из дескрипторных таблиц.

Тип дескриптора может быть таким, который инициирует выполнение новой задачи после сохранение состояния текущей. Имеется 2 кода типов, определяющих дескрипторы сегментов состояния задачи (TSS) и шлюза задачи. Когда управление передается любому из дескрипторов этих типов, происходит переключение задачи.

Дескрипторы шлюзов хранят только заполненные байты прав доступа и селектор соответствующего обьекта в глобальной таблице дескрипторов, помещенный на место 2-х младших байтов базового адреса. При каждом переключении задачи процессор может перейти с другой локальной дескрипторной таблицы, что позволяет назначить каждой задаче свое отображение логических адресов на физические.

Переключение задачи состоит из действий выполняемых одной из команд JMPPAR, CALLTAR или RET при NT=1:

6. проверка, разрешено ли уходящей задачи переключиться на новую

7. проверка файла, что дескриптор TSS приходящей задачи отмечен как присутствующий и имеет правильный предел (не меньше 67H)
8. сокращение состояния уходящей задачи
9. загрузка в регистр TR селектора TSS входящей задачи
10. загрузка состояния входящей задачи из ее сегмента TSS и продолжения выполнения.

При переключении задачи всегда сохраняется состояние уходящей задачи

Перед тем, как переключить процессор в защищённый режим, надо выполнить некоторые подготовительные действия, а именно:

- Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим. Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, разумеется, она работает в нулевом кольце защиты). Программа может модифицировать имеющиеся дескрипторы или добавить новые, загрузив заново регистр GDTR.
- Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по адресу 0040h:0067h, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по адресу 0040h:0067h.
  - Запретить все маскируемые и немаскируемые прерывания.
  - Открыть адресную линию A20.
  - Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
  - Загрузить регистр GDTR.

Это самый простой этап. Для перевода процессора i80286 из реального режима в защищённый можно использовать специальную команду LMSW, загружающую регистр состояния процессора (Mashine Status Word). Младший бит этого регистра указывает режим работы процессора. Значение, равное 0, соответствует реальному режиму работы, а значение 1 - защищённому.

Если установить младший бит регистра состояния процессора в 1, процессор переключится в защищённый режим:

```
mov ax, 1
lmsw ax
```

Робота програм для обробки преривань в захищенному режимі

В цьому режимі звичайно в таблицю дискретних преривань заносимо дискретний шлюз преривань, в якому зберігається адреса слова сегмента стану задачі TSS для задачі обробки преривань. В ОС може бути одна чи декілька сегментів задач. При переключені задачі управління передач за новим сегментом стану задачі запамьятовується адреса переваної задачі. В цьому випадку новий сегмент TSS буде повьязан з іншим адресним простором і буде включати в себе новий стек для нової задачі. Регітри переривань програми запамьятовуються в старому TSS і таким чинов в обробчику переривань в захисному режимі нема необхідності зберігати регістри перивань задачі. При виконанні команди IRET наприкінці обробки переривань відбувається перехід до перерваної задачі з відновленого старого TSS.

Білет №24

1. Методи низхідного розбору.

Нисходящий разбор заключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в заполнении промежутка между начальным нетерминалом и символами входной цепочки правилами, выводимыми из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом. Подставляемое правило в общем случае выбирается произвольно. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил. В последнем случае входная цепочка отвергается. Построение дерева разбора подтверждает принадлежность входной цепочки данному языку. При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что грамматика данного языка является недетерминированной. Эти рассуждения иллюстрируются следующим примером. Пусть будет дана грамматика G: $\mathbf{G_6 = \{\{S\}, \{a, +, *\}, P, S\}}$, где P определяется как: 1) S **(ρ)** a; 2) S **(ρ)** S + S; 3) S **(ρ)** S * S

Цепочки, порождаемые данной грамматикой можно интерпретировать как выражения, состоящие из операндов **"a"**, а также операций **"+"** и **"*"**. Недетерминированность грамматики позволяет порождать одну и ту же терминальную цепочки с использованием различных выводов. Например, выражение **"a+a*a+a"** можно получить следующими способами:

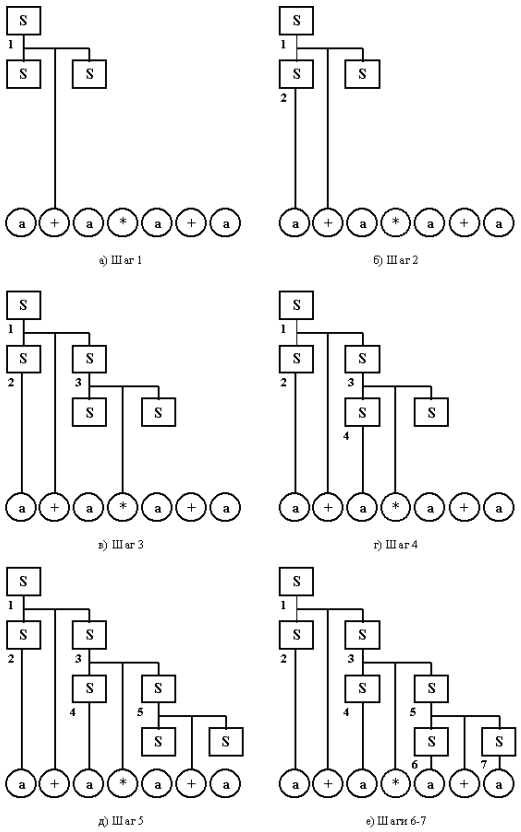


Рис. 6.2. Нисходящий разбор слева направо.

- S ⇒ S+S ⇒ a+S ⇒ a+S*S ⇒ a+ a*S ⇒ a+a*S+S ⇒ a+a*a+S ⇒ a+a*a+a
- S ⇒ S+S ⇒ S+a ⇒ S*S+a ⇒ S*a+a ⇒ S+S*a+a ⇒ S+a*a+a ⇒ a+a*a+a (6.1)
- S ⇒ S*S ⇒ S+S*S ⇒ S+S*S+S ⇒ a+ S*S+S ⇒ a+a*S+S ⇒ a+a*S+a ⇒ a+a*a+a

И так далее. В этом пример число вариантов одной и той же произвольной цепочки вывода настолько велико, что не имеет и смысла говорить о практическом применении данной грамматики. Но в данном случае она позволяет показать, каким образом могут порождаться различные деревья при нисходящем разборе. Пошаговое построение различных деревьев показано на рис. Можно отметить, что процесс построения дерева совпадает с последовательностью шагов вывода входной цепочки.

Методи низхідного розбору:

- Метод рекурсивного спуску
- Метод синтаксичних графів

(по конспекту)

Метод низхідного розбору в якому аналіз конструкції викнується починаючи від найбільш складних конструкції доо термінальних позначень. В цьому випадку за базу можуть бути взяті правила підстановки. Один з найпоширеніших – метод рекурсивного спуску. В цьому методі розбір починається з кінцевого позначення граматики. При виконанні такого розбору аналізатор звертається до підлеглого ресурсу, щоб розібрати спочатку перший а потім наступні позначення правої частини правила підстановки.Рекурсивні правила у формі Бекуса прийнято, так що рекурсивні звертання записуються з правого боку, що утворює ліворекурсивні правила. Однак при такому розборі ми будмо просуватись в глибину рекурсії , не просуваючись вздовж вхідного потоку даних, що фактично призводить до за циклювання аналізатора. Тому для використованого методу рекурсивного спуску необхідно перетворити правила на право рекурсивну форму. Альтернативним методом є метод синтаксичних графів.

```
Number>::=[<Sign>]<digit>{<digit>}[<Separator><digit>{<digit>}]
           [<Exponent>[<Sign>]<digit>{<digit>}]
<digit>::='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<Sign>::='+'|'-'
<Separator>::='.'
<Exponent>::='E'|'e'
```

Теперь на основе этих правил напишем функцию IsNumber, которая в качестве параметра принимает строку и возвращает True, если эта строка удовлетворяет правилам записи числа, и False, если не удовлетворяет.

```
// Проверка символа на соответствие <digit>
function IsDigit(Ch:Char):Boolean;
 begin
```

```
  Result:=Ch in ['0'..'9']
 end;

// Проверка символа на соответствие <Sign>
function IsSign(Ch:Char):Boolean;
 begin
  Result:=(Ch='+') or (Ch='-')
 end;

// Проверка символа на соответствие <Separator>
function IsSeparator(Ch:Char):Boolean;
 begin
  Result:=Ch='.'
 end;

// Проверка символа на соответствие <Exponent>
function IsExponent(Ch:Char):Boolean;
 begin
  Result:=(Ch='E') or (Ch='e')
 end;

function IsNumber(const S:string):Boolean;
 // Номер символа выражения, который сейчас проверяется
 var P:Integer;
  begin
   Result:=False;
   // Проверка, что выражение содержит хотя бы один символ.
   // пустая строка не является числом
   if Length(S)=0 then
    Exit;
   // Начинаем проверку с первого символа
   P:=1;
   // Если первый символ - <Sign>, переходим к следующему
   if IsSign(S[P]) then
    Inc(P);
   // Проверяем, что в данной позиции стоит хотя бы одна цифра
   if (P>Length(S)) or not IsDigit(S[P]) then
    Exit;
   // Переходим к следующей позиции, пока не достигнем
   // конца строки или не встретим не цифру
   repeat
    Inc(P)
   until (P>Length(S)) or not IsDigit(S[P]);
   // Если достигли конца строки, выражение корректно - число,
   // не имеющее дробной части и экспоненты
   if P>Length(S) then
    begin
     Result:=True;
     Exit
    end;
   // Если следующий символ - <Separator>, проверяем,
   // что после него стоит хотя бы одна цифра
   if IsSeparator(S[P]) then
    begin
     Inc(P);
     if (P>Length(S)) or not IsDigit(S[P]) then
      Exit;
     repeat
      Inc(P)
     until (P>Length(S)) or not IsDigit(S[P]);
```

```
    // Если достигли конца строки, выражение корректно - число
    // без экспоненты
    if P>Length(S) then
     begin
      Result:=True;
      Exit
     end
   end;
  // Если следующий символ - <Exponent>, проверяем,
  // что после него стоит всё то, что требуется правилами
  if IsExponent(S[P]) then
   begin
    Inc(P);
    if P>Length(S) then
     Exit;
    if IsSign(S[P]) then
     Inc(P);
    if (P>Length(S)) or not IsDigit(S[P]) then
     Exit;
    repeat
     Inc(P)
    until (P>Length(S)) or not IsDigit(S[P]);
    if P>Length(S) then
     begin
      Result:=True;
      Exit
     end
   end
  // Если выполнение дошло до этого места, значит,
  // в выражении остались ещё какие-то символы. Т.к. никакие
  // дополнительные символы синтаксисом не предусмотрены,
  // такое выражение не считается корректным числом.
 end;
```

Для каждого нетерминального символа мы ввели отдельную функцию, разбор начинается с символа самого верхнего уровня - <Number> - и следует правилам, записанным для этого символа. Такой способ синтаксического анализа называется левосторонним рекурсивным нисходящим анализом. Левосторонним потому, что символы в выражении перебираются слева направо, нисходящим - потому, что сначала анализируются символы верхнего уровня, а потом - символы нижнего. Рекурсивность метода на данном примере не видна, т.к. наша грамматика не содержит рекурсивных определений, но мы с этим столкнёмся в последующих примерах.

2. Особливості пересилок і перетворень в розширенні MMX.

MMX-расширение предназначено для поддержки приложений, ориентированных на работу с большими массивами данных целого и вещественного типов, над которыми выполняются одинаковые операции. С данными такого типа обычно работают мультимедийные, графические, коммуникационные программы – от этого и название MultiMedia eXtensions. Важное отличие MMX-команд от обычных команд процессора в том, как они реагируют на ситуации переполнения и заема. Возникают ситуации, когда результат арифм. операции выходит за размер разрядной сетки исходных операндов. В этом случае производится усечение старших бит результата и возвращаются только те биты, которые умещаются в пределах исходного операнда (арифметика с циклическим переносом). Некоторые MMX-команды в такой ситуации действуют иначе. В случае выхода значения результата за пределы операнда, в нем фиксируется максимальное или минимальное значение (арифметика с насыщением). MMX_расширение имеет команды, которые выполняют арифметические операции с использование обоих принципов. При этом среди них есть команды, учитывающие знаки элементов операндов. ПРИМЕР:

```
include mmx16.inc
.data
mem dw 4444h
df 111122223333h
.code
movd rmmx0, mem ; rmmx0=0000 0000 3333 4444
movq rmmx0, mem ; rmmx0=1111 2222 3333 4444
```

Команды для пересылки.

MOVD приемник,источник – пересылка двойных слов

Команда копирует двойное слово из источника (регистр MMX, обычный регистр или переменная) в приемник (регистр MMX, обычный регистр или переменная, но хотя бы один из операндов обязательно должен быть регистром MMX). Если приемник — регистр MMX, двойное слово записывается в его младшую половину (биты 31 – 0), а старшая заполняется нулями. Если источник — регистр MMX, в приемник записывается младшее двойное слово этого регистра.

MOVQ приемник,источник – пересылка учетверенных слов

Копирует учетверенное слово (64 бита) из источника (регистр MMX или переменная) в приемник (регистр MMX или переменная, оба операнда не могут быть переменными).

Команды преобразования данных.

PACKSSWB приемник,источник – упаковка байт со знаковым насыщением

PACKSSDW приемник, – упаковка слов со знаковым насыщением

Команды упаковывают и насыщает слова со знаком в байты (PACKSSWB) или двойные слова со знаком в слова (PACKSSDW). Команда PACKSSWB копирует четыре слова (со знаком), находящиеся в приемнике (регистр MMX), в 4 младших байта (со знаком) приемника и копирует четыре слова (со знаком) из источника (регистр MMX или переменная) в старшие четыре байта (со знаком) приемника. Если значение какого-нибудь слова больше +127 (7Fh) или меньше -128 (80h), в байты помещаются числа +127 и -128 соответственно. Команда PACKSSDW аналогично копирует два двойных слова из приемника в два младших слова приемника и два двойных слова из источника в два старших слова приемника. Если значение какого-нибудь двойного слова больше +32 767 (7FFFh) или меньше -32 768 (8000h), в слова помещаются числа +32 767 и -32 768 соответственно.

PACKUSWB приемник,источник – упаковка с беззнаковым насыщением

Копирует четыре слова (со знаком), находящиеся в приемнике (регистр MMX), в 4 младших байта (без знака) приемника и копирует четыре слова (со знаком) из источника (регистр MMX или переменная) в старшие четыре байта (без знака) приемника. Если значение какого-нибудь слова больше 255 (FFh) или меньше 0 (00h), в байты помещаются числа 255 и 0 соответственно.

PUNPCKHBW приемник,источник – распаковка и объединение старших элементов (байт)

PUNPCKHWD приемник,источник – распаковка и объединение старших элементов (двойное слово)

PUNPCKHDQ приемник,источник – распаковка и объединение старших элементов (учетверенное слово)

Команды распаковывают старшие элементы источника (регистр MMX или переменная) и приемника (регистр MMX) и записывают их в приемник через один. Команда PUNPCKHBW объединяет по 4 старших байта источника и приемника, команда PUNPCKHWD объединяет по 2 старших слова, и команда PUNPCKHDQ копирует в приемник по одному старшему двойному слову из источника и приемника.

Если источник содержит нули, эти команды фактически переводят старшую половину приемника из одного формата данных в другой, дополняя увеличиваемые элементы нулями. PUNPCKHBW переводит упакованные байты в упакованные слова, PUNPCKHWD переводит слова в двойные слова, и PUNPCKHDQ переводит единственное старшее двойное слово приемника в учетверенное.

PUNPCKLBW приемник,источник – распаковка и объединение младших элементов (байтов)

PUNPCKLWD приемник,источник – распаковка и объединение младших элементов (двойное слов)

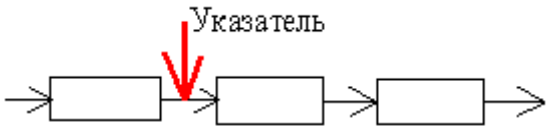
PUNPCKLDQ приемник,источник – распаковка и объединение младших элементов (учетверенное слово)

Команды распаковывают младшие элементы источника (регистр MMX или переменная) и приемника (регистр MMX) и записывают их в приемник через один аналогично предыдущим командам. Команда PUNPCKLBW объединяет по 4 младших байта источника и приемника, команда PUNPCKLWD объединяет по 2 младших слова, и команда PUNPCKLDQ копирует в приемник по одному младшему двойному слову из источника и приемника. Если источник содержит только нули, эти команды, аналогично PUNPCKH*, фактически переводят младшую половину приемника из одного формата данных в другой, дополняя увеличиваемые элементы нулями.

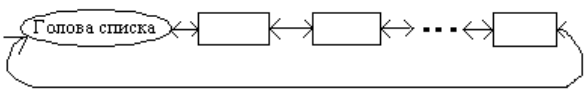
3. Побудова таблиць у вигляді списків.

Другая линия развития базовых структур таблиц основана на ссылочных структурах. Использование ссылок и указателей избавляет от необходимости множественных пересылок при сортировки таблиц. Наличие одно- и двунаправленных ссылок вдоль массива элементов приводит к построению одно- и двусвязных списков, в которых практически невозможно получить эффект от упорядочения при поиске.

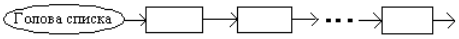
Классический пример структуры данных последовательного доступа, в которой можно удалять и добавлять элементы в середине структуры, — это линейный список. Различают однонаправленный и двунаправленный списки (иногда говорят односвязный и двусвязный).
Элемены списка как бы выстроены в цепочку друг за другом. У списка есть начало и конец. Имеется также указатель списка, который располагается между элементами. Если мысленно вообразить, что соседние элементы списка связаны между собой веревкой, то указатель — это ленточка, которая вешается на веревку. В любой момент времени в списке доступны лишь два элемента — элементы до указателя и за указателем.



В однонаправленном списке указатель можно передвигать лишь в одном направлении — вперед, в направлении от начала к концу. Кроме того, можно установить указатель в начало списка, перед его первым элементом. В отличие от однонаправленного списка, двунаправленный абсолютно симметричен, указатель в нем можно передвигать вперед и назад, а также устанавливать как перед первым, так и за последним элементами списка.
В двунаправленном списке можно добавлять и удалять элементы до и за указателем. В однонаправленном списке добавлять элементы можно также с обеих сторон от указателя, но удалять элементы можно только за указателем.
Удобно считать, что перед первым элементом списка располагается специальный пустой элемент, который называется головой списка. Голова списка присутствует всегда, даже в пустом списке. Благодаря этому можно предполагать, что перед указателем всегда есть какой-то элемент, что упрощает процедуры добавления и удаления элементов.
В двунаправленном списке считают, что вслед за последним элементом списка вновь следует голова списка, т.е. список зациклен в кольцо.



Можно было бы точно так же зациклить и однонаправленной список. Но гораздо чаще считают, что за последним элементом однонаправленного списка ничего не следует. Однонаправленный список, таким образом, представляет собой цепочку, начинающуюся с головы списка, за которой следует первый элемент, затем второй и так далее вплоть до последнего элемента, а заканчивается цепочка ссылкой в никуда.



Ссылочная реализация списка

Мы рассмотрели абстрактное понятие списка. Но в программировании зачастую отождествляют понятие списка с его ссылочной реализацией на базе массива или непосредственно на базе оперативной памяти.

Основная идея реализации двунаправленного списка заключается в том, что вместе с каждым элементом хранятся ссылки на следующий и предыдущий элементы. В случае реализации на базе массива ссылки представляют собой индексы ячеек массива. Чаще, однако, элементы списка не располагают в каком-либо массиве, а просто размещают каждый по отдельности в оперативной памяти, выделенной данной задаче. (Обычно элементы списка размещаются в так называемой **динамической памяти**, или **куче** — это область оперативной памяти, в которой можно при необходимости захватывать куски нужного размера, а после использования освобождать, т.е. возвращать обратно в кучу.) В качестве ссылок в этом случае используют адреса элементов в оперативной памяти.

Голова списка хранит ссылки на первый и последний элементы списка. Поскольку список зациклен в кольцо, то следующим за головой списка будет его первый элемент, а предыдущим — последний элемент. Голова списка хранит только ссылки и не хранит никакого элемента. Это как бы пустой ящик, в который нельзя ничего положить и который используется только для того, чтобы написать на нем адреса следующего и предыдущего ящиков, т.е. первого и последнего элементов списка. Когда список пуст, голова списка зациклена сама на себя.

Указатель списка реализуется в виде ссылки на следующий и предыдущий элементы, он просто отмечает некоторое место в цепочке элементов.

В случае однонаправленного списка хранится только ссылка на следующий элемент, таким способом экономится память. Голова однонаправленного списка хранит ссылку на первый элемент списка. Последний элемент списка хранит нулевую ссылку, т.е. ссылку в никуда, т.к. в программах нулевой адрес никогда не используется.

Ценность ссылочной реализации списка состоит в том, что процедуры добавления и удаления элементов не приводят к массовым операциям. Рассмотрим, например, операцию удаления элемента за указателем. Читая ссылку на следующий элемент в удаляемом элементе, мы находим, какой элемент должен будет следовать за указателем после удаления текущего элемента. После этого достаточно связать элемент до указателя с новым элементом за указателем. А именно, обозначим через X адрес элемента до указателя, через Y — адрес нового элемента за указателем. В поле следующий для элемента с адресом X надо записать значение Y, в поле предыдущий для элемента с адресом Y — значение X. Таким образом, при удалении элемента за указателем он исключается из цепочки списка, для этого достаточно лишь поменять ссылки в двух соседних элементах. Аналогично, для добавления элемента достаточно включить его в цепочку, а для этого также нужно всего лишь модифицировать ссылки в двух соседних элементах. Добавляемый элемент может располагаться где угодно, следовательно, нет никаких проблем с захватом и освобождением памяти под элементы.

Білет 25

1. Метод рекурсивного спуску

рекурсия — способ общего определения объекта или действия через себя, с использованием ранее заданных частных определений. Рекурсия используется, когда можно выделить самоподобие задачи.

Метод рекурсивного спуска - это один из методов определения принадлежности входной строки к некоторому формальному языку, описанному LL(k) контекстно-свободной грамматикой

Идея метода

Для каждого нетерминального символа *K* строится функция, которая для любого входного слова *x* делает 2 вещи:

- Находит наибольшее начало *z* слова *x*, способное быть началом выводимого из *K* слова
- Определяет, является ли начало *z* выводимым из *K*

Такая функция должна удовлетворять следующим критериям:

- считывать из еще необработанного входного потока максимальное начало *A*, являющегося началом некоторого слова, выводимого из *K*
- определять является ли *A* выводимым из *K* или просто невыводимым началом выводимого из *K* слова

В случае, если такое начало считать не удается (и корректность функции для нетерминала *K* доказана), то входные данные не соответствуют языку, и следует остановить разбор.

Разбор заключается в вызове описанных выше функций. Если для считанного нетерминала есть составное правило, то при его разборе будут вызваны другие функции для разбора входящих в него терминалов. Дерево вызовов, начиная с самой “верхней” функции эквивалентно дереву разбора.

Условия применения

Пусть в данной формальной грамматике *N* - это конечное множество нетерминальных символов; Σ - конечное множество терминальных символов, тогда метод рекурсивного спуска применим только, если каждое правило этой грамматики имеет следующий вид:

- или $A \to \alpha$, где $\alpha \subset (\Sigma \cup N)*$, и это единственное правило вывода для этого нетерминала
- или $A \to a_1\alpha_1|a_2\alpha_2|...|a_n\alpha_n$ для всех $i = 1,2...n; a_i \neq a_j, i \neq j; \alpha \subset (\Sigma \cup N)*$

Метод рекурсивного спуска учитывает особенности современных языков программирования, в которых реализован рекурсивный вызов процедур. Если обратиться к дереву нисходящего разбора слева направо то можно заметить, что в начале происходит анализ всех поддеревьев, принадлежащих самому левому нетерминалу. Затем, когда самым левым становится другой нетерминал, то анализ происходит для него. При этом используются и полностью раскрываются все нижележащие правила. Это навязывает определенные ассоциации с иерархическим вызовом процедур в программе, следующих друг за другом в охватывающей их процедуре. Поэтому, все нетерминалы можно заменить соответствующими им процедурами или функциями, внутри которых вызовы других процедур - нетерминалов и проверки терминальных символов будут происходить в последовательности, соответствующей их расположению в правилах. Такая возможность подкрепляется и другой ассоциацией.Вызов процедуры или функции реализуется через занесение локальных данных в стек, который поддерживается системными средствами. Заносимые данные определяют состояние обрабатываемого нетерминала, а машинный стек соответствует магазину автомата. *Использование рекурсивного спуска позволяет достаточно быстро и наглядно писать программу распознавателя на основе имеющейся грамматики*. Главное, чтобы последняя соответствовала требуемому виду. Использование рекурсивного спуска позволяет написать программу быстрее, так как не надо строить автомат. Ее текст может быть и менее ступенчатым, если использовать инверсные условия проверки или другие методы компоновки текста. Не имеет смысла заменять рекурсивный спуск автоматом с магазинной памятью, особенно в том случае, если грамматику задавать с использованием диаграмм Вирта.

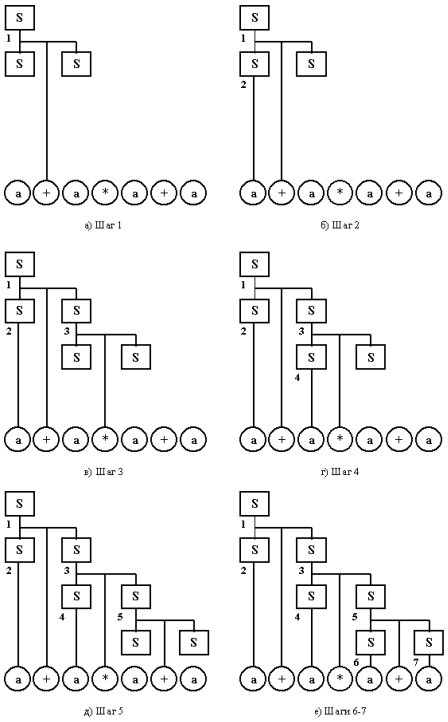


Рис. 6.2. Нисходящий разбор слева направо.

<ВІДПОВІДЬ ПРО РЕКУРСИВНИЙ СПУСК, НЕ ЗВ’ЯЗАНА ІЗ СИНТАКСИЧНИМ РОЗБОРОМ:>

Порождение все новых копий рекурсивной подпрограммы до выхода на граничное условие называется *рекурсивным спуском*. Максимальное количество копий рекурсивной подпрограммы, которое одновренно может находиться в памяти компьютера, называется *глубиной рекурсии*. Завершение работы рекурсивных подпрограмм, вплоть до самой первой, инициировавшей рекурсивные вызовы, называется *рекурсивным подъёмом*.

Классический пример рекурсии — определение факториала. С одной стороны, факториал определяется так: $n! = \begin{cases} 1, & \text{если } n \le 1, \\ (n-1)! \times n, & \text{если } n > 1. \end{cases}$ *n*!=1*2*3*...**n*. С другой стороны, Граничным условием в данном случае является *n*<=1.

```
    /* Функция на С */     // Рекурсивный спуск – повторный вызов метода Factorial
 double Factorial(int N)
 {
  double F;
  if (N<=1) F=1.; else F=Factorial(N-1)*N;
  return F;
 }.
```

2. Розширення MMX

MMX (Multimedia Extensions — мультимедийные расширения) — коммерческое название дополнительного набора инструкций, выполняющих характерные для процессов кодирования/декодирования потоковых аудио/видео данных действия за одну машинную инструкцию. Впервые появился в процессорах Pentium MMX. Разработан в лаборатории Intel в Хайфе, Израиль, в первой половине 1990-х.

::: Основа аппаратной компоненты расширения mmx – *восемь* новых *регистров*, которые на самом деле являются регистрами *сопроцессора*, только вместо 80-ти разрядов *используется 64 младших разряда* (мантисса). при работе со стеком сопроцессора в режиме mmx он рассматривается как обычный массив регистров с произвольным доступом. использовать стек сопроцессора

по его прямому назначению и как регистры mmx-расширения одновременно невозможно. Основным принципом работы команд mmx является одновременная обработка нескольких единиц однотипных данных одной командой.
Допустим, у нас есть два массива: A и B, длиной по 8 байтов каждый. Поставим себе задачу — прибавить к каждому элементу массива B соответствующий ему элемент массива A. Это приведет нас к такой программе:

```
mov  esi, offset A
   mov  edi, offset B
   mov  ecx, 8
OurLoop:
   mov  al, [esi]
   add  [edi], al
   inc  esi
   inc  edi
   loop OurLoop
```

Как видим, не смотря на простую постановку задачи, процессор будет вынужден выполнить восемь раз нетривиальную последовательность команд. Вот если бы операцию сложения двух массивов можно было выполнить одной командой... На этой идее и был построен MMX. Аналогичная программа с его использованием имела бы такой вид: movq mmreg1, A

```
   movq  mmreg2, B
   paddb mmreg2, mmreg1
   movq  B, mmreq2
```

Допустим, выполняется программа обрабатывающая 256-цветное изображение. Обычно она обрабатывает каждый пиксел по отдельности, используя же MMX, она может обрабатывать восемь пикселей сразу! Так что же такое MMX? MMX - это расширение, включающее в себя 57 новых команд и восемь 64-разрядных регистров. В основу положен принцип SIMD (Single Instruction Multiple Data) - одна инструкция - множество данных. MMX предоставляет инструкции для складывания, умножения и даже выполнения комбинированных операций. К примеру, команда PMADDWD перемножает, а затем складывает четыре слова данных, при этом она выполняется намного быстрее программы использующей только набор "стандартных" инструкций. Технология MMX предоставляет простую, гибкую программную модель, не требуя при этом перевода процессора в какой-то особенный режим. Все существующие программы будут корректно исполняться на процессорах с MMX без каких-либо изменений, даже если в системе присутствуют приложения, использующие новую технологию.
Регистры.
В процессорах использующих MMX добавлено 8 новых 64-разрядных регистров MM0-MM7. Они могут быть использованы только для выполнения операций с типами данных MMX. Команды MMX позволяют задавать в качестве операндов как регистры общего назначения (EAX, EBX, ECX, EDX, EBP, ESI, EDI и ESP), так и переменные в памяти, используя для этого стандартную схему адресации принятую в процессорах x86. Хотя MMX регистры и имеют ни с чем не совпадающие названия, на самом деле, они являются "псевдонимами" регистров сопроцессора (st0-st7). Это означает, что, изменяя один из регистров MMX, в своей программе, мы в то же время изменяем регистры сопроцессора.
ТИПЫ ДАННЫХ.
MMX поддерживает данные в упакованном формате. Это означает, что каждый 64 битовый регистр MMX может интерпретироваться как:
1. Восемь байтов
2. Четыре слова
3. Два двойных слова
4. Одна 64-разрядная переменная

Все инструкции можно разбить на следующие **группы:**
арифметические команды
  сложение PADDB PADDD PADDSB PADDSW PADDUSB PADDUSW PADDW
  сложение и умножение PMADDWD
  умножение PMULHW PMULLW
  вычитание PSUBB PSUBD PSUBSB PSUBSW PSUBUSB PSUBUSW PSUBW
команды очищающие регистры
  EMMS
команды сравнения
  проверка равенства PCMPEQB PCMPEQD PCMPEQW
  сравнение PCMPGTB PCMPGTD PCMPGTW
команды упаковки/распаковки
  PUNPCKHBW PUNPCKHDQ PUNPCKHWD PUNPCKLBW PUNPCKLDQ PUNPCKLWD PACKSSDW PACKSSWB PACKUSWB
логические команды
  PAND PANDN POR PXOR
команды передачи данных
  MOVD MOVQ
команды сдвига
  логический сдвиг влево PSLLD PSLLQ PSLLW
  арифметический сдвиг вправо PSRAD PSRAW
  логический сдвиг вправо PSRLD PSRLQ PSRLW
Команды MMX имеют такой формат: инструкция mmreg1, mmreg2/mem64

То есть источником может быть как переменная памяти, так и регистр MMX. А целевым может быть только регистр MMX. К тому же MOVD и MOVQ допускают пересылки из регистров MMX в память. Одним из первых вопросов у меня был: "зачем так много команд?". Ведь количество операций не так уж и велико... Дело в том, что название команды формируется из двух

частей. Первая часть говорит о том, что она делает (MOV,PSUB,PADD). Вторая же говорит о том, как она интерпретирует операнды. Рассмотрим команду PACKUSWB. Первая часть строки - PACK указывает, что будем что-то упаковывать. Вторая же - USWB в свою очередь разбивается на две US и WB. US говорит о том, что результат будет беззнаковым с сатурацией , WB же указывает, что источник - упакованные слова, результат - упакованные байты. Благодаря такой записи смысл инструкции схватывается "на лету", по этой же причине команд так много. Приведем полный перечень мнемоник:

P-упакованные данные (Packed data)
B-байт (byte)
W-слово (word)
D-двойное слово (DoubleWord)
Q-64 битовая переменная (QuadWord)
S-знаковая (Signed)
U-беззнаковая (Unsigned)
SS-знаковая с сатурацией (Signed Saturation)
US-беззнаковая с сатурацией (Unsigned Saturation)

Однако разрядность MMX регистров, к сожалению, мала. Поэтому некоторые классы задач сложно приспособить к использованию этой мощи. К примеру, в графике часто нужно выполнить операцию умножения матрицы на матрицу, или вектора на матрицу. Казалось бы, вот где MMX понадобится. Однако в большинстве задач в качестве элементов матрицы или вектора используются числа с плавающей точкой. Вывод - MMX тут не применишь.

Правда можно в качестве элементов использовать числа с фиксированной точкой . Размер такого числа - четыре байта. То есть в один регистр поместится только два элемента вектора. Тонкость тут в том, что в случае трехмерных преобразований скалярное произведение нужно считать от двух трехэлементных векторов... Решить проблему можно - использовать для этого два MMX регистра: в первом два элемента, во втором один. Однако выигрыш при этом будет не столь значителен, как хотелось бы.

КРАТКОЕ ОПИСАНИЕ КОМАНД.

MOVD

Формат

MOVD mmreg1, reg32/mem32

MOVD reg32/mem32, mmreg1

Описание

Инструкция MOVD пересылает 32 младших бита регистра MMX в регистр общего назначения или память. Или же из регистра общего назначения/памяти в регистр MMX. В последнем случае кроме собственно пересылки биты 32-64 соответствующего регистра обнуляются.

PADDB

Формат

PADDB mmreg1, mmreg2/mem64

Описание

Складывает восемь 8-разрядных чисел операнда источника (MMX регистра или 64-разрядного поля памяти) и восемь 8-разрядных чисел операнда-получателя (регистра MMX). В случае переполнения полученные значения заворачиваются (то есть 255+10=9) без установки флага переноса при подсчете последующих байтов. Результат помещается в операнд-получатель.

PADDSW

Формат

PADDSW mmreg1, mmreg2/mem64

Описание

Складывает четыре 16-битных чисел со знаком операнда источника (регистра MMX или 64-разрядного поля памяти) и четыре соответствующих значения операнда-получателя (регистра MMX). Если сумма каких-либо двух из них менее чем -32768 (8000h), то результат сложения -32768 (8000h). Аналогично в случае, когда сумма превышает 32767 (7FFFh), то возвращаемый результат 32767 (7FFFh). Результат заносится в операнд-получатель.

PAND

Формат

PAND mmreg1, mmreg2/mem64

Описание

Инструкция PAND выполняет логическую операцию "и" над операндами. Результат заносится в операнд-получатель. Если соответствующие биты источника и получателя равны единице, то значение итогового бита единица. Эта команда может использоваться для распаковки упакованных переменных при помощи маски полученной от инструкций сравнения PCMPEQ и PCMPGT.

PCMPEQB

Формат

PCMPEQB mmreg1, mmreg2/mem64

Описание

Сравнивает источник с получателем, рассматривая операнды как упакованные байты. Если биты операндов эквивалентны, то все биты 8-разрядной части получателя устанавливаются равными единице, в противном случае они обнуляются.

PCMPGTD

Формат

PCMPGTB mmreg1, mmreg2/mem64

Описание

Аналогична PCMPEQ, но в отличие от нее биты в целевом операнде устанавливается в том случае, когда байт целевого операнда больше соответствующего байта операнда источника.

PACKUSWB

Формат

PACKUSWB mmreg1, mmreg2/mem64

Описание

Переводит восемь знаковых слов задаваемые аргументами (по четыре слова в каждом) в восемь беззнаковых байт. После чего результат сохраняется в mmreg1

3. Організація підсистем введення – виведення

Початковим поштовхом до розробки ОС були проблеми автоматизації завантаження програм та використання узагальнених механізмів введення-виведення. На початковому етапі розроблення ОС найбільш дорогою частиною був ЦП, тому головною вважалася задача ефективного використання процесору Для цього основною проблемою була організація ефективного введення-виведення з одночасною роботою ЦП над розв’язанням інших задач. Для цього необхідно механізм апаратного переривання, який замінював цикл ЦП з очікуванням готовності даних.

Структура підпрограми драйверів пристроїв включала наступні блоки:

- Видача команди на пристрій для його підготовки до обміну
- Очікування готовності пристрою до обміну
- Виконання власне обміну
- Видача на пристрій команди для закінчення операції
- Організація обміну драйвера даними з програмою, яка його використовує.

При створенні мікропроцесорів фактично було повторено процес створення програм обміну для зовнішніх пристроїв, але з деякою систематизацією.

Підключення зовнішніх пристроїв до мікропроцесору виконувалось шляхом визначення вихідного командного порту, вхідного порту стану, які мали однакові номер та вхідного і вихідного порту для введення і виведення даних, які мали однакову адресу. Щоб написати узагальнений драйвер введення-виведення треба визначити адреси портів за допомогою

```
CMPRT EQU   41H
STPRT  EQU   CMPRT
INPRT  EQU   42H
OUTPRT      EQU   INPRT
```

Возвращает в ax 1 байт данных введенных с некоторого устройства

```
drIn    Proc
        mov     al,cmOn
        out     CMPRT,al
lwr:    in      al,STPRT
        test    al,RdyBit
        jnz     lwr
        in      al,INPRT
        push    ax
        mov     al,cmOff
        out     CMPRT
        pop     ax
        ret
drIn    endp
```

Білет №26

1. Метод синтаксичних графів.

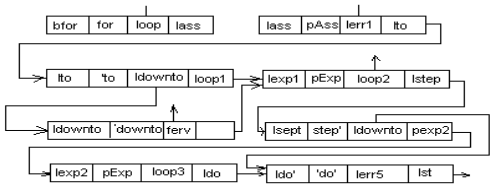
Результати синтаксичного розбору подаються у вигляді зв'язаних вузлів графа розбору, в якому кожний вузол відповідає окремій лексемі, а зв'язки визначають підлеглість операндів до операцій. Вузол об'єднує 4 поля:

- Ідентифікація або мітка вузла
- Прототип співставлення (термінал або нетермінал)
- Мітка продовження обробки (при успішному результаті синтаксичного аналізу)
- Вказівник на альтернативну вітку, яку можна перевірити (якщо аналіз був невдачним)

Синтаксично буває зручним використовувати змішані алгоритми аналізу. Як раз для обробки операторів мов програмування краще використовувати синтаксичний граф, а для обернених виразів – висхідний розбір.

Також існують зв'язки передачі управління в операторах розгалуження та блоках циклів та варіантному блоці. Таким чином будь-який закінчений фрагмент програми має головний вузол, в якому як підлеглі вузли різних рівнів зберігаються лексеми та синтаксичні структури, які входять до цього блоку. Такий граф розбору є основою для всіх видів подальшої семантичної обробки.

Методи низхідного розбору єфективніші при обробці структурованих операторів(for)



2. Організація циклів на базі процесора з плаваючою точкою

Для организации ветвлений и проверки условий окончания циклов используется группа команд сравнения FPU и команды передачи слова состояния в центральный процессор. FCOM сравнивает st[0] с операндом src, а FICOM – с целым оперндом, после чего исключает его из стека FPU, а FICOM производит те же действия с целочисленным операндом. FOCMPP сравнивает два верхних регистра стека с последующим исключением их из стека. Результат сравнения в регистрах SW.

Команда FTST сравнивает st[0] с 0.Перечисленные команды меняют содержимое регистра состояния в соответствии с с результаттом выситания второго операнда из первого и могут использоватся для организации условной передчи управления по арифметическим отношениям.
При программировании условных переходов по результатам работы АЗГ нужно учитывать, что команды в режиме эмуляции трансоируются в модифицированную форму, в которой сохраняется изоморфность к программе FPU по ко-ву байтов, а значит и по размещению команд, а также сохраня.тся биты, в которых определены операции FPU. Условные переходы по результатам сравнений в FPU можно организоавать следующиим макроопределением:

```
FJ MACRO CD,LB ; прототип макровызова
FSTSW STSW   ;запись в память регистра состояния
FPU и команды центрального процессора
FWAIT ; ожидание окончания пересылки
MOV AH,BYTE PTR STSW+1 ;копирование
;регистра состояния
SAHF   ;Сохранение содержимого регистра ah в регистреF
J&CD LB          ; условный переход
ENDM
```

Для программированния условного перехода по результату сравнения достаточно использовать макровызов вида:
[метка] fj условие перех, метка прехода.

3. Особливості роботи з БПП
(нашел в какой-то книге, вставил все, где упоминалось БПП ☺, что подходит, что нет НЕЗНАЮ, но то что нужно здесь есть 100%)

Типовое аппаратное обеспечение системы прерываний включает:

- специальные команды обращения к процедурам процессора, программируемым на языках высокого уровня с определением атрибута процедуры interrupt (для процессоров ix86 команда INT);
- специальные средства защиты процессора от ошибок и аварийных ситуаций, вызывающих автоматическое обращение к процедурам обработки прерываний при возникновении особых ситуаций;
- специальные средства отладки программ, которые позволяют определять на машинном уровне контрольные точки и режимы отладки программ;
- блок приоритетного прерывания (БПП), формирующий по сигналам от внешних устройств управляющие сигналы на центральный процессор и номера векторов прерывания.

Аппаратные прерывания, используемые для обработки вектора, определяются конструкцией компьютера и, прежде всего, способом подключения сигналов прерывания к БПП и способом его программирования в BIOS. Для машин типа IBM/AT и последующих BIOS использует такие векторы прерываний внешних устройств.

8 – прерывание системного таймера IRQ0, стандартно запускаемое 18,2 раза в секунду по каналу 0 системного таймера 8254 и используемое для обновления текущего системного времени и даты.
9 – прерывание IRQ1, свидетельствующее о готовности данных на клавиатуре.
0ah – прерывание IRQ2, свидетельствующее о приходе сигналов с устройств через подчиненный БПП.
0bh – прерывание IRQ3, возникающее при необходимости обслуживания последовательного порта COM2.
0ch – прерывание IRQ4, возникающее при необходимости обслуживания последовательного порта COM1.
0dh – прерывание IRQ5, сильно изменяется от системы к системе.
0h – прерывание IRQ6, формируемое контроллером гибких дисков по завершении операции.
0fh – прерывание IRQ7, формируемое параллельным портом LPT1.
70h – прерывание часов реального времени IRQ8, обеспечивающее возможность подачи сигнала будильника или периодического инициирующего прерывания.
71h – прерывание IRQ9 перенаправляется BIOS на INT 0ah.
72h – прерывание IRQ10 зарезервировано IBM, но может использо-

ваться для подключения нестандартных устройств.

73h – прерывание IRQ11 зарезервировано IBM, но может использоваться для подключения нестандартных устройств.

74h – прерывание IRQ12 обрабатывает сигнал от мыши или другого координатного устройства.

75h – прерывание IRQ13 обрабатывает исключительные ситуации математического сопроцессора.

76h – прерывание IRQ14 возникает при завершении работы жесткого диска.

77h – прерывание IRQ15 зарезервировано IBM, но может использоваться для подключения нестандартных устройств.

Организация обработки аппаратных прерываний обеспечивается процедурами, получившими название обработчиков прерываний (interruption handler) и выполняющими самостоятельные вычислительные процессы, инициированные сигналами с внешних устройств. Вообще последовательность действий обработчика близка к последовательности действий драйвера, но имеет несколько существенных особенностей:

- при входе в обработчик прерываний обработка новых прерываний обычно запрещается;
- необходимо сохранить содержимое регистров, изменяемых в обработчике;
- поскольку прерывание может приостановить любую задачу, то нужно позаботиться о корректном состоянии сегментных регистров в начале обработчика или в процессе переключения задач;
- выполнить базовые действия драйвера;
- для того, чтобы разрешить обработку новых прерываний через этот блок необходимо выдать на БПП последовательность кодов окончания обработки прерывания (end of interruption) EOI;
- выдать синхронизирующую информацию готовности буферов для последующих обменов со смежными процессами;
- если есть необходимость обратиться к действиям, которые связанны с другими аппаратными прерываниями, то это целесообразно сделать после формирования EOI, разрешив после этого обработку аппаратных прерываний командой STI;
- перед возвратом к прерванной программе нужно восстановить регистры, испорченные при обработке прерывания.

В конце кода каждого из обработчиков аппаратных прерываний необходимо включать следующие 2 строчки кода для главного БПП, если обслуживаемое прерывание обрабатывается главным БПП:

```
MOV   AL,20H
OUT   20H,AL    ; Выдача EOI на главный БПП
```

и еще 2 дополнительные строчки, если обслуживаемое прерывание обрабатывается вспомогательным БПП компьютеров типа AT и более поздних:

```
MOV  AL,20H
OUT  0A0H,AL   ; Выдача EOI на БПП–2
```

Два одинаковых числа (20H) в обеих строках – это простое совпадение. Если аппаратное прерывание не заканчивается этими строками, то микросхема 8259 в режиме применяемом в MS DOS не очистит информацию регистра обслуживания, с тем чтобы была разрешена обработка прерываний с более низкими уровнями, чем только что обработанное. Отсутствие этих строк легко может привести к краху программы, так как прерывания от клавиатуры, скорее всего, окажутся замороженными и даже Ctrl-Alt-Del окажется бесполезным. Более того, эти строки должны быть выданы вместе с командой STI, разрешающей прерывания перед любым обращением к программам DOS, связанным с этими БПП, чтобы обеспечить возможность выполнения ввода-вывода по прерыванию.

Одной из причин написания обработчика прерывания может быть использование какого-либо отдельного аппаратного прерывания. Это прерывание автоматически вызывается при возникновении определенных условий в вычислительной системе. В некоторых случаях BIOS инициализирует вектор этого прерывания так, что он указывает на процедуру, которая вообще ничего не делает, так как содержит один оператор IRET. Вы можете написать свою процедуру и изменить вектор прерываний, чтобы он обеспечивал ее выполнение при возникновении аппаратного прерывания. Наконец, вы можете написать прерывание, которое полностью заменит одну из процедур операционной системы, приспособленное к вашим программным нуждам.

В ОС для работы с дисковыми файлами часто включают специальные драйверы, выполняющиеся на уровне отдельных буферных задач ввода-вывода по информационному обмену между устройствами и буферами. Построение драйверов с использованием системы прерываний требует дополнительных действий по синхронизации вычислительных процессов. При этом наиболее распространена ситуация, когда один из процессов является источником данных, а другой приемником или потребителем.

Билет №27

1. Построение алгоритмов анализа с использованием функций предшествования.

Другим *средством задания транслятору старшинства операций* являются *таблицы предшествования*, которые устанавливают между операциями отношения предшествования. Отношение .=. *устанавливается между операциями одного порядка старшинства*. Отношение <. устанавливается, если после данной операции в выражении следует более приоритетная операция. Например, если после операции "+" следует операция "*". Если же после данной операции следует менее приоритетная операция, то между ними устанавливается отношение .>. *Алгоритм использует два стека - стек операций и стек операндов*.
Выражения просматриваются слева направо, операнды и операции заносятся в стеки до тех пор, пока между символом на вершине стека и входным символом не выполниться отношение .=. или .>. После этого выделяется тройка, для которой два символа берутся из стека операндов и один из стека операций. На вершину стека операндов заносится вспомогательная переменная, обозначающая результат, и описанные выше действия повторяются. Для выражения A+(B-C)*D ниже приводится последовательность шагов обработки.
В матрицях передування визначаються 3 можливі відношення передування:

- * =
- R <* L
- L *> R

Вони визначають різні співвідношення, бо елементи беруться за напрямком.
R $\cup$ L – відсутність передування – забороняє передування двох елементів синтаксичної конструкції.
Матриця передування – квадратна матриця, координатами якої по X і Y є повна сукупність термінальних та не термінальних позначень. На перетині координат матриці записується одне з чотирьох відношень передування. Таким чином для висхідного розбору можна визначити граматику будь-якої контекстно-залежної мови, але в таблиці буде багато надлишкових елементів, що визначають можливі помилкові варіанти конструкцій. Тобто проектування висхідного синтаксичного аналізу методом матриць передування призводить до необхідності перетворення правил опису стандартної мови на відповідну матрицю.
Звичайно кількість термінальних та нетермінальних позначень включає кілька десятків таких елементів, що дозволяю отримати відносно невеликі таблиці
Потужні комп'ютерні мови включають декілька сотень граматичних правил, а розміри матриць дещо збільшуються. В тих випадках коли матрицю передування можна перетворити на функцію передування кажуть, що такі граматики можна лінізувати, або зробити лінійними.
Операторне передування – коли нема дужок і спеціальних символів.

2. Особенности команд логических операций и сдвигов в расширениях MMX.

Основа аппаратной компоненты **расширения mmx** – *восемь* новых *регистров*, которые на самом деле являются регистрами *сопроцессора*, только вместо 80-ти разрядов *используется 64 младших разряда* (мантисса).
Основным принципом работы команд mmx является одновременная обработка нескольких единиц однотипных данных одной командой.
Важное отличие MMX-команд от обычных команд процессора в том, как они реагируют на ситуации переполнения и заема.

3. Программно-аппаратные взаимодействия при обработке прерываний в машинах IBM PC.

В реальном режиме имеются программные и аппаратные прерывания. ПП инициируются командой INT, АП - внешними событиями, по отношению к выполняемой программе. Обычно АП инициируются аппаратурой ввода/вывода после завершения выполнения текущей операции. Кроме того, некоторые прерывания зарезервированы для использования самим процессором - прерывания по ошибке деления, прерывания для пошаговой работы.
Для обработки прерываний в реальном режиме процессор использует Таблицу Векторов Прерываний. Эта таблица располагается в самом начале оперативной памяти, т.е. её физический адрес - 00000. ТВП реального режима состоит из 256 элементов по 4 байта, таким образом её размер составляет 1 килобайт. Элементы таблицы - дальние указатели на процедуры обработки прерываний. Указатели состоят из 16-битового сегментного адреса процедуры обработки прерывания и 16-битового смещения. Причём смещение хранится по младшему адресу, а сегментный адрес - по старшему.
Когда происходит программное или аппаратное прерывание, текущее содержимое регистров CS, IP а также регистра флагов FLAGS записывается в стек программы (который, в свою очередь, адресуется регистровой

парой SS:SP). Далее из таблицы векторов прерываний выбираются новые значения для CS и IP, при этом управление передаётся на процедуру обработки прерывания.
Перед входом в процедуру обработки прерывания принудительно сбрасываются флажки трассировки TF и разрешения прерываний IF. Поэтому если процедура прерывания сама должна быть прерываемой, то необходимо разрешить прерывания командой STI. В противном случае, до завершения процедуры обработки прерывания все прерывания будут запрещены.
Завершив обработку прерывания, процедура должна выдать команду IRET, по которой из стека будут извлечены значения для CS, IP, FLAGS и загружены в соответствующие регистры. Далее выполнение прерванной программы будет продолжено.
Что же касается аппаратных маскируемых прерываний, то в компьютере IBM AT и совместимых с ним существует всего шестнадцать таких прерываний, обозначаемых IRQ0-IRQ15. В реальном режиме для обработки прерываний IRQ0-IRQ7 используются вектора прерываний от 08h до 0Fh, а для IRQ8-IRQ15 - от 70h до 77h.
В защищённом режиме все прерывания разделяются на два типа - обычные прерывания и исключения (exception - исключение, особый случай). Обычное прерывание инициируется командой INT (программное прерывание) или внешним событием (аппаратное прерывание). Перед передачей управления процедуре обработки обычного прерывания флаг разрешения прерываний IF сбрасывается и прерывания запрещаются.
Исключение происходит в результате ошибки, возникающей при выполнении какой-либо команды, например, если команда пытается выполнить запись данных за пределами сегмента данных или использует для адресации селектор, который не определён в таблице дескрипторов. По своим функциям исключения соответствуют зарезервированным для процессора внутренним прерываниям реального режима. Когда процедура обработки исключения получает управление, флаг IF не изменяется. Поэтому в мультизадачной среде особые случаи, возникающие в отдельных задачах, не оказывают влияния на выполнение остальных задач.

В защищённом режиме прерывания могут приводить к переключению задач.

Механизм обработки прерываний и исключений в защищённом режиме.

Если при работе с внешним устройством вычислительная система не пользуется методом опроса его состояния, а использует механизм прерываний, то при возникновении прерывания, как мы уже говорили раньше, процессор, частично сохранив свое состояние, передает управление специальной программе обработки прерывания. Давайте теперь подробнее остановимся на том, что скрывается за словами "обработка прерывания". Одна и та же процедура обработки прерывания может использоваться для нескольких устройств ввода-вывода (например, если эти устройства используют одну линию прерываний, идущую от них к контроллеру прерываний), поэтому первое действие собственно программы обработки состоит в определении того, какое именно устройство выдало прерывание. Зная устройство, мы можем выявить процесс, который инициировал выполнение соответствующей операции. Поскольку прерывание возникает как при удачном, так и при неудачном ее выполнении, следующее, что мы должны сделать - это определить успешность завершения операции, проверив значение бита ошибки в регистре состояния устройства.