1. Архітектура пристрою з плаваючою точкою

Регистровый стек (восемь 80-битных регистров R0-R7), слово тегов, регистр управления, регистр состояния, указатель команды и указатель данных.

Для хранения данных в сопроцессоре предназначены регистры R0-R7. Эти регистры организованы в стек, и доступ к ним производится относительно вершины стека - ST. Номер регистра, соответствующего вершине стека, хранится в регистре состояния (поле TOS). Как и у ЦП, стек сопроцессора растет к регистрам с меньшими адресами. Команды, которые производят запоминание и извлечение из стека, передают данные из текущего регистра ST, а затем производят инкремент поля TOS в регистре состояния. Многие команды сопроцессора допускают неявное обращение к вершине стека, обозначаемой ST или ST(0). Для указания i-го регистра относительно вершины используется обозначение ST(i), где I = 0,:,7.

Если результат численной операции не может быть точно представлен в выбранном форм., сопроц. выполняет округл. в соответ. с полем RC. По умол. RC = 00.
00 Округление к ближайшему (или четному)
01 Округление вниз (к ∞)
10 Округление вверх (к +∞)
11 Округление к нулю (усечение)

Регистр тегов содержит 8 тегов - признаков, хар-щих содержимое соответствующего численного регистра сопроцессора. Тег принимает знач:

00 - в регистре находится действительное число;

01 - нулевое число в регистре;

10 - недействительное число (∞, денормализованное число, нечисло);

11 - пустой регистр.

2. Особливості використання регістрів з плав. точкою

Реализация численных алгоритмов на основе регистрового стека позволяет получить существенный выигрыш в скорости вычислений.

Многие команды сопроцессора допускают неявное обращение к вершине стека, обозначаемой ST или ST(0). Для указания i-го регистра относительно вершины используется обозначение ST(i), где I = 0,:,7. Например, если поле TOS регистра состояния содержит значение 011 (вершиной стека является регистр R3), то команда FADD ST,ST(2) суммирует содержимое регистров R3 и R5. Стековая организация упрощает программирование подпрограмм, допуская передачу параметров в регистровом стеке сопроцессора.

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.

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

3. Формати даних (код№2)

Для представления вещественных чисел – стандарт IEEE 754-1985. Старший разряд двоичного представления вещественного числа всегда кодирует знак числа. Остальная часть разбивается на две части: экспоненту и мантиссу. Вещественное число вычисляется как: $(-1)^S \cdot 2^E \cdot M$, где S - знаковый бит числа, E - экспонента, M - мантисса. Если 1<=M<2, то такое число называется нормализованным. При хранении нормализованных чисел сопроцессор отбрасывает целую часть мантиссы (она всегда 1), сохраняя лишь дробную часть. Экспонента кодируется со сдвигом на половину разрядной сетки, таким образом, удается избежать вопроса о кодировании знака экспоненты. Т.е. при 8-битной разрядности экспоненты код 0 соответствует числу -127, 1 - числу -126, ..., 255 числу +126 (экспонента вычисляется как код 127).

Вещ. ординарной точности - 32 бит $1{,}18 \cdot 10^{-38}$... $3{,}40 \cdot 10^{38}$
Вещ. двойной точности - 64 бит $2{,}23 \cdot 10^{-308}$... $1{,}79 \cdot 10^{308}$
Вещ.расширенной точности - 80 бит $3{,}37 \cdot 10^{-4932}$... $1{,}18 \cdot 10^{4392}$

В общем случае все множество двоичных комбинаций делится на следующие классы

- нормализованные вещественные числа со знаком;
- денормализованные вещественные числа со знаком;
- ноль со знаком;
- бесконечность со знаком;
- нечисла (NaN - not a number)

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.

4. Команди пересилань

MOVD mm, mm/m32/ir32 команда копирует 32 бита из младших разрядов MMX-регистра, либо из памяти, либо из целочисленного регистра в младшие 32 разряда MMX-регистра (старшие разряды заполняются нулями).

MOVD m32/ir32, mm команда копирует 32 бита из младших разрядов MMX-регистра в память либо в целочисленный регистр.

MOVQ mm, mm/m64 команда пересылки данных в MMX-регистр.

MOVQ mm/m64, mm команда пересылки данных из MMX-регистра.

Командой обмена является команда FXCH она осуществляет обмен двух регистров сопроцессора (следовательно, для обмена данных необходимо загрузить их сначала в стек). 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), $\log_2(10)$, $\log_2(e)$, $\log_{10}(2)$, $\ln(2)$, π, нуль соответственно. Для снятия результата используются команды FST (FSTP) приемник – 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. Пример:

```
.data                                   FLD X
X dd (dq) 0.5                           FLDL2E
.code                                   FXCH
```

5. Команди арифметичних операцій

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.

```
fld arg1                    fadd arg2 ; st[0] = st[1] + arg2
fld arg2                    fadd ; st[0] = st[0] + st[1]
```

6. Команди перевірки умов за результатами операцій з плаваючою точкою

Старший байт регистра состояния содержит 4 бита кода условия (биты 14, 10, 9, 8), аналогичные флажкам состояния FLAGS у IA-32, отражающие результат арифметических операций. Эти флажки могут быть использованы для условных переходов.

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.

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

FSTSW wrd ; Запись в память регистра состояния сопроцессора и команды центрального процессора

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.

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

7. Обчислення часткових математ. функцій

Процессор 8087 может вычислить любую элементарную трансцендентную. 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 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 лежит вне этого диапазона, то в начале программы необходимо выполнить преобразования по сведению аргумента к требуемому диапазону и запомнить данные для обратного пребразования. Следующие два преобразования могут быть проведены в безусловной форме, так как модиф-ый 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 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 u и v . Тогда

tg(V) = 2*x*y/(x*x-y*y) и u = 2*x*y; v=x*x-y*y 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); cosec(Y) = (1+(u/v)^2) / 2(u/v);

cos(Y) = (1-(u/v)^2) / (1+(u/v)^2); 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); ctg(x) = 1 / tg(x).

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

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
```

8. 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.

Команды технологии MMX работают с 64-разрядными целочисленными данными, а также с данными, упакованными в группы (векторы) общей длиной 64 бита. Такие данные могут находиться в памяти или в восьми MMX-регистрах.

Команды технологии MMX работают со следующими типами данных:

- упакованные байты (восемь байтов в одном 64-разрядном регистре)
- упакованные слова (четыре 16-разрядных слова в 64-разрядном регистре)
- упакованные двойные слова (два 32-разрядных слова в 64-разрядном регистре) 64-разрядные слова

MMX-команды имеют следующий синтаксис: instruction [dest, src] Здесь **instruction** — имя команды, **dest** обозначает выходной операнд, **src** — входной операнд.

В систему команд введено 57 дополнительных инструкций для одновременной обработки нескольких единиц данных. Большинство команд имеют суффикс, который определяет тип данных и используемую арифметику:

- US (unsigned saturation) — арифметика с насыщением, данные без знака.
- S или SS (signed saturation) — арифметика с насыщением, данные со знаком. Если в суффиксе нет ни S, ни SS, используется циклическая арифметика (wraparound).
- B, W, D, Q указывают тип данных. Если в суффиксе есть две из этих букв, первая соответствует входному операнду, а вторая — выходному.
- Команды пересылки данных (Data Transfer Instructions) между регистрами MMX и целочисленными регистрами и памятью;
- Команды преобразования типов
- Арифметические операции (Arithmetic Instructions), включающие сложение и вычитание в разных режимах, умножение и комбинацию умножения и сложения;
- Команды сравнения (Comparison Instructions) элементов данных на равенство или по величине;
- Логические операции (Logical Instructions)- И,И-НЕ,ИЛИ и Исключающие ИЛИ, выполняемые над 64 битными операндами;
- Сдвиговые операции (Shift Instructions) логические и арифметические;
- Команды управления состоянием (Empty MMX State) очистка MMX - установка признаков пустых регистров в слове тегов.

Инструкции MMX не влияют на флаги условий. Команды MMX доступны из любого режима процессора.

9. Особливості арифм. операцій в MMX

mm - MMX-регистр; m32, m64 - память объема 32 и 64 бит соответственно;

+ PADDB mm, mm/m64; PADDW mm, mm/m64; PADDD mm, mm/m64

- PSUBB mm, mm/m64; PSUBW mm, mm/m64; PSUBD mm, mm/m64

Если сумма выходит за границу допустимого диапазона, то по правилам циклической арифметики избыток отсчитывается от другой границы диапазона. "Переноса" единицы из одного элемента данных в другой не происходит.

+ PADDSB mm, mm/m64; PADDSW mm, mm/m64

+ PADDUSB mm, mm/m64; PADDUSW mm, mm/m64

- PSUBSB mm, mm/m64; PSUBSW mm, mm/m64

- PSUBUSB mm, mm/m64; PSUBUSW mm, mm/m64

Если сумма выходит за граничное значение допустимого диапазона, то результатом считается это граничное значение.

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

movq mmregl, A

movq mmreg2, B

paddb mmregZ, mmregl

movq B, mmreq2

Умножение

Все команды попарно перемножают 16-разрядные слова со знаком входного и выходного операндов. Это дает четыре 32-разрядных произведения

PMADDWD mm, mm/m64

Затем первое произведение складывается со вторым, а третье с четвертым. Суммы записываются в 32-разрядные слова выходного операнда.

PMULHW mm, mm/m64

Старшие разряды произведений записываются в 16-разрядные слова выходного операнда. Младшие разряды произведений теряются.

PMULLW mm, mm/m64

Младшие разряды произведений записываются в 16-разрядные слова выходного операнда. Старшие разряды произведений теряются.

10. Особливості лог. операцій в MMX

PAND mm, mm/m64

поразрядное логическое И своих операндов.

PANDN mm, mm/m64

поразрядное НЕ выходного операнда, а затем поразрядное логическое И между входным операндом и обращенным значением выходного.

POR mm, mm/m64

поразрядное логическое ИЛИ своих операндов.

PXOR mm, mm/m64

поразрядное логическое исключающее ИЛИ своих операндов.

Сдвиги (imm - непосредственный операнд)

PSLLW mm, mm/m64/imm; PSLLD mm, mm/m64/imm; PSLLQ mm, mm/m64/imm

Освободившиеся младшие разряды заполняются нулями.

PSRLW mm, mm/m64/imm; PSRLD mm, mm/m64/imm; PSRLQ mm, mm/m64/imm

Освободившиеся старшие разряды заполняются нулями.

PSRAW mm, mm/m64/imm; PSRAD mm, mm/m64/imm

Если сдвигается положительное число, то освободившиеся старшие разряды заполняются нулями, а если отрицательное, то единицами.

http://www.codenet.ru/progr/optimize/mmx.php#top

11. Класифікація системних програм

Системное программное обеспечение (System Software) - совокупность программ и программных комплексов для обеспечения работы компьютера и сетей ЭВМ.

СПО управляет ресурсами компьютерной системы и позволяет пользователям программировать в более выразительных языках, чем машинных язык компьютера. Состав СПО мало зависит от характера решаемых задач пользователя.

Системное программное обеспечение предназначено для:

- создания операционной среды функционирования других программ (другими словами, для организации выполнения программ);
- автоматизации разработки (создания) новых программ;
- обеспечения надежной и эффективной работы самого компьютера и вычисл-ой сети;
- проведения диагностики и профилактики аппаратуры компьютера и вычислительных сетей;
- выполнения вспомогательных технологических процессов

Классификация:

2 вида классификаций:

- *сист. управляющие* (организуют корректное функционирование всех устройств системы, отвечают за автоматизацию процессов системы)
- *сист.обрабатывающие программы*. (выполняются как специальные прикладные задачи, или приложения)

- *Базовое ПО* (base software минимальный набор программных средств, обеспечивающих работу компьютера. Относятся операционные системы и драйверы в составе ОС; интерфейсные оболочки для взаимодействия пользователя с ОС (операционные оболочки) и программные среды; системы управления файлами)
- *Сервисное* ПО (расширяют возможности базового ПО и организуют более удобную среду работы пользователя: утилиты (архивирование, диагностика, обслуживание дисков); антивирусное ПО; система программирования (редактор;·транслятор с соответствующего языка; компоновщик (редактор связей); отладчик; библиотеки подпрограмм).

12. Системні управляючі програми

Управляющая программа – системная программа, реализующая набор функций управления, который включает в себя управление ресурсами и взаимодействие с внешней средой СОИ, восстановление работы системы после проявления неисправностей в технических средствах.

Основные системные функции управляющих программ - управление вычислительными процессами и вычислительными комплексами; работа с внутренними данными ОС.

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).

При розв'язанні задач повинні бути виділені ресурси оперативної або віртуальної пам'яті, в яку завантажується задача

При наявності функцій В/В ОС повинна забезпечити монопольне або розділене між декотрими задачами, підключення ресурсів В/В.

Вхідні файли можуть розділятись та використовуватись сумісно декількома процесами, а вихідні файли виділяються для задачі монопольно.

Після того, як задача одержала деякі ресурси, їй необхідно надавати ресурси ЦПУ та процесора обміну даними (за необх.)

Ці ресурси надається планувальниками, для яких визначають стратегію планування і порядок обробки наявних запитів.

Супервізори – програми, що управляють вибором чергової активної задачі. Переключення активних задач виконуються через використання апаратних переривань або за готовністю зовнішніх пристроїв, або за таймером.

13. Системні обробляючі програми

Сист. обраб. пр. выполняются под управлением управляющей системы. Это значит, что она в полном объеме может пользоваться услугами управляющей программы и не может самостоятельно выполнять системные функции. Так, обрабатывающая программа не может самостоятельно осуществлять собственный В-В. Операции В-В обрабатывающая программа реализует с помощью запросов к управляющей программе, которая и выполняет непосредственно ввод и вывод данных.

Сист. обраб. пр. относятся программы, входящие в состав ОС: редактор текста, ассемблеры, трансляторы, редакторы связей, отладчик, библиотеки подпрограмм, загрузчик, программы обслуживания и ряд других.

14. Структура системних програм

Обычно как для системной управляющей, так и для системной обрабатывающей программ входные данные подаются в виде директив. Для операционной системы это командная строка, а для системы обработки это программа на входном языке программирования для компил. и интерпрет., и программа в машинных кодах для компановщика и загрузщика.

Через це звичайно системна програма виконується у декілька етапів , які у найбільш загальному випадку включають: ЛА, СА, Сем.обробка, оптиміз (в компіл.), ген. кодів

В системных программах используются таблицы имен и констант, которые предназначены для СА и Сем. О. . 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).

15. Задача лексичного аналізу

ЛА предназначен для преобразования текста на входном языке во внутреннюю форму, при этом текст разбивается на лексемы.

Группы лексем:
Разделители (знаки операций, именованные элементы языка)
Вспомогательные разделители (пробел, табуляция, ентер)
Код разделителя | код внутр.представления
Ключевые слова языка программирования
Код слова | код внутр.представления
Стандартные имена объектов и пользователей
Константы
Имя | тип, адрес
Комментарии

ЛА может работать в двух основных режимах: либо как подпрограмма, вызываемая синтаксическим анализатором для получения очередной лексемы, либо как полный проход, результатом которого является файл лексем.

На этапе ЛА обнаруживаются некоторые (простейшие) ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и др.).

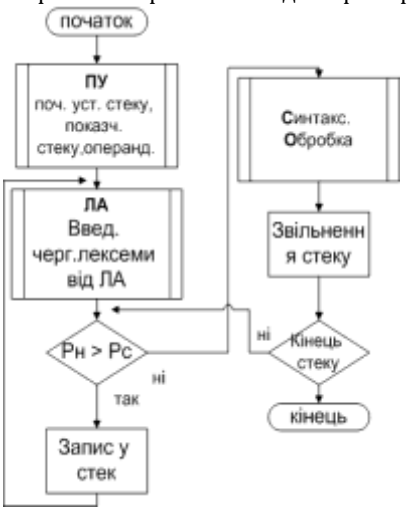
+ см. вопрос 29.

16. Задача синтаксичного аналізу

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.

Последовательный анализ лексем и сравнение приоритетов.

Если у новой операции приоритет больше, то предыдущая операция вместе с её операндом, сохраняется в стеке для дальнейшей обработки.

Иначе – выполняется семантическая обработка или устанавливается связь подчиненности для пред.операции.

И в том, и в другом случае, необх. вернуться к пред. операнду цикла, после записи в стек до введения след. пары лексем.

После семантической обработки – вернуться к сравнению приор. операций в стеке с приор. последней операции.

```
int nxtProd(struct lxNode*nd, // вказівник на початок масиву вузлів
      int nR, // номер кореневого вузла
      int nC) // номер поточного вузла
{int n=nC-1;   // номер попереднього вузла
 enum tokPrec pC = opPrF[nd[nC].ndOp],// передування поточного вузла
  *opPr=opPrG;//F;// nd[nC].prvNd = nd+n;
 while(n!=-1) // цикл просування від попереднього вузла до кореню
 {if(opPr[nd[n].ndOp]<pC//)// порівняння функцій передувань
   &&nd[n].ndOp</*_ctbz*/_frkz)
 {if(n!=nC-1&&nd[n].pstNd!=0)   // перевірка необхідності вставки
   {nd[nC].prvNd = nd[n].pstNd; // підготовка зв'язків
    nd[nC].prvNd->prnNd=/*nd+*/nC;} // для вставки вузла
  if(opPrF[nd[n].ndOp]==pskw&&nd[n].prvNd==0)nd[n].prvNd = nd+nC;
   else nd[n].pstNd = nd+nC;
  nd[nC].prnNd=/*nd+*/n; // додавання піддерева          return nR;}
```

17. Задача семантичної обробки

Семантическая обработка:

1. семантич. анализ – проверяет корректность соединения типов данных во всех операциях и операторах и определяет типы данных для промеж. результатов.

Нужно иметь таблицы соответствия операндов/аргументов и типа результата. Ключевая часть – код операции, тип аргумента. Функц. часть – тип результата.

Алгоритм анализа строится при проходе дерева, в результате будет сформировано несколько результатов. Алгоритм может быть построен через рекурсивные вызовы той же самой рек. функции или процедуры для всех поддеревьев. При достижении терм. Обозначений, все рекурс. Вызовы останавливаются.

- каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;
- при вызове функции число фактических параметров и их типы должны соответствовать числу и типам формальных параметров;
- обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке; на типы левой и правой частей в операторе присваивания; … и т.п.

2. интепретация – сборка конст. выражений. В случае реализации языка программ. в виде интерпретатора, данные для обработки получаются из констант и результатов операций ввода.
3. машинно-независимая оптимизация. Основные действия должны сократить объёмы графов внутр. представления путем удаления повтор. И неисполз. фрагментов графа. Кроме этого, могут быть выполнены действия упрощ.экви превращений. Этап требует сохранения доп. информации.
4. генерация объектных кодов выполняется: в языке высокого уровня; на уровне команд асма. Для каждого узла дерева генерируется соотвеств. последовательность команд по спец. шаблонам, а потом испол-ся трансляция асм или языка, кот. включает асм-вставки.
5. машинно-зависимая опт-ция учитывает архи и специфику команд целевого устройства

Первичная семантическая обработка в процессе синт. анализа

Это построение деревоподобных структур или графа подчиненности операций.

Если пред. операция имела высший приор., то она размещается в графе послед. операций, как подчиненная операции низшего приоритета

Если пред. операция низшего приор., то её надо продвигать ближе к корню подграфа, пока не встретим опер. с таким же приор.

```
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
…
// 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; };
```

18. Типові об'єкти системних 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 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.

19. Таблиці та операції над ними

В системных программах используются таблицы имен и констант, 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. Записи таблицы содержат поля: аргументы – по которым ведется поиск; функциональные поля (тип, адрес, код внутреннего представления). Так при трансляции программы создаются таблицы вида: Ключ (переменная/метка) и характеристики: сегмент (данных / кода), смещение (XXXX), тип (байт, слово или двойное слово / близкий или дальний).

Поиски в таблицах: линейный, упорядоч. таблицы по ключам/индексам, индексы двоичных и В-деревьев, поиск по прямому адресу, хэш-поиск).

Если размеры таблиц велики, их целесообразно создавать и заполнять по сегментам. Используются динамические структуры, в которых с созданием нового эл, формируются ссылки с пред. строк на след. Таким образом, создаются списки или деревья элементов, или древоподобные индексы.

Реализация таблиц как объектов в рамках языка C/C++ требует определения структуры для строки таблицы, в которой должны сохраняться ключевые и функциональные поля. Саму же таблицу следует определять как объект класса, включающий строки таблицы как динамическую составляющую в форме массива записей языка Pascal или структур языка C/C++ или Ассемблера.

```
struct keyStr // ключевая часть записи
{char* str;  // ключевые поля
 int nMod;};// (уточняется по варианту)
struct fStr;// функциональная часть записи
{long double _f;};//f-поле
struct recrd //структура строки таблицы
{struct keyStr key;// экземпляр структуры ключа
 struct fStr func;// экземпляр функциональной части
 char _del;};  // признак удаления
```

В соответствии с традициями программирования работы с таблицами базовыми операциями являются: select – выборка данных из таблицы; insert; delete; update.

```
// обработка таблиц по прямому адресу
// выборка по прямому адресу
struct recrd* selNmb(struct recrd*, int nElm);

// вставка по прямому адресу
struct recrd* insNmb(struct recrd*pElm,
struct recrd*tb, int nElm, int*pQnElm);
// удаления по прямому адресу
struct recrd* delNmb(struct recrd*, int nElm);

// коррекция по прямому адресу
struct recrd* updNmb(struct recrd *pElm,
struct recrd*tb, int nElm, int*pQnElm);
// сравнение строк за отношением порядка
int cmpStr(unsigned char* s1, unsigned char* s2);
// сравнение ключей за отношением неравенства
int neqKey(struct recrd*, struct keyStr);
// сравнение ключей за отношением порядка
int cmpKey(struct recrd*, struct keyStr);
// сравнения по отношению сходства
int simKey(struct recrd*, struct keyStr);
// выборка линейным поиском
struct recrd* selLin(struct keyStr kArg,
 struct recrd*tb, int ln);
// выборка двоичным поиском
struct recrd*selBin(struct keyStr kArg,
                    struct recrd*tb, int ln);
```

20. Автомати та моделі роботи автоматів

Автомат — последовательность (кортеж) из пяти элементов (Q,Σ,δ,S_0,F), где:

- Q — конечное множество состояний автомата
- Σ — алфавит языка, который понимает автомат
- δ — функция перехода, такая что $\delta : Q \times \Sigma \to Q$
- S_0 — начальное состояние, состояние когда автомат еще не прочитал ни одного символа
- F — множество состояний, называемое «конечные состояния».

Основу простейшей программной реализации конечного автомата составляют коды состояния автомата и так называемая матрица переходов автомата. Коды состояния, чаще всего, определяются перечислимым типом с именованными значениями состояний.

```
enum autStat
{S0,   // S0 - Начальное состояние
 S1,   // S1 - Первое состояние
 S2,   // S2 - Второе состояние
 Se    // Se - Последнее состояние
};
```

Коды сигналов также удобно определять другим перенумерованным типом с именованными значениями кодов сигналов. Пример описания типа для представления 5-ти видов сигналов,приведен ниже.

```
enum autSgn
{sg0, // sg0 - Начальный сигнал
 sg1,  // sg1 - Первый сигнал
 sg2,  // sg2 - Второй сигнал
 sg3,  // sg3 - Третий сигнал
 sg    // sg - Последний сигнал
};
```

Матрица переходов определяется двухмерным массивом типа enum autStat, первый индекс которого определяет целое число, соответствующее предшествующему состоянию автомата, а второй индекс – число, которое соответствует сигналу или классу сигнала для перевода автомата в следующее состояние.

```
enum autStat nxtSts[Se+1][ sg+1] =
{{S0,S1,S2,S0,S0},          // для S0
 {S1,S1,S1,S2,Se},          // для S1
 {S1,S2,S2,S2,S2},          // для S2
 {S1,Se,Se,Se,Se}           // для Se
};
```

Функция переходов определена в лабораторной работе следующим образом:

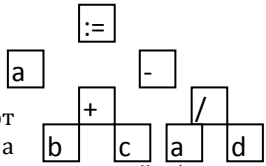
```
enum autStat nxtStat(enum autSgn sgn)
{static enum autStat s=S0;//текущее состояние лексемы
 return s=nxtSts[s][sgn];} // новое состояние лексемы
```

21. Графи та їх використання для внутрішнього подання

В результате синтаксической обработки как правило создаются графы синтаксического разбора, которые показывают связи между терминальными и нетерминальными выражениями, что выражается при синтаксическом разборе.

С другой стороны, для использования всех видов семантической обработки более удобными являются графы подчиненности (підлеглості) операций.

a := b + c - a / d.



Граф подчиненности однозначно определяет порядок выполнения операций в конструкции. Чтобы иметь возможность получения графов подчиненности, к структурам, которые отвечают узлам лексем необходимо добавить ссылки к подчиненным узлам, а для терминальных указателей … могут быть заменены на указатель на символьный образ и указатель на внутреннее представление соответствующего терминального указателя.

```
struct lxNode        //узел дерева, САГ или УСГ
{int ndOp;           //код операции или типа лексемы
 unsigned stkLength;  // номер модуля для терминалов
 struct lxNode* prvNd, *pstNd;   // связи с подчиненными
 int dataType;       // код типа возвращаемых данных
 unsigned resLength;  //длина результата
 int x, y, f;        //координаты размещения в входном файле
 struct lxNode*prnNd; //связь с родительским узлом
};
```

Обычно записи таблиц записываются один за другим в заранее определенных или динамически созданных массивах:

```
char *imgs[]={"b","a","1","f","c","g","k","ky","n","nD", "nU","el"};
struct lxNode pic1[]=                    // b=f(a-1)+a-1 – не соответствует графу выше ^  !!!
{{_nam,(struct lxNode*)imgs[0], NULL, 0, 0, 0, 0, 0, &token[1], 0},
 {_ass,&pic1[0], &pic1[2], 0, 0, 0, 0, 0, NULL, 0},
 {_nam, (struct lxNode*)imgs[3], NULL, 0, 0, 0, 0, 0, NULL, 0},
 {_brkt,&pic1[2], &pic1[5], 0, 0, 0, 0, 0, NULL, 0},
 {_nam,(struct lxNode*)imgs[1], NULL, 0, 0, 0, 0, 0, NULL, 0},
 {_sub, &pic1[4], &pic1[6], 0, 0, 0, 0, 0, NULL, 0},
 {_srcn,(struct lxNode*)imgs[2], NULL, 0, 0, 0, 0, 0, NULL, 0},
 {_add, &pic1[3], &pic1[5], 0, 0, 0, 0, 0, NULL, 0} };
```

22. Способи організації таблиць та індексів

Индексы создаются в таблицах с помощью ссылки в древовидную структуру. Для построения индексов м.б. использованы разные структуры с ссылками – списки, где пред. эл. связан с последующим через ссылку. Используются динамические структуры, в которых с созданием нового эл, формируются ссылки с пред. строк на след. Таким образом, создаются списки или деревья элементов, или древоподобные индексы (исп. В БД для повышения скорости поиска) – индексы двоичных и B-деревьев.

Таблицы – сложные структуры данных, с помощью которых можно значительно увеличить эффективность программы. Их основное назначение состоит в поиске информации о зарегистрированном объекте по заданному аргументу поиска. Результатом поиска обычно является элемент таблицы, ключ которого совпадает с аргументом поиска в таблице. Есть несколько способов организации таблиц, для этого используют директивы определения элементов таблиц: struc и record. **Директива struc** предназначена для определения структурированных блоков данных. Структура представляется последовательностью полей. Поле структуры – последовательность данных стандартных ассемблерных типов, которые несут информацию об одном элементе структуры. Определение структуры задает шаблон структуры:

имя_структуры struc

последовательность директив DB,DW,DD,DQ,DT

имя_структуры ends

шаблон структуры представляет собой только прототип или предварительную спецификацию элемента таблицы. Для статического резервирования памяти и инициализации значений используется оператор вызова структуры:

имя_переменной имя_структуры <спецификация_инициализации>

переменная ассоциируется с началом структуры и используется с именами полей для обращения к различным полям в структуре:

student iv_groups <’timofey’,19,5>

mov ax,student.age ; ax => 19

с помощью коэффициента кратности с ключевым словом dup можно резервировать статическое количество памяти. Директива record предназначена для определения двоичного набора в байте или слове. Ее применение аналогично директиве struc в том отношении, что директива record только формирует шаблон, а инициализация или резервирование памяти выполняется с помощью оператора вызова записи:

имя_записи record имя_поля : длина поля [= начальное значение],...

длина поля задается в битах, сумма длин полей не должна превышать 16, например:

iv_groups record number:5=3,age:5=19,add:6

оператор вызова записи выглядит следующим образом:

имя_переменной имя_записи <значение полей записи>

к полям записи применимы следующие операции:

width поля – длина поля, mask поля – значение байта или слова, определяемого переменной, в которой установлены в 1 биты данного поля, а остальные равны нулю. Для получения в регистре al значения поля необходимо сделать следующее:

mov al, student

and al, mask age

mov cl, age

shr al,cl ; выравнивание поля age вправо существует несколько методов поиска в таблицах – линейный, двоичный и прямой (хэш-поиска). смотри ниже.

23. Організація таблиць як масивів записів

Простейший способ построения информационной базы состоит в определении структуры отдельных элементов, которые встраиваются в структуру таблицы. Как уже отмечалось, аргументом поиска в общем случае можно использовать несколько полей. Каждый элемент обычно сохраняет несколько (m) характеристик и занимает в памяти последовательность адресованных байтов. Если элемент занимает k байтов и надо сохранять N элементов, то необходимо иметь хотя бы kN байтов памяти.

Все элементы разместить в k последовательных байтах и построить таблицу с N элементов в виде массива. Пример такой таблицы приведен ниже, где элементы задаются структурой `struct` в языке C, записями `record` в языке Pascal, длиной k байтов, определяемой суммой размеров ключевой и функциональной части элемента таблицы.

```
struct keyStr// ключевая часть записи
{char* str;  // ключевые поля
 int  nMod;};// (уточняется по варианту)
struct fStr;// функциональная часть записи
{long double _f;};//f-поле
struct recrd // структура строки таблицы
{struct keyStr key;// экземпляр структуры ключа
 struct fStr func;// экземпляр функциональной части
   char _del;};     // признак удаления
```

24. Організація таблиць у вигляді структур з покажчиків

Реализация, в которой для хранения N элементов по k байт использует kN байт памяти часто бывает избыточной. В больших таблицах данные часто повторяются, что наталкивает на мысль вынесения повторяемых данных в отдельные таблицы и организации некоторого механизма связывания этих таблиц. Такой подход используется при построении баз данных и получил название нормализации. Но каким же образом происходит связывание двух или нескольких таблиц? Итак, мы вынесли некоторые повторяемые элементы из основной таблицы во вспомогательную. Теперь на их место введем дополнительное поле (естественно, оно должно быть меньше замещенных данных). Оно будет указывать на вынесенный элемент вспомогательной таблицы. Данное поле удобно представлять в виде указателя или некоторого уникального идентификатора.

Частным случаем такой структуры может быть древовидная структура. Вместо того, чтобы полностью хранить данные связанных «листочков» дерева, необходимо использовать указатели на эти элементы. Это не только значительно упрощает реализацию такой структуры, но и позволяет сэкономить память.

```
  struct lxNode          //узел дерева, САГ или УСГ
  {int ndOp;             //код операции или типа лексемы
   unsigned stkLength;   // номер модуля для терминалов
   struct lxNode* prvNd, *pstNd;   // связи с подчиненными
   int dataType;         // код типа возвращаемых данных
   unsigned resLength;   //длина результата
   int x, y, f;          //координаты размещения в входном файле
   struct lxNode*prnNd; //связь с родительским узлом
  };                     //пример взят с вопроса #21
```

25. Організація пошуку

В большинстве системных программ главной целью поиска является определение характеристик, связанных с символическими обозначениями элементов входного языка (ключевых слов, имен, идентификаторов, разделителей, констант и т.п.). Эти элементы рассматр-ся как аргументы поиска или ассоциативные признаки информации обозначений.

Поиски в таблицах: линейный, упорядоч. таблицы по ключам/индексам, индексы двоичных и В-деревьев, поиск по прямому адресу, хэш-поиск).

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.

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.

Самым быстрым методом поиска в больших таблицах является *прямой*, основанный на обращении к элементу с ключевой частью Кі 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+ Н(Кі), где хеш-функция Н(Кі) - 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

26. Лінійний пошук

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.

```
// порівняння за відношенням порядку (neq = not_equal)
int neqKey(struct recrd* el, struct keyStr kArg) {
 return (strcmp(el->key.str, kArg.str)
 || el->key.nMod != kArg.nMod);
}
// вибірка за лінійним пошуком
struct recrd* selLin(struct keyStr kArg, struct recrd* tb, int ln) {
  int i;
  int pos = 0;
  struct recrd* temp = (struct recrd*)malloc(100*sizeof(*tb));
  struct recrd* res  = NULL;
  for (i = 0; i < ln; i++)
    if (!neqKey(&tb[i], kArg)) {
      temp[pos] = tb[i];
      pos++;
    }
  res = (struct recrd*) malloc((pos+1)*sizeof(*tb));
  for(i = 0; i < pos; i++) {
    res[i] = temp[i];
  }
  free(temp);
  res[pos] = emptyElm;
  res[pos].key.str = NULL;
  return res; }
```

27. 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

```
// сортування для двійкового пошуку
struct recrd* srtBin(struct recrd*tb, int ln){
  int n, n1;
  struct recrd el;
  for(n = 0; n < ln; n++)
    for(n1 = n+1; n1 < ln; n1++)
     if(cmpKey(&tb[n],tb[n1].key) > 0){
        el = tb[n];
        tb[n] = tb[n1];
        tb[n1]= el;
      }
  return tb;
}
// вибірка за двійковим пошуком
struct recrd* selBin(struct keyStr kArg, struct
recrd* tb, int ln){
  int i, nD = -1, nU = ln, n = nD + nU>>1;

  int pos = 0;
  struct recrd* temp = (struct
recrd*)malloc(100*sizeof(*tb));
  struct recrd* res  = NULL;

  while(i = cmpKey(&tb[n], kArg)){
    if(i > 0)
      nU = n;
    else
      nD = n;
    n = (nD + nU) >> 1;
    if(n == nD) {
      return NULL;
    }
  }

  while (!cmpKey(&tb[n],kArg))
    n--;
  n++;
  while (!cmpKey(&tb[n], kArg)) {
    temp[pos] = tb[n];
    pos++;
    n++;
  }

  res = (struct recrd*)malloc((pos+1)
*sizeof(*tb));
  for(i = 0; i < pos; i++)
    res[i] = temp[i];
  free(temp);
  temp = NULL;

  res[pos] = emptyElm;
  res[pos].key.str = NULL;

  return res;
}
```

28. Пошук за прямою адресою, хеш-пошук

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.

Прямой поиск: mov eax, [ebx][edi] xlat

Метод деления (размер таблицы hashTableSize - простое число). Этот метод использован в примере ниже. Хеширующее значение hashValue, изменяющееся от 0 до (hashTableSize - 1), равно остатку от деления ключа на размер хеш-таблицы. Вот как это может выглядеть:

```
makehashkey macro value,hashkey
xor ax,ax
xor dx,dx
  mov ax, value
  div hash_table_size
  mov hashkey,dx ; остаток от деления
хранится в dx
endm        ; деления хранится в dx
; запись значения в хеш-таблицу
pushtotable macro value
makehashkey value,cx ; получаем хеш-ключ
xor si,si        ; (индекс для хеш-таблицы)
mov dx,cx
l3:
add si,2 ; вычисляем смещение в хеш-
таблице по индексу
loop l3
mov bx,hash_table[si] ; записываем
значение элемента хеш-таблицы (ссылка
на список)
mov dx,value
mov [bx],dx записываем в список значение
value
endm
```

29. Основні методи лекс.аналізу

ЛА можно построить методами теории автоматов

Состояния автоматов – последовательность целый чисел или перенум. тип данных (enum).

Для перевода автомата из одного состояния в другое, необходимо определить набор сигналов, для него следует определить другой enum.

Состояния законч.(имена, ключ.слова, константы с фикс./плав. тчк, дробной частью, показ. экпоненты, буквенные и строчные конст.) и незакончен. («..) лексем, комментарии.

```
enum ltrType
{dgt,      //c0 десятичная цифра
 ltrexplt,//c1 буква–признак экспоненты
 ltrhxdgt,//c2 литера–шестнадцатеричная цифра…
dlobrct, //c14 открытые скобки
 dlcbrct, //c15 закрытые скобки
 ltrcode=16//c16 признак возможности кодирования
};
```

Сигналы – по матрице передувань. матрица переходов автомата определяется двумерным массивом типу enum autStat, первый индекс которого определяет целое число, соответствующее предыдущему состоянию, а второй индекс – число, соответствующее сигналу или классу сигнала для перевода в следующее состояния.

Коды состояния чаще всего определяются перечислимым типом с именованными значениями всех возможных заключительных и промежуточных состояний:

```
enum autStat
{Eu,   // Eu – Неклассифицированный объект
 S0,   // S0 – Разделитель
 S1g,  // S1g – Знак числовой константы …
 En,   // En – Неправильное имя
 Eo};  // Eo – Недопустимое сочетание операций
```

Функция лексического анализа очередной лексемы

```
int lxAnlzr(void)
{static int LexNmb = 0;
 static enum autStat s=S0, sP;
                    // текущее и предыдущее состояние лексемы
 char l;                    // очередная литера
 enum ltrType c;            // класс очередной литеры
 lxInit();          // заполнение позиции в тексте и таблицах
 do{sP=s;                   // запоминание состояния
       l=ReadLtr();// чтение литеры
       c=ltCls[l];          // определение класса литеры
s=nxtSts[s][c<dlmaux?c:dlmaux];
// состояние лексемы
 }while(s!=S0);// проверка конца лексемы
 switch (sP)
 {case S1n:// поиск ключевых слов и имен
       frmNam(sP, x);
break;
  default:          // не дошли до классифицированных ошибок
  case Eu: Ec: Ep: Eq: En: Eo:// обработка ошибок
eNeut(lxNmb);               // фиксация ошибки
  case S1c: S2c: S1p: S2s:// формирование констант
frmCns(sP, x); break;
  case S0:  dGroup(lxNmb);// анализ групповых разделителей       } return lxNmb++; }
```

30. Граматики та їх застосування

Грамматикой называется четверка G = (Vt, Vn, R, e є Vn), где Vn - конечное множество нетерминальных символов (все обозначения, кот. определяются через правила), Vt - множество терминалов (не пересекающихся с Vn), e - символ из Vn, называемый начальным/конечным, R - конечное подмножество множества: $(Vn \cup Vt)^* Vn (Vn \cup Vt)^* x (Vn \cup Vt)^*$, называемое множеством правил. Множество правил R описывает процесс порождения цепочек языка. Элемент $r_i = (\alpha, \beta)$ множества R 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.

Терминалы – обозначения или элементы грамм, кот. не подлежат дальнейшему анализу (разделители, лексемы).

Нетерминалы – требуют для своего опрделения правил в некотором формате, чаще всего в формате правил текстовой подстановки.

Формальная грамматика или просто грамматика в теории формальных языков (это множество конечных слов (син. строк, цепочек) над конечным алфавитом) — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита. Различают порождающие и распознающие (или аналитические) грамматики — первые задают правила, с помощью которых можно построить любое слово языка, а вторые позволяют по данному слову определить, входит оно в язык или нет.

Порождающие грамматики

Vt, Vn, e є Vn, R — набор правил вида: «левая часть» «правая часть», где:

«левая часть» — непустая последовательность терминалов и нетерминалов, содержащая хотя бы один нетерминал; «правая часть» — любая последовательность терминалов и нетерминалов

Применение

1. Контекстно-свободные грамматики широко применяются для определения грамматической структуры в грамматическом анализе.
2. Регулярные грамматики (в виде регулярных выражений) широко применяются как шаблоны для текстового поиска, разбивки и подстановки, в т.ч. в лексическом анализе.

Терминальный алфавит:

Vt={'0','1','2','3','4','5','6','7','8','9','+','-','*','/','(',')'}.

Нетерминальный алфавит:

{ ФОРМУЛА, ЗНАК, ЧИСЛО, ЦИФРА }

Правила:

1. ФОРМУЛА ФОРМУЛА ЗНАК ФОРМУЛА (формула есть две формулы, соединенные знаком)
2. ФОРМУЛА ЧИСЛО (формула есть число)
3. ФОРМУЛА (ФОРМУЛА) (формула есть формула в скобках)
4. ЗНАК + | - | * | / (знак есть плюс или минус или умножить или разделить)
5. ЧИСЛО ЦИФРА (число есть цифра)
6. ЧИСЛО ЧИСЛО ЦИФРА (число есть число и цифра)
7. ЦИФРА 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (цифра есть 0 или 1 или ... 9)

Начальный нетерминал:

ФОРМУЛА

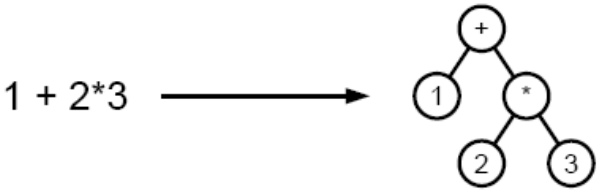
Аналитические грамматики

Порождающие грамматики - не единственный вид грамматик, однако наиболее распространенный в приложениях к программированию. В отличие от порождающих грамматик, аналитическая (распознающая) грамматика задает алгоритм, позволяющий определить, принадлежит ли данное слово языку. Например, любой регулярный язык может быть распознан при помощи грамматики, задаваемой конечным автоматом, а любая контекстно-свободная грамматика — с помощью автомата со стековой памятью. Если слово принадлежит языку, то такой автомат строит его вывод в явном виде, что позволяет анализировать семантику этого слова.

31. Трансляція шляхом граматичного аналізу

Грамматический анализ (грамматический разбор). Процесс сопоставления линейной последовательности лексем (слов, лексем) языка с его формальной грамматикой. Результатом обычно является дерево разбора или абстрактное синтаксическое дерево. Для грамматического разбора компьютерных языков используются контекстно-свободные грамматики. Это обоснованно тем, что грамматики более общих типов по иерархии Хомского (контекстно-зависимые и, тем более, неограниченные) гораздо труднее поддаются определённому анализу, а более простые (регулярные грамматики) не позволяют описывать вложенные конструкции языка, и поэтому недостаточно выразительны.

Методы грамматического разбора можно разбить на 2 больших класса - восходящие и нисходящие - в соответствии с порядком построения дерева грамматического разбора. Нисходящие методы (методы сверху вниз) начинают с правила грамматики, определяющего конечную цель анализа с корня дерева грамматического разбора и пытаются его наращивать, чтобы последующие узлы дерева соответствовали синтаксису анализируемого предложения. Восходящие методы (методы снизу вверх) начинают с конечных узлов дерева грамматического разбора и пытаются объединить их построением узлов все более и более высокого уровня до тех пор, пока не будет достигнут корень дерева.



+ см вопрос 29.

32. Класифікація граматик за Хомським

По иерархии Хомского, грамматики делятся на 4 типа, каждый последующий является более ограниченным подмножеством предыдущего (но и легче поддающимся анализу):

тип 0. неограниченные грамматики — возможны любые правила. 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. контекстно-зависимые грамматики — левая часть может содержать один нетерминал, окруженный «контекстом» (последовательности символов, в том же виде присутствующие в правой части); сам нетерминал заменяется непустой последовательностью символов в правой части.

Грамматика G = (VT, VN, P, S) называется *неукорачивающей грамматикой*, если каждое правило из P имеет вид $\alpha \to \beta$, где $\alpha \in (VT \cup VN)^+$, $\beta \in (VT \cup VN)^+$ и $|\alpha| \subset |\beta|$.

Грамматика G = (VT, VN, P, S) называется *контекстно-зависимой (КЗ)*, если каждое правило из P имеет вид $\alpha \to \beta$, где $\alpha = \xi_1 A \xi_2$; $\beta = \xi_1 \gamma \xi_2$; $A \in VN$; $\gamma \in (VT \cup VN)^+$; $\xi_1, \xi_2 \in (VT \cup VN)^*$.

Грамматику типа 1 можно определить как неукорачивающую либо как контекстно-зависимую.

Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых неукорачивающими грамматиками, совпадает с множеством языков, порождаемых КЗ-грамматиками.

тип 2. контекстно-свободные грамматики — левая часть состоит из одного нетерминала. Грамматика G = (VT, VN, P, S) называется *контекстно-свободной (КС)*, если каждое правило из P имеет вид $A \to \beta$, где $A \in VN$, $\beta \in (VT \cup VN)^+$.

Грамматика G = (VT, VN, P, S) называется *укорачивающей контекстно-свободной (УКС)*, если каждое правило из P имеет вид $A \to \beta$, где $A \in VN$, $\beta \in (VT \cup VN)^*$.

Грамматику типа 2 можно определить как контекстно-свободную либо как укорачивающую контекстно-свободную. Возможность выбора обусловлена тем, что для каждой УКС-грамматики существует почти эквивалентная КС-грамматика.

тип 3. регулярные грамматики — более простые, эквивалентны конечным автоматам.

Грамматика G = (VT, VN, P, S) называется *праволинейной*, если каждое правило из P имеет вид $A \to tB$ либо $A \to t$, где $A \in VN$, $B \in VN$, $t \in VT$.

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

*Грамматику типа 3(регул-ую,Р-грамматику)*можно опр-ть как праволин-ую либо как леволин-ую.

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

- **праволинейной**, если каждое правило из **P** имеет вид: **A→xB** или **A→x**, где **A**, **B** - нетерминалы, **x** - цепочка, состоящая из терминалов;

G2 = ({S,}, {0,1}, P, S), где P:

1) $S \to 0S$; 2) $S \to 1S$; 3) $S \to \varepsilon$, определяет язык {0, 1}*.

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; Данная грамматика порождает простейшие арифметические выражения.

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) $F \to a$.

- **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);

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 $\{ a^n b^n c^n \}, n \ge 1$.

- **грамматикой свободного вида**, если в ней отсутствуют выше упомянутые ограничения.

Примечание1.Согласно определению каждая правол-ая грамматика есть 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.

33. Граматики для лексичного аналізу

Регулярные грамматики широко применяются как шаблоны для текстового поиска, разбивки и подстановки, в т.ч. в лексическом анализе.

Для решения задачи ЛА 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.

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.

Обычно все лексемы делятся на классы. Примерами таких классов являются числа (целые, восьмеричные, шестнадцатиричные, действительные и т.д.), идентификаторы, строки. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители). Как правило, ключевые слова - это некоторое конечное подмножество идентификаторов.

Для построения автомата лексического анализа нужно определить сигналы его переключения по таблицам классификаторов литер с кодами, удобными для использования в дальнейшей обработке. Тогда каждый элемент таблицы классификации должен определить код, существенный для анализа лексем, приведенный в форме кода сигналов автомата лексического анализа соответствующего типа.

```
enum ltrType
{dgt,     //c0 десятичная цифра
 ltrexplt,//c1 буква–признак экспоненты
 ltrhxdgt,//c2 литера–шестнадцатеричная цифра
 ltrtpcns,//c3 литера–определитель типа константы
 ltrnmelm,//c4 литеры, допустимые только в именах
 ltrstrlm,//c5 литеры для ограничения строк и констант
 ltrtrnfm,     //c6 литеры начала перекодирования литер строк
 nc,     //c7 неклассифицированные литеры
 dldot,     //c8 точка как разделитель и литера констант
 ltrsign, //c9 знак числа или порядка
 dlmaux,     //c10 вспомогательные разделители типа пробелов
 dlmunop, //c11 одиночные разделители операций
 dlmgrop, //c12 элемент начала группового разделителя
 dlmbrlst,//c13 разделители элементов списков
 dlobrct, //c14 открытые скобки
 dlcbrct, //c15 закрытые скобки
 ltrcode=16//c16 признак возможности кодирования
 };
```

Выходом лексического анализатора является таблица лексем (или цепочка лексем). Эта таблица образует вход синтаксического анализатора, который исследует только один компонент каждой лексемы — ее тип. Остальная информация о лексемах используется на более поздних фазах компиляции при семантическом анализе, подготовке к генерации и генерации кода результирующей программы.

34. Граматики для синтаксичного аналізу

Синтаксический анализатор (синт. разбор) — это часть компилятора, которая отвечает за выявление основных синтаксических конструкций входного языка. В задачу синтаксического анализа входит: найти и выделить основные синтаксические конструкции в тексте входной программы, установить тип и проверить правильность каждой синтаксической конструкции и, наконец, представить синтаксические конструкции в виде, удобном для дальнейшей генерации текста результирующей программы.

В основе синтаксического анализатора лежит распознаватель текста входной программы на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью КС-грамматик, реже встречаются языки, которые, могут быть описаны с помощью регулярных грамматик.

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.

```
struct lxNode//вузол дерева, САГ або УСГ
{int x, y, f;//координати розміщення у вхідному файлі
 int ndOp;        //код типу лексеми
 int dataType;   // код типу даних, які повертаються
 unsigned resLength;  //довжина результату
 struct lxNode* prnNd;//зв'язок з батьківським вузлом
 struct lxNode* prvNd, pstNd;// зв'язок з підлеглими
 unsigned stkLength;//довжина стека обробки семантики
};
```

Результатом работы распознавателя КС-грамматики входного языка является последовательность правил грамматики, примененных для построения входной цепочки. По найденной последовательности, зная тип распознавателя, можно построить цепочку вывода или дерево вывода. В этом случае дерево вывода выступает в качестве дерева синтаксического разбора и представляет собой результат работы синтаксического анализатора в компиляторе.

Однако ни цепочка вывода, ни дерево синтаксического разбора не являются целью работы компилятора. Дерево вывода содержит массу избыточной информации, которая для дальнейшей работы компилятора не требуется. Эта информация включает в себя все нетерминальные символы, содержащиеся и узлах дерева, — после того как дерево построено, они не несут никакой смысловой нагрузки и не представляют для дальнейшей работы интереса.

35. Граматики висхідного синтаксичного розбору

В общем случае для восходящего разбора строится так называемые грамматики предшествования. В грамматике предшествования строится матрица по парных 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).

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. 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 понятием 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.

Методы восходящего анализа (см. ниже, если вопрос звучит иначе):

1. Простое предшествование
2. Свертка-перенос

```
int nxtProd(struct lxNode*nd,      // вказівник на початок масиву вузлів
              int nR,       // номер кореневого вузла
              int nC)       // номер поточного вузла
{int n=nC-1;         // номер попереднього вузла
 enum tokPrec pC = opPrF[nd[nC].ndOp],// передування поточного вузла
     *opPr=opPrG;
 while(n!=-1)        // цикл просування від попереднього вузла до кореню
 {if(opPr[nd[n].ndOp]<pC// порівняння функцій передувань
 {if(n!=nC-1&&nd[n].pstNd!=0)              // перевірка необхідності вставки
    {nd[nC].prvNd = nd[n].pstNd;  // підготовка зв'язків
     nd[nC].prvNd->prnNd=/*nd+*/nC;}      // для вставки вузла
     if(opPrF[nd[n].ndOp]==pskw&&nd[n].prvNd==0)
           nd[n].prvNd = nd+nC;
     else nd[n].pstNd = nd+nC;
           nd[nC].prnNd=/*nd+*/n; // додавання піддерева
return nR;}
```

35. Методи висхідного розбору при синтаксичному аналізі (если вопрос звучит иначе)

Методы восходящего анализа:

1. Простое предшествование

Построение отношения предшествования начинается с перечисления все пар соседних символов правых частой правил, которые и определяют все варианты отношений смежности для границ возможных выстраиваемых на них деревьях. (Сразу отметим, что правила с одним символом в правой части являются для этой цели непродуктивными). Далее, в каждой паре возможны следующие комбинации терминальных/нетерминальных символов и продуцируемые из них элементы отношений:

1. Все пары соседних символов находятся в отношении «равенства» **=** или одинаковой глубины. При этом для метода «свертка-перенос» нетерминальный символ не может являться символом входной строки, поэтому пары с нетерминалом справа в построении отношения предшествования для такого метода не участвуют.
2. Для пары нетерминал-терминал правая граница поддерева, выстраиваемая на основе нетерминала (множество **LAST**$^+$) находится «глубже» правого терминала, т.е.
3. Аналогичное обратное отношение выстраивается для пары терминал-нетерминал: левая нижняя граница поддерева, выстраиваемого на нетерминале «глубже» левого терминала
4. Наиболее сложное, но и самое «продуктивное» соотношение – два рядом стоящих нетерминала, которые производят сразу два отношения: правая граница левого поддерева «глубже» левой нижней границы правого смежного поддерева, но при этом корневая вершина левого поддерева «выше» той же самой левой нижней границы правого смежного поддерева.

2. Свертка-перенос

Основные принципы восходящего разбора с использованием магазинного автомата (МА), именуемого также методом **«свертка-перенос»**:

- Первоначально в стек помещается первый символ входной строки, а второй становится текущим;
- МА выполняет два основных действия: **перенос** (сдвиг - **shift**) очередного символа из входной строки в стек (с переходом к следующему);
- Поиск правила, правая часть которого хранится в стеке и замена ее на левую – **свертка (reduce);**
- Решение, какое из действий – перенос или свертка выполняется на данном шаге, принимается на основе анализа пары символов – символа в вершине стека и очередного символа входной строки. Свертка соответствует наличию в стеке **основы**, при ее отсутствии выполняется перенос. Управляющими данными МА является таблица, содержащая для каждой пары символов грамматики указание на выполняемое действие (свертка, перенос или недопустимое сочетание -ошибка) и сами правила грамматики.
- Положительным результатом работы МА будет наличие начального нетерминала грамматики в стеке при пустой входной строке.

Как следует из описания, алгоритм не строит синтаксическое дерево, а производит его обход «снизу-вверх» и «слева-направо».

36. Матриці передувань

В общем случае для восходящего разбора строится так называемые грамматики предшествования. В грамматике предшествования строится матрица по парных отношений всех терминальных и не терминальных символов. Чтобы использовать стековый алгоритм в случае скобок или операторов языком прогр., следует использовать функции предшествования f(op), g(op). Для мат. операций эти функции имеют одно значения, для скобок – другое. Однако этот метод не даёт однозначного решения для построения алгоритмов обрабатывания сложных операторов.

Более общий подход – построение матрицы предшествования. В матрице определяют отношение предш. для всех возможных пар пред. и след. терм. и нетерм. обозначений.

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).

Матрица предш. строится так, что на пересечении определяется приоритет отношения. Матрица квадратная и каждая размерность включает номера терм. и нетерм. обозначений. Матрица предш. – универсальный механизм определения грамматик разбора.

В связи с тем, что след. обозначение м.б. не терминальным, это вызывает необходимость повторения для полного разбора и может вызвать неоднозначность грамматик.

Линеаризация грамматик предш. – действия по превращению матрицы предш. на функции предш. в большинстве случаев, полная Л. невозможна.

Матрица предш.

```
enum autStat nxtSts[Se+1][ sg+1] =
{{S0,S1,S2,S0,S0},        // для S0
 {S1,S1,S1,S2,Se},        // для S1
 {S1,S2,S2,S2,S2},        // для S2
 {S1,Se,Se,Se,Se}         // для Se
};
```

Функция предш.

```
enum autStat nxtStat(enum autSgn sgn)
{static enum autStat s=S0;//текущее
состояние лексемы
 return s=nxtSts[s][sgn];} // новое
состояние лексемы
```

37. Нисхідний розбір (код см. №16/35)

Идея: правила определения грамм. в формулах Бекуса необх. превратить в матрицы/функции предш.

При доказательстве корректности конструкций текстов правилам подстановки необходимо выполнять анализ входного текста любыми из альтернативних частей правой части правила.

Управление передаётся отдельным программам ресурса доведення, кот. могут обращаться к другим ресурсам довед, ил к рес. довед, которые вызываются рекурсивно.

Отсюда возникает проблема зацикливания при обработке леворекурсивных правил.

1. Метод рекурсивного спуска (имеет модификации, испол. в большинстве систем автоматиз. построения компилятора)
2. Метод синтаксических графов
3. LL-парсер
4. Парсер старьёвщика

Рекурсивный спуск: Для объяснения 1 используют понятие ресурса доведення. Нисходящий разбор начинается с конечного нетерминала грамматики, кот. должен быть получен в результате анализа последовательности лексем.

Для каждого нетерминала исп. 1 или несколько правил. Несколько правил подстановки м.б. объединены в одно с помощью "|". Для опред. соответствия последовательности лексем, обращаемся к ресурсам доведення правой части правила.

Если исп. один ресур доведення, то он начинает обрабатывать эл. слева направо.

Если рекурс. обращение выполняется в начале или слева правой части правила, то это приводит к возможности возникновения зацикливания. Для того, чтоб представить правила подстановки для метода рекурсивного спуска, их превращают в правила с правостор. рекурсией.

Чтоб реализовать алгоритм рек.спуска, необходимо иметь входную послед. лексем и набор управляющих правил в виде направленного графа. Такой подход позволяет строить дерево разбора, в котором из распознанных нетерм. узлов, формируются указатели на поддеревья и/или терм.узлы.

Некоторые конструкции (case/if) в графах пидлеглости, следует дополнить спец. связями – указатели на результаты общего выражения условия и обратные свзи для выхода из цикла.

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 $i = 1,2...n; a_i \neq a_j, i \neq j; \alpha \subset (\Sigma \cup N)*$

38. Метод синтаксичних графів

Для представления грамматики используется списочная структура, называемая синтаксическим графом. Можно использовать спец. узлы с информацией об использовании синт. разбора.

Эти узлы имеют 4 элемента:

- Индентификатор/имя узла
- Распознаватель терминала/нетерминала
- Ссылка на смежные узлы для продолжения разбора при успешном распознавании.
- Ссылка на альтернативную ветку в случае неудачи при распознавании.

Узел в программе определяется структурой.

```
struct lxNode//вузол дерева, САГ або УСГ
{int x, y, f;//координати розміщення у вхідному файлі
 int ndOp;          //код типу лексеми
 int dataType;  // код типу даних, які повертаються
 unsigned resLength;  //довжина результату
 struct lxNode* prnNd;//зв'язок з батьківським вузлом
 struct lxNode* prvNd, pstNd;// зв'язок з підлеглими
 unsigned stkLength;//довжина стека обробки семантики
};
```

В процессе разбора формируется дерево разбора, которое, в отличие от дерева пидлеглости, может иметь больше двоичных ответвлений.

Чтобы превратить дерево разбора в дерево пидлеглости, можно использовать значения предшествований для отдельных терминалов и нетерминалов.

В случае унарных операций, связь с другим операндом не устанавливается.

Кроме того, каждый нетерминальный символ представлен узлом, состоящим из одной компоненты, которая указывает на первый символ в первой правой части, относящейся к этому символу.

Отсутствие альтернативных вариантов в графе помечает место обнаружения ошибки, компилятор занимается нейтрализацией ошибок, кот. включает в себя следующие действия:

1. Пропуск дальнейшего контекста до места, с которого можно продолжить программу (нейтрализация ошибок)
2. Накопление диагностики для ее последующего представления с текстом исх. программы

Некоторые компиляторы передают управление текстовому редактору подсказкой варианта обрабатывающего ошибки.

39. Формування графів синтаксичного розбору при використанні синтаксичного аналізу

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.

Методы синтаксического разбора – восходящие, нисходящие.

Восходящие начинаются с анализа правил, которые находят ближайшее к терминальному обозначение, и оканчиваются анализом конечного правила грамм.

Нисходящие начинают анализ с конечного обозначения грамматики.

Метод синт. графов – нисходящий метод разбора. Для представления грамматики используется списочная структура, называемая синтаксическим графом. Можно использовать спец. узлы с информацией об использовании синт. разбора.

Эти узлы имеют 4 элемента:

- Индентификатор/имя узла
- Распознаватель терминала/нетерминала
- Ссылка на смежные узлы для продолжения разбора при успешном распознавании.
- Ссылка на альтернативную ветку в случае неудачи при распознавании.

Узел в программе определяется структурой.

```
struct lxNode//вузол дерева, САГ або УСГ
{int x, y, f;//координати розміщення у вхідному файлі
 int ndOp;          //код типу лексеми
 int dataType;  // код типу даних, які повертаються
 unsigned resLength;  //довжина результату
 struct lxNode* prnNd;//зв'язок з батьківським вузлом
 struct lxNode* prvNd, pstNd;// зв'язок з підлеглими
 unsigned stkLength;//довжина стека обробки семантики
};
```

В процессе разбора формируется дерево разбора, которое, в отличие от дерева пидлеглости, может иметь больше двоичных ответвлений.

Чтобы превратить дерево разбора в дерево пидлеглости, можно использовать значения предшествований для отдельных терминалов и нетерминалов.

В случае унарных операций, связь с другим операндом не устанавливается.

Кроме того, каждый нетерминальный символ представлен узлом, состоящим из одной компоненты, которая указывает на первый символ в первой правой части, относящейся к этому символу.

Отсутствие альтернативных вариантов в графе помечает место обнаружения ошибки, компилятор занимается нейтрализацией ошибок, кот. включает в себя следующие действия:

40. Загальний підхід до організації семантичної обробки

Для того щоб надати можливість використання базових методів семантичної обробки та синтаксичного аналізу мов, використовують уніфіковані внутрішні подання.

Кроме деревьев подчиненности или направленнях ацикличных графов, использовались:

- Обратная польская запись позволяет записывать мат. выражения и присвоения в однозначной постфиксной форме без скобок. Операция над аргументами запис. после аргументов и объединяет 2 вида аргументов (терм. и нетерм). A+B –> AB+
- Форматы, похожие на представления маш.операций.

До задач семантичної обробки на різних етапах роботи компілятора відносять:

6. **семантич. анализ** – проверяет корректность соединения типов данных во всех операциях и операторах и определяет типы данных для промеж. результатов.

Нужно иметь таблицы соответствия операндов/аргументов и типа результата. Ключевая часть – код операции, тип аргумента. Функц. часть – тип результата.

Алгоритм анализу строится при проходе дерева, в результате будет сформировано несколько результатов. Алгоритм может быть построен через рекурсивные вызовы той же самой рек. функции или процедуры длявсех поддеревьев. При достижении терм. Обозначений, все рекурс. Вызовы останавливаются.

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

7. **интепретация** – сборка конст. выражений. В случае реализации языка программ. в виде интерпретатора, данные для обработки получаются из констант и результатов операций ввода.
8. **машинно-независимая оптимизация**. Основные действия должны сократить объёмы графов внутр. представления путем удаления повтор. И неисполз. фрагментов графа. Кроме этого, могут быть выполнены действия упрощ.экви превращений. Этап требует сохранения доп. информации.
9. **генерация объектных кодов выполняется**: в языке высокого уровня; на уровне команд асма. Для каждого узла дерева генерируется соотвеств. Последовательность команд по спец. шаблонам, а потом используется трансляция асм или языка, кот. включает асм-вставки.
10. **машинно-зависимая оптимизация** учитывает архи и специфику команд целевого устройства.

Повне табличне визначення семантичної обробки мови потребує повного покриття всіх операцій мови в таблицях семантичної відповідності операцій та операторів вхідної мови транслятора та операцій та підпрограм у вихідній мові системи трансляції. Для інтерпретації можна обмежитись таблицею відповідності розміщення даних в пам'яті інтерпретатора, структура якої наведена в табл 1, та таблицею виконавчих підпрограм для кожної з операцій, структура якої наведена в табл.2.

Структура для управління доступом до даних в інтерпретаторі

Ім'я даного	Адреса розміщення	Довжина	Тип	Блок визначення

Для спрощення та стандартизації семантичної обробки вузлів аж до рівня даних Multimedia доцільно узагальнити структуру **struct** lxNode для термінального вузла дерева так, щоб вона могла бути використана при інтерпретації з потрібними для неї

полями, замінивши вказівники на структури сполучення вказівників, які для попередника мають вигляд:

```
struct lxNode//вузол дерева, об'єкта або термінал
union prvTp    //адреса або вказівник на попередника
{char            *namNd;// зв'язок з текстом імені
 struct lxNode*grpNd;// зв'язок в графі
 void           *funNd();// зв'язок з виконавчим кодом
};
```

а для наступника, який може посилатись і на виконавчу процедуру:

```
union pstTp    //адреса або вказівник на наступника
{void            *datNd;// зв'язок з даними
 struct lxNode*grpNd;// зв'язок в графі
 void           *funNd();// зв'язок з виконавчим кодом
};
```

Тоді узагальнена структура термінального та нетермінального вузлів графа прийме вигляд

```
struct lxNode//вузол дерева, САГ, УСГ або термінал
{int ndOp;      //код типу лексеми
 union prvTp prvNd;// зв'язок з попередником
 union pstTp pstNd;// зв'язок з наступником
 int dataType; // код типу даних, які повертаються
 unsigned resLength;  //довжина результату
 unsigned auxNmb;//довжина стека обробки семантики
                 //або номер модуля визначення
 int x, y, f;//координати розміщення у вхідному файлі
 struct lxNode* prnNd;//зв'язок з батьківським вузлом
};
```

Структура для управління доступом до виконавчих кодів в інтерпретаторі

Операція	Ім'я підпрограми	Адреса розміщення	Тип результату

41. Організація семантичного аналізу (см. код №40)

Как правило, на этапе семантического анализа используются различные варианты деревьев синтаксического разбора, поскольку семантический анализатор интересует, прежде всего, структура входной программы.

Семантический анализ обычно выполняется на двух этапах компиляции: на этапе синтаксического разбора и в начале этапа подготовки к генерации кода. В первом случае всякий раз по завершении распознавания определенной синтаксической конструкции входного языка выполняется её семантическая проверка на основе имеющихся в таблице идентификаторов данных (такие конструкции – процедуры, функции и блоки операторов входного языка).

Нужно иметь таблицы соответствия операндов/аргументов и типа результата. Ключевая часть – код операции, тип аргумента. Функц. часть – тип результата.

Алгоритм анализа строится при проходе дерева, в результате будет сформировано несколько результатов. Алгоритм может быть построен через рекурсивные вызовы той же самой рек. ф-ции или процедуры для всех поддеревьев. При достижении терм. обозначений, все рекурс. вызовы останавливаются.

Во втором случае, после завершения всей фазы синтаксического разбора, выполняется полный семантическим анализ программы на основании данных в таблице идентификаторов (сюда попадает, например, поиск неописанных идентификаторов). Иногда семантический анализ выделяют в отдельный этап (фазу) компиляции.

Этапы семантического анализа

Семантический анализатор выполняет следующие основные действия:

- ❖ проверка соблюдения семантических соглашений входного языка;
- ❖ дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка;
- ❖ проверка элементарных семантических норм языков прогр., напрямую не связанных с входным языком.

Проверка соблюдения во входной программе семантических соглашений входного языка заключается в сопоставлении входных цепочек программы с требованиями семантики входного языка программирования. Каждый язык прогр. имеет четко заданные семантические соглашения, кот. не могут быть проверены на этапе синтаксического разбора. Именно их в первую очередь проверяет семантический анализатор.

Примерами соглашении являются следующие требования:

- ▪ каждая метка, на которую есть ссылка, должна один раз присутствовать в программе;
- ▪ каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;
- ▪ при вызове функции число фактических параметров и их типы должны соответствовать числу и типам формальных параметров;
- ▪ обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке; на типы левой и правой частей в операторе присваивания; на тип параметра цикла; на тип условия в операторах цикла и условном операторе и т.п.

```
   struct recrdSMA tTbl[179]=   /
таблиця припустимості типів для
операцій
   {{_if,_ui,32,_ui,32,_v,0},
   {_if,_ui,32,_si,32,_v,0},
   {_if,_ui,32,_f,32,_v,0},
   {_ass,_ui,32,_ui,32,_ui,32},
   {_ass,_ui,32,_si,32,_ui,32},
   {_sub,_d,32,_f,32,_d,64},
   {_sub,_d,32,_d,32,_d,64},
   {_mul,_ui,32,_ui,32,_ui,32},
   {_mul,_ui,32,_si,32,_si,32},}
   struct recrdSMA // структура
рядка таблиці припустимості типів
для операцій
   {enum tokType oprtn;
    int oprd1, ln1;
    int oprd2, ln2;
    int res, lnRes;
    _fop       *pintf;
    char       *assCd;
   };
```

42. Організація інтерпретації вхідної мови

При реалізації інтерпретації константних виразів доцільно створити для кожної операції з істотно різними парами даних окремі процедури, які будуть повертати результат строго визначеного типу, а перед використанням цих процедур треба вирівняти тип аргументів до більш загального (розробити процедуру для більш загальних форматів даних – з фіксованою і плаваючою точкою). Потім можна формувати результат. В кінці треба перетворити результат із загального типу в необхідний.

Для інтерпретації можна обмежитись таблицею відповідності розміщення даних в пам'яті інтерпретатора, структура якої наведена в табл 1, та таблицею виконавчих підпрограм для кожної з операцій, структура якої наведена в табл.2.

Структура для управління доступом до даних в інтерпретаторі

Ім'я даного	Адреса розміщення	Довжина	Тип	Блок визначення

Структура для управління доступом до виконавчих кодів в інтерпретаторі

Операція	Ім'я підпрограми	Адреса розміщення	Тип результату

Для роботи інтерпретатора необхідно додатково виділити пам'ять для зберігання даних, завантажити необх. константи. Будь-яка реалізація мови програмування має програми підготовки середовища інтерпретації і звертання до головної програми. При коректному завершенні роботи програми необх. відновити стек ОС до інтерпретації програми.

Формат структури елемента таблиці для семантичного аналізу, інтерпретації та генерації кодів через макроси виведення функцій форматного виведення:

```
typedef union gnDat _fop(union gnDat*,union gnDat*);
struct recrdSMA // структура рядка таблиці операцій
{enum tokType oprtn;// код операції
unsigned oprd1,ln1;//код типу та довжина першого аргументу
unsigned oprd2,ln2;//код типу та довжина другого аргументу
unsigned res,lnRes;//код типу та довжина результату
_fop    *pintf; // покажчик на функцію інтерпретації
char*assCd;     // покажчик на текст макроса
};
```

43. Організація генерації кодів

Генерация кода - последняя фаза трансляции. Результатом ее является либо ассемблерный модуль, либо объектный (или загрузочный) модуль. В процессе генерации кода могут выполняться некоторые локальные оптимизации, такие как распределение регистров, выбор длинных или коротких переходов, учет стоимости команд при выборе конкретной последовательности команд. Для генерации кода разработаны различные методы, такие как таблицы решений, сопоставление образцов, включающее динамическое программирование, различные синтаксические методы.

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

Генератор кодів формує команди з дрібних фрагментів команд, операцій, імен або адрес даних, індексних і базових регістрів. 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. наборі машинних команд

Більш ранні компілятори одразу формували машинні коди. Компілятори з мови Паскаль формували свої результати у так званих P-кодах. На етапі виконання цей код оброблювався середовищем Паскаль, яке включало підпрограми для обробки всіх операндів та операторів. Для даних виділяють як правило фіксовану пам'ять, а до кодів звертається виконуюча програма, яка дозволяє формувати результат виконання програми. 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.

FOR **var**:=**Value**$_1$	push dx mov dx, Value$_1$	Инициализируем индексную переменную - регистр dx, сохраняя прежнюю в стеке
TO|DOWNTO **Value**$_2$	loop_XXX: cmp dx,Value$_2$ ja|jb loop_exit_XXX	Здесь мы генерируем уникальную метку (на самом деле только суффикс XXX), которую мы ложим на стек вместе с ярлыком _FOR_ и пометкой, увеличивается или уменьшается в цикле индексная переменная
DO ...	... ;тело цикла	в dx - индексная переменная
... ;	inc|dec dx jmp loop_XXX loop_exit_XXX: pop dx	Вынимаем из стека управляющих операторов ярлык _FOR_, генерируем инкремент или декремент индексной переменной, генерируем переход к началу цикла, метку окончания цикла и восстанавливаем регистр, содержащий индексную переменную

44. Машинно-незалежна оптимізація (см. код №45)

Машинно-независимые оптимизации нельзя считать полностью оторванными от конкретной архитектуры. Многие из них разрабатывались с учётом общих представлений о свойствах некоторого класса машин (как правило - это машина с большим количеством регистров, фон-неймановской архитектуры, с относительно большой по размерам и медленной памятью). В современных компиляторах они, как правило, нацелены на достижение предельных скоростных характеристик программы, в то время как для встраиваемых систем к критическим параметрам также относится и размер получаемого кода.

Практически каждый компилятор производит определённый набор машинно-независимых оптимизаций.

1.*Исключение общих подвыражений*: если внутреннее представление генерируется последовательно для каждого подвыражения, оно обычно содержит большое количество избыточных вычислений. Вычисление избыточно в данной точке программы, если оно уже было выполнено ранее. Такие избыточности могут быть исключены путём сохранения вычисленного значения на регистре и последующего использования этого значения вместо повторного вычисления.

2.*Удаление мёртвого кода*: любое вычисление, производящее результат, который в дальнейшем более не будет использован, может быть удалено без последствий для семантики программы. 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.*Вынесение инвариантов циклов*: вычисления в цикле, результаты которых не зависят от других вычислений цикла, могут быть вынесены за пределы цикла как инварианты с целью увеличения скорости.

4.*Вычисление константных подвыражений*: вычисления, которые гарантированно дают константу могут быть произведены уже в процессе компиляции.

45. Машинно-залежна оптимізація

Машинно-залежна оптимізація полягає у ефективному використанні ресурсів цільового компа, тобто: РОН, st[i], ММХ, СОЗУ. Треба ефективно використ. сист команд, надаючи перевагу тим, для яких швидше організувати пошук

Для машинно-залежної оптимізації можна виділити такі види: вилучення ділянок програми, результати яких не використ. в кінц. результатах прогр., вилучення ділянок прогр., до яких не можна фактично звернутися через якісь умови, або помилки програми., оптимізація шляхом еквівал.перетворень складніших виразів на простіші.

Таким чином при машинно-залежній оптимізації таблиці реалізації операндів та операцій стають складнішими і мають декілька варіантів яких обирають найкращий за часом виконання. Потрібно ефективно виконувати операції пошуку, бо вони є критичними по часу. При обробці виразів проміжні дані краще зберігати у стеці регістра з плаваючою точкою.

Все описанные ниже алгоритмы, за исключением межпроцедурного анализа, работают на этапе машинно-зависимых оптимизаций. Здесь следует особо отметить, что их эффективность зачастую напрямую зависит от наличия той или иной дополнительной информации об исходной программе, а именно:

1. **«Программная конвейеризация»** и сворачивание в «аппаратные циклы» напрямую зависят от информации о циклах исходной программы и их свойствах (цикл for, цикл while, фиксированное или нет число шагов, ветвление внутри цикла, степень вложенности). Таким образом, необходимо, во-первых, исключить машиннонезависимые преобразования, разрушающие структуру цикла, и, во-вторых, обеспечить передачу информации об этих циклах через кодогенератор.
2. **Алгоритмы «SOA» и «GOA»** используют информацию о локальных переменных программы. Требуется информация, что данное обращение к памяти суть обращение к переменной и эта переменная локальная.
3. **Алгоритмы раскладки локальных переменных по банкам памяти** требуют информации об обращениях к локальным переменным, плюс важно знать используется ли где-либо адрес каждой из локальных переменных (т.е. возможно ли обращение по указателю)
4. **Алгоритм «Array Index Allocation»** и «частичное дублирование данных» должен иметь на входе информацию о массивах и обращениях к ним.

```
До оптимизации:
.L1:
.L2:

jmp  .LI
movl buffer,   %eax
movl %edx,   %eax
movl %edx,   %eax
movl %eax,   iecx
movl %ecx,   %edx
movzbl       [%eax] ,   %ecx
movl $0,   %eax
je   .L2
movzbl       [%ecx),   %edx
cmpb %dl,    (%esi)
jmp  .L4
jmp  .13

После оптимизации:
addl %edx,   %eax
movl %eax,   %edx
movzbl       [%eax) ,   %ecx
xorl %eax,   %eax
je   .L3
movzbl       [%ecx),   %edx
cmpb %dl,    (%esi)
jmp  .L4
```

46. Типові складові ОС

ОС – базовый комплекс компьютерных программ, обеспечивающий интерфейс с пользователем, управление аппаратными средствами компьютера, работу с файлами, ввод и вывод данных, а также выполнение прикладных программ и утилит.

В составе ОС различают три группы компонентов:

- ядро, содержащее планировщик; драйверы устройств, непосредственно управляющие оборудованием; сетевую подсистему, файловую систему;
- базовая системы ввода-вывода,
- системные библиотеки
- оболочка с утилитами.

Ядро — центральная часть (ОС), выполняющеяся при максимальном уровне привилегий, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память и внешнее аппаратное обеспечение. Также обычно ядро предоставляет сервисы файловой системы и сетевых протоколов, процедуры, выполняющие манипуляции с основными ресурсами системы и уровнями привилегий процессов, а также критичные процедуры.

Базовая система ввода-вывода (**BIOS**) — набор программных средств, обеспечивающих взаимодействие ОС и приложений с аппаратными средствами. Обычно BIOS представляет набор компонент — драйверов. Также в BIOS входит уровень аппаратных абстракций, минимальный набор аппартно-зависимых процедур ввода-вывода, необходимый для запуска и функционирования ОС. Драйвер – с операционными системами поставляются драйверы для ключевых компонентов аппаратного обеспечения, без которых система не сможет работать.

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

Файловая система — регламент, определяющий способ организации, хранения и именования данных на носителях информации. Она определяет формат физического хранения информации, которую принято группировать в виде файлов. Конкретная файловая система определяет размер имени файла (папки), максимальный возможный размер файла и раздела, набор атрибутов файла.

Библиотеки системных функций позволяющие многократное применение различными программными приложениями

Интерфейс пользователя – совокупность средств, при помощи которых пользователь общается с различными устройствами.

- Интерфейс командной строки: инструкции компьютеру даются путём ввода с клавиатуры текстовых строк (команд).
- Графический интерфейс пользователя: программные функции представляются графическими элементами экрана.

47. Типи ОС та їх режими

Операционные системы можно классифицировать на основании многих признаков. Наиболее распространенные способы их классификации далее.

Разновидности ОС:

по целевому устройству:

1. Для мейн-фреймов
2. Для ПК
3. Для мобильных устройств

по количеству одновременно выполняемых задач:

1. Однозначные
2. Многозадачные

по типу интерфейса:

1. С текстовым интерфейсом
2. С графическим интерфейсом

по количеству одновременно обрабатываемых разрядов данных:

1. 16-разрядные
2. 32-разрядные
3. 64-разрядные

До основних функцій ОС відносять:

1) управління процесором шляхом передачі управління програмам.
2) обробка переривань, синхронізація доступу до ресурсів.
3) Управління пам'яттю
4) Управління пристроями вводу-виводу
5) Управління ініціалізацією програм, між програмні зв'язки
6) Управління даними на довготривалих носіях шляхом підтримання файлової системи.

До функцій програм початкового завантаження відносять:

1) первинне тестування обладнання необхідного для ОС
2) запуск базових системних задач
3) завантаження потрібних драйверів зовнішніх пристроїв, включаючи обробники переривань.

В результаті завантаження ядра та драйверів ОС стає готовою до виконання задач визначених програмами у формі виконанні файлів та відповідних вхідних та вихідних файлів програми. При виконанні задач ОС забезпечує інтерфейс між прикладними програмами та системними програмами введення-виведення. Програмою найнижчого рівня в багатозадачних системах є так звані обробники переривань, робота яких ініціалізується сигналами апаратних переривань.

Режим супервизора — привилегированный режим работы процессора, как правило используемый для выполнения ядра операционной системы. В данном режиме работы процессора доступны привилегированные операции, как то операции ввода-вывода к периферийным устройствам, изменение параметров защиты памяти, настроек виртуальной памяти, системных параметров и прочих параметров конфигурации

Реальный режим (или режим реальных адресов) — это название было дано прежнему способу *адресации памяти* после появления процессора 80286, поддерживающего защищённый режим.

В реальном режиме при вычислении линейного адреса, по которому процессор собирается читать содержимое памяти или писать в неё, сегментная часть адреса умножается на 16 (или, что то же самое, сдвигается влево на 4 бита) и суммируется со

смещением. Таким образом, адреса 0400h:0001h и 0000h:4001h ссылаются на один и тот же физический адрес, так как 400h×16+1 = 0×16+4001h.

Такой способ вычисления физического адреса позволяет адресовать 1 Мб + 64 Кб − 16 байт памяти (диапазон адресов 0000h…10FFEFh). Однако в процессорах 8086/8088 всего 20 адресных линий, поэтому реально доступен только 1 мегабайт (диапазон адресов 0000h…FFFFFh).

В реальном режиме процессоры работали только в DOS. Адресовать в реальном режиме дополнительную память за пределами 1 Мб нельзя. Совместимость 16-битных программ, введя ещё один специальный режим — режим виртуальных адресов V86. При этом программы получают возможность использовать прежний способ вычисления линейного адреса, в то время как процессор находится в защищённом режиме.

Защищённый режим Разработан Digital Equipment (DEC), Intel. Данный режим позволил создать многозадачные операционные системы — Microsoft Windows, UNIX и другие.

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.

С появлением 32-разрядных процессоров 80386 фирмы Intel процессоры могут работать в трех режимах: реальном, защищённом и виртуального процессора 8086. В защищённом режиме используются полные возможности 32-разрядного процессора — обеспечивается непосредственный доступ к 4 Гбайт физического адресного пространства и многозадачный режим с параллельным выполнением нескольких программ (процессов).

48. Організація роботи планувальника задач і процесів. Супервізори

Планирование выполнения задач является одной из ключевых концепций в многозадачности и многопроцессорности как в операционных системах общего назначения, так и в операционных системах реального времени. Планирование заключается в назначении приоритетов процессам в очереди с приоритетами. Программный код, выполняющий эту задачу, называется **планировщиком.**

Самой важной целью планирования задач является наиболее полная загрузка процессора. Производительность — количество процессов, которые завершают выполнение за единицу времени. Время ожидания — время, которое процесс ожидает в очереди готовности. Время отклика — время, которое проходит от начала запроса до первого ответа на запрос.

Стратегия планирования определяет, какие процессы мы планируем на выполнение для того, чтобы достичь поставленной цели. Стратегии:

- по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;
- отдавать предпочтение более коротким процессам;
- предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

Известно большое количество правил (дисциплин диспетчеризации), в соответствии с которыми формируется список (очередь) готовых к выполнению задач, различают два больших класса дисциплин обслуживания — *бесприоритетные и приоритетные.*

- При *бесприоритетном* обслуживании выбор задачи производятся в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания.

При реализации *приоритетных* дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения. Приоритетные:

- с фиксированным приоритетом (с относительным пр, с абсолютным пр, адаптивное обслуживание, пр. зависит от t ожидания)
- с динамическим приоритетом (пр. зависит от t ожидания, пр. зависит от t обслуживания)

Супервизор ОС – центральный управляющий модуль ОС, который может состоять из нескольких модулей например супервизор ввода/вывода, супервизор прерываний, супервизор программ, диспетчер задач и т. п. Задача посредством специальных вызовов команд или директив сообщает о своем требовании супервизору ОС, при этом указывается вид ресурса и если надо его объем. Директива обращения к ОС передает ей управление, переводя процессор в привилегированный режим работы (если такой существует).

Не все ОС имеют 2 режима работы. Режимы работы бывают привилегированными (режим супервизора), пользовательскими, режим эмуляции.

49. Способи організації переключення задач

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.

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.

50. Підходи для реалізації систем В/В

Основная идея организации программного обеспечения ввода-вывода состоит в разбиении его на несколько уровней, причем нижние уровни обеспечивают экранирование особенностей аппаратуры от верхних, а те, в свою очередь, обеспечивают удобный интерфейс для пользователей.

Вид программы не должен зависеть от того, читает ли она данные с гибкого диска или с жесткого диска. Другим важным вопросом для программного обеспечения ввода-вывода является обработка ошибок. Еще один ключевой вопрос - это использование блокирующих (синхронных) и неблокирующих (асинхронных) передач. Большинство операций физического ввода-вывода выполняется асинхронно - процессор начинает передачу и переходит на другую работу, пока не наступает прерывание. Последняя проблема состоит в том, что одни устройства являются разделяемыми, а другие - выделенными. (диски vs принтеры)

Для решения поставленных проблем целесообразно разделить программное обеспечение ввода-вывода на четыре слоя

- Обработка прерываний,
- Драйверы устройств,
- Независимый от устройств слой операционной системы,
- Пользовательский слой программного обеспечения.

Побудова драйверів введення-виведення в реальному режимі

На верхньому рівні зв'язків прикладних програм з ОС використовують інтерфейсні програми, які звертаються до системних програм з запитами про введення-виведення. На нижньому рівні взаємодії з пристроями вводять драйвери, які відповідають за введення-виведення одного фізичного запису. Фізичні записи в свою чергу визначаються технологією обміну – на диск / на дисплей тощо. Тому постає задача перетворення фізичних записів на логічні і навпаки, що дасть можливість виконувати паралельний обмін між зовнішніми пристроями. Побудова драйверів спирається на відносно прості механізми, бо в ОС реального режиму непередбачено суворого розділення пам'яті між задачами. Драйвер повинен включати в себе такі блоки:

- Видача команди на підготовку до роботи зовнішнього пристрою
- Видача сигналів на порти управління командою OUT через будь-яку програмі
- Очікування готовності роботи зовнішнього пристрою
- Реалізується читанням слова стану пристрою і перевіркою в ньому біта готовності.
- Виконання операції обміну
- Видача команд призупинення роботи пристрою
- Формування фізичного запису введеної інформації для передачі задачі-споживачу
- Вихід

Приклад реалізації драйверу однобайтного каналу обміну даними:

```
Kanal PROC
        MOV AL, DeviceOn; загружаємо код ввімкнення пристрою
        OUT ComPort, AL  ; пересилання у порт управління коду ввімкнення

    L:  IN AL, StatPort      ; завантаження регістру стану
        TEST AL, ErMask  ; перевірка аварійного стану
        JNZ GoError              ; перехід на обробник помилки

        TEST AL,ReadyIN  ; перевірка готовності даних для введення
```

```
        JZ L                        ; нова ітерація опросу готовності пристрою

        IN AL, DestPort     ; введення даних
        PUSH AX
        MOV AL, DeviceOff ; загружаємо код вимкнення пристрою
        OUT ComPort, AL  ; пересилання у порт управління коду вимкнення
        POP AX
        RET
    Kanal ENDP
```

Побудова драйверів введення-виведення в захищеному режимі

Вимоги до драйверів такі ж як і в реальному режимі (питання 42), однак є декілька уточнень і поправок. А саме :

- Команди підготовки пристрою видаються тільки програмами, запущеними на 0 кільці захисту

51. Стан виконання задачі

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.

52. Стан задачі в захищеному режимі

– сохраняются все регистры задачи
– каталог таблиц страниц процесса

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.

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.

-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 переключения процессора из реального режима в защищённый можно использовать, например, такую последовательность команд:

```
   mov     ax,     cr0
   or      ax,     1
   mov     cr0,    ax

   Обеспечение возможности возврата
в реальный режим:
   push ds ; готовим адрес возврата
   mov   ax,40h   ;   из   защищённого
режима
   mov ds,ax
```

```
      mov       [WORD       67h],OFFSET
   shutdown_return
      mov [WORD 69h],cs
      pop ds
      Запрет прерываний:
      cli
      in al, INT_MASK_PORT
      and al, 0ffh
      out INT_MASK_PORT, al
      mov al,8f
      out CMOS_PORT,al
```

53. Стан задачі в реальному режимі

Команда INT в реальном у режимі виконується як звертання до підпрограми обробника переривань, адреса якой записана в 1 КБ пам'яті як чотири адреси входи в програму переривань, ця адреса складається з сегментів адреси та зміну в середині сегмента. В стеці переривань задачі запам'ятовують адресу повернення, а перед цим в стеку запам'ятовується вміст регістру прапорців.

Для виходу використовується команда RET, яка відновлює адресу команди переривання задачі та стан регістру прапорців.

Для уникнення постійних зчитувань регістру стану для перевірки готовності пристрою на головних платах встановлюються Блоки Програмних Переривань, на входи яких подаються сигнали готовності пристроїв. БПП програмуються при завантаженні ОС і BIOS через порти 20/21 H, 0A0/0A1 H : встановлюються пріорітетні пристрої – вони маркуються «1» у регістрі масок. Коли БПП готовий і працює, він передає сигнали до процесора, який обробляє їх тільки якщо був активний прапорець IF (= 1). Це досягається командою STI. Але перехід на обробку в реальному режимі виконується через таблицю адрес обробників, яка знаходиться в першому кілобайті пам'яті і команда записується в ній у вигляді 4 байт адреси входу в програму-переривання (сегмент адреси + зміщення). Схема обробки переривання наступна:

- Закінчується команда, що виконувалась в процесорі
- Процесор підтверджує переривання
- Блок програмного переривання формує сигнал блоку пріоритетних переривань, тобто видає номер вектора переривань
- INT

Для того, чтобы вернуть процессор 80286 из защищённого режима в реальный, необходимо выполнить аппаратный сброс (отключение) процессора.

```
PROC _real_mode NEAR

; Сброс процессора

cli
mov [real_sp], sp
mov al, SHUT_DOWN
out STATUS_PORT, al
rmode_wait:
hlt
jmp rmode_wait
LABEL shutdown_return FAR

; Вернулись в реальный режим

mov ax, DGROUP
mov ds, ax
assume ds:DGROUP
mov ss,[real_ss]
mov sp,[real_sp]

; Размаскируем все прерывания

in al, INT_MASK_PORT
and al, 0
out INT_MASK_PORT, al
call disable_a20
mov ax, DGROUP
mov ds, ax
mov ss, ax
mov es, ax
mov ax,000dh
out CMOS_PORT,al
sti
ret
ENDP _real_mode
```

54. Механізми переключення задач

Реал. режим: При використанні синхронізації примітивів програми обробки переривань звертаються до функції зміни стану примітиву, щоб далі звернутися до супервізора і змінити стан виконуваної задачі. В 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.

Защ.режим: Переключение задач осуществляется или программно (командами CALL или JMP), или по прерываниям (например, от таймера). 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. При переходе к выполнению процедуры, процессор запоминает (в стеке) лишь точку возврата (CS:IP).

Поддержка многозадачности обеспечивается:

· Сегмент состояния задачи (TSS).

· Дескриптор сегмента состояния задачи.

· Регистр задачи (TR).

· Дескриптор шлюза задачи.

Переключение по прерываниям.

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.

Как и при обращении к любому другому дескриптору, при обращении к шлюзу проверяется условие *CPL* < *DPL.*

Программное переключение

Переключение задач выполняется по инструкции межсегментного перехода *(JMP)* или вызова *(CALL).* Для того чтобы произошло переключение задачи, команда *JMP* или *CALL* может передать управление либо дескриптору TSS, либо шлюзу задачи.

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 JMP, должна заканчиваться командой обратного перехода. В случае CALL - возврат должен происходить по команде IRET.

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 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.

55. Організація захисту пам'яті в ОС

Защита памяти делится на защиту при управлении памятью и защиту по привилегиям.

1. Средства **защиты при управлении памятью** осуществляют проверку превышения эффективным адресом длины сегмента, прав доступа к сегменту на запись или только на чтение, функционального назначения сегмента.

Основна проблема роботи з пам'яттю як правило була обмеженість обсягів ОП. Для того щоб одержати можливість виконувати задачі здатні обробляти великі обсяги даних стали використовувати так звану віртуальну пам'ять, адресний простір якої відповідав потребам задачі, а фізична реалізація використовувала наявний обсяг вільної ОП та зберігав дані для яких не вистачало ОП на дискових носіях.

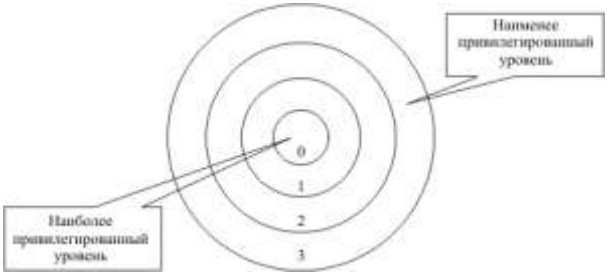
Для ефективної організації ОП використовували сегментний та сторінковий підходи. При сегментному підході виділявся спец. сегмент обміну даних, який розміщувався на диску і встановлювалась відповідність блоків сегментів обміну сегментам віртуальної пам'яті. Сегментна організація була характерна для Windows 3.x.

В подальших версіях Windows і більшості інших ОС використовується так звана сторінкова організація віртуальної пам'яті, для якої будується таблиця сторінок пам'яті для кожної із задач. Сторінки мають 4Кб і адресуються 12 бітами, номери сторінок використовують 20 додаткових бітів. В таблиці сторінок для кожної сторінки фіксується фізична адреса.

2**. Защита по привилегиям** фиксирует более тонкие ошибки, связанные, в основном, с разграничением прав доступа к той или иной информации.

Различным объектам, которые должны быть распознаны процессором, присваивается идентификатор, называемый уровнем привилегий. Процессор постоянно контролирует, имеет ли текущая программа достаточные привилегии, чтобы

- выполнять некоторые команды,
- выполнять команды ввода-вывода на том или ином внешнем устройстве,
- обращаться к данным других программ,
- вызывать другие программы.



На аппаратном уровне в процессоре различаются 4 уровня привилегий. Наиболее привилегированными являются программы на уровне 0.

уровень 0 - ядро ОС, обеспечивающее инициализацию работы, управление доступом к памяти, защиту и ряд других жизненно важных функций, нарушение которых полностью выводит из строя процессор;

уровень 1 - основная часть программ ОС (утилиты);

уровень 2 - служебные программы ОС (драйверы, СУБД, специализированные подсистемы программирования и др.);

уровень 3 - прикладные программы пользователя.

ОС UNIX работает с двумя кольцами защиты: супервизор (уровень 0) и пользователь (уровни 1,2,3).

OS/2 поддерживает три уровня: код ОС работает в кольце 0, специальные процедуры для обращения к устройствам ввода-вывода действуют в кольце 1, а прикладные программы выполняются в кольце 3.

56. Особливості визначення пріоритетів задач.

Приоритеты дают возможность индивидуально выделять каждую задачу по важности.

Сервис **планировщик задач** — программа, которая запускает другие программы в зависимости от различных критериев, как например:

- наступление определенного времени
- операционная система переходит в определенное состояние (бездействие, спящий режим и тд)
- поступил административный запрос через пользовательский интерфейс или через инструменты удаленного администрирования.

Стратегия планирования определяет, какие процессы мы планируем на выполнение для того, чтобы достичь поставленной цели. Стратегии:

- ❖ по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;
- ❖ отдавать предпочтение более коротким процессам;
- ❖ предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

Известно большое количество правил (дисциплин диспетчеризации), в соответствии с которыми формируется список (очередь) готовых к выполнению задач, различают два больших класса дисциплин обслуживания — *бесприоритетные и приоритетные.* При *бесприоритетном* обслуживании выбор задачи производятся в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания. При реализации *приоритетных* дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения. Приоритетные:

- с фиксированным приоритетом (с относительным пр, с абсолютным пр, адаптивное обслуживание, пр. зависит от t ожидания)
- с динамическим приоритетом (пр. зависит от t ожидания, пр. зависит от t обслуживания)

Одна из проблем, которая возникает при выборе подходящей дисциплины обслуживания, — это **гарантия обслуживания.** Дело в том, что при некоторых дисциплинах, например при использовании дисциплины абсолютных приоритетов, низкоприоритетные процессы оказываются обделенными многими ресурсами и, прежде всего, процессорным временем, могут быть не выполнены.

Как реализован механизм динамических приоритетов в **ОС UNIX.**

Каждый процесс имеет два атрибута приоритета, с учетом которого и распределяется между исполняющимися задачами процессорное время: *текущий приоритет,* на основании которого происходит планирование, и заказанный *относительный приоритет.*

- Текущий приоритет процесса – в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет).
- Для режима задачи приоритет меняется в диапазоне 0-65, для режима ядра — 66-95 (системный диапазон).
- Процессы, приоритеты которых лежат в диапазоне 96-127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой.
- Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение *приоритета сна,* выбираемое ядром из диапазона системных приоритетов и связанное с событием, вызвавшее это состояние.

57. Ієрархія організації програм В/В

Были разработаны различные системные инструментальные программы. К ним относятся библиотеки функций, редакторы связей, загрузчики, отладчики и драйверы ввода-вывода, существующие в виде программного обеспечения, общедоступного для всех пользователей.

Для решения проблем многозадачности потребовалось разработать аппаратное обеспечение, поддерживающее прерывания ввода-вывода, и прямой доступ к памяти. Используя эти возможности, процессор генерирует команду ввода-вывода для одного задания и переходит к другому на то время, пока контроллер устройства выполняет ввод-вывод. После завершения операции ввода-вывода процессор получает прерывание, и управление передается программе обработки прерываний из состава операционной системы. Затем операционная система передает управление другому заданию.

Рассмотрим структуру програм ввода-вывода одного физического элемента, обрабатываемого внешним устройством. Общая схема процедуры обмена включает такую последовательность действий:

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

Эту последовательность действий для драйвера ввода устройства, можно записать для однобайтного канала обмена таким образом:

```
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
IN AL, dtPrt ; ввод данных
PUSH AX ;
MOV AL,cmOff ; загрузка управляющего кода включения устройства
OUT cmPrt,AL ; пересылка кода включения в порт управления
POP AX ;
RET
DrIn ENDP
```

58. Способи організації драйверів (код см. №59)

Операционная система управляет некоторым «виртуальным устройством», которое понимает стандартный набор команд. Драйвер переводит эти команды в команды, которые понимает непосредственно устройство. Эта идеология называется «абстрагирование от аппаратного обеспечения».

Драйвер состоит из нескольких функций, которые обрабатывают определенные события операционной системы. Обычно это 7 основных событий:

- загрузка драйвера – др. регистрируется в системе, производит первичную инициализацию и т. п.;
- выгрузка – освобождает захваченные ресурсы — память, файлы, устройства и т. п.;
- открытие драйвера – начало основной работы. Обычно драйвер открывается программой как файл, функциями CreateFile() в Win32 или fopen() в UNIX-подобных системах;
- чтение;
- запись – программа читает или записывает данные из/в устройство, обслуживаемое драйвером;
- закрытие – операция, обратная открытию, освобождает занятые при открытии ресурсы и уничтожает дескриптор файла;
- управление вводом-выводом – драйвер поддерживает интерфейс ввода-вывода, специфичный для данного устройства.

Найпростіший драйвер включає в свою структуру наступні блоки:

1. Видача команди на підготовку до роботи зовнішнього пристрою.
2. Очікування готовності зовнішнього пристрою для виконання операцій.
3. Виконання власне операцій обміну.
4. Видача команд призупинки роботи пристрою.
5. Вихід з драйверу.

В простих системах підготовка пристрою до роботи виконується звичайно видачею відповідних сигналів на порти управління командами OUT. В реальному режимі ці команди можуть бути використані в будь-якій задачі, а в захищеному лише на так званому нульовому рівні захисту. Очікування готовності зовнішнішнього пристрою в найпростіших драйверах організовано шляхом зчитування біту стану зовнішнього пристрою з наступним переведенням відповідного біту готовності. Для того, щоб уникнути такого бігання по циклу, в подальших серіях драйверів стали використовувати систему апаратних переривань.

Драйвери Windows.

Звичайно драйвери розділяють на статичну та динамічну складові. Статична складова або ініціалізація драйвера готує драйвер до роботи, виконуючи відкриття файлів ті настроюючи динамічну частину драйвера, або переривання. Динамічна частина драйвера являє собою фактично обробник переривань, обробник переривань захищеного режиму може бути побудований як прилевійований обробник переривань в нульовому кінці запису, а може бути побудований в режимі задачі. Обробник переривань драйверів будуються як служби або сервіси ОС, які розглядаються як спеціальний обьєкт, що мають бути зареєстровані.

59. Роль переривань в побудові драйверів

Заголовок драйвера
Область данных драйвера
Программа СТРАТЕГИЙ
Вход в программу ПРЕРЫВАНИЙ
Обработчик команд
Программа обработки прерываний
Процедура инициализации

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. Заголовок запроса содержит всю необходимую информацию для корректной обработки каждой команды и сообщает вызывающей 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, программа стратегии и программа.**

Процедура обслуживания прерывания (interrupt service routine — ISR) обычно выполняется в ответ на получение прерывания от аппаратного устройства и может вытеснять любой код с более низким приоритетом. Процедура обслуживания прерывания должна использовать минимальное количество операций, чтобы центральный процессор имел свободные ресурсы для обслуживания других прерываний.

Программа прерыв. это не тоже самое, что программа обработки прерываний, которая может присутствовать в качестве необязательной части работающего по прерываниям драйвера. На самом деле, программа прерыв. - это точка входа в драйвер для обработки получаемых команд.

```
   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 ; Тчк входа в
прог.STRATEGY
   dw offset INTERRUPT ;Тчк входа в
прог.INTERRUPT
   db   8   dup   (?)   ;   Кол-во
устройств/поле имени
   ;*******    РЕЗИДЕНТНАЯ    ЧАСТЬ
ДРАЙВЕРА
   req_ptr  dd  ?  ;  Указатель  на
заголовок запроса
   ;******* ПРОГРАММА СТРАТЕГИИ
   ; Сохр. адрес заг. запр. для
прог.СТРАТЕГИЙ
   ; в REQ_PTR.
   ;   На   входе   адрес   заг.запр.
находится в рег. 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 заг. запр. Адрес заг.
запр. содержится в REQ_PTR в форме
   ; СМЕЩЕНИЕ:СЕГМЕНТ.
   INTERRUPT PROC FAR
   pusha ; Сохранить все регистры
   lds  bx,cs:[req_ptr]  ;  Получить
адрес заг. запр.
   …
   INTERRUPT ENDP
   …
   DRIVER ENDS
   END
```

60. Необхідність синхронізації даних

Для синхронізації обміну даними між процесами-споживачами та процесами-постачальниками важливо забезпечити гарантовану передачу повністю підготовлених даних, щоб уникнути помилок при зміні ще необроблених даних. Такі задачі покладаються на спеціальні системні об'єкти – синхронізуючі примітиви. Назва примітив відображає той факт, що операції з такими об'єктами розглядаються як неподільні (не можна переривати поки не закінчаться). До таких об'єктів в Windows відносять взаємні виключення (Mutex), критичні секції, семафори, події (event).

Взаємні виключення це такі об'єкти які можуть бути в двох станах (вільному та зайнятому). При запиті до взаємного виключення блокується можливість повторного зайняття цього об'єкту іншим процесом, аж до його звільнення спеціальною операцією. Це дозволяє виконувати доступ до об'єкту, який використовується в декількох процесах лише однією задачею, наприклад буфер введення-виведення може бути зайнятим або обробником переривань або інтерфейсною задачею аж доки його не звільнить попередня задача. Звичайно mutex системний об'єкт який формується в ОС і може бути доступним за іменем з будь-якої задачі, тобто він дозволяє синхронізувати процеси взаємодії з будь-яких автономних процесів або задач.

Критичні секції виконують ті ж самі функції, але є всього лише внутрішніми об'єктами одного процесу або задач. Тому з цього боку вони обробляються швидше, але мають обмежене використання.

Семафори можна розглядати як узагальнення взаємних виключень на випадок використання груп схожих об'єктів (груп буферів або буферних пулів). Для цього в семафорі створюється лічильник зайнятих ресурсів, який підраховує наявність вільних ресурсів і блокує доступ лише при їх відсутності. З такої позиції mutex можна розглядати як семафор з одним можливим значенням лічильника. Однак при використанні цих трьох примітивів потрібно враховувати що вони не пов'язані з тими об'єктами для яких вони використовуються, тобто відповідальність при роботі з примітивами покладається на програміста.

Події вважаються більш складними примітивами і вони змінюють відповідний елемент пам'яті з 0 на 1 за спеціальними функціями. Це дозволяє перевірити закінчення процесів і організувати операцію очікування.

```
   Крит секции на С
   typedef struct
_RTL_CRITICAL_SECTION {
   PRTL_CRITICAL_SECTION_DEBUG
DebugInfo; // Используется
операционной системой
   LONG LockCount;// Счетчик
использования этой критической
секции
   LONG RecursionCount;// Счетчик
повторного захвата из нити-
владельца
   HANDLE OwningThread;//
Уникальный ID нити-владельца
   HANDLE LockSemaphore;// Объект
ядра используемый для ожидания
   ULONG_PTR SpinCount;//
Количество холостых циклов перед
вызовом ядра
   } RTL_CRITICAL_SECTION,
*PRTL_CRITICAL_SECTION;
```

```
   // Нить №1
   void Proc1()
   {
   ::EnterCriticalSection(&m_lockOb
ject);
   if (m_pObject)
   m_pObject->SomeMethod();
   ::LeaveCriticalSection(&m_lockOb
ject);
   }

   // Нить №2
   void Proc2(IObject *pNewObject)
   {
   ::EnterCriticalSection(&m_lockOb
ject);
   if (m_pObject)
   delete m_pObject;
   m_pObject = pNewobject;
   ::LeaveCriticalSection(&m_lockOb
ject);
   }
```

61. Способи організації трансляторів з мов програмування.

Транслятор — 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.

Трансляторы подразделяют:

- *Многопроходной*. Формирует объектный модуль за несколько просмотров исходной программы.
- *Однопроходной*. Формирует объектный модуль за один последовательный просмотр исходной программы.
- *Обратный*. То же, что детранслятор,
- *Оптимизирующий*. Выполняет оптимизацию кода в создаваемом объектном модуле.
- *Синтаксически-ориентированный (синтаксически-управляемый)*. Получает на вход описание синтаксиса и семантики языка и текст на описанном языке, который и транслируется в соответствии с заданным описанием.

Компилятор – транслятор, который преобразует программы в машинный язык, принимаемый и исполняемый непосредственно процессором.

Процесс компиляции как правило состоит из нескольких этапов (ЛА, СА, Сем.О., оптимиз., ген.кодов).

+: программа компилируется один раз и при каждом выполнении не требуется доп. преобразований.

-: отдельный этап компиляции замедляет написание и отладку

Интерпретатор программно моделирует машину, цикл выборки-исполнения которой работает с командами на языках высокого уровня, а не с машинными командами

- **Чистая интерпретация – создание вирт.машины, реализующей язык**

+: отсутствие промежут. действий для трансл. упрощает реализацию интерпр. и делает его удобнее

- ин-тор должен быть на машине, где должна исполняться программа.

- **Смешанная реализация –** интерпретатор перед исполнением программы транслирует её на промежуточный язык (например, в байт-код или р-код), более удобный для интерпретации (то есть речь идёт об интерпретаторе со встроенным транслятором).

62. Особливості роботи з БПП

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: 08h-0fh – прерывания IRQ0-IRQ7; 70h-77h – прерывания IRQ8-IRQ15

Организация обработки аппаратных прерываний обеспечивается процедурами – обработчиками прерываний и выполняющими самостоятельные вычислительные процессы, инициированные сигналами с внешних устройств. 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
```

63. Программно-аппаратные взаимодействия при обработке прерываний в машинах IBM PC.

В реальном режиме имеются программные и аппаратные прерывания. Прогр.Прер. инициируются командой INT, Апп.Прер. - внешними событиями, по отношению к выполняемой программе. 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. Состоит из 256 элементов по 4 байта, т.е. 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 ПП или АП, 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. По своим функциям исключения соответствуют зарезервированным для процессора внутренним прерываниям реального режима. Когда процедура обработки исключения получает управление, флаг IF не изменяется.

Содержание