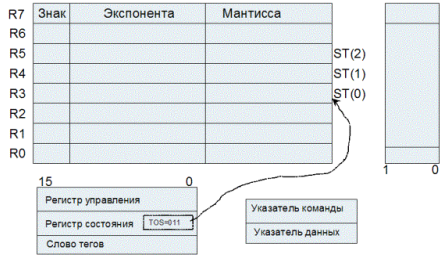
1. Архітектура пристрою з плаваючою точкою

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

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
```

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.

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.

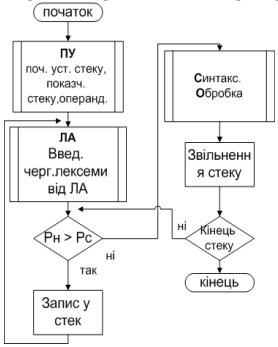
Супервізори – програми, що управляють вибором чергової активної задачі. Переключення активних задач виконуються через використання апаратних переривань або за готовністю зовнішніх пристроїв, або за таймером.

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;}
```

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.

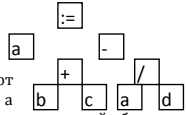
```
// обработка таблиц по прямому адресу
// выборка по прямому адресу
struct recrd* selNmb(struct recrd*, int nElm);

// 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);
```

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} };
```

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
```

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;
}
```

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
```

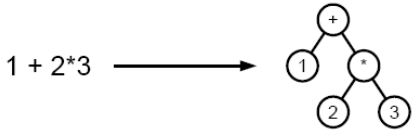
Функция лексического анализа очередной лексемы

```
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++; }
```

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.

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.

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;}
```

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
```

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.

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 неиспольз. 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

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;
   };
```

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 Р-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

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 оперантів 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:                                        cmpb %dl,    (%esi)
.L2:                                        jmp  .L4
                                            jmp  .13

jmp  .LI                                    После оптимизации:
movl buffer,   %eax                         addl %edx,   %eax
movl %edx,   %eax                           movl %eax,   %edx
movl %eax,   iecx                           movzbl       [%eax) ,   %ecx
movl %ecx,   %edx                           xorl %eax,   %eax
movzbl       [%eax] ,   %ecx                je   .L3
movl $0,   %eax                             movzbl       [%ecx),   %edx
je   .L2                                    cmpb %dl,    (%esi)
movzbl       [%ecx),   %edx                 jmp  .L4
```

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

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.

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
```

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
```

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.

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.

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.

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);
   }
```

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
```

Содержание