1. Системні програми

1.1. Класифікація системних програм

Системное программное обеспечение (СПО) – совокупность программ для обеспечения работы компьютера и их сетей. 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. СПО предназначено для:

- создания среды функционирования других программ;
- автоматизации разработки (создания) новых программ;
- обеспечения надежной и эффективной работы компьютера и их сетей;
- проведения диагностики и профилактики аппаратуры.

2 вида классификаций:

- управляющие (прозрачные) организуют корректное функционирование всех устройств системы;
- обрабатывающие выполняются как специальные прикладные приложения.
- базовое ПО – это минимальный набор программных средств, обеспечивающих работу компьютера. К нему относятся операционные системы и драйверы; интерфейсные оболочки для взаимодействия пользователя с ОС и программные среды; системы управления файлами;
- сервисное ПО расширяет возможности базового и организуют более удобную работу: утилиты (архивирование, диагностика, обслуживание дисков); антивирусы; среда программирования (редактор; транслятор; компоновщик (редактор связей); отладчик; библиотеки подпрограмм).

1.2. Системні обробляючі програми

Системные обрабатывающие программы (СОП) предназначены для автоматизации подготовки программ для компьютера и выполняются под управлением операционной системы. Это значит, что они могут пользоваться услугами управляющих программ и не могут самостоятельно выполнять системные функции. Например, СОП не может осуществлять собственный ввод-вывод. Эти операции реализуются с помощью запросов к управляющим программам. К СОП относят:

- трансляторы (компиляторы и интерпретаторы);
- компоновщики – объединяют оттранслированные программные модули с программными модулями библиотек и порождают выполняемые коды;
- текстовые редакторы и процессоры;
- оболочки, автоматизирующие работу по созданию программных проектов;
- отладчики.

1.3. Системні управляючі програми

Системные управляющие программы (СУП) реализуют набор функций управления, который включает в себя управление ресурсами ЭВМ, восстановление работы системы после проявления неисправностей, загрузку программ на выполнение, ввод-вывод данных на внешние устройства, управление доступом и защитой информации в системе. Основные функции СУП – управление вычислительными процессами и вычислительными комплексами; работа с внутренними данными ОС. К СУП относят:

- программы начальной загрузки;
- программы управления файловой системой;
- супервизоры управляют переключением задач. Переключение происходит по таймеру или с использованием аппаратных прерываний или по готовности внешних устройств;
- программы контроля оборудования.

Как правило, СУП находятся в основной памяти. Это резидентные программы, составляющие ядро ОС. СУП, которые загружаются в память непосредственно перед выполнением, называют транзитными. Сейчас они поставляются в виде инсталляционных пакетов ОС и драйверов специальных устройств.

1.4.Узагальнена структура системної програми

Обычно для СПО входные данные подаются в виде директив. Для ОС это командная строка, а для системы обработки – это программа на входном языке программирования для компилятора или интерпретатора, и программа в машинных кодах для компоновщика или загрузчика. Выходные данные формируются либо в главной памяти, либо в виде файлов на дисковых накопителях. СПО выполняется в несколько этапов:

- лексический анализ (ЛА) – разбиение входных данных на лексемы;

- синтаксический анализ (СА) – проверка соответствия данных синтаксическим правилам входного языка и построение деревьев разбора – графов выполняемых операций;
- семантическая обработка (СО) – проверка соответствия типов данных в операциях и формирование результата. Второй тип СО – это интерпретация или исполнение;
- машинно-независимая оптимизация (только в компиляторах) – удаление неиспользуемых фрагментов кода;
- генерация кодов.

1.5.Типові елементи і об'єкти системних програм

Більшість СОП базується на використанні таблиць та правил обробки. Таблиці – складні структури даних, завдяки яким можна підвищити ефективність програми. Їх основним призначенням є пошук інформації по заданому аргументу. Тому вони мають схожу з базами даних структуру та мають задану кількість полів з даними, що є ключовими і функціональними. До ключового поля висувається умова однозначності. Використовуються таблиці імен і констант, що призначені для СА і СО. Для генерації кодів використовують таблиці відповідності внутрішнього подання. Вони звичайно організовуються як масиви чи структури з покажчиками. Ці таблиці складаються з елементів з такими полями: ключ (змінна/мітка) та характеристика: сегмент (даних/коду), зміщення, тип (байт, слово чи подвійне слово/близька чи дальня). Різниця таблиць СОП від таблиць реляційних баз даних полягає в тому, що для зберігання таблиць баз даних використовується носії інформації, а для СОП – пам'ять. В процесі виконання частина бази даних переноситься до оперативної пам'яті, а частина СОП, якщо їй не вистачає пам'яті, зберігається на накопичувачі. Таблиці в СОП використовують також для того, щоб повертати дані, значення полів як аргументи пошуку. Приклад:

```
struct recrd // структура строки
{struct keyStr key; // ключ
 struct fStr func;}; // функциональная часть
```

2. Структури даних

2.1. Основні способи організації таблиць та індексів

Индексы создаются в таблицах при необходимости упорядочивания или связзки. Чаще всего они древообразные (которые упорядочены в лексикографическом порядке) или в виде hash-функций. Таблицы – сложные структуры данных, с помощью которых значительно увеличивается эффективность программы. Их основное назначение состоит в поиске информации об объекте. Результатом 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:

```
имя_структуры struc
последовательность директив DB (1 байт), DW (2), DD (4), DQ (8), DT (10)
имя_структуры ends
```

Шаблон структуры представляет собой только спецификацию элемента таблицы. Для 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 ним:

```
student iv_groups <'timofey', 19, 5>
mov ax, student.age ; ax => 19
```

С помощью ключевого слова dup резервируют n 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 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 так: 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 ?
```

2.2. Організація таблиць у вигляді масивів записів

Таблицы – сложные структуры данных, с помощью которых значительно увеличивается эффективность программы. Их основное назначение состоит в поиске информации об объекте. Результатом обычно является элемент таблицы, ключ которого совпадает с аргументом поиска в таблице. Простейший способ построения информационной базы состоит в определении структуры отдельных элементов, которые встраиваются в структуру таблицы. 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 kN байтов памяти. Если нужно все элементы разместить в k последовательных байтах и построить таблицу с N элементов в виде массива, то пример такой таблицы приведен ниже, где элементы задаются структурой struct в языке С, длиной k байтов, определяемой суммой размеров ключевой и функциональной части элемента таблицы.

```
struct keyStr // ключевая часть записи
{char* str; // ключевые поля
 int  nMod;};
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
```

2.3. Організація таблиць у вигляді структур з покажчиками

Таблицы – сложные структуры данных, с помощью которых значительно увеличивается эффективность программы. Реализация, в которой для хранения N элементов по k байт использует kN байт памяти, часто бывает избыточной. В больших таблицах данные часто повторяются, поэтому повторяемые данные выносятся в отдельные таблицы и связываются – нормализируются. Если повторяемые элементы были вынесены, то вместо их можно ввести дополнительное поле (должно быть меньше замещенных данных). Оно будет указывать на вынесенный элемент вспомогательной таблицы и его удобно представлять в виде указателя или некоторого уникального идентификатора. Частным случаем такой структуры может быть древовидная структура. Это не только значительно упрощает ее реализацию, но и позволяет сэкономить память. Также это избавляет от множественных операций при сортировке таблиц, добавлении и удалении элементов. Приведем пример:

```
struct lxNode // узел дерева
{int ndOp; // код операции/лексемы
 unsigned stkLength; // номер модуля
 struct lxNode* prvNd, *pstNd; // связи с подчиненными
 int dataType; // тип возвращаемых данных
 unsigned resLength;
 int x, y, f; // координаты в файле
 struct lxNode*prnNd;}; // связь с родительским узлом
```

2.4.Організація роботи з таблицями в системних програмах

Таблиці – складні структури даних, завдяки яким можна підвищити ефективність програми. Їх основним призначенням є пошук інформації по заданому аргументу. Тому вони мають схожу з базами даних структуру та мають задану кількість полів з даними, що є ключовими і функціональними. До ключового поля висувається умова однозначності. Таблиці системних програм (СП) зазвичай зберігаються в ОП, адже створюються на період виконання. На кожному етапі роботи використовуються свої набори таблиць. Наприклад на етапі ЛА: таблиці імен, констант, сегментних регістрів. На етапі СА і СО: таблиці імен, констант, а для генерації кодів використовують таблиці відповідності внутрішнього подання. При

створенні таблиці імен з врахуванням можливості різної видимості ключ поля складається з образу імені та, частіше за все, номеру блоку та його визначення. Згідно з традиціями програмування роботи з таблицями, базовими операціями є: s select – вибірка даних; insert – додавання рядків; delete – вилучення рядків; update – заміна даних в рядку таблиці. Приклади сигнатур операцій:

```
// 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);
```

2.5. Основні способи організації пошуку в таблицях

В большинстве СП главной целью поиска является определение характеристик, связанных с символическими обозначениями элементов входного языка (ключевых слов, идентификаторов, разделителей, констант и т.п.). Линейный поиск заключается в последовательном сравнении ключа искомого с ключами элементов таблицы до совпадения или достижения конца. При работе с упорядоченной таблицей отсутствие может быть установлено без просмотра всех. При больших объемах эффективным будет двоичный, при котором определяется адрес среднего элемента, с ключевой частью которого сравнивается ключ искомого. При совпадении поиск удачный, а иначе – из дальнейшего поиска исключается половина элементов, в которой искомый находиться не может. Процедура повторяется, пока длина оставшейся части таблицы не станет 0. Но для его корректной работы необходима сортировка, а время выполнения – логарифмическое. Прямой – самый быстрый, но он эффективен только для малых таблиц. Хэш поиск (hash – крошить), основанный на обращении к элементу с ключевой частью Кі, выполняется в таблице с начальным адресом Ан, в которой каждый элемент находится по адресу = Ан + Н(Кі), где функция Н(Кі) дает неповторяющиеся значения для разных элементов и плотно размещает их в памяти. Пример поиска по линейному алгоритму. Аргумент загружен в AX, начальный адрес таблицы – ES:DI, а количество элементов таблицы – в CX.

```
MOV BX, DI ; сохранение начального адреса
REPNZ SCASW ; сканирование
JNZ NotFnd ; если не найден
SUB DI, BX ; определение индекса найденного элемента
SUB DI, 2 ; компенсация
SRL DI, 1 ; получения индекса
```

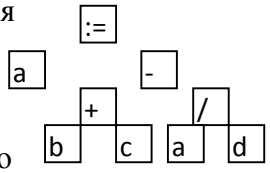
2.6. Робота з таблицями на базі структур з покажчиками на рядки

Див. № 2.3.

3. Графи внутрішнього подання

3.1. Графи та способи їх обробки

В результате СА, как правило, создаются графы синтаксического разбора (Directed Acyclic Graph – DAG), отображающие связи между терминальными и нетерминальными выражениями. Для использования в СО более удобными являются графы подчиненности операций, которые однозначно определяют порядок выполнения операций в конструкции. Пример: a := b + c - a/d.



Такой граф разбора является основой для всех видов СО. Пример внутреннего представления:

```
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}};
```

3.2.Вибір структури для представлення вузла дерева підлеглості операцій та спрямованих ациклічних графів

Результати СА 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:

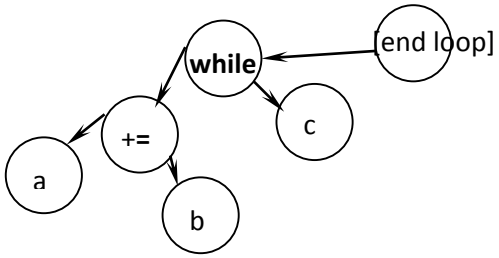
- ідентифікація або мітка вузла;
- прототип співставлення (термінал (Т) або нетермінал (НТ));
- мітка продовження обробки (при успішному результаті СА);
- вказівник на альтернативну вітку, яку можна перевірити (якщо СА невдалий).

Часто буває зручно використовувати змішані алгоритми аналізу. Якраз 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 СО. Приклад:

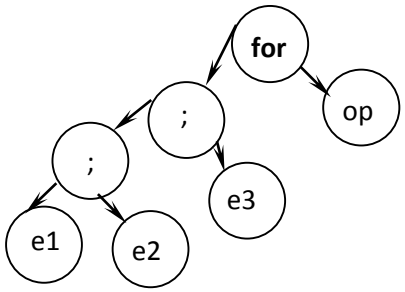
```
struct lxNode // вузол
{int ndOp; // код операції або типу лексеми
 unsigned stkLength; // номер модуля для терміналів
 struct lxNode* prvNd, *pstNd; // до підлеглих вузлів
 int dataType; // код типу даних, які повертаються
 unsigned resLength; // довжина результату
 int x, y, f; // координати розміщення
 struct lxNode*prnNd;}; // до батьківського вузла
```

3.3. Особливості структур дерева внутрішнього подання для операторів циклу

Різниця між поданнями операторів з передумовами і післяумовами полягає у приєднанні повторюваних операторів та умов до протилежних гілок графа. Вони повинні бути відображені вузлами різних типів: _whileP та _untilP для передумов або _whileN та _untilN для післяумов. Цикли типу for також є циклами з передумовами, e1 – задавання початкових установок, e2 – умова продовження, e3 – зміна параметрів. Оператор break закінчує цикл, а continue – переходить на наступну ітерацію (граф перестає бути ациклічним). Вузол [endloop] є взагалі необов’язковим і відображає закінчення циклу. Приклад для do a+= b; while(c);:

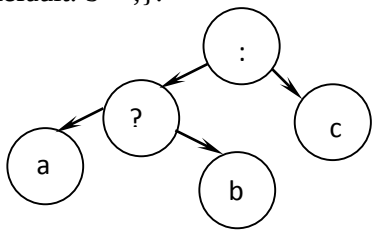


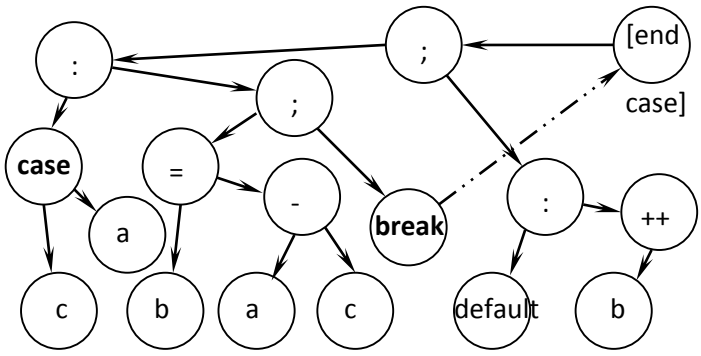

```
struct lxNode pic5[] =
{{_endloop, &pic5[4], NULL},
 {_nam, (struct lxNode*)imgs[1], NULL},
 {_asAdd, &pic5[1], &pic5[3] },
 {_nam, (struct lxNode*)imgs[0], NULL},
 {_whileN, &pic5[2], &pic5[5]},
 {_nam, (struct lxNode*)imgs[4], NULL}};
```



3.4. Особливості структур дерева внутрішнього подання для умовних операторів

Існують такі умовні оператори: ?, if-then, if-then-else, switch-case. Приклад a?b:c та switch(c) {case a: b = a - c; break; default: b++;}:






```
struct lxNode pic2[] =
{{_nam, (struct lxNode*)imgs[0], NULL},
 {_ass, &pic2[0], &pic2[5]},
 {_nam, (struct lxNode*)imgs[1], NULL},
 {_qmrk, &pic2[1], &pic2[4]},
```

```
 {_nam, (struct lxNode*)imgs[0], NULL},
 {_cln, &pic2[3], &pic2[6]},
 {_nam, (struct lxNode*)imgs[4], NULL}};
struct lxNode pic4[] =
{{_nam, (struct lxNode*)imgs[4], NULL},
 {_case,&pic4[0], &pic4[2] },
 {_nam, (struct lxNode*)imgs[1], NULL},
 {_cln, &pic4[1], &pic4[9] },
 {_nam, (struct lxNode*)imgs[0], NULL},
 {_ass, &pic4[4], &pic4[7]},
 {_nam, (struct lxNode*)imgs[1], NULL},
 {_sub, &pic4[6], &pic4[8]},
 {_nam, (struct lxNode*)imgs[4], NULL},
 {_EOS, &pic4[5], &pic4[10]},
 {_break, NULL, NULL},
 {_EOS, &pic4[3], &pic4[13]},
 {_default, NULL, NULL},
 {_cln, &pic4[12],&pic4[15]},
 {_nam, (struct lxNode*)imgs[0], NULL},
 {_inr, &pic4[14], NULL},
 {_endcase, &pic4[11], NULL}};
```

3.5.Задача реконструкції вхідного тексту за внутрішнім поданням

Важливо побудувати програми аналізу та реконфігурації на основі єдиних таблиць, зв'язаних з реалізованою мовою так, щоб для кожної нової мови будувався мінімум схожих таблиць на основі стандарту внутрішнього подання, прийнятого в багатомовній системі програмування. Важливо забезпечити подання всіх змістовних синонімів та омонімів в окремих кодах лексем. Приклад типу лексем з прив'язкою до їх позначень в мовах C/C++, Pascal, що можна використовувати при реконструкції вхідних текстів, при ЛА, СА, СО.

```
#define begOprtr 0x50 // зміщення початку виконуваних операторів
enum tokType
{_nil, _nam, // зовнішнє подання
 _srcn,_cnst, // вхідне і внутрішнє кодування
 _if, _then, _else, _elseif,
// if then else elseif
 _case, _switch, _default, _endcase,
//case switch defualt endcase
 _break, _return, _whileP, _whileN,
// break return while while
 _continue, _repeat, _untilN, _endloop,
// continue repeat until do
... };
```

3.6.Внутрішнє представлення дерева внутрішнього подання для об'яв та описів в програмах

В результаті СА формуються дерева розбору (parse tree), вузли яких відображують термінальні та нетермінальні позначення, розрізнені шляхом використання правил підстановки. Вимоги до кодування внутрішнього подання типів:

- однозначний код для будь-яких типів з різними варіантами модифікаторів;
- легкість розбиття елементів коду на окремі поля;
- легкість визначення або підрахунку номера типу користувача та типу і рівня покажчика та забезпечення загальної кількості типів.

```
struct lxNode // вузол
{int ndOp; // код операції або типу лексеми
 unsigned stkLength; // номер модуля для терміналів
```

```
struct lxNode* prvNd, *pstNd; // до підлеглих вузлів
int dataType; // код типу даних, які повертаються
unsigned resLength; // довжина результату
int x, y, f; // координати розміщення
struct lxNode*prnNd;}; // до батьківського вузла
```

4. Транслятори мов програмування

4.1. Поняття граматик та їх використання для розв'язання задач

Грамматикой (Г) называется четверка G = (Vt, Vn, R, e є Vn), где Vn – конечное множество НТ символов (все обозначения, определяющиеся через правила), Vt – множество Т (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 Т и НТ. Данная запись может читаться одним из следующих способов:

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

Т – обозначения или элементы Г, которые не подлежат дальнейшему анализу (разделители, лексемы). НТ – требуют для своего определения правил в некотором формате, чаще всего в формате правил текстовой подстановки. Формальная Г или просто Г в теории формальных языков (это множество конечных слов (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) Г — 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 Г

Vt, Vn, e є Vn, R — набор правил вида: «левая часть» ::= «правая часть», где: «левая часть» — непустая последовательность Т и НТ, содержащая хотя бы один НТ; «правая часть» — любая последовательность Т и НТ.

Применение

- Контекстно-свободные Г широко применяются для определения грамматической структуры в грамматическом анализе.
- Регулярные Г (в виде регулярных выражений) широко применяются как шаблоны для текстового поиска, разбивки и подстановки, в ЛА.

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 НТ: ФОРМУЛА

Аналитические Г

Порождающие Г – не единственный вид Г, однако наиболее распространенный в приложениях к программированию. В отличие от порождающих Г, аналитическая (распознающая) задает алгоритм, позволяющий определить, принадлежит ли данное слово языку. Например, любой регулярный язык может быть распознан при помощи Г, задаваемой конечным автоматом, а любая контекстно-свободная Г — 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.

4.2. Класифікація граматик за Хомським

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

тип 0. неограниченные Г — возможны любые правила. G = (VT, VN, P, S) типа 0, если на правила вывода не накладывается никаких ограничений (кроме указанных в определении).

тип 1. контекстно-зависимые Г — левая часть может содержать один НТ, окруженный «контекстом» (последовательности символов, в том же виде присутствующие в правой части); сам НТ заменяется непустой последовательностью символов в правой части.

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

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

тип 2. контекстно-свободные Г — левая часть состоит из одного НТ. G = (VT, VN, P, S) контекстно-свободна (КСГ), если каждое правило из P имеет вид $A \to \beta$, где $A \in VN$, $\beta \in (VT \cup VN)^+$.

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

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

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

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

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

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

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

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

Данная Г порождает простейшие арифметические выражения.

Пример КЗГ: 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 $\{ a^n b^n c^n \}$, $n \geq 1$.

Каждая праволинейная Г есть КСГ. КЗГ не допускает правил: $A \to \varepsilon$, где ε – пустая цепочка. КСГ с пустыми цепочками в правой части правил не является КЗГ. Наличие пустых цепочек ведет к Г без ограничений. Если язык L порождается Г типа G, то L называется языком типа G. Пример: L(G3) – КС язык типа G3. Наиболее широкое применение при разработке трансляторов нашли КСГ и порождаемые ими языки.

4.3. Представлення правил в граматиках та їх застосування

Див. питання № 4.1.

4.4. Задачі лексичного аналізу

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

- разделители (знаки операций, именованные элементы языка);
- вспомогательные разделители (пробел, табуляция, enter);
- код разделителя | код внутреннего представления;
- ключевые слова;
- код слова | код внутреннего представления;
- стандартные имена объектов и пользователей;
- константы;
- имя | тип, адрес;
- комментарии.

ЛА может работать в основных режимах: либо как подпрограмма, вызываемая во время СА для получения очередной лексемы, либо как полный проход. На этапе ЛА обнаруживаются простейшие ошибки

(недопустимые символы, неправильная запись чисел, идентификаторов и др.). Выходом лексического анализатора является таблица лексем. Она образует вход СА, который исследует только тип каждой лексемы. Остальная информация используется на более поздних фазах компиляции при СО, подготовке к генерации и генерации кода.

4.5. Граматики, що використовуються для лексичного аналізу

Для решения задачи ЛА могут использоваться разные подходы, один из них основан на теории Г. К ней 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 спецлексемы. Класифікацію можна робити за допомогою таблиці, в якій кожній літері відповідає 1 або декілька ознак. Якщо код ознак займає не більше 1 байта, то класифікацію можна виконувати за допомогою xlat – використовується вміст регістру al, як зміщення відносно початку таблиці перекодування, адреса якої знаходиться в bx. Байт функції, одержаний з відповідного місця в таблиці записується в al. 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) Неклассифицируемые 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. Для построения автомата ЛА нужно определить сигналы его переключения по таблицам классификаторов литер с кодами, удобными для использования в дальнейшей обработке.

```
enum ltrType
{dgt, // десятичная цифра
 ltrexplt, // буква-признак экспоненты
 ltrhxdgt, // литера-шестнадцатеричная цифра
 ltrtpcns, // литера-определитель типа константы
 ltrnmelm, // литеры, допустимые только в именах
 ltrstrlm, // литеры для ограничения строк и констант
 ltrtrnfm, // литеры начала перекодирования литер строк
 nc, // неклассифицированные литеры
 dldot, // точка как разделитель и литера констант
 ltrsign, // знак числа или порядка
 dlmaux,  // вспомогательные разделители типа пробелов
 dlmunop, // одиночные разделители операций
 dlmgrop, // элемент начала группового разделителя
 dlmbrlst,// разделители элементов списков
 dlobrct, // открытые скобки
 dlcbrct, // закрытые скобки
 ltrcode = 16}; // признак возможности кодирования
```

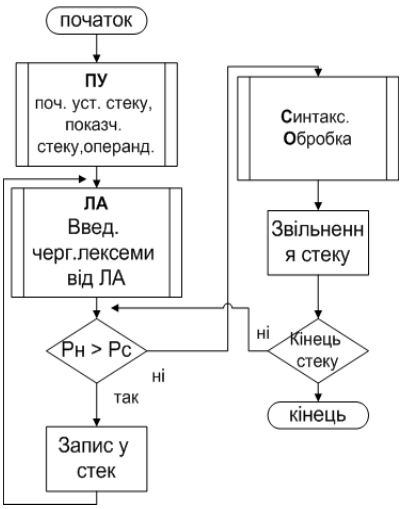
Під регулярною Г будемо розуміти якусь ліволінійну. Це така Г, що її Р правила мають вигляд A → Bt або A → t, де A є N, B є N, t є T. Для Г такого типу передбачений алгоритм розбору для отримання лексеми: перший символ вихідного ланцюжка замінюємо НТ А, для якого в Г є правило A → a. Далі проводимо ітерації до кінця ланцюжка за наступною схемою: отриманий раніше НТ А і розташований правіше від нього Т а вихідного ланцюжка замінюємо НТ В, для якого в граматиці є правило B → Aa. Стан автомату повинен відповідати типу розпізнаної лексеми або типу помилки ЛА. Вхідними сигналами автомату повинні бути класифікаційні ознаки літери вхідної послідовності. Для побудови аналізатора доцільно мати двомірну матрицю переходів, що визначає код наступного стану автомату, одним з яких є поточний а другим – класифікатор вхідної літери. Програма повинна виділяти чергову лексему шляхом циклічного прогляду літер. Цикл закінчується в тому випадку, коли знайдена помилка або лексема закінчується роздільником. Результати роботи ЛА треба рознести по таблицях імен, по таблицях констант і, можливо,

по таблицях модулів. Результати доцільно розміщувати в спеціалізованих структурах, в яких зберігається інформація про код вхідної лексеми та її внутрішнє подання.

4.6. Використання регулярної граматики для лексичного аналізу

Див. № 4.5.

4.7.Задачі синтаксичного аналізу



В задачи входят: найти и выделить основные конструкции в тексте входной программы, установить тип и проверить правильность каждой из них и представить их в виде, удобном для дальнейшей генерации кода.Алгоритм:

- послідовний аналіз лексем і порівняння пріоритетів;
- якщо в нової операції пріоритет вищий, то попередня зберігається у стеці;
- інакше виконується СО і встановлюється зв'язок підлеглості для попередньої операції;
- в будь-якому випадку необхідно до введення наступної лексеми.

Можна додати з 3.2.

```
int nxtProd(struct lxNode*nd,□□□ // 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
&& nd[n].ndOp < _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 = 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 = n; // додавання піддерева
 return nR;}
```

4.8. Граматики, що використовуються для синтаксичного аналізу

В основе синтаксического (С) анализатора лежит распознаватель текста входной программы на основе Г входного языка. Как правило, С конструкции языков программирования могут быть описаны с помощью КСГ, реже встречаются языки, которые, могут быть описаны с помощью регулярных Г. На этапе СА нужно установить, имеет ли цепочка Л структуру, заданную С языка, и зафиксировать ее. Надо решать задачу разбора: дана цепочка Л, и надо определить, выводима ли она в Г, определяющей С языка. Однако структура таких конструкций как выражение, описание, оператор и т.п., более сложная, чем структура идентификаторов и чисел. Поэтому для описания С языков программирования нужны более мощные У, чем регулярные. Обычно для этого используют УКСГ, правила которых имеют вид $A \to \alpha$, где $A \in VN$, $\alpha \in (VT \cup VN)^*$. Г этого класса, с одной стороны, позволяют достаточно полно описать С структуру реальных языков программирования; с другой стороны, для разных подклассов УКСГ существуют достаточно эффективные алгоритмы разбора.

```
struct lxNode // вузол дерева
{int x, y, f; // координати розміщення
 int ndOp; // код типу лексеми
 int dataType; // код типу даних, які повертаються
 unsigned resLength;
 struct lxNode* prnNd; //зв'язок з батьківським вузлом
 struct lxNode* prvNd, pstNd; // зв'язок з підлеглими
 unsigned stkLength;}; //довжина стеку
```

Цель разбора в том, чтобы ответить на вопрос: принадлежит ли анализируемая цепочка множеству правильных цепочек заданного языка. Ответ "да" дается, если такая принадлежность установлена. Получение ответа "нет" связано с понятиям отказа. Единственный отказ на любом уровне ведет к общему отказу. Чтобы получить ответ "да" относительно всей цепочки, надо его получить для каждого правила, обеспечивающего разбор отдельной подцепочки. Так как множество правил образуют иерархическую структуру, возможно с рекурсиями, то процесс получения общего положительного ответа можно интерпретировать как сбор по определенному принципу ответов для листьев, лежащих в основе дерева разбора, что дает положительный ответ для узла, содержащего эти листья. Далее анализируются обработанные узлы, и уже в них полученные ответы складываются в общий ответ нового узла. С теоретической точки зрения существует алгоритм, который по любой данной КСГ и данной цепочке выясняет, принадлежит ли цепочка языку, порождаемому этой Г. Но время работы такого алгоритма (СА с возвратами) экспоненциально зависит от длины цепочки, что с практической точки зрения совершенно неприемлемо.

4.9.Методи висхідного синтаксичного аналізу

В общем случае для восходящего разбора (ВР) строится Г предшествования. В Г предшествования строится матрица попарных отношений всех Т и НТ. При этом определяется три вида отношений: R< S; R >S и R = S;4 отсутствует (это говорит о недопустимости использования R и S рядом в тексте записи на этом языке). Найпростіший алгоритм висхідного розбору для аналізу математичних виразів на основі порівняння пріоритетів окремих операцій. Якщо нижчий пріоритет – записуємо в стек. Якщо пріоритет високий – виконати інтерпретацію чи саму операцію. Результатом такого розбору буде дерево розбору чи підлеглості. Если, в результате полного перебора всех возможных правил, мы не сможем построить требуемое дерево разбора, то рассматриваемая входная цепочка не принадлежит данному языку. ВР также непосредственно связан с любым возможным выводом цепочки из начального НТ. Однако, эта связь, по сравнению с нисходящим разбором (НР), реализуется с точностью до "наоборот".

1. Простое предшествование

Построение отношения предшествования начинается с перечисления всех пар соседних символов правых частой правил, которые и определяют все варианты отношений смежности для границ возможных выстраиваемых на них деревьях. Далее, в каждой паре возможны следующие комбинации Т/НТ символов и продуцируемые из них элементы отношений:

- Все пары соседних символов находятся в отношении =. При этом для метода «свертка-перенос» НТ не может являться символом входной строки, поэтому пары с НТ справа в построении отношения предшествования для такого метода не участвуют.
- Для пары НТ-Т правая граница поддерева, выстраиваемая на основе НТ находится «глубже» правого Т.
- Аналогичное обратное отношение выстраивается для пары Т-НТ: левая нижняя граница поддерева, выстраиваемого на НТ «глубже» левого Т.
- Наиболее сложное, но и самое «продуктивное» соотношение – два НТ, 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 ВР 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 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;
- 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 ошибка) и сами правила грамматики.
- Положительным результатом работы МА будет наличие начального НТ грамматики в стеке при пустой

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

4.10. Представлення результатів синтаксичного аналізу у вигляді дерева розбору

Для представления Г используется списочная структура, называемая С графом. Можно использовать спецузлы с информацией. Вузол об'єднує 4 поля:

- ідентифікація або мітка вузла;
- прототип співставлення (Т або НТ);
- мітка продовження обробки (при успішному результаті СА);
- вказівник на альтернативну вітку, яку можна перевірити (якщо СА невдалий).

```
struct lxNode // вузол
{int ndOp; // код операції або типу лексеми
 unsigned stkLength; // номер модуля для терміналів
 struct lxNode* prvNd, *pstNd; // до підлеглих вузлів
 int dataType; // код типу даних, які повертаються
 unsigned resLength; // довжина результату
 int x, y, f; // координати розміщення
 struct lxNode*prnNd;}; // до батьківського вузла
```

В процессе разбора формируется дерево, которое, в отличие от дерева приоритетов операций, может иметь больше двоичных ответвлений. Чтобы превратить дерево разбора в дерево приоритетов, можно использовать значения предшествований для отдельных Т и НТ. В случае унарных операций, связь с другим операндом не устанавливается. Кроме того, каждый НТ представлен узлом, состоящим из одной компоненты, которая указывает на первый символ в первой правой части, относящейся к этому символу. Результатом работы распознавателя КСГ входного языка является последовательность правил, 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. Однако ни цепочка вывода, ни дерево не являются целью работы компилятора. Дерево вывода содержит массу избыточной информации, которая для дальнейшей работы не требуется. Она включает в себя все НТ, содержащиеся и узлах дерева.

4.11. Відмінності дерева синтаксичного розбору від графів підлеглості операцій

Див. № 4.10.

4.12. Перетворення дерева синтаксичного розбору на дерева підлеглості операцій

Див. № 4.10.

4.13. Алгоритми синтаксичного аналізу з використанням матриць передування

В матрице предшествования (МП) определяют отношение предшествования для всех возможных пар предыдущих и следующих Т и НТ. При этом определяется 4 вида отношений. МП строится так, что на пересечении определяется приоритет отношения. МП квадратная и каждая размерность включает номера Т и НТ. МП – универсальный механизм определения Г разбора. В связи с тем, что следующее обозначение может быть НТ, это вызывает необходимость повторения для полного разбора и может вызвать неоднозначность Г. Линеаризация Г предшествования. – действия по превращению МП на функции предшествования, в большинстве случаев, полная линеаризация невозможна. Пример МП и функции:

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

Алгоритм использует два стека – операций и операндов. Выражения просматриваются слева направо, всё заносятся в стеки до тех пор, пока между символом на вершине стека и входным символом не выполниться

отношение = или >. После этого выделяется тройка (два символа из стека операндов и один из стека операций). На вершину стека операндов заносится вспомогательная переменная-результат и действия повторяются. Операторне передування – коли нема дужок і спеціальних символів.

4.14. Алгоритми низхідного синтаксичного розбору

НР заключается в построении дерева разбора, начиная от корневой вершины до Т. НР заключается в заполнении промежутка между начальным НТ и символами входной цепочки правилами, выводимыми из начального НТ. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой Т и НТ одного из альтернативных правил, порождаемых начальным НТ. Подставляемое правило в общем случае выбирается произвольно. Вместо новых НТ вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил. В последнем случае входная цепочка отвергается. Построение дерева разбора подтверждает принадлежность входной цепочки данному языку. При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что Г данного языка является недетерминированной. Идея: правила определения Г в формулах Бекуса превратить в МП или функции. Отсюда возникает проблема зацикливания при обработке леворекурсивных правил.

1. Метод рекурсивного спуска (модификации для систем автоматизации построения компилятора);
2. Метод синтаксических графов;
3. LL-парсер.

4.15. Метод рекурсивного спуску

Рекурсия — способ общего определения объекта или действия через себя, с использованием ранее заданных частных определений. Розбір починається з кінцевого позначення Г. При його виконанні аналізатор звертається до підлеглого ресурсу, щоб розібрати спочатку перший, а потім наступні позначення правої частини правила підстановки. Рекурсивні правила у формі Бекуса, так що рекурсивні звертання записуються з правого боку, що утворює ліворекурсивні правила. Однак при цьому ми будемо просуватись в глибину рекурсії, не просуваючись вздовж вхідного потоку даних, що фактично призводить до за циклювання аналізатора. Тому для використання необхідно перетворити правила на право рекурсивну форму. Альтернативним є метод синтаксичних графів.

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

Нисходящий разбор начинается с конечного НТ Г, который должен быть получен в результате анализа последовательности Л. Чтоб реализовать алгоритм, необходимо иметь входную последовательность Л и набор управляющих правил в виде направленного графа. Такой подход позволяет строить дерево, в котором из распознанных НТ узлов, формируются указатели на поддеревья и/или Т узлы. Некоторые конструкции в графах, следует дополнить спецсвязями – указатели выхода из цикла.

Пусть в данной формальной Г N — это конечное множество НТ; Σ — конечное множество Т, тогда метод применим только, если каждое правило этой Г имеет следующий вид:

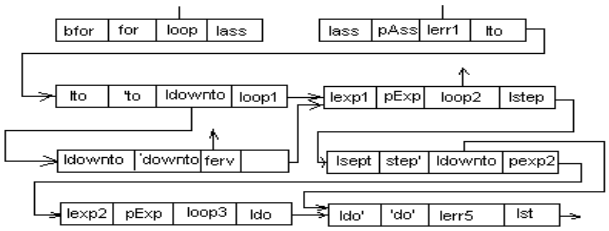
$A \to \alpha$, где $\alpha \subset (\Sigma \cup N)*$, и это единственное правило вывода для этого НТ или $A \to a_1\alpha_1|a_2\alpha_2|...|a_n\alpha_n$ для всех $i = 1,2...n; a_i \neq a_j, i \neq j; \alpha \subset (\Sigma \cup N)*$

Метод учитывает особенности современных языков программирования, в которых реализован рекурсивный вызов процедур. Если обратиться к дереву НР слева направо, то можно заметить, что в начале происходит анализ всех поддеревьев, принадлежащих самому левому НТ. При этом используются и полностью раскрываются все нижележащие правила. Это навязывает ассоциации с иерархическим вызовом процедур в программе. Поэтому, все НТ можно заменить соответствующими им процедурами или функциями, внутри которых вызовы других процедур будут происходить в последовательности, соответствующей их расположению в правилах. Также вызов процедуры или функции реализуется через занесение локальных данных в стек, который поддерживается системными средствами. Заносимые данные определяют состояние обрабатываемого НТ. Использование метода позволяет достаточно быстро и

наглядно писать программу распознавателя на основе имеющейся Г. Использование метода позволяет написать программу быстрее, так как не надо строить автомат. Ее текст может быть и менее ступенчатым, если использовать инверсные условия проверки или другие методы компоновки текста.

4.16. Метод синтаксичних графів для синтаксичного аналізу

В цьому методі послідовність правил С розбору організована у формі графа. Використання цього методу дозволяє комбінувати методи висхідного та низхідного розбору. Приклад синтаксичного графу:



Отсутствие альтернативных вариантов в графе помечает место обнаружения ошибки, компилятор занимается нейтрализацией ошибок:

1. Пропуск дальнейшего контекста до места, с которого можно продолжить программу;
2. Накопление диагностики для ее последующего представления с текстом программы.

Некоторые компиляторы передают управление текстовому редактору подсказкой варианта обрабатывающего ошибки.

4.17. Формування графів підлеглості операцій при синтаксичному аналізі

Див. № 3.2.

4.18. Задачі семантичної обробки

1. семантический анализ – проверяет корректность соединения типов данных во всех операциях и операторах и определяет типы данных для промежуточных результатов. Алгоритм может быть построен через рекурсивные вызовы той же самой функции или процедуры для всех поддеревьев. При достижении T, все рекурсивные 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. интерпретация – сборка константных 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 внутреннего представления путем удаления повторяющихся и неиспользуемых фрагментов графа. Кроме этого, могут быть выполнены действия упрощения эквивалентных превращений.

4. генерация объектных кодов выполняется: в языке высокого уровня; на уровне команд асемблера. Для каждого узла дерева генерируется соотвествующая последовательность команд по шаблонам, а потом используется трансляция асемблера.

5. машинно-зависимая оптимизация учитывает архитектуру и специфику команд целевого устройства.

```
enum datType // кодування типів даних
{_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
{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;};
```

Задачі СА: аналіз сумісності операндів окремих операцій, формування типів результатів виконання операцій та формування довжин результатів. В об'єктному коді необов'язково знаходиться весь код програми, в ньому можуть бути зовнішні посилання на бібліотечні процедури або процедури у інших модулях. Найчастіше як внутрішнє подання програми використовувались формат польського інверсного запису для виразу або постфіксна форма подання.

4.19. Загальний підхід до організації семантичної обробки в трансляторах

Транслятор — это программа, которая принимает исходную и порождает на своем выходе объектную программу. В частном случае объектным может служить машинный язык, и в этом случае полученную программу можно сразу же выполнить на ЭВМ. В общем случае объектный язык необязательно должен быть машинным или близким к нему. В качестве объектного может служить и некоторый промежуточный язык. Транслятор с языка высокого уровня называют компилятором. Для того щоб створити і використовувати узагальнену програму СО необхідно побудувати набори функцій СО для кожного з вузлів графа розбору. Програма повинна перевірити аргументи за таблицею відповідностей аргументів та типів результатів. В більш ранніх компіляторах часто використовували тетрадні та тріадні подання, які складалися з команд віртуальної машини реалізації мови. Вони включали код операції та адреси або вказівники на 2 операнди. В них використовувались 3 окремі посилання: 2 на операнди, а 3-й на результат, тобто вони відповідали 2-х та 3-х адресним командам комп'ютерів попередніх поколінь. Такі форми є залежними від обраної архітектури віртуальних машин. Внутрішні подання Паскалю у формі Р-кодів ближче до 1-адресної машини, а початкова форма більш спрямована до циклічної є взагалі архітектуронезалежною, тому більш загальною. Кроме деревьев подчиненности или направленнях ацикличных графов, использовались:

- Обратная польская запись. A+B –> AB+
- Форматы, похожие на представления машинных операций.

Повне табличне визначення СО 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.

```
struct lxNode // вузол дерева
{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;
 unsigned auxNmb; // довжина стека
 int x, y, f; // координати розміщення
 struct lxNode* prnNd;}; // зв'язок з батьківським вузлом
```

4.20. Організація семантичного аналізу в компіляторах

На этапе СО используются различные варианты деревьев разбора. Семантический анализ обычно выполняется на двух этапах компиляции: на этапе СА и в начале этапа подготовки к генерации кода. В первом случае всякий раз по завершении распознавания определенной С конструкции входного языка выполняется её семантическая проверка на основе имеющихся в таблице идентификаторов данных. Во втором случае, 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. Проверка соблюдения во входной программе семантических соглашений входного языка заключается в сопоставлении входных цепочек

программы с требованиями семантики входного языка программирования. 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.

```
// Елемент таблиці модифікованих типів
struct recrdTPD // структура рядка
{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;};
```

4.21. Організація інтерпретації вхідної мови

Интерпретатор — это программа, которая получает исходную и по мере распознавания конструкций входного языка реализует действия, описываемые этими конструкціями. При реалізації інтерпретації 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). В кінці треба перетворити результат із загального типу в необхідний. 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

Будь-яка реалізація мови програмування має програми підготовки середовища інтерпретації і звертання до головної програми. При коректному завершенні роботи програми необхідно відновити стек ОС до інтерпретації програми.

```
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; // другого аргументу
 unsigned res,lnRes; // результату
 _fop *pintf; // покажчик на функцію інтерпретації
 char *assCd;}; // покажчик на текст макроса
```

4.22. Особливості генерації кодів для обробки даних з плаваючою комою

Это последняя фаза трансляции. Результатом ее является либо ассемблерный, либо загрузочный модуль. Для генерации кода разработаны различные методы, такие как таблицы решений, сопоставление образцов, включающее динамическое программирование, различные С методы. Для генерації слід використовувати машинні команди або підпрограми з відповідними аргументами. Більшість компіляторів системних програм включає такі коди в об’єктні файли OBJ. Такі файли включають 4 групи записів:

- Записи двійкового коду – двійкові коди констант, машинні коди, відносні адреси відносно початку відповідного сегменту;

- Записи 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.

При генерації виконується напрямлений перегляд ациклічного графу, де генератор породжує команди обробки цих даних. Для реалізації генераторів необхідно визначити повні таблиці відповідності усіх можливих вузлів графу у необхідні набори машинних команд. Ранні компілятори одразу формували машинні коди. Компілятори з мови Pascal формували свої результати у P-кодах. На етапі виконання цей код оброблювався середовищем, яке включало підпрограми для обробки всіх операндів та операторів. Для даних виділяють фіксовану пам'ять, а до кодів звертається виконуюча програма, яка дозволяє формувати результат виконання. Але щоб раціонально організовувати модульне програмування необхідно, щоб поєднувані модулі подавалися в однаковому форматі, тому вони добре поєднувалися з модулями на цій же мові, але погано з іншими.

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

Для виклику стандартних підпрограм використовують бібліотеки періоду виконання.

```
int main(int argc, char* argv[])
{float b = 1.4;
 float a[2] = {2, 5};
 int n = 1;
 _asm {
  mov eax,n
  sub eax, 1
  mov n,eax
  fld b
  mov esi,n
  fld a[esi*4]
  fadd b
  fstp b}
 printf("%f", b);
 return 0;} // n = n-1; b = b + a[1];
```

4.23. Особливості генерації кодів для обробки даних цілих типів

Див. № 4.22.

4.24. Особливості організації генерації кодів для роботи з покажчиками

Див. № 4.22.

4.25. Особливості генерації кодів для індексних виразів

Див. № 4.22.

4.26. Машинно-незалежна оптимізація

Машинно-независимые оптимизации (МНО) нельзя считать полностью оторванными от конкретной архитектуры. Они разрабатывались с учётом общих представлений о свойствах некоторого класса машин. В компиляторах они, 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 МНО.

1. Исключение избыточных вычислений, если оно уже было выполнено ранее.
2. Удаление мёртвого кода: любое вычисление, производящее результат, который в дальнейшем более не будет использован, может быть удалено.
3. Вычисления в цикле, результаты которых не зависят от других вычислений цикла, могут быть вынесены за пределы цикла с целью увеличения скорости.
4. Вычисление, которые дают константу, могут быть произведены уже в процессе компиляции.

Наиболее хорошо проработаны алгоритмы для некоторых частных случаев избыточности, однако в общем случае оптимизация связана с анализом смысла и поиском решения задачи. Фазы оптимизации не всегда присутствуют в составе компиляторов – все зависит от целей проектирования.

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

4.27. Машинно-залежна оптимізація

Машинно-залежна оптимізація (МЗО) полягає у ефективному використанні ресурсів компа. Треба ефективно використовувати систему команд. Для МЗО можна виділити такі види: вилучення ділянок програми, результати яких не використовується, вилучення недосяжних ділянок програми, еквівалентні перетворення складніших виразів на простіші. При МЗО таблиці реалізації операндів та операцій стають складнішими і мають декілька варіантів. Потрібно ефективно виконувати операції пошуку, бо вони є критичними по часу. При обробці виразів проміжні дані краще зберігати у стеці регістра з плаваючою крапкою. При переполнении стека возникает прерывание, где сохраняем состояние регистра сопроцессора в главном стеке задачи и продолжать работу с оновленым стеком. Эффективность алгоритмов напрямую зависит от наличия той или иной дополнительной информации об исходной программе, а именно:

- «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).
- Алгоритмы «SOA» и «GOA» используют информацию о локальных переменных программы. Требуется информация, что эта переменная локальная.
- Алгоритмы раскладки локальных переменных по банкам памяти требуют информации об обращениях к локальным переменным.
- Алгоритм «Array Index Allocation» должен иметь на входе информацию о массивах.

4.28. 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, СО, оптимизация, генерація 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 виртуальной машины, реализующей язык.
+: отсутствие промежуточных действий для трансляции упрощает реализацию интерпретатора и делает его удобнее;
- инерпритатор должен быть на машине, где должна исполняться программа.
Смешанная реализация – интерпретатор перед исполнением программы транслирует её на промежуточный язык (например, в байт-код или p-код), более удобный для интерпретации (то есть речь идёт об интерпретаторе со встроенным транслятором).

5. Операційні системи

5.1. Типи ОС і режими їх роботи

ОС можно классифицировать на основании многих признаков. Наиболее распространенные способы их классификации далее.
Разновидности ОС:
по целевому устройству:
1. Для мейнфреймов
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). Существует ряд различных путей построения ОС. Монолитная ОС. Наиболее ранние ОС не имели определенной структуры. ОС представляет собой просто большую программу, состоящую из множества процедур; не существовало ограничений на то, какие процедуры могут вызываться, была сделана только робкая попытка ограничить совокупность системных данных. Поддерживать такие системы сложно, так как модификация одной программы может породить проблемы в других частях ОС. В таких системах трудно вылавливание ошибок; любая попытка решить одну проблему приводит к новой проблеме. Уровневая ОС. Другой способ структурирования ОС состоит в том, чтобы разделить ее на модули, которые составляют уровни; каждый уровень обеспечивает множество функций, которые зависят только от нижних уровней. Самые нижние уровн – это те, которые наиболее критичны по надежности, мощи и производительности. Таким образом, к чистой машине разработчик добавляет слой программного обеспечения. Программное обеспечение вместе с низлежащей аппаратурой обеспечивает выполнение некоторого множества команд, определяющих новую виртуальную машину. На следующем шаге выделяется новое нужное свойство, добавляется новый слой программного обеспечения и получается еще более удобная виртуальная машина. Слои программного обеспечения добавляются последовательно, причем каждый слой реализует одно или несколько желаемых свойств до тех пор, пока не будет получена требуемая виртуальная машина. Преимуществом этой структуры является то, что модульный подход уменьшает зависимость между различными компонентами системы, сокращая нежелательные взаимодействия. Только несколько используемых сейчас ОС являются образцом такого построения в чистом виде. Значимыми примерами таких ОС являются T.H.E. и RC-4000. В коммерческом мире – это VME и VMS.

5.2.Типовий склад програм 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). ОС, 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.

5.3. Особливості визначення пріоритетів задач в ОС

Приоритеты дают возможность индивидуально выделять каждую задачу по важности. Планирование выполнения задач является одной из ключевых концепций в многозадачности и многопроцессорности как в ОС общего назначения, так и в ОС 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.

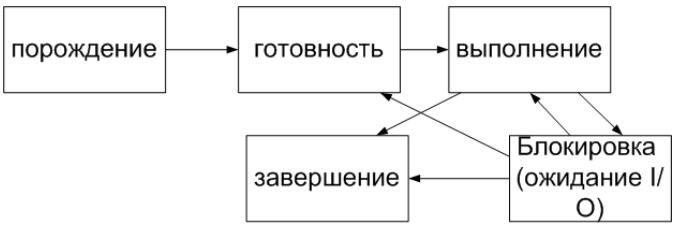
Известно большое количество правил (дисциплин диспетчеризации), в соответствии с которыми формируется список (очередь) готовых к выполнению задач, различают два больших класса дисциплин обслуживания — бесприоритетные и приоритетные. При бесприоритетном выбор задачи производятся в некотором заранее установленном 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, с абсолютным, адаптивное обслуживание, приоритет зависит от t ожидания);
- с динамическим приоритетом (приоритет зависит от t ожидания или 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. 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.

5.4. Основні стани виконання задач в ОС



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.

5.5.Збереження стану задач в реальному режимі

Команда INT в реальном у режимі (РР) виконується як звертання до підпрограми обробника переривань, адреса якої записана в 1 КБ пам'яті як 4 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 (БПП), на входи яких подаються сигнали готовності пристроїв. БПП програмуються при завантаженні ОС і BIOS через порти 20/21H, 0A0/0A1H: 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
```

5.6. Збереження стану задач в захищеному режимі

- сохраняются все регистры задачи;
- каталог таблиц страниц процесса.

В захищеному режимі (ЗР) 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 переключенні задачі управління передач за новим сегментом стану задачі запам'ятовується адреса перерваної задачі. В цьому випадку новий сегмент TSS буде пов'язаний з іншим адресним простором і буде включати в себе новий стек для нової задачі. Регістри переривань програми запам'ятовуються в старому TSS і таким чином 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
cli ; запрет прерываний
in al, INT_MASK_PORT
and al, 0ffh
out INT_MASK_PORT, al
mov al, 8f
out CMOS_PORT, al
```

5.7. Основні архітектурні елементи захищеного режиму та їх призначення

Див. № 5.6.

5.8. Способи організації переключення задач

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

5.9. Організація роботи планувальників задач і процесів

Див. № 5.3.

5.10. Способи організації планувальників задач

Див. № 5.9.

5.11. Механізми переключення задач в архітектурі процесора

Див. № 5.8.

5.12. Організація захисту пам’яті в процесорах

NX (XD) — атрибут страницы памяти в архитектурах x86 и x86-64, который может применяться для более надежной защиты системы от программных ошибок, а также использующих их вирусов, троянских коней и прочих вредоносных программ. NX (No eXecute) — терминология AMD. Intel называет этот атрибут XD-бит (eXecution Disable). Поскольку в современных компьютерных системах память разделяется на страницы, имеющие определенные атрибуты, разработчики процессоров добавили ещё один: запрет исполнения кода на странице. То есть, такая страница может быть использована для хранения данных, но не программного кода. При попытке передать управление на такую страницу процессор сформирует особый случай ошибки страницы и программа (чаще всего) будет завершена аварийно. Атрибут защиты от исполнения давно присутствовал в других микропроцессорных архитектурах, однако в x86-системах такая защита реализовывалась только на уровне программных сегментов, механизм которых давно не используется современными ОС. Теперь она добавлена ещё и на уровне отдельных страниц. Современные программы четко разделяют на сегменты кода («text»), данных («data»), неинициализированных данных («bss»), а также динамически распределяемую область памяти, которая подразделяется на кучу («heap») и программный стек («stack»). Если программа написана без ошибок, указатель команд никогда не выйдет за пределы сегментов кода, однако, в результате программных ошибок, управление может быть передано в другие области памяти. При этом процессор перестанет выполнять какие-то запрограммированные действия, а будет выполнять случайную последовательность команд, за которые он будет принимать хранящиеся в этих областях данные, до тех пор, пока не встретит недопустимую последовательность, или попытается выполнить операцию, нарушающую целостность системы, которая вызовет срабатывание системы защиты. В обоих случаях программа завершится аварийно. Также процессор может встретить последовательность, интерпретируемую как команды перехода к уже пройденному адресу. В таком случае процессор войдет в бесконечный цикл, и программа «зависнет», забрав 100 % процессорного времени. Для предотвращения подобных случаев и был введен этот дополнительный атрибут: если некоторая область памяти не предназначена для хранения программного кода, то все её страницы должны помечаться NX-битом, и в случае попытки передать туда управление процессор сформирует особый случай и ОС тут же аварийно завершит программу, сигнализировав выход за пределы сегмента (SIGSEGV). Основным мотивом введения этого атрибута было не столько обеспечение быстрой реакции на подобные ошибки, сколько то, что очень часто такие ошибки использовались злоумышленниками для несанкционированного доступа к компьютерам, а также написания вирусов. Появилось огромное количество таких вирусов и червей, использующих уязвимости в распространенных программах. Один из сценариев атак состоит в том, что воспользовавшись переполнением буфера в программе (зачастую это демон, предоставляющий некоторый сетевой сервис), специально написанная вредоносная программа (эксплоит) может записать некоторый код в область данных уязвимой программы таким образом, что в результате ошибки этот код получит управление и выполнит действия, запрограммированные злоумышленником (зачастую это запрос выполнить программу-оболочку ОС, с помощью которой злоумышленник получит контроль над уязвимой системой с правами владельца уязвимой программы, очень часто это root). Переполнение буфера часто возникает, когда разработчик программы выделяет некоторую область данных (буфер) фиксированной длины, считая, что этого будет достаточно, но потом, манипулируя данными, никак не проверяет выход за её границы. В результате поступающие данные займут области памяти им не предназначенные, уничтожив имеющуюся там информацию. Очень часто временные буферы выделяются внутри процедур (подпрограмм), память для которых выделяется в программном стеке, в котором также хранятся адреса возвратов в вызывающую подпрограмму. Тщательно изучив код программы, злоумышленник может обнаружить такую ошибку, и теперь ему достаточно передать в программу такую последовательность данных, обработав которую программа ошибочно заменит адрес возврата в стеке на адрес, требуемый злоумышленнику, который также передал под видом данных некоторый программный код. После завершения подпрограммы инструкция возврата (RET) вытолкнет из стека в указатель команд адрес входа в процедуру злоумышленника. Контроль над компьютером получен. Благодаря атрибуту NX такое

становится невозможным. Область стека помечается NX-битом и любое выполнение кода в нём запрещено. Теперь же, если передать управление стеку, то сработает защита. Хоть программу и можно заставить аварийно завершиться, но использовать её для выполнения произвольного кода становится очень сложно (для этого потребуется ошибочное снятие программой NX-защиты). Однако некоторые программы используют выполнение кода в стеке или куче. Такое решение может быть связано с оптимизацией, динамической компиляцией или просто оригинальным техническим решением. Обычно, операционные системы предоставляют системные вызовы для запроса памяти с разрешенной функцией исполнения как раз для таких целей, однако многие старые программы всегда считают всю память исполнимой. Для запуска таких программ под Windows приходится отключать функцию NX на весь сеанс работы, и чтобы включить её вновь, требуется перезагрузка. Хотя в Windows и предусмотрен механизм белого списка приложений, для которых отключен DEP, тем не менее данный метод не всегда работает корректно. NX-бит является самым старшим разрядом элемента 64-битных таблиц страниц, используемых процессором для распределения памяти в адресном пространстве. 64-разрядные таблицы страниц используются операционными системами, работающими в 64-битном режиме, либо с включенным расширением физических адресов (PAE). Если ОС использует 32-разрядные таблицы, то возможности использовать защиту страниц от исполнения нет. Защита программ и данных в процессоре i286. Взаимная защита программ и данных в MS DOS основана на "джентельменском" соглашении о неиспользовании чужих областей. Но проблема заключается в том, что соглашение может быть нарушенно непреднамеренно всвязи с ошибкой в программе или из-за того что ОС не всегда может учесть требования всех загруженных потребителей адресного пространства.

Для организации защиты информации необходимо обеспечить:

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

Для решения этих задач используется:

1 Схема "супервизор-пользователь":

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

2 Система ключей:

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

3 Система привелигированных и чувствительных команд;

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

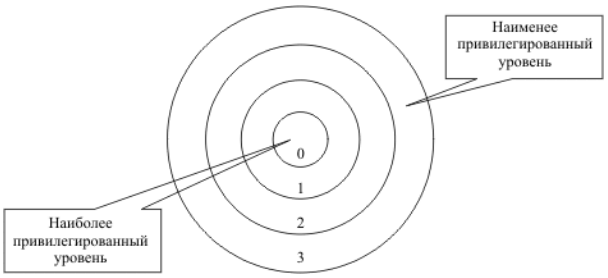
5.13. Організація захисту пам’яті в ОС

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.

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:



- 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. Для гарантированного разделения данных и кодов необходимо использовать гораздо больше информации о сегменте, чем это возможно в реальном режиме. В защищенном режиме загружается так называемый селектор, в котором хранится указатель на восьмибайтный блок памяти, называемый дескриптором, в котором хранится вся нужная информация о сегменте. При выполнении команд загрузки сегментных регистров содержимое дескриптора для получения нормальной скорости работы процессора переносится в теневой регистр, связанный с сегментным, но непосредственно недоступный программисту .Эти блоки хранятся в сегментах памяти, называемых таблицами дескрипторов: глобальной – GDT, локальной – LDT, определяемой для каждой задачи и прерываний – IDT, замещающей таблицу векторов прерываний реального режима. Селектор содержит относительный адрес в таблице бит типа таблицы: bl = 0 – для LDT, bl = 1 – для GDT, а также двухбитовое поле запрашиваемого уровня привилегий для контроля правильности доступа в механизме защиты. Соотношение запрашиваемого и предоставляемого уровня привилегий определяют 4 кольца защиты : 00 – кольцо ядра ОС, 01 – кольцо обслуживания аппаратуры, 10 – кольцо системы программирования базами данных и расширениями ОС, 11 – кольцо прикладных программ пользователя. Поля дескриптора используются со следующим назначением: базовый адрес сегмента определяет начальный линейный адрес и занимает байты 2,3,4 и 7 дескриптора.20-битовое поле предела определяет границу сегмента и занимает байты 0 и1, а также младшие 4 бита байта 6 дескриптора. Поле предела позволяет аппаратно контролировать выход используемого адреса за пределы сегмента. В 5 дескриптора закодированы права доступа к сегменту. Бит присутствия p дает возможность управления виртуальной памятью: p = 1, когда описаный сегмент присутствует в физической памяти, p = 0, когда он перемещен на диск.

5.14. Організація віртуальної пам’яті в ОС

Задача віртуальної пам’яті виникла у зв’язку з тим, що на різних комп’ютерах не завжди вистачало фізичної пам’яті для розв’язання складних задач. Для забезпечення такої ідеї використовуються наступні засоби:

- використання overlay компанувальників, що забезпечує часткове завантаження окремих модулів програми по мірі її виконання;
- динамічне завантаження виконавчих модулів виконується програмістом з використанням спеціальних функцій.

Віртуальна пам'ять в основному розташовується на жорсткому диску, тобто коди і адреси – в ОП, а інші дані – на жорсткому диску. Але так як процесор може обробляти лише коди, занесені в ОП, то треба розв’язати задачу розміщення в ОП тих фрагментів даних, що були на жорсткому диску. Використовуються або методи заміщення сегментів, або методи заміщення сторінок. Для цього були

введені дескриптори сегментів для прямого доступу до пам’яті. Довжина сторінки пам’яті – 4 Кб. Старші 20 розрядів виконавчої адреси визначають номер сторінки. Щоб використовувати сторінкову організацію, треба завантажити таблицю сторінок. Якщо якась сторінка у віртуальній пам’яті відсутня, виконується переривання і послідовність таких дій:

- пошук вільної рідко необхідної сторінки;
- збереження сторінки на жорсткий диск;
- запис сторінки в ОП;
- корекція таблиці сторінок.

Віртуальна пам'ять реалізується як віртуальний драйвер, який збуджується при відсутності сторінок в ОП. Ядро ОС керує цим драйвером. Для правильної роботи такого драйверу при завантаженні будь-якої програми в ОП потрібно сформувати таблицю сторінок для програми і розмістити в ОП, це дозволить зробити зв'язок між ОП і жорстким диском коли це буде потрібно в процесі виконання програми. 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 спецсегмент обміну даних, який розміщувався на диску і встановлювалась відповідність блоків сегментів обміну сегментам віртуальної пам’яті. В процесорах типу Pentium було передбачено включення до дескрипторів сегментів біту їх наявності в ОП, коли відповідна пам’ять відтворюється на диску цей біт установлюється в 0, і при спробі звернення до такої пам’яті запускався обробник переривань, який ініціює збереження змінних даних на диску і перенесення потрібних даних з області обміну в ОП. Така 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 4 Кб і адресуються 12 бітами, номери сторінок використовують 20 додаткових бітів. В таблиці сторінок для кожної сторінки фіксується фізична адреса для якої відповідні дані. Перерахунок логічних даних на фізичні виконується апаратурою процесора, що не потребує додаткового часу, не впливає на швидкість. Якщо відповідна сторінка відсутня в ОП, разом з блоком суміжних сторінок зчитується з дискової області обміну, або з дискової проекції в ОП. Таким чином для ефективної роботи віртуальної пам’яті в сучасній ОП важливо зберегти файли інформації в форматах що відповідає проекціям сторінок на диск. Для підвищення надійності виконання прикладних програм в версіях Windows NT і вище крайні адреси віртуальної пам’яті розміром 64К не використовуються для розміщення кодів і даних програм. Вони розглядаються як резервні, щоб уникнути помилкових звертань до пам’яті. Фактично це блокує виконання неефективних машинних програм, які можуть виникнути в результаті помилок програмування. Адресний простір задач (може охоплювати до 4Гб) розбивається на 2 частини: одна зберігає інформацію задачі; друга – використовує інтерфейс програм введення-виведення. При побудові сучасних ОС віртуальна пам’ять також ініціалізується як спеціальна служба ОС, яка при створенні нової задачі будує відповідні таблиці задач і виконує розподіл даних між ОП та дисковими областями обміну. При виконанні переривань за відсутності сторінок в ОП сервер захищеного режиму виконує потрібне заміщення сторінок між ОП.

5.15. Організація роботи користувачів, реєстрів та аудиту в ОС

У системах розділення часу пропонується рівноправне обслуговування всіх процесів по черзі, щоб користувач бачив просування своєї задачі. Для цього кожному користувачу по черзі надається квант процесорного часу, за який відбувається виконання якоїсь частки задачи. І маємо комбінацію різних способів управляння задачами. Распространение многопользовательских систем потребовало решения задачи разделения полномочий, позволяющей избежать возможности модификации исполняемой программы или данных одной программы в памяти компьютера другой (содержащей ошибку или злонамеренно подготовленной) программы, а также модификации самой ОС прикладной программой. Реализация разделения полномочий в ОС была поддержана разработчиками процессоров, предложивших архитектуры с двумя режимами работы процессора — «реальным» (в котором исполняемой программе доступно всё адресное пространство компьютера) и «защищённым» (в котором доступность адресного пространства ограничена диапазоном, выделенном при запуске программы на исполнение).

5.16. Ієрархічна організація програм введення-виведення

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
```

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

```
MOV DX,cmPrt ; подготовка адреса порта
OUT DX,AX
```

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

5.17. Необхідність синхронізації даних в задачах введення-виведення

Для синхронізації обміну даними між процесами 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). Mutex це об'єкт, який може бути в двох станах (вільному та зайнятому). При запиті 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 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. (Mutex – 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.

```
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_lockObject);
if (m_pObject) m_pObject->SomeMethod();
::LeaveCriticalSection(&m_lockObject);}
// Нить №2
void Proc2(IObject *pNewObject) {
::EnterCriticalSection(&m_lockObject);
if (m_pObject) delete m_pObject;
m_pObject = pNewobject;
::LeaveCriticalSection(&m_lockObject);}
```

5.18. Способи організації драйверів в ОС

ОС 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 – регистрация в системе, первичная инициализация 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. 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.

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

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

жестко определен, поэтому большинство драйверов может работать друг с другом совместно. Драйверы бывают двух типов: ориентированные на символьный либо блоковый обмен данными. Драйверы, обслуживающие хранение и/или доступ к данным на дисках либо других устройствах прямого доступа, обычно ориентированы на блоковый обмен данными. Блоковые драйверы передают данные фиксированными порциями. Драйвер состоит из трех основных частей: заголовка, программы стратегий и программы прерываний. Заголовок описывает возможности и атрибуты драйвера, присваивает имя драйверу символьного устройства и содержит NEAR (только смещение, одно слово) указатели на программы стратегий и прерываний, а также FAR (смещение и сегмент, двойное слово) указатель на следующий драйвер в цепочке драйверов (цепочка является однонаправленной, что затрудняет поиск ее начала). Указатель на следующий драйвер устанавливает MS-DOS сразу после завершения процедуры инициализации, а в самом драйвере ему должно быть присвоено начальное значение -1 (FFFFFFFFH). Драйвер должен помещаться в 64K памяти, т.к. программы стратегий и прерываний содержат лишь смещения внутри выделенного драйверу сегмента. Программа стратегий вызывается, когда драйвер устанавливается при загрузке операционной системы, а также при каждом сгенерированном операционной системой запросе на ввод/вывод. Единичный запрос на ввод/вывод от прикладной программы может породить несколько запросов к драйверу. Задача данной программы заключается лишь в том, чтобы сохранить где-нибудь некоторый адрес для дальнейшей обработки. Этот адрес, передаваемый в паре регистров ES:BX, указывает на структуру, называемую заголовком запроса, которая содержит информацию, сообщающую драйверу, какую операцию он должен выполнить. Существенно то, что программа стратегий не осуществляет никаких операций ввода/вывода, а лишь сохраняет адрес заголовка запроса для последующей обработки программой прерываний. В мультизадачной системе этот адрес должен был бы храниться в некотором массиве, который при последующем вызове процедуры прерываний подвергался бы сортировке с целью оптимального использования устройства. Под управлением MS-DOS программа прерываний вызывается сразу после программы стратегий. Следует обратить внимание на то, что между вызовами программ стратегий и прерываний допустимы прерывания, а это может создать трудности, если драйвер написан в предположении, что между этими двумя вызовами проходит "нулевое время". Если вместо цикла ожидания использовать систему апаратных прерываний по готовности, то нет необходимости менять структуру драйвера, но цикл ожидания надо заменить на обращения к системной программе, обеспечивающей взаимодействие с обработчиком прерываний. Для реализации системы прерываний (в частности апаратных прерываний по готовности внешних устройств). В интерфейсные схемы включен блок програмируемых прерываний. Этот блок, как и ВУ, подключается через порты ввода/вывода, т.е. в нем есть порты программирования и порты настройки. В машинах типа IBM PC настройка этих устройств стандартизована, т.е. она выполняется так, что обращение к аппаратным прерываниям осуществляется по адресам(номерам векторов прерываний с 08h по 0Fh и с 0C0h...0C7h). Таким образом, блок программируемого прерывания имеет 16 входов, к которым можно подключать сигналы готовности ВУ. При поступлении сигнала готовности блок ПП генерирует команду int<Nвектора>.

5.19. 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 прерываний это не тоже самое, что программа обработки прерываний, которая может присутствовать в качестве необязательной части работающего по прерываниям драйвера. На самом деле, 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 ; Точка входа в STRATEGY
dw offset INTERRUPT ; Точка входа в INTERRUPT
db 8 dup (?) ; Количество устройств/поле имени
; РЕЗИДЕНТНАЯ ЧАСТЬ ДРАЙВЕРА
req_ptr dd ? ; Указатель на заголовок запроса
; ПРОГРАММА СТРАТЕГИИ
; Сохранить адрес заголовка запроса для СТРАТЕГИЙ в REQ_PTR. Находится в
ES:BX.
STRATEGY PROC FAR
mov cs:word ptr [req_ptr], bx
mov cs:word ptr [req_ptr + 2], bx
ret
STRATEGY ENDP
; ПРОГРАММА ПРЕРЫВАНИЙ
; Обработать команду, находящуюся в заголовке запроса Адрес в REQ_PTR в форме
СМЕЩЕНИЕ:СЕГМЕНТ.
INTERRUPT PROC FAR
pusha
lds bx, cs:[req_ptr] ; Получить адрес заголовка запроса
INTERRUPT ENDP
DRIVER ENDS
END
```

5.20. Програмно-апаратні взаємодії при обробці переривань в машинах IBM PC

В реальном режиме имеются программные (ПП) и аппаратные (АП) прерывания. ПП инициируются командой INT, АП – 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. Обычное инициируется командой INT (ПП) или внешним событием (АП). 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. Поэтому в мультизадачной среде особые случаи, возникающие в отдельных задачах, не оказывают влияния на выполнение остальных задач. В защищённом режиме прерывания могут приводить к переключению задач. Одна и та же процедура обработки прерывания может использоваться для нескольких устройств ввода-вывода (например, если они используют одну линию прерываний, идущую к контроллеру прерываний), поэтому первое действие собственно программы обработки состоит в определении устройства. Зная его, мы можем выявить процесс, который инициировал выполнение соответствующей операции. Поскольку прерывание возникает как при удачном, так и при неудачном ее выполнении, следующее действие – определить успешность завершения операции, проверив значение бита ошибки в регистре состояния устройства.

5.21. Особливості роботи з КПП

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 БПП
```

Если обслуживаемое прерывание обрабатывается вспомогательным БПП компьютеров типа AT и более поздних, то уще 2 строчки:

```
MOV AL, 20H
OUT 0A0H, AL; Выдача EOI на БПП-2
```

5.22. Організація драйверів в ОС за схемою “клієнт-сервер”

Другим подходом, также заключающимся в делении системы на модули, является наделение модулей некоторым множеством функций. Однако, вместо ранжирования по уровням, модули являются более или менее равноправными. Они взаимодействуют не вызовом процедур из друг друга, а посылкой сообщений через центральный обработчик сообщений. Сообщения идут в обоих направлениях, результаты возвращаются то тому же пути, что и запрос. Модуль, посылающий первоначальное сообщение известен как клиент, модуль, получающий его (и предоставляющий соответствующий сервис) известен как сервер. Статус данного модуля не всегда один и тот же; сервер может быть запросом для предоставления искомого сервиса, для предоставления которого может быть необходимо послать запрос еще другому модулю; и поэтому в этом случае этот модуль выступает как клиент. Этот подход предлагает те же преимущества, что и уровневый подход, но с увеличением изоляции и сокращением зависимости между модулями (с расширением возможности запуска клиента и сервера на разных процессорах или даже разных системах). Количество критического кода также сокращается. Центральной компонентой такой системы является, конечно, та, что производит обработку сообщений и обеспечивает базисные функции; т.е. то, что часто называют микроядром. Одним из примеров клиент-сервер ОС является Windows NT.