Системні програми (1-5)

1. Класифікація системних програм

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

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

Системное программное обеспечение предназначено для:

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

Классификация:

2 вида классификаций:

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

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

2. Системні управляючі програми

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

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

Как правило, они находятся в основной памяти. Это резидентные программы, составляющие ядро ОС. Управляющие программы, которые загружаются в память непосредственно перед выполнением, называю транзитными (transitive).

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

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

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

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

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

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

3. Системні обробляючі програми

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

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

4. Структура системних програм

Обычно как для системной управляющей, так и для системной обрабатывающей программ входные данные подаются в виде директив. Для операционной системы это командная строка, а для системы обработки это программа

на входном языке программирования для компил. и интерпрет., и программа в машинных кодах для компановщика и загрузщика.

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

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

5. Типові об'єкти системних програм

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

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

Структури даних (6-16)

6. Способи організації таблиць та індексів

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

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

имя_структуры struc

последовательность директив DB,DW,DD,DQ,DT

имя_структуры ends

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

имя_переменной имя_структуры <спецификация_инициализации>

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

student iv_groups <'timofey',19,5>

mov ax,student.age ; ax => 19

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

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

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

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

оператор вызова записи выглядит следующим образом:

имя_переменной имя_записи <значение полей записи>

к полям записи применимы следующие операции:

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

```
mov al, student
and al, mask age
mov cl, age
shr al,cl ; выравнивание поля age вправо существует несколько методов поиска в таблицах – линейный, двоичный и
```

прямой (хэш-поиска). смотри ниже.

7. Організація таблиць як масивів записів

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

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

8. Організація таблиць у вигляді структур з покажчиків

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

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

9. Організація роботи з таблицями в системних програмах

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

Основные операции:

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

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

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

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

10. Організація пошуку

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

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

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

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

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

11. Лінійний пошук

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

Алгоритм линейного поиска в неупорядоченной таблице:

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

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

12. Двійковий пошук

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

алгоритм:

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

При определении адреса среднего элемента следует выполнить следующие действия:

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

13. Пошук за прямою адресою, хеш-пошук

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

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

- метод перехода к новому основанию, при котором ключ Ki преобразуется в код по правилам системы счисления с другим основанием. Полученное число усекается до размера адреса. Алгоритм:

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

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

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

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

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

14. Хэш ф- ции

Хеш-функция должна отображать ключ в целое число из диапазона 0...n-1. При этом количество коллизий должно быть ограниченным, а вычисление самой хеш-функции – очень быстрым. Некоторые методы удовлетворяют этим требованиям.

Метод деления

Наиболее часто используется метод деления (division method), требующий двух шагов. Сперва ключ должен быть преобразован в целое число, а затем полученное значение вписывается в диапазон 0...n-1 с помощью оператора получения остатка. На практике метод деления используется в большинстве приложений, работающих с хешированием.

Предположим, что ключ – пятизначное число. Хеш-функция извлекает две младшие цифры. Например, если это число равно 56389, то HF(56389) = 89. Две младшие цифры являются остатком от деления на 100.

```
int HF(int key)
{
  return key % 100; // метод деления на 100
}
```

Эффективность хеш-функции зависит от того, обеспечивает ли она равномерное распределение ключей в диапазоне 0...n-1. Если две последние цифры соответствуют году рождения, то будет слишком много коллизий при идентификации подростков, играющих в юношеской бейсбольной лиге.

Другой пример – ключ-символьная строка C++. Хеш-функция отображает эту строку в целое число посредством суммирования первого и последнего символов и последующего вычисления остатка от деления на 101 (размер таблицы).

```
// хеш-функция для символьной строки.
// Возвращает значение в диапазоне от 0 до 100
int HF(char *key)
{
  int len = strlen(key), hashf = 0;

  // если длина ключа равна 0 или 1, возвратить key[0].
  // иначе сложить первый и последний символ
  if (len <= 1)
    hashf = key[0];
  else
    hashf = key[0] + key[len-1];

  return hashf % 101;
}
```

Эта хеш-функция приводит к коллизии при одинаковых первом и последнем символах строки. Например, строки «start» и «slant» будут отображаться в индекс 29. Так же ведет себя хеш-функция, суммирующая все символы строки.

```
int HF(char *key)
{
  int hashf = 0;
```

```
  // просуммировать все символы строки и разделить на 101
  while (*key)
    hashf += *key++;

  return hashf % 101;
 }
```

Строки «bad» и «dab» преобразуются в один и тот же индекс. Лучшие результаты дает хеш-функция, производящая перемешивание битов в символах.

В общем случае при больших n индексы имеют больший разброс. Кроме того, математическая теория утверждает, что распределение будет более равномерным, если n – простое число.

Другие методы хеширования

Метод середины квадрата (midsquare technique)

предусматривает преобразование ключа в целое число, возведение его в квадрат и возвращение в качестве значения функции последовательности битов, извлеченных из середины полученного числа. Предположим, что ключ есть целое 32-битное число. Тогда следующая хеш-функция извлекает средние 10 бит возведенного в квадрат ключа.

```
 // возвратить средние 10 бит произведения key*key
 int HF(int key);
 {
   key *= key;          // возвести ключ в квадрат
   key >>= 11;          // отбросить 11 младших бит
   return key % 1024    // возвратить 10 младших бит
 }
```

При мультипликативном методе (multiplicative method)

используется случайное действительное число f в диапазоне от $0 \le f \le 1$. Дробная часть произведения f * key лежит в диапазоне от 0 до 1. Если это произведение умножить на n (размер хеш-таблицы), то целая часть полученного произведения даст значение хеш-функции в диапазоне 0...n-1.

```
 // хеш-функция, использующая мультипликативный метод;
 // возвращает значение в диапазоне 0...700
 int HF(int key);
 {
   static RandomNumber rnd;
   float f;

   // умножить ключ на случайное число из диапазона 0...1
   f = key * rnd.fRandom();
   // взять дробную часть
   f = f - int(f);
   // возвратить число в диапазоне 0...n-1
   return 701*f;
 }
```

15. Разрешение коллизий

Несмотря на то, что два или более ключей могут хешироваться одинаково, они не могут занимать в хеш-таблице одну и ту же ячейку. Нам остаются два пути: либо найти для нового ключа другую позицию в таблице, либо создать для каждого значения хеш-функции отдельный список, в котором будут все ключи, отображающиеся при хешировании в это значение. Оба варианта представляют собой две классические стратегии разрешения коллизий – открытую адресацию с линейным перебором и метод цепочек. Мы проиллюстрируем на примере открытую адресацию, а сосредоточимся главным образом на втором методе, поскольку эта стратегия является доминирующей.

Открытая адресация с линейным перебором

Эта методика предполагает, что каждая ячейка таблицы помечена как незанятая. Поэтому при добавлении нового ключа всегда можно определить, занята ли данная ячейка таблицы или нет. Если да, алгоритм осуществляет перебор по кругу, пока не встретится «открытый адрес» (свободное место). Отсюда и название метода. Если размер таблицы велик относительно числа хранимых там ключей, метод работает хорошо, поскольку хеш-функция будет равномерно распределять ключи по всему диапазону и число коллизий будет минимальным. По мере того как коэффициент заполнения таблицы приближается к 1, эффективность процесса заметно падает.

Проиллюстрируем линейный перебор на примере семи записей.

Предположим, что данные имеют тип DataRecord и хранятся в 11-элементной таблице.

```
 struct DataRecord
 {
   int key;
   int data;
 };
```

В качестве хеш-функции HF используется остаток от деления на 11, принимающий значения в диапазоне 0-10.

```
HF(item) = item.key % 11
```

В таблице хранятся следующие данные. Каждый элемент помечен числом проб, необходимых для его размещения в таблице.

```
Список: {54,1}, {77,3}, {94,5}, {89,7}, {14,8}, {45,2}, {76,9}
```

Хеширование первых пяти ключей дает пять различных индексов, по которым эти ключи запоминаются в таблице. Например, HF({54,1}) = 10, и этот элемент попадает в Table[10]. Первая коллизия возникает между ключами 89 и 45, так как оба они отображаются в индекс 1.

Элемент данных {89,7} идет первым в списке и занимает позицию Table[1]. При попытке записать {45,2} оказывается, что место Table[1] уже занято. Тогда начинается последовательный перебор ячеек таблицы с целью нахождения свободного места. В данном случае это Table[2]. На ключе 76 эффективность алгоритма сильно падает. Этот ключ хешируется в индекс 10 – место, уже занятое. В процессе перебора осуществляется просмотр еще пяти ячеек, прежде чем будет найдено свободное место в Table[4]. Общее число проб для размещения в таблице всех элементов списка равно 13, т.е. в среднем 1,9 проб на элемент.

Метод цепочек

При другом подходе к хешированию таблица рассматривается как массив связанных списков или деревьев. Каждый такой список называется блоком (bucket) и содержит записи, отображаемые хеш-функцией в один и тот же табличный адрес. Эта стратегия разрешения коллизий называется методом цепочек (chaining with separate lists).

Если таблица является массивом связанных списков, то элемент данных просто вставляется в соответствующий список в качестве нового узла. Чтобы обнаружить элемент данных, нужно применить хеш-функцию для определения нужного связанного списка и выполнить там последовательный поиск.

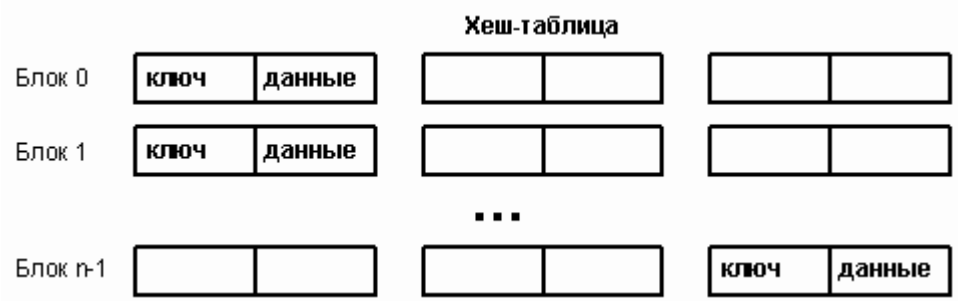


Рис. 29.

Проиллюстрируем метод цепочек на семи записях типа DataRecord и хеш-функции HF.

```
Список: {54,1}, {77,3}, {94,5}, {89,7}, {14,8}, {45,2}, {76,9}
HF(item) = item.key % 11
```

Каждый новый элемент данных вставляется в хвост соответствующего связанного списка. На рисунке 30 каждое значение данных сопровождается (в скобках) числом проб, требуемых для запоминания этого значения в таблице.

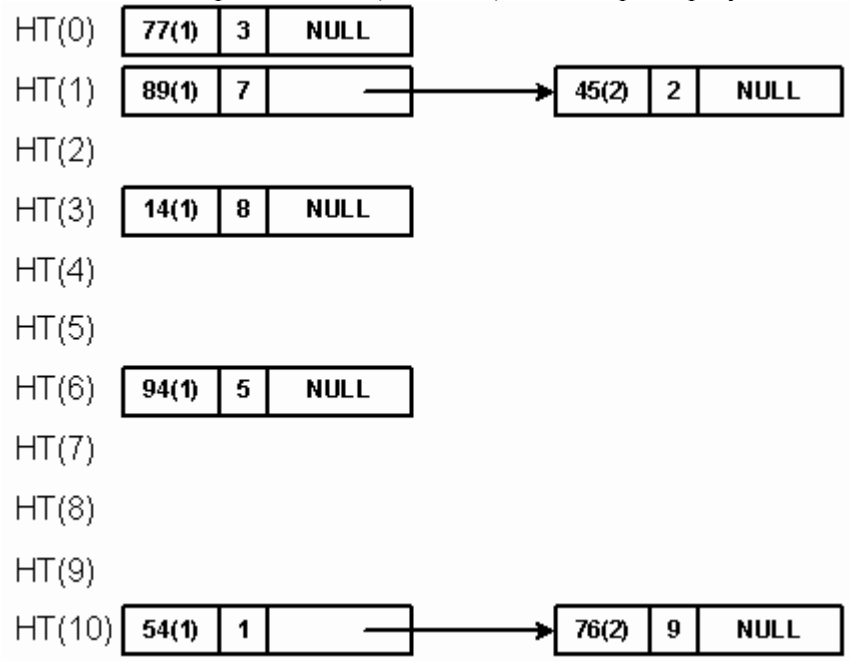


Рис. 30.

Заметьте, что если считать пробой вставку нового узла, то их общее число при вставке семи элементов равно 9, т.е. в среднем 1,3 пробы на элемент данных.

В общем случае метод цепочек быстрее открытой адресации, так как просматривает только те ключи, которые попадают в один и тот же табличный адрес. Кроме того, открытая адресация предполагает наличие таблицы

фиксированного размера, в то время как в методе цепочек элементы таблицы создаются динамически, а длина списка ограничена лишь количеством памяти. Основным недостатком метода цепочек являются дополнительные затраты памяти на поля указателей. В общем случае динамическая структура метода цепочек более предпочтительна для хеширования.

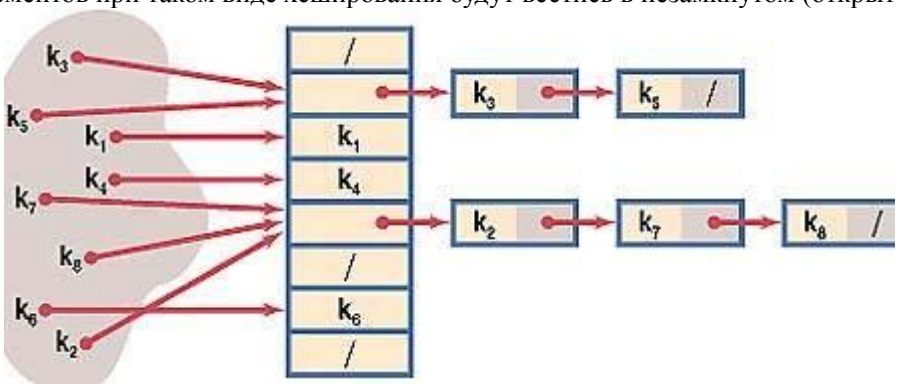
14. Хеш фции

Схемы хеширования

Традиционно принято выделять две схемы хеширования:

- хеширование с цепочками (со списками);
- хеширование с открытой адресацией.

В первом случае выбирается некая хеш-функция h(k) = i, где i трактуется как индекс в таблице списков t. Поскольку нельзя гарантировать, что не встретится двух разных ключей, которым соответствует один и тот же индекс i (конфликт, коллизия), такие «однородные» ключи просто помещаются в список, начинающийся в i-ячейке хеш-таблицы t (см. рисунок). Очевидно, что процесс заполнения хеш-таблицы будет достаточно простым, но при этом доступ к элементам потребует двух операций: вычисления индекса и поиска в соответствующем списке. Операции по занесению и поиску элементов при таком виде хеширования будут вестись в незамкнутом (открытом) пространстве памяти.



Отображение ключей путем хеширования с цепочками

Во втором случае все операции производятся в одном измерении, и таблица t является обычным одномерным массивом. Однако в этом случае подход к разрешению коллизий индексов иной: либо элементы с «однородными» ключами пытаются размещать в непосредственной близости от полученного индекса, либо осуществляют многократное хеширование (обычно двойное), когда для хорошего перемешивания последовательно применяется набор разных (взаимосвязанных) хеш-функций. Очевидно, что здесь и заполнение хеш-таблицы, и доступ к элементам будет весьма замысловатым. Хеш-адрес элемента с данным ключом как бы открыт: он постепенно уточняется. При этом все операции ведутся в замкнутом пространстве (в одномерном массиве).

Как нетрудно заметить, традиционная классификация методов хеширования может быть заменена другими: хеширование с цепочками можно относить по классификации Ахо, Хопкрофта и Ульмана [6] к так называемому «открытому» хешированию (в данной статье — к многомерному хешированию, поскольку при использовании списков речь идет о нескольких измерениях), а хеширование с открытой адресацией — к «закрытому» хешированию (одномерному).

Выбор хеш-функции

— непростая задача. Такая функция должна удовлетворять двум требованиям: предусматривать быстрое вычисление и минимизировать количество коллизий. Самый простой вид хеш-функции: h(k) = k MOD M, где k — ключ-число, M — размер хеш-таблицы (простое число), MOD — остаток от целочисленного деления. Если ключ k — составной (состоит из нескольких слов/символов x_1 ... x_s), можно воспользоваться идеей Дж. Картера и М. Вегмана (1977): $h(k) = (h_1(x_1) + h_2(x_2) + ... + h_s(x_s))$ MOD M. На практике для выбора хеш-функции применяются различные эвристические подходы, учитывающие специфику задач.

Вопросы вычисления и уточнения хеш-адреса с алгоритмической точки зрения являются ключевыми, и на них мы остановимся при разборе схем хеширования.

Одномерное хеширование

В одномерном хешировании (открытая адресация) можно выделить два основных метода:

- линейное исследование — В.Петерсон (1957), А.П.Ершов (1957);
- двойное хеширование — Гюи де Бальбин (1968), Г.Кнотт (1968), Белл, Каман (1970), Р.Брент (1973).

Идея метода линейного исследования состоит в том, чтобы в случае коллизии просматривать соседние ячейки таблицы размером M до тех пор, пока не будет найден искомый ключ k или же пустая позиция. Обычно просмотр ведется в виде последовательности проб: h(k), h(k)–1, h(k)–2, ..., 0, M–1, ... h(k)+1. Чтобы избежать эффекта скучивания, шаг просмотра можно выбирать не равным 1, а в виде числа, взаимно простого с M. Это приводит к идее квадратичного исследования. Здесь для i-й пробы $h(k,i) = (h(k) + c_1xi + c_2xi^2)$ MOD M.

В случае линейного исследования нужно быть крайне осторожным с реализацией функции удаления (Del): можно потерять другой ключ. По этой причине прибегают к приему логического удаления: ячейку в таблице помечают соответствующим признаком. Таким образом, ячейки становятся трех видов: пустые, занятые и удаленные. Стоит заметить, что среднее число проб при успешном поиске с помощью этого метода зависит не от порядка вставки

ключей, а только от числа ключей с конкретным хеш-адресом. Алгоритм хорошо работает в начале заполнения таблицы, но затем все чаще встречаются длинные серии проб. И все же он достаточно прост и эффективен: при заполнении таблицы на 90% для поиска элемента в среднем требуется около 5,5 пробы.

Двойное хеширование на этапе уточнения хеш-адреса использует не просмотр, а вычисление значения другой хеш-функции, т. е. применяет $h_1(k)$ и $h_2(k)$. Значения $h_1(k)$ должны опять-таки лежать в диапазоне 0...M–1, а вот функция $h_2(k)$ должна порождать значения от 1 до M–1, причем взаимно простые с M (если M — простое число, то $h_2(k)$ — любое в указанном диапазоне, а если $M = 2^p$, то $h_2(k)$ — нечетное). Если число занятых ячеек обозначить N, то среднее количество проб в этом алгоритме будет составлять (M+1) / (M–N+1).

Многомерное хеширование

Схема многомерного хеширования (метод цепочек, Ф. Уильямс, 1959) довольно проста: в случае возникновения коллизий после вычисления хеш-функции ключи с одним хеш-адресом соединяются в цепочку. Здесь приходится решать такую проблему, как обеспечение равномерности заполнения хеш-таблицы. К тому же было бы неплохо, если бы она оказалась достаточно сбалансированной, чтобы содержать предельно короткие цепочки. Интересно, что если таблица будет заполнена наполовину, среднее число проб при неудачном поиске составит 1,18. Если таблица будет заполнена полностью, то для нахождения элемента потребуется в среднем 1,8 пробы. Метод цепочек экономичен с точки зрения проб, но неэффективно расходует память. Интересный обзор эффективности методов вместе с демонстрационным Java-аплетом приведен в курсе канадского университета McGill по алгоритмам и структурам данных (*www.cs.mcgill.ca/~cs251/OldCourses/ 1997/topic12*).

Во всех наших рассуждениях следует, однако, иметь в виду, что среднее время, основанное на теории вероятностей, может значительно отличаться от реального в каждом конкретном случае. Так что несмотря на стройные математические выкладки, жизнь вносит свои существенные коррективы. Современные процессоры работают при тактовой частоте порядка 1 ГГц, соответственно каждый такт занимает 1 нс. Время доступа к оперативной памяти составляет 7—10 нс. Отсюда следует, что на каждое обращение к памяти времени требуется в 7—10 раз больше, чем на обработку инструкции процессора. При наличии кэш-памяти проблема частично решается, однако в общем случае такой разрыв на порядок будет сохраняться. Для хеширования желательно, чтобы элементы хранились как можно более компактно и ближе друг к другу, а это возможно как раз при одномерном хешировании.

Области применения и другие методы

Одно из побочных применений хеширования состоит в том, что оно создает своего рода слепок, «отпечаток пальца» для сообщения, текстовой строки, области памяти и т. п. Такой «отпечаток пальца» может стремиться как к «уникальности», так и к «похожести» (яркий пример слепка — контрольная сумма CRC). В этом качестве одной из важнейших областей применения является криптография. Здесь требования к хеш-функциям имеют свои особенности. Помимо скорости вычисления хеш-функции требуется значительно осложнить восстановление сообщения (ключа) по хеш-адресу. Соответственно необходимо затруднить нахождение другого сообщения с тем же хеш-адресом. При построении хеш-функции однонаправленного характера обычно используют функцию сжатия (выдает значение длины n при входных данных больше длины m и работает с несколькими входными блоками). При хешировании учитывается длина сообщения, чтобы исключить проблему появления одинаковых хеш-адресов для сообщений разной длины. Наибольшую известность имеют следующие хеш-функции [7]: MD4, MD5, RIPEMD-128 (128 бит), RIPEMD-160, SHA (160 бит). В российском стандарте цифровой подписи используется разработанная отечественными криптографами хеш-функция (256 бит) стандарта ГОСТ Р 34.11—94.

Хеширование можно рассматривать и как расстановку, и как снятие отпечатков, и как раскрашивание. В самом деле, то, что каждому ключу ставится в соответствие целое число, дает основание говорить о хеш-адресе как о хеш-краске.

Многомерное хеширование отражает известный принцип Дирихле: при любом размещении (n+1) предметов по «n» ящикам всегда найдется ящик с двумя предметами. В случае, если нас интересует наличие k предметов в одном ящике, он формулируется так: при любом размещении (rxk – r + 1) предметов по r ящикам найдется k предметов в одном ящике. Несмотря на свою тривиальность, принцип Дирихле весьма полезен — на нем основаны многие теоремы математики фундаментального характера (теоремы Матиясевича, Дирихле, Эйлера – Ферма, Рамсея). Если большая структура разбивается на непересекающиеся части, то наличие какой подструктуры можно гарантировать в одной из частей? И обратная задача: сколь богатой должна быть большая структура, чтобы любое ее разбиение содержало часть предписанной природы? Здесь разбиение множества ключей на цепочки («ящики») приводит к смежным задачам, в частности, к теории Рамсея — специальной ветви комбинаторики, которая имеет дело со структурами, сохраняющимися под действием разбиений.

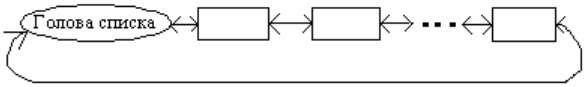
16. . Побудова таблиць у вигляді списків

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

Классический пример структуры данных последовательного доступа, в которой можно удалять и добавлять элементы в середине структуры, — это линейный список. Различают однонаправленный и двунаправленный списки (иногда говорят односвязный и двусвязный).

Элемены списка как бы выстроены в цепочку друг за другом. У списка есть начало и конец. Имеется также указатель списка, который располагается между элементами. Если мысленно вообразить, что соседние элементы списка связаны между собой веревкой, то указатель — это ленточка, которая вешается на веревку. В любой момент времени в списке доступны лишь два элемента — элементы до указателя и за указателем.

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



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

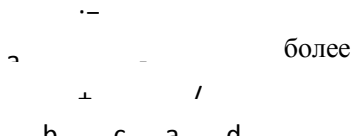
Графи внутрішнього подання (17-22)

17. Графи та їх використання для внутрішнього подання

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

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

a := b + c - a / d.



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

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

Транслятори мов програмування (23-47)

23. Граматики та їх застосування

Грамматикой называется четверка G = (Vt, Vn, R, e є Vn), где Vn - конечное множество нетерминальных символов (все обозначения, кот. определяются через правила), Vt - множество терминалов (не пересекающихся с Vn), e - символ из Vn, называемый начальным/конечным, R - конечное подмножество множества: (Vn □ Vt)* Vn (Vn □ Vt)* x (Vn □ Vt)*, называемое множеством правил. Множество правил R описывает процесс порождения цепочек языка. Элемент ri = (□, □) множества R называется правилом (продукцией) и записывается в виде □□□□□. Здесь □□и □ - цепочки, состоящие из терминалов и нетерминалов. Данная запись может читаться одним из следующих способов:

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

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

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

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

Порождающие грамматики

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

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

Применение

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

Терминальный алфавит:

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

Нетерминальный алфавит:

{ ФОРМУЛА, ЗНАК, ЧИСЛО, ЦИФРА }

Правила:

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

Начальный нетерминал:

ФОРМУЛА

Аналитические грамматики

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

24. Класифікація граматик за Хомським

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

тип 0. неограниченные грамматики — возможны любые правила. Грамматика G = (VT, VN, P, S) называется грамматикой типа 0, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики).

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

Грамматика G = (VT, VN, P, S) называется неукорачивающей грамматикой, если каждое правило из P имеет вид □ □ □, где □ □ (VT □ VN)+, □ □ (VT □ VN)+ и | □ | □ | □ |.

Грамматика G = (VT, VN, P, S) называется контекстно-зависимой (КЗ), если каждое правило из P имеет вид □ □ □, где □ = □1A□2; □ = □1□□2; A □ VN; □ □ (VT □ VN)+; □1,□2 □ (VT □ VN)*.

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

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

тип 2. контекстно-свободные грамматики — левая часть состоит из одного нетерминала. Грамматика G = (VT, VN, P, S) называется контекстно-свободной (КС), если каждое правило из P имеет вид A □ □, где A □ VN, □ □ (VT □ VN)+.

Грамматика G = (VT, VN, P, S) называется укорачивающей контекстно-свободной (УКС), если каждое правило из P имеет вид A □ □, где A □ VN, □ □ (VT □ VN)*.

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

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

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

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

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

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

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

G2 = ({S,}, {0,1}, P, S), где P:

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

контекстно-свободной (КС) или бесконтекстной, если каждое правило из P имеет вид: A□□, где A □ N, a □ □□(N □ T)*, то есть является цепочкой, состоящей из множества терминалов и нетерминалов, возможно пустой; Данная грамматика порождает простейшие арифметические выражения.

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

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

- контекстно-зависимой или неукорачивающей, если каждое правило из P имеет вид: □ □ □, где □□□□□□□□□□ То есть, вновь порождаемые цепочки не могут быть короче, чем исходные, а, значит, и пустыми (другие ограничения отсутствуют);

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

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

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

Примечание1.Согласно определению каждая правол-ая грамматика есть контекстно-свободной.

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

25. Побудова графів для правил підстановки в різних формах Бекуса

Побудова графів для правил підстановки в різних формах Бекуса

Мова правил Бекуса, як і інші комп'ютерні мови, чітко розбивається на різні лози- або лексеми. Для відображення правил на цій мові головними є термінальні та нетермінальні лози, а також операції, які вик. при запис. правил.

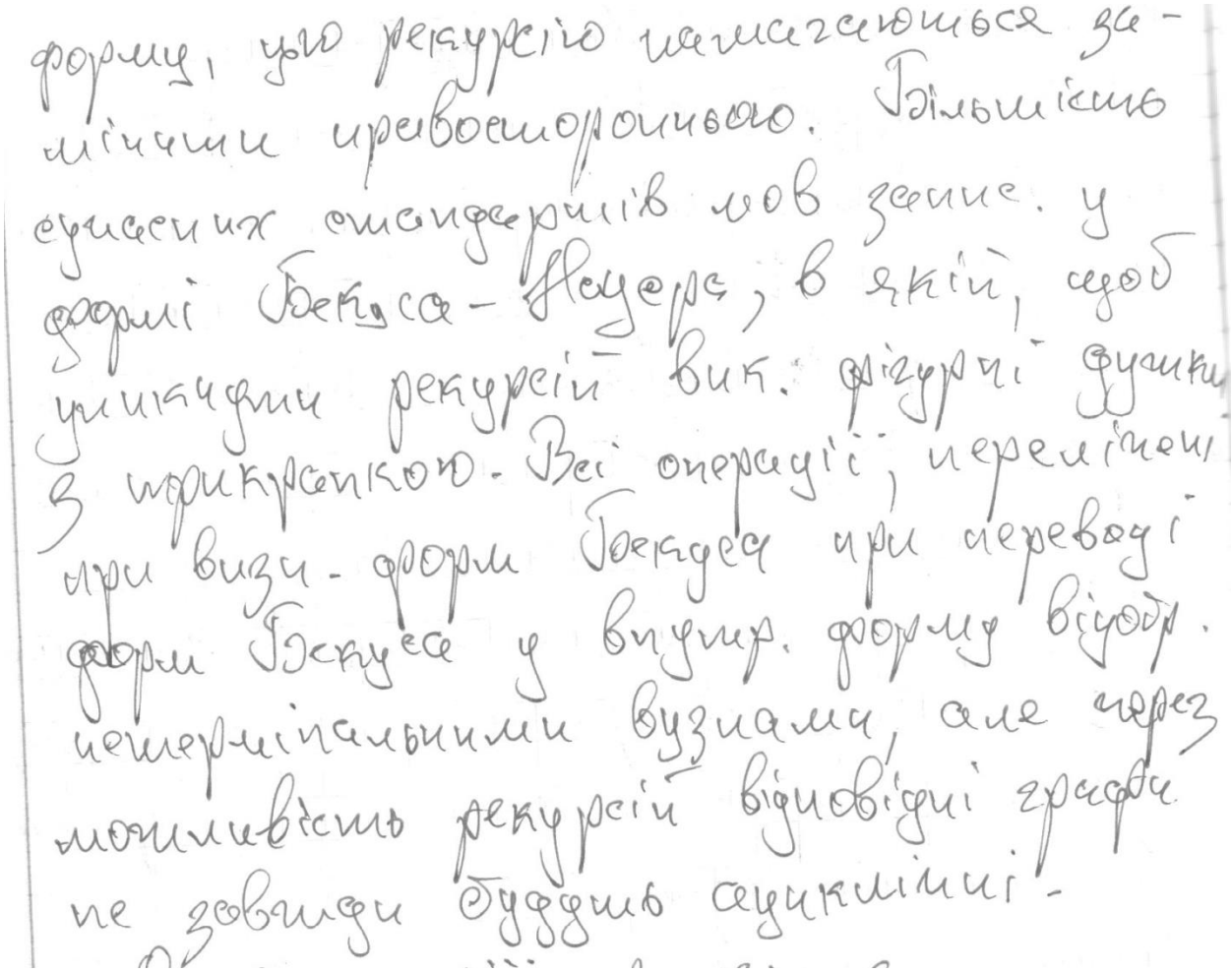
::= операція визначення, запис. в заголовку і відділяє ліву част. від правої:

| – операція альтернативи.

[ ] – факультатив, необов'язкова конструкція.

{ } – показ повторювання конструкцій, або показ проміжного списку альтернатив.

В нормальній формі Бекуса повторювані елементи констр. записуються рекурсивними правилами. Якщо ліва частина повт. зліва списку альтернатив правила, то таку рекурсію наз. лівосторонньою, а якщо в правій альтернативі – то правосторонньою. В більшості стандартних описів, зроблених в нормальній формі Бекуса вик. лівостороння рекурсія, але при перетв. цих правил на частину



26. Задача лексичного аналізу

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

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

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

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

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

27. Граматики для лексичного аналізу

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

Для решения задачи ЛА могут использоваться разные подходы, один из них основан на теории грамматик. К этой задаче можно подойти через классификацию лексем как элементов или типов входных данных. В качестве лексем входного языка обычно объявляют разделители, ключевые слова, имена пользователя, операторы, константы и спец. лексемы.

Чаще всего удобно построить спец. классификационную таблицу. Входной текст может быть в ASCII (каждый символ 1 байт), в UNICODE (2 байта) и др. Основные классы символов:

1) Вспомогательные разделители (пробел, таб., enter, ...);
2) Одно-символьные операции (+, -, *, /, ..., (,));
3) Много символьные операции (>=, <=, <>...);
4) Буквы, которые можно использовать в именах (латиница):
5) Не классифицируемые символы (для представления чисел или констант необходимо определить цифры и признаки основных систем счисления).

При формировании внутреннего представления, кроме кода желательно формировать информацию о приоритете или значении предшествующих операторов.

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

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

28. Трансляція шляхом граматичного аналізу

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

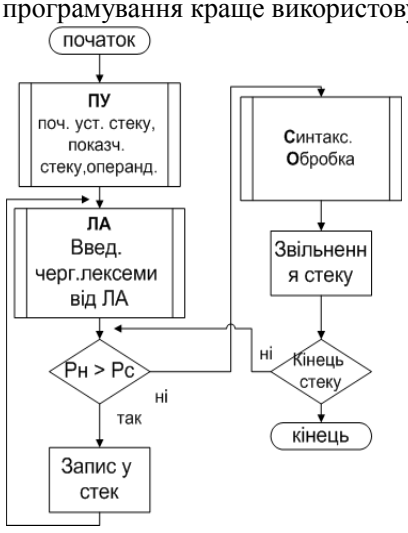
Методы грамматического разбора можно разбить на 2 больших класса - восходящие и нисходящие - в соответствии с порядком построения дерева грамматического разбора. Нисходящие методы (методы сверху вниз) начинают с правила грамматики, определяющего конечную цель анализа с корня дерева грамматического разбора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

29. Задача синтаксичного аналізу

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

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

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



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

Последовательный анализ лексем и сравнение приоритетов.

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

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

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

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

30. Граматики для синтаксичного аналізу

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

генерации текста результирующей программы.

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

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

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

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

31. Методи висхідного розбору при синтаксичному аналізі

Методы восходящего анализа:

1. Простое предшествование

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

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

2. Свертка-перенос

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

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

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

Методи висхідного синтаксичного аналізу.

Алгоритм должен содержать в себе блоки обращения к лексическому анализу для получения лексем, и возвращения к семантической обработке. Для того чтобы алгоритм имел более общую форму, в начальных установках алгоритма целесообразно сформировать фиксированный

элемент с низким приоритетом фиктивной операции, который будет служить признаком конца обработки выражения. Ображения к процедуре, в результате можно получить или одну лексему или пару (операнд и операция). Сравнение приоритетов текущей операции с предыдущей. Если новой выше, то ее рядом с операндом следует запомнить в стэке и снова обратится к выражению, если нет, то к программе семантической обработки с тем, чтобы получить со стэка предыдущую операцию и выполнить необходимые действия.
Программы с таким алгоритмом можно записать в трех вариантах :
1) программа которая использует специальный стэк в виде массива;
2) рекурсивная процедура на языке высшего уровня, которая использует системный стэк для сохранения лексем как аргументов рекурсивных вызовов и адресов возврата в программу;
3) циклическую программу на языке асм., которая будет использовать в середине процедуры только для сохранения промежуточных значений.
При восходящем разборе дерево начинает строиться от терминальных листьев путем подстановки правил, применимых к входной цепочке, опять таки, в общем случае, в произвольном порядке. На следующем шаге новые узлы полученных поддеревьев используются как листья во вновь применяемых правилах. Процесс построения дерева разбора завершается, когда все символы входной цепочки будут являться листьями дерева, корнем которого окажется начальный нетерминал. Если, в результате полного перебора всех возможных правил, мы не сможем построить требуемое дерево разбора, то рассматриваемая входная цепочка не принадлежит данному языку.Восходящий разбор также непосредственно связан с любым возможным выводом цепочки из начального нетерминала. Однако, эта связь, по сравнению с нисходящим разбором, реализуется с точностью до "наоборот". На рис. приведены примеры построения деревьев разбора для грамматики G6 и процессов порождения цепочек, представленных выражениями. В общем случае для восходящего разбора строится так называемые грамматики предшествования. В грамматике предшествования строится матрица попарных отношений всех терминальных и не терминальных символов. При этом определяется три вида отношений:
R предшествует S (R< S);
S предшествует R•
(R >S);
и операция с• Одинаковым предшествованием (R =S);•
четвертый вариант отношение предшествия отсутствует (это говорит о недопустимости использования R и S рядом в тексте записи на этом языке)
(з конспекту)
Найпростіший алгоритм висхідного розбору для аналізу математичних виразів. На основі пріоритетів окремих операцій. Порівнюємо пріоритет наступної з пріоритетом попередньої.
Незалежно від результатів порівняння цього пріоритету треба обрати варіант подальшої роботи - якшо має низький пріоритет - відкласти обробку операції записуючи її в стек. Якщо пріоритет більш високий - виконати інтерпретацію чи саму операцію. Результат такого розбору буде дерево розбору (дерево підлеглості)

32. Матриці передувань

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

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

При этом определяется три вида отношений: R предшествует S (R<•S); S предшествует R (R•>S); и операция с одинаковым предшествованием (R•=S); четвертый вариант отношение предшествия отсутствует (это говорит о недопустимости использования R и S рядом в тексте записи на этом языке).

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

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

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

33. Нисхідний розбір

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

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

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

Отсюда возникает проблема зацикливания при обработке леворекурсивных правил.

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

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

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

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

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

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

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

Условия применения:

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

или , где , и это единственное правило вывода для этого нетерминала

или для всех

34. Метод рекурсивного спуску

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

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

Идея метода

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

1.Находит наибольшее начало z слова x, способное быть началом выводимого из K слова

2.Определяет, является ли начало z выводимым из K

Такая функция должна удовлетворять следующим критериям:

1.считывать из еще необработанного входного потока максимальное начало A, являющегося началом некоторого слова, выводимого из K

2.определять является ли A выводимым из K или просто невыводимым началом выводимого из K слова

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

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

Условия применения

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

-или $A \to \alpha$, где $\alpha \subset (\Sigma \cup N)*$, и это единственное правило вывода для этого нетерминала

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

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

35. Метод синтаксичних графів.

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

• мітка, що призначена для зв’ язування вузлів графа

• розпізнавач, який може розпізнати термінальні або не термінальні елементи синтаксичної конструкції для синтаксичного аналізу в компіляторах. За термінальні позначення обирають імена, константи, ключові слова, тощо. За нетермнальні – вирази, списки, оператори, тощо.

• покажчик на наступний вузол графу, до якого відбувається перехід у випадку успішної роботи розпізновача

• показник на альтернативне відгалуження синтаксичного графу, коли робота розпізнувача не буде успішною. По формі структур даних синтаксичний граф може бути представлений як сукупність вузлів з двійковим розг. У випадку коли синтаксична конструкція виявиться неправильною необхідно передбачити нейтралізацію помилок, тобто видачу діагностики з пошуком конструкції після якої можна продовжити синтаксичний аналіз

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

1.Ідентифікація або мітка вузла

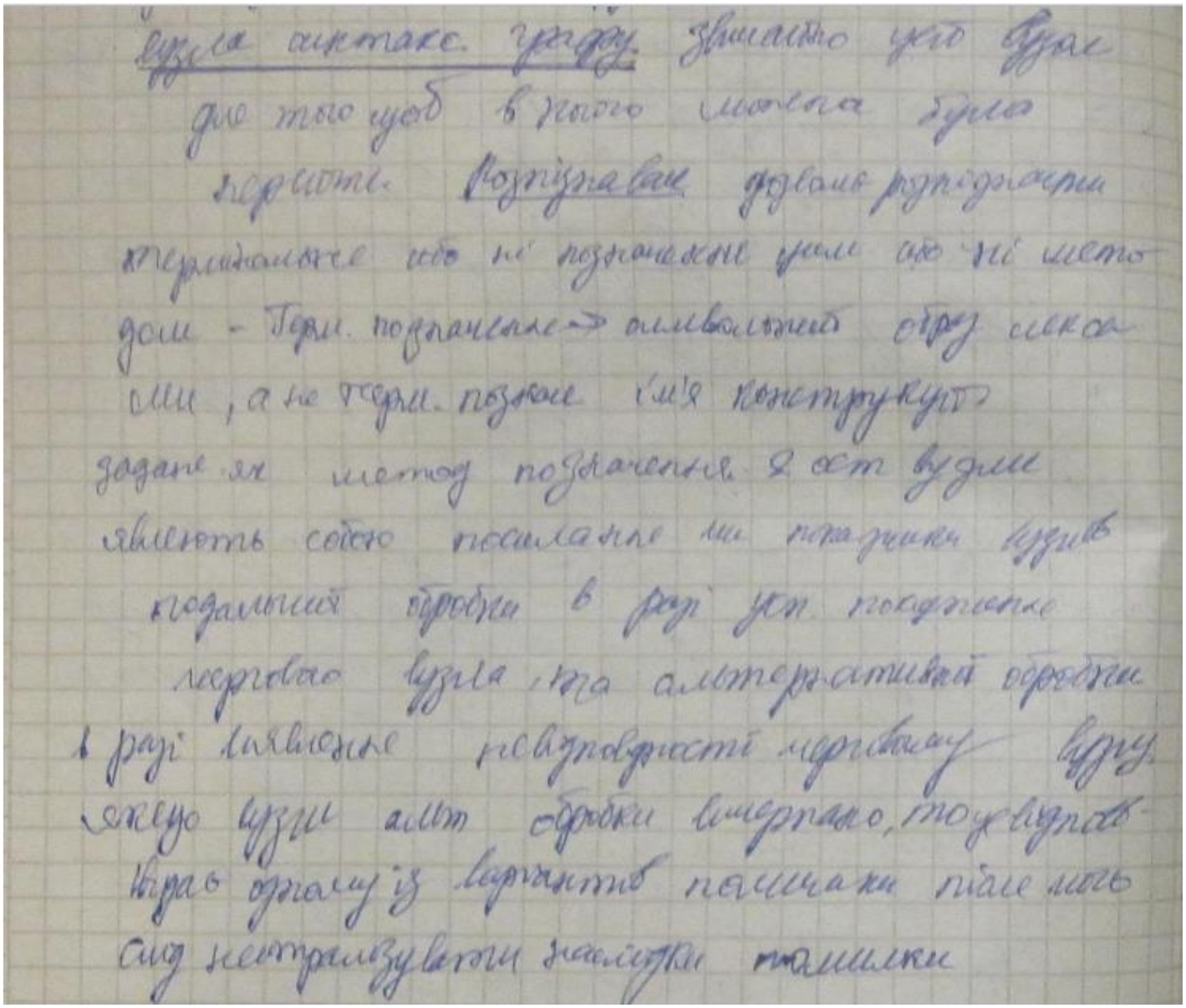
2.Прототип співставлення (термінал або нетермінал)

3.Мітка продовження обробки (при успішному результаті синтаксичного аналізу)

4.Вказівник на альтернативну вітку, яку можна перевірити (якщо аналіз був невдачним)

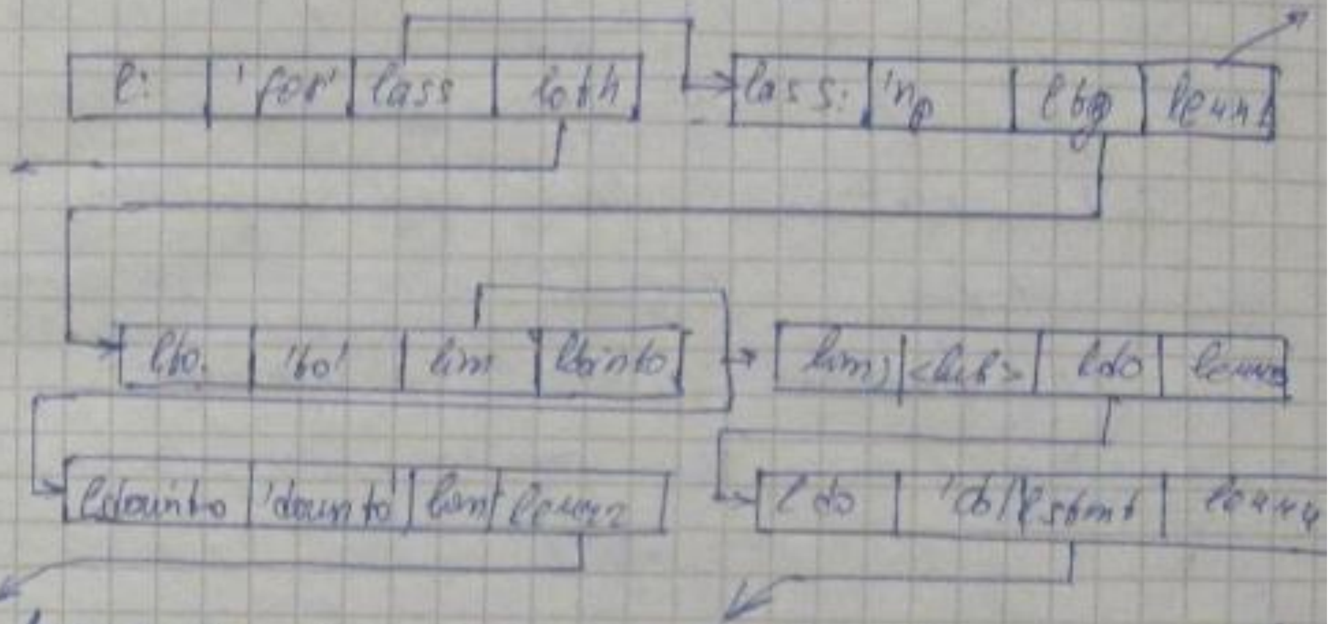
Синтаксично буває зручним використовувати змішані алгоритми аналізу. Як раз для обробки операторів мов програмування краще використовувати синтаксичний граф, а для обернених виразів – висхідний розбір.

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



Методи низхідного розбору єфективніші при обробці структурованих операторів(for) :

Приклад побудови синтаксичного графу для оператора циклу for в мові Pascal

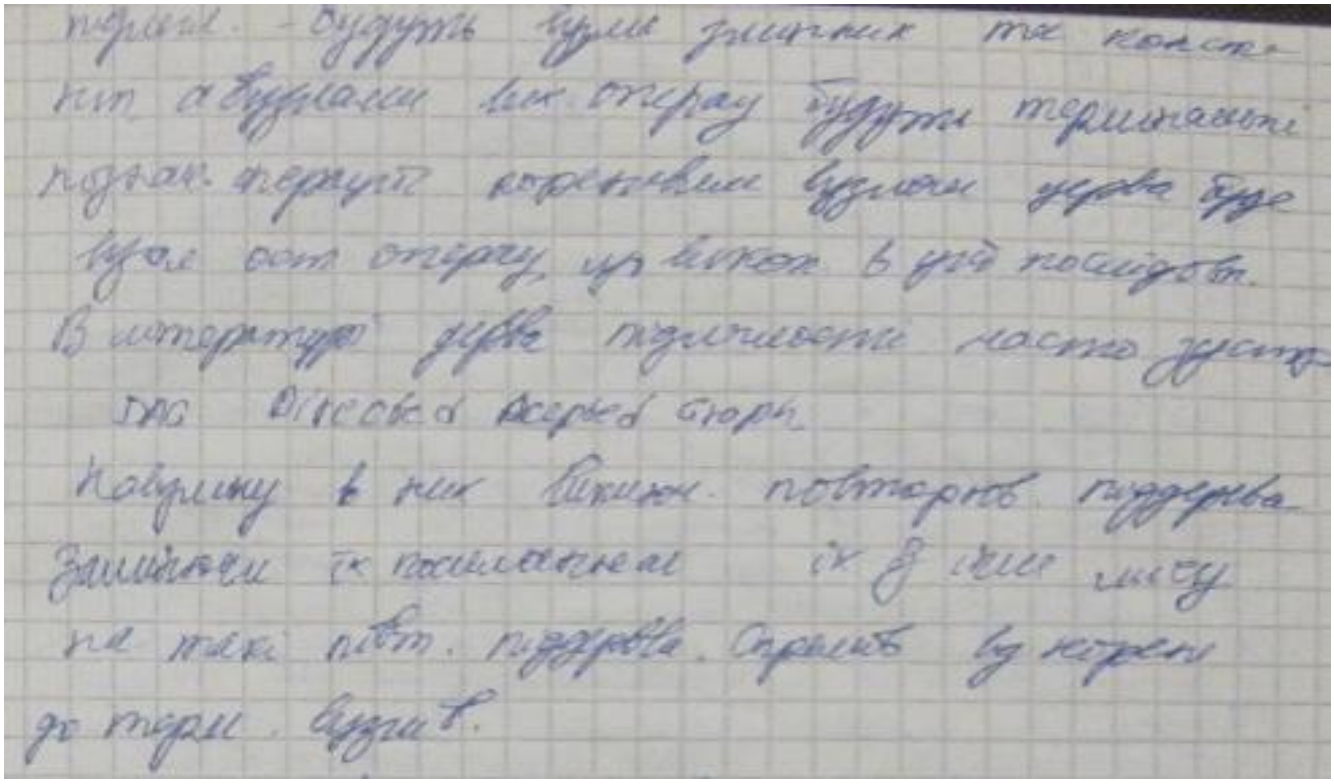


Використ. синтакс. графу передбачає можливість помилок. Використ алгоритмів висхідного та низхідного розбору, тобто програма за правила ли цим тут виконується за правилами низхідного розбору, а аналіз нетермінальних розпізнавачів, або виразу або умовного виконув. методами висх. розбору

Для формування результатів синтакс аналізу використов. 2 види інформації, формув. → для формування діагностики

про коректн. тексту виходят правила
→ підстановки

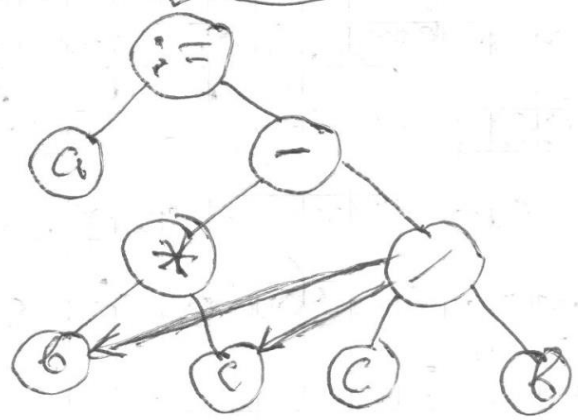
Пашкров: в ситуац. коли для вх. тексту мова не існує правил, що підтверджу. належність вхідн. фрази до мови. Друга сторона синт. аналізу полягає в побудові деревоподібних результатів аналізу. В результ. аналізу за правилами підстановки можна сформулювати дерева розбору в яких, для кожного нетермінального вузла визначається розгалуження на позиц... термін. і нетерм. вузлів наст. рівня. Для підлишених нетерм. вузлів, в дерев. розбору, повинні бути побуд. під дерева, що склад. із нетерм. і терм. вузлів. Однак для подальш. семант. обробки дерев зручніше будувати дерева підлеглості операцій. У відповідності зі значеннями передбачено вх переходу від дерев розбору до дерев підлеглості, нетермінальні позначення заміщувати піддеревами. Таким чином в результ. таблиці вузлами дерев



Граф (дерево) підлеглості операцій

В цьому дереві проміжними вершинами є вузли мовних операцій, що вик. для одержання проміжних результатів, а кінцевими вершинами є вузли імен або константи у виразах.

a := b * c - e / b ОПБ



Такі дерева і графи зручні для всіх видів семантичної або змістовної обробки, мовно семантичного аналізу, інтерпретації, пошук. коду, оптимізації і генерації об'єктних кодів.

Особливості побудови дерев для обробки складних операторів процедурних мов програмування

Звич. при вик. синтаксичного аналізу уже. дерева розбору, вузли яких предст. собою нетермінальні позн., за винятком термінальних вузлів, які є термін. по значенню. Правило визн. нетермін.-означень беруться з форм Бекуса, із них формують дерева при деталізації або розборі відповідних нетерміналів. Тобто якщо якесь речення класифікувала як нетерм. що в проц. доведення його коректності ми повинні вивести ієрархію нетермінальних нетерміналів, що в кінці будуть завершуватись терм. вузлами з терм. позн. мова.

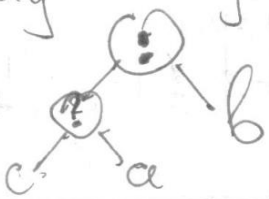
<вир. присв> ::= <l value> =<вираз>

Частіше за все в літературі вихід синт. семантичного описується так, що має вигляд дерева розбору. Один з мінусів одерж. такого дерева — вик. правил форми Бекуса. Однак такі дерева незручні для змістовної обробки. Тому краще перед семантичною обробкою «вирощувати» syntax tree → Directed Acyclic Graph (DAG). Однак цю форму доцільно розширити вузлами, що міст. ключовими словами складних виразів мов програмування. Заг. правило для таких вузлів — що ближче до кореню лежить вузол, тим вик. операція.

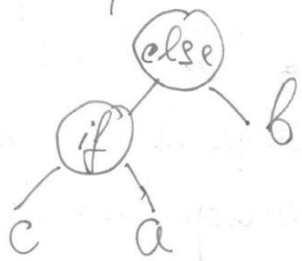
Для умовних виразів

c ? a : b

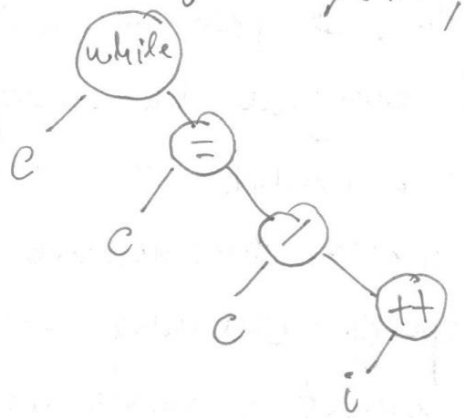
Якщо вираз в не дор. нулю, то як результат повертається вираз a, в іншому випадку повертається c.



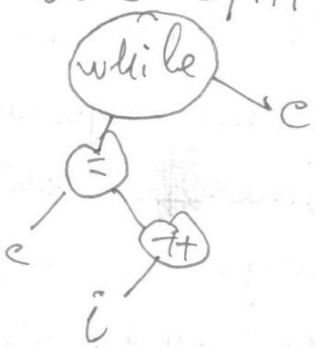
Це аналогічно оператор if else :
if (c) a; else b;



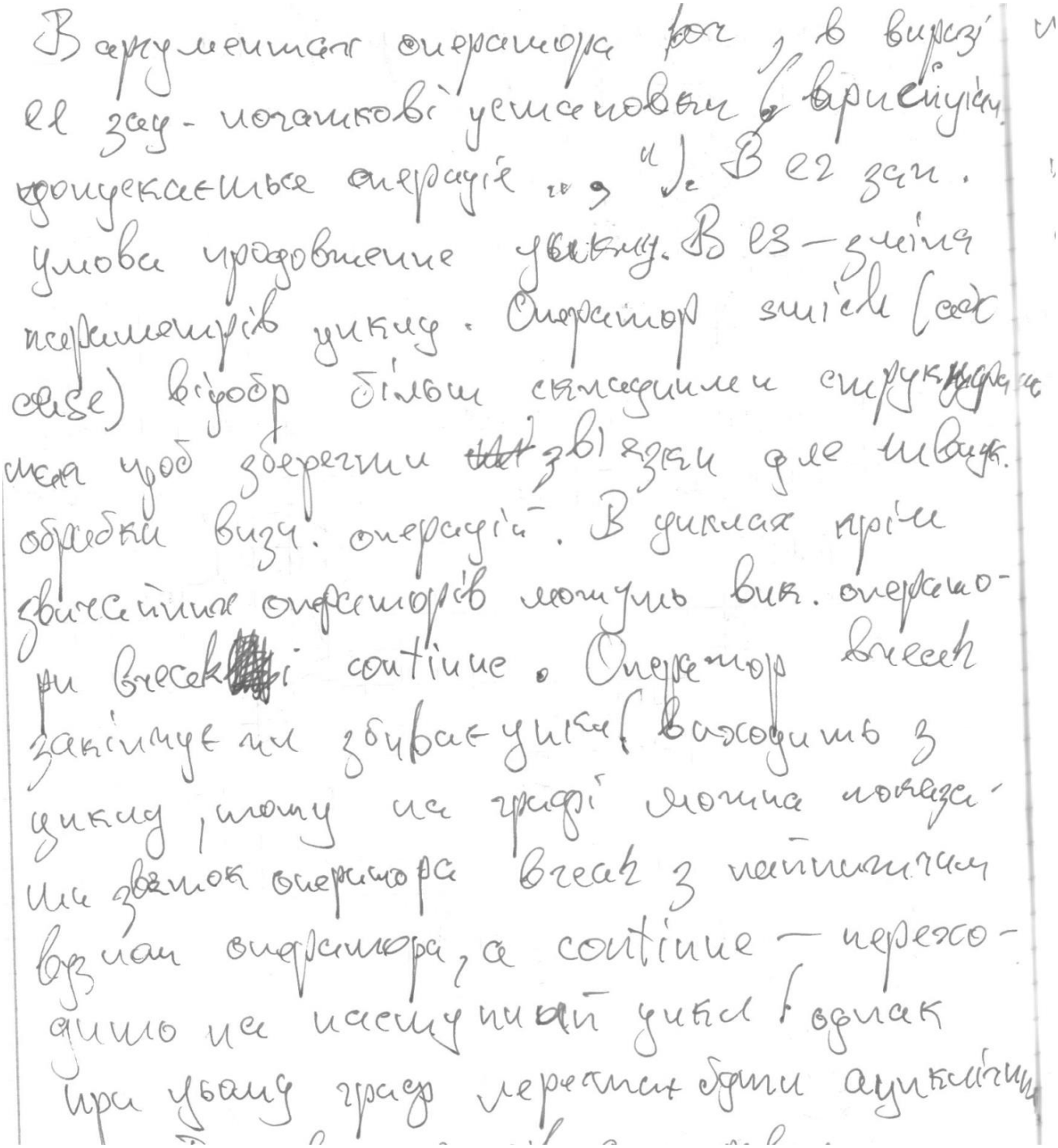
Для циклів передумова
while(c) c = c/c++;



постумова
do c=c/i++; while(c);



Цикли з вик. індексів
for



37. Задача семантичної обробки

Семантическая обработка:

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

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

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

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

1. интепретация – сборка конст. выражений. В случае реализации языка программ. в виде интерпретатора, данные для обработки получаются из констант и результатов операций ввода.
2. машинно-независимая оптимизация. Основные действия должны сократить объёмы графов внутр.

представления путем удаления повтор. И неисполз. фрагментов графа. Кроме этого, могут быть выполнены действия упрощ.экви превращений. Этап требует сохранения доп. информации.

3. генерация объектных кодов выполняется: в языке высокого уровня; на уровне команд асма. Для каждого узла дерева генерируется соотвеств. последовательность команд по спец. шаблонам, а потом испол-ся трансляция асм или языка, кот. включает асм-вставки.

4. машинно-зависимая опт-ция учитывает архи и специфику команд целевого устройства

Первичная семантическая обработка в процессе синт. анализа

Это построение деревоподобных структур или графа подчиненности операций.

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

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

38. Загальний підхід до ораганізації семантичної обробки в трансляторах

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

При аналізі задач оптимізації можна виділити семантичний аналіз у вигляді термінальних вузлів, або у вигляді обробки нетерм. Вузлів доцільно використю табл. Відповідності аргументів чи операндів разом з полями визначених вихідних типів. Аргументами пошукутакої табл. Буде: внутр. Код(подання операції), тип і довжина 1-гота 2-го операнду. Функціональними полями в табл. Повинні бути тип і спосіб визначення довжини результату. Спосіб обчислення довжини може обчислюватись за допомогою якоїсь процедури обчислення довжини. Кожен алгоритм обробки використовується і для інших видів семант. Обробки, тобто використ. Та сама рекурсія, але використ. Інші проц. Для семантичного аналізу програм може бути використана 1 табл. Відповідностей,в якій серед функц. Полів будують тип результату і функції визначення довжини результатута ф-ції інтерпретацій визначення операцій. Для реалізації повного інтерпретатора мови необх. Визначити в табл. Відповідності рядки для кожної з операцій, або фрагмента оператора мови. Найчастіше як внутр. Подання програми використовувались формат польського інверсного запису для виразу або постфіксна форма подання виразу. Деревовидна форма подання , в якій кореневі вузли піддерев визначають обчислення, що виконуються в підпідлеглих структурах. Різновидом таких дерев можна вважати спрямовані ациклічні графи(DAG). Звичайно такі графи мінімізують деревовидні подання, замінюючи повторювані піддерева відповідними посиланнями. В більш ранніх компіляторах часто використовували тетрадні та тріадні подання, які складалися з команд(послідовності команд) віртуальної машини реалізації мови. Тріадні подання включали код операції та адреси або вказівники на 2 операнди.В тетрадних поданнях використовувались 3 окремі посилання: 2-на операнди, а 3-й на результат, тобто вони відповідали 2-х та 3-х адресним командам комп'ютерів попередніх поколінь. Такі форми є залежними від обраної архітектури віртуальних машин. Внутрішні подання Паскалю у формі Р-кодів ближче до 1-адресної машини, а початкова форма більш спрямована до циклічної є взагалі архітектуронезал., атому більш загальною.

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

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

39. Організація семантичного аналізу

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

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

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

аргумента. Функц. часть – тип результата.

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

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

Этапы семантического анализа

Семантический анализатор выполняет следующие основные действия:

проверка соблюдения семантических соглашений входного языка;

дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка;

проверка элементарных семантических норм языков прогр., напрямую не связанных с входным языком.

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

Примерами соглашении являются следующие требования:

каждая метка, на которую есть ссылка, должна один раз присутствовать в программе;

каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;

при вызове функции число фактических параметров и их типы должны соответствовать числу и типам формальных параметров;

обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке; на типы левой и правой частей в операторе присваивания; на тип параметра цикла; на тип условия в операторах цикла и условном операторе и т.п.

40. Організація інтерпретації вхідної мови

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

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

Структура для управління доступом до даних в інтерпретаторі

Ім’я даного

Адреса розміщення

Довжина

Тип

Блок визначення

Структура для управління доступом до виконавчих кодів в інтерпретаторі

Операція

Ім’я підпрограми

Адреса розміщення

Тип результату

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

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

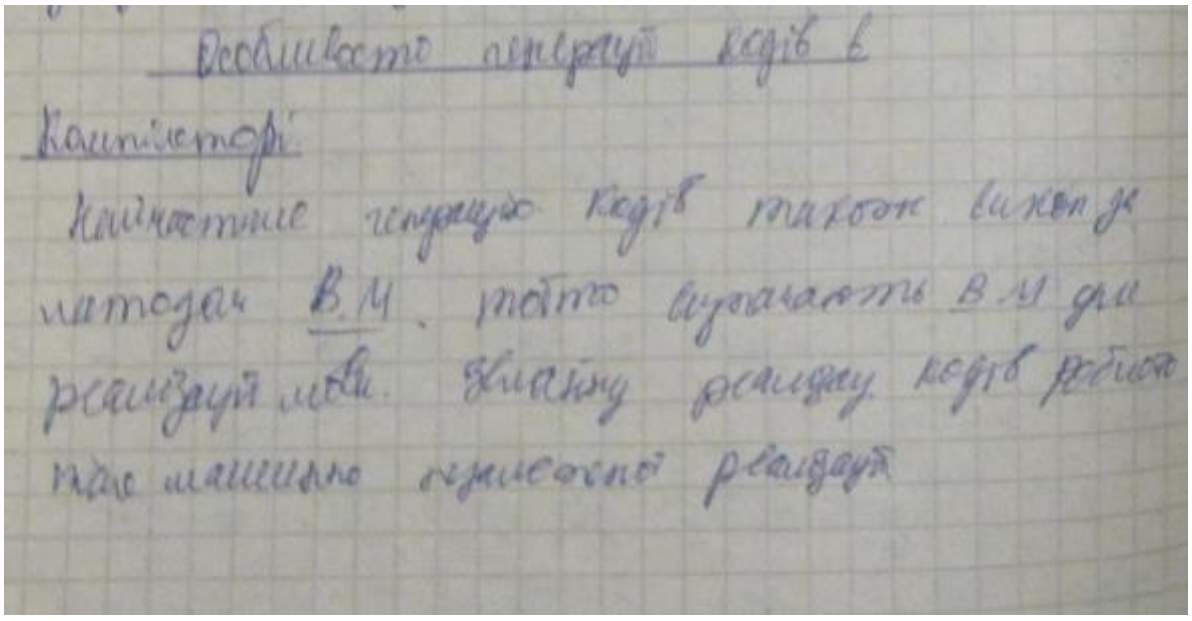
41.-44 Організація генерації кодів

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

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

-Записи (елементи) переміщуванності – спосіб настроювання відповідної відносної

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



Для викон. будучих кодів семант. обробки компілятор необхідно використов. таблицю відповідності типів. Вона визначає відповідність типів для семант. аналізу та розширюється для інтерпретації попатсиками в вбудовані підпрограми.

А при генерац. кодів попатсиками на шаблони кодів, які компіляторами. Шаблони кодів генеруються задав на мові Assembler або безпосередньо у двійкових кодах. Звичайно аргументи операції повинні перетворюватись на параметри кодів, що формуються, при виконанні операції над тими аргументами, що так в ОП [illegible] спос. завантаження першого аргумент операц., найчастіше завантаження відбув. до акумулятора в процесорі

завантаження відбув. до акумулятора в процесорі

до вершини стеку. А потім викон. операц. над акумулятором і з тим операндом команду фактично крім таких операцій треба могти викон. операції над даними, що запис. в акумуляторі та стеку результат операції [illegible] в акумул.

а потім науки до необхідності до стеку, або до адресованої пам'яті. При роботі регістрах з плав. точкою заватажу нових аргументів заміщає попередні результ в пам'яті. А виконання операції над вершиною стеку і даними стеку - можна звільнити вершину стеку. Через це використовувати коди для пристрою з плаваючою точкою доцільно використовувати для регістрів звільнить пристрій

Якщо для викон. операції необхідно здійснити перетвор даних до більш загальної форми, - слід використовувати спец операції

- для цілих даних: cbw cwd

з двобайтного слова на подвійне слово

для перетворення даних із цілого формату на формат із плав точкою цілого використовувати команди завантаження цілих даних до регістрів із плаваючою точкою - команда F [illegible] FILD

Ця команда перетворює ціле дане у формат [illegible] четь [illegible] на формат з плаваючою точкою. При [illegible] вивантаженні до [illegible] із плаваючою точкою

При присвоєнні формату результату якщо в більшості випадків може виникать потрібні перетвор. для формат сприймання.

Дані із плаваючою точкою fist- перетворення цілі.

Точності і округлення в процесах з плав. точкою задаються командами стандарту оточуючого в середовищі програми.

Точність може бути визначена для роботи з 4-ми 8-ми і 10-ти байтними даними.

В реальних пристроях різниця в цій дозі істотна. Округлення звичайно робиться на рівні одиниці наступного. При реалізації індексних операцій значення завантажуються до рег. заг. приз.

де для розборів даних 1, 2, 4, 8 – визначено автоматичне множення індексу, якщо довж. елемента не вдпов. вел. ступеню. дії, то може на довж. елем. мав використ. перед використ. індексу окремими командами.

Інша механізм викор. при розробці кодів.

28.10.10 лекція №8

Для перетвор. даних із цілих форм. у на форм. у плаваючого точкого числа використов. регістри у плаваючою точкою при додат. цілих даних із плав. точкою команда fild перетвор. цілі дані в 4 або 8 байтному форматі на дані із плаваючою точкою. Зворотня команда fist записує дані із регістру у плав. точку в цілі дані в пам'ять.

Для виконання функціональних викликів генератор коду формує послідовність викликів, яка оформлюється у відповідності стрімкими у мові програм. угодами про зв'язки. В мові С існує можливість передачі різної кількості аргументів для деяких ф-цій. (форматне введення виведення) Через це

ф-цій. (форматне введення виведення) Через це

зручно організувати запис аргументів таким чином щоб вони розташов. в послід. порядку в стеку за адресою повернення. тому в мові С аргументи до стеку запис. в порядку із остан. аргументу і закінчуючи першим. Після цього до стеку (Call) записується повернення

і таким чином.

```
call
add esp, <зн зн>
```

Програма на мові паскаль передбачає розміщення кількості [illegible] тому послідовність запису аргументів до стеку почина першим, [illegible] останнім

```
ret <зн дв> call
```

Тоді немає необхідності [illegible] додавати до стеку

Для запису в стек цілих даних

Зазвичай використовують 2- або 4-байт команди push або pop

```
push ax ; pop eax
push eax ;
```

При запису до стеку даних із [illegible] можна використовувати регістр сп як покажчик стеку при запису операції для цього спочатку регістр сп потрібно буде додати адресу, а потім або командами пересилки або [illegible] кількох команд [illegible] пересилати відразу значення

В С [illegible] стеку [illegible] [illegible] [illegible] [illegible] використання умови [illegible] мова pascal з відповідним порядком запису

45. Машинно-незалежна оптимізація

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

Практически каждый компилятор производит определённый набор машинно-независимых оптимизаций.

1. Исключение общих подвыражений: если внутреннее представление генерируется последовательно для каждого подвыражения, оно обычно содержит большое количество избыточных вычислений. Вычисление избыточно в данной точке программы, если оно уже было выполнено ранее. Такие избыточности могут быть исключены путём сохранения вычисленного значения на регистре и последующего использования этого значения вместо повторного вычисления.
2. Удаление мёртвого кода: любое вычисление, производящее результат, который в дальнейшем более не будет использован, может быть удалено без последствий для семантики программы. Щоб усунути повторні обчислення в програмі необхідно проаналізувати граф програми на наявність однакових під графів, для яких використовуються однакові значення змінних підграфа. Однакові підграфи можна замінити посиланнями на перше використання підграфа. Для цього треба вміти організовувати пошук чергового підграфа серед підграфів програми. Для того щоб реалізувати такий пошук відносно швидким доцільно побудувати індекс над вершинами підграфа за визначеним відношенням підгарафа. Однак аналіз областей існування значень змінних потребує додаткових інформаційних структур, які використовувались в тому чи іншому іншому піддереві.
3. Вынесение инвариантов циклов: вычисления в цикле, результаты которых не зависят от других вычислений цикла, могут быть вынесены за пределы цикла как инварианты с целью увеличения скорости.
4. Вычисление константных подвыражений: вычисления, которые гарантированно дают константу могут быть произведены уже в процессе компиляции.

46. Машинно-залежна оптимізація

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

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

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

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

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

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

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

Наиболее часто встречаются две разновидности трансляторов:

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

Интерпретаторы — исполняют программу после разбора (в этом случае в роли объектного кода выступает внутреннее представление программы интерпретатором). Исполняется она построчно. В данном случае программа
a.транслируется (интерпретируется) при каждом запуске (если объектный код кешируется, возможны варианты)
b.требует для исполнения наличия на машине интерпретатора и исходного кода
Помимо «чисто» трансляторов и интерпретаторов, существует множество промежуточных вариантов. Так, большинство современных интерпретаторов перед исполнением переводят программу в байт-код (так как его покомандно выполнять гораздо проще, а значит, быстрее) или даже прямо в машинный код (в последнем варианте от интерпретатора остался только автоматический запуск, поэтому такой «интерпретатор» называется JIT-компилятором)

Операційні системи (48-67)

48. Типи ОС та їх режими

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

Разновидности ОС:
по целевому устройству:
1. Для мейн-фреймов
2. Для ПК
3. Для мобильных устройств
по количеству одновременно выполняемых задач:
1. Однозначные
2. Многозадачные
по типу интерфейса:
1. С текстовым интерфейсом
2. С графическим интерфейсом
по количеству одновременно обрабатываемых разрядов данных:
1. 16-разрядные
2. 32-разрядные
3. 64-разрядные

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

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

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

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

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

В реальном режиме при вычислении линейного адреса, по которому процессор собирается читать содержимое памяти или писать в неё, сегментная часть адреса умножается на 16 (или, что то же самое, сдвигается влево на 4 бита) и суммируется со смещением. Таким образом, адреса 0400h:0001h и 0000h:4001h ссылаются на один и тот же физический адрес, так как 400h×16+1 = 0×16+4001h.

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

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

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

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

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

49. Типові складові ОС

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

В составе ОС различают три группы компонентов:

ядро, содержащее планировщик; драйверы устройств, непосредственно управляющие оборудованием; сетевую подсистему, файловую систему;

базовая системы ввода-вывода,

системные библиотеки

оболочка с утилитами.

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

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

Командный интерпретатор — необязательная, но существующая в подавляющем большинстве ОС часть, обеспечивающая управление системой посредством ввода текстовых команд (с клавиатуры, через порт или сеть). Операционные системы, не предназначенные для интерактивной работы часто его не имеют. Также его могут не иметь некоторые ОС для рабочих станций

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

Библиотеки системных функций позволяющие многократное применение различными программными приложениями

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

Интерфейс командной строки: инструкции компьютеру даются путём ввода с клавиатуры текстовых строк (команд).

Графический интерфейс пользователя: программные функции представляются графическими элементами экрана.

50. Особливості визначення пріоритетів задач.

Приоритеты дают возможность индивидуально выделять каждую задачу по важности.

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

наступление определенного времени

операционная система переходит в определенное состояние (бездействие, спящий режим и тд)

поступил административный запрос через пользовательский интерфейс или через инструменты удаленного администрирования.

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

по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;

отдавать предпочтение более коротким процессам;

предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

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

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

с динамическим приоритетом (пр. зависит от t ожидания, пр. зависит от t обслуживания)

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

Как реализован механизм динамических приоритетов в ОС UNIX.

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

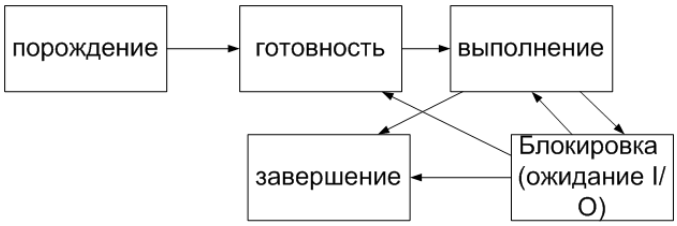
Текущий приоритет процесса – в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет).

Для режима задачи приоритет меняется в диапазоне 0-65, для режима ядра — 66-95 (системный диапазон).

Процессы, приоритеты которых лежат в диапазоне 96-127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой.

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение приоритета сна, выбираемое ядром из диапазона системных приоритетов и связанное с событием, вызвавшее это состояние.

51. Стан виконання задачі



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

активна

готова – має всі ресурси для виконання, але чекає звільнення процесора

очікує дані

призупинена – тимчасово-вилучена з процесу виконання

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

52. Стан задачі в реальному режимі

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

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

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

Закінчується команда, що виконувалась в процесорі

Процесор підтверджує переривання

Блок програмного переривання формує сигнал блоку пріоритетних переривань, тобто видає номер вектора

переривань

INT

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

53. Стан задачі в захищеному режимі

сохраняются все регистры задачи каталог таблиц страниц процесса

Робота програм для обробки преривань в захищенному режимі

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

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

Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим.

Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по адресу 0040h:0067h, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по адресу 0040h:0067h.

Запретить все маскируемые и немаскируемые прерывания.

Открыть адресную линию A20.

Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.

Загрузить регистр GDTR.

55. Способи організації переключення задач

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

активна

готова – має всі ресурси для виконання, але чекає звільнення процесора

очікує дані

призупинена – тимчасово-вилучена з процесу виконання

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

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

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

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

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

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

1. проверка, разрешено ли уходящей задачи переключиться на новую

2. проверка файла, что дескриптор TSS приходящей задачи отмечен как присутствующий и имеет правильный предел (не меньше 67H)

3. сокращение состояния уходящей задачи

4. загрузка в регистр TR селектора TSS входящей задачи

5. загрузка состояния входящей задачи из ее сегмента TSS и продолжения выполнения.

При переключении задачи всегда сохраняется состояние уходящей задачи.

56. Організація роботи планувальника задач і процесів. Супервізори

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

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

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

по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;

отдавать предпочтение более коротким процессам;

предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

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

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

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

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

с динамическим приоритетом (пр. зависит от t ожидания, пр. зависит от t обслуживания)

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

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

58. Механізми переключення задач

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

активна

готова – має всі ресурси для виконання, але чекає звільнення процесора

очікує дані

призупинена – тимчасово-вилучена з процесу виконання

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

Защ.режим: Переключение задач осуществляется или программно (командами CALL или JMP), или по прерываниям (например, от таймера). Тип дескриптора может быть таким, который инициирует выполнение новой задачи после сохранение состояния текущей. Имеется 2 кода типов, определяющих дескрипторы сегментов состояния задачи (TSS) и шлюза задачи. Когда управление передается любому из дескрипторов этих типов, происходит переключение задачи. При переходе к выполнению процедуры, процессор запоминает (в стеке) лишь точку возврата (CS:IP).

Поддержка многозадачности обеспечивается:

· Сегмент состояния задачи (TSS).

· Дескриптор сегмента состояния задачи.

· Регистр задачи (TR).

· Дескриптор шлюза задачи.

Переключение по прерываниям.

Может происходить как по аппаратным, так и программным прерываниям и исключениям. Для этого соответствующий элемент в IDT должен являться дескриптором шлюза задачи. Шлюз задачи содержит селектор, указывающий на дескриптор TSS.

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

Программное переключение

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

либо шлюзу задачи.

JMP dword ptr adr_sel_TSS(/adr_task_gate)

CALL dword ptr adr_sel_TSS(/adr_task_gate)

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

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

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

При переключении задачи всегда сохраняется состояние уходящей задачи.

59. Организация защиты памяти в процессорах

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).

Технические детали

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).

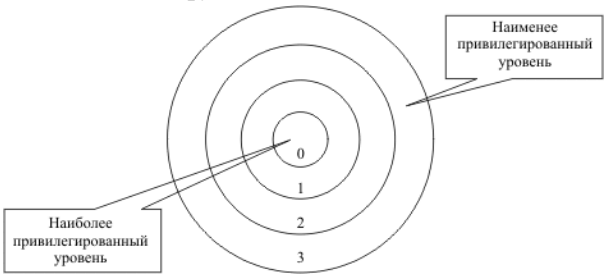
Однако, некоторые программы используют выполнение кода в стеке или куче. Такое решение может быть связано с оптимизацией, динамической компиляцией или просто оригинальным техническим решением. Обычно, операционные системы предоставляют системные вызовы для запроса памяти с разрешенной функцией исполнения 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.[источник не указан 151 день] Примером такой программы может служить Iris.

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.

60. Організація захисту пам'яті в ОС

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

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



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

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

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

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

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

выполнять некоторые команды,

выполнять команды ввода-вывода на том или ином внешнем устройстве,

обращаться к данным других программ,

вызывать другие программы.

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

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

уровень 1 - основная часть программ ОС (утилиты);

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

уровень 3 - прикладные программы пользователя.

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

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

61. Ієрархія організації програм В/В

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

Для решения проблем многозадачности потребовалось разработать аппаратное обеспечение, поддерживающее

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

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

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

62. Необхідність синхронізації даних

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

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

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

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

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

63. Способи організації драйверів

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

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

загрузка драйвера – др. регистрируется в системе, производит первичную инициализацию и т. п.;

выгрузка – освобождает захваченные ресурсы — память, файлы, устройства и т. п.;

открытие драйвера – начало основной работы. Обычно драйвер открывается программой как файл, функциями CreateFile() в Win32 или fopen() в UNIX-подобных системах;

чтение;

запись – программа читает или записывает данные из/в устройство, обслуживаемое драйвером;

закрытие – операция, обратная открытию, освобождает занятые при открытии ресурсы и уничтожает дескриптор файла;

управление вводом-выводом – драйвер поддерживает интерфейс ввода-вывода, специфичный для данного устройства.

Найпростіший драйвер включає в свою структуру наступні блоки:

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

В простих системах підготовка пристрою до роботи виконується звичайно видачею відповідних сигналів на порти управління командами OUT. В реальному режимі ці команди можуть бути використані в будь-якій задачі, а в захищеному лише на так званому нульовому рівні захисту. Очікування готовності зовнішнішнього пристрою в

найпростіших драйверах організовано шляхом зчитування біту стану зовнішнього пристрою з наступним переведенням відповідного біту готовності. Для того, щоб уникнути такого бігання по циклу, в подальших серіях драйверів стали використовувати систему апаратних переривань.

Драйвери Windows.

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

64. Роль переривань в побудові драйверів

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

В программе прерываний выполняется вся фактическая работа драйвера, поэтому она является наиболее сложной его частью. При вызове этой программы исследуется командный байт (третий байт) ранее сохраненного заголовка запроса и в зависимости от его значения выполняются те или иные действия. Программа прерываний обычно использует командный байт в качестве индекса для некоторой управляющей таблицы, вызывая, таким образом, нужную процедуру для каждой команды. Заголовок запроса содержит всю необходимую информацию для корректной обработки каждой команды и сообщает вызывающей о состоянии запроса после завершения соответствующей процедуры. Слово, хранящее состояние после возврата, разбито на несколько полей. Оно содержит бит ошибки, указывающий на то, что в оставшейся части содержится специфический код ошибки, бит выполнения, сигнализирующий о том, что требуемая операция была завершена, и бит занятости,призванный в первую очередь сигнализировать о текущем состоянии устройства. Обязательно должны присутствовать три раздела драйвера – заголовок, программа стратегии и программа.

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

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

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

В реальном режиме имеются программные и аппаратные прерывания. Прогр.Прер. инициируются командой INT, Апп.Прер. - внешними событиями, по отношению к выполняемой программе. Кроме того, некоторые прерывания зарезервированы для использования самим процессором - прерывания по ошибке деления, прерывания для пошаговой работы.

Для обработки прерываний в реальном режиме процессор использует Таблицу Векторов Прерываний. Эта таблица располагается в самом начале оперативной памяти, т.е. её физический адрес - 00000. Состоит из 256 элементов по 4 байта, т.е. её размер составляет 1 килобайт. Элементы таблицы - дальние указатели на процедуры обработки прерываний. Указатели состоят из 16-битового сегментного адреса процедуры обработки прерывания и 16-битового смещения. Причём смещение хранится по младшему адресу, а сегментный адрес - по старшему.

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

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

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

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

В защищённом режиме все прерывания разделяются на два типа - обычные прерывания и исключения (exception - исключение, особый случай). Обычное прерывание инициируется командой INT (программное прерывание) или внешним событием (аппаратное прерывание). Перед передачей управления процедуре обработки обычного прерывания флаг разрешения прерываний IF сбрасывается и прерывания запрещаются.

Исключение происходит в результате ошибки, возникающей при выполнении какой-либо команды. По своим функциям исключения соответствуют зарезервированным для процессора внутренним прерываниям реального режима. Когда процедура обработки исключения получает управление, флаг IF не изменяется.

66. Особливості роботи з БПП

Блок приоритетного прерывания (БПП), формирующий по сигналам от внешних устройств управляющие сигналы на центральный процессор и номера векторов прерывания.

Аппаратные прерывания, используемые для обработки вектора, определяются конструкцией компьютера и, прежде всего, способом подключения сигналов прерывания к БПП и способом его программирования в BIOS. Для машин типа IBM/AT и последующих BIOS использует такие векторы прерываний внешних устройств: 08h-0fh – прерывания IRQ0-IRQ7; 70h-77h – прерывания IRQ8-IRQ15

Организация обработки аппаратных прерываний обеспечивается процедурами – обработчиками прерываний и выполняющими самостоятельные вычислительные процессы, инициированные сигналами с внешних устройств. Последовательность действий обработчика близка к последовательности действий драйвера, но имеет несколько существенных особенностей:

при входе в обработчик прерываний обработка новых прерываний обычно запрещается;

необходимо сохранить содержимое регистров, изменяемых в обработчике;

поскольку прерывание может приостановить любую задачу, то нужно позаботиться о корректном состоянии сегментных регистров в начале обработчика или в процессе переключения задач;

выполнить базовые действия драйвера;

для того, чтобы разрешить обработку новых прерываний через этот блок необходимо выдать на БПП последовательность кодов окончания обработки прерывания (end of interruption) EOI;

выдать синхронизирующую информацию готовности буферов для последующих обменов со смежными процессами;

если есть необходимость обратиться к действиям, которые связанны с другими аппаратными прерываниями, то это целесообразно сделать после формирования EOI, разрешив после этого обработку аппаратных прерываний командой STI;

перед возвратом к прерванной программе нужно восстановить регистры, испорченные при обработке прерывания.