# ЛК – 1. ВВОДНОЕ ЗАНЯТИЕ. ОСНОВНЫЕ КОНЦЕПЦИИ. ИСТОРИЧЕСКАЯ СПРАВКА

**Операционная система** – набор программ для управления ресурсами ЭВМ, взаимодействия с пользователей, другими ПК и программными средствами.

Первый ЯП создан Адой Лавейс (дочь Байрона)

Он состоял из **4 операторов:**

* присваивания (=);
* проверка условий (if);
* цикл (do/while);
* go to

Работала у Чарльза Бебедж, который разрабатывал архитектуру прототип современной ЭВМ. Но в 1864г. Не создал реальную машину.

**В начале 20 века** создали триодную лампу (состояние 0/1). Бондж Бруевич придумал триодную лампу.

Также появляются электромеханические счетные машины. Электромеханические т.к. используются электрические и механика.

Атанасов разработал конструкцию (схему) электронной машины, которая попала американцам.

В последствии **в 1945 г** на основе схемы Атанасова американцы построили первую электронную машину. Она использовалась для управления защиткой. Команды вводились вручную. Автор – Джон фон Нейман.

**В 1952г.** В СССР была построена первая ЭВМ «Стрела» под руководством Лебедева.

«Стрела» была 3-значной (0,1,2) для большей производительности (но это не эффективно, т.к.схемы получались очень сложными).

**В 1957-1958 г.** Бэкусом Науром был создан первый ЯП – фортран.

Фортран – формула трансляций.

Примерно в это же время появились первые ОС – «Атлас», «Сиггейт».

**Транзистор** – полупроводниковая схема.

+: маленький размер.

+:обеспечивает скачок производительности.

**В середине 60х** появилось большое количество ЯП.

C(си) появился в конце 70-х.

В середине 60х появилась ОС System/360 фирмы IBM.

IBM также выпускала ЭВМ (большие машины, как комната).

Впоследствии была разработана ОС System/370.

В СССР эта ОС также использовалась, но машины были отечественные (EC-1861…..)

Первая машина в СССР – «Минск» (Минск-1, Минск -2 и др).

**В 70-е годы** появляется ЯП Паскаль (Никлас Вирт) => Delphi

В это же время появился язык С (автор Ричи).

Также в это время появляются микропроцессоры и микроконтроллеры от корпорации Intel. Разрядность постепенно увеличивалась (4->8->16).

В то же время появляется ОС DOS (USA, Microsoft).

Страуструп разработал С++. А первый объектно-ориентированный язык программирования – smalltalk.

Бил Гейтс купил ОС klaridos и адаптировал ее для микропроцессоров. Появился цветной дисплей, мышь.

Windows пришла на смену DOS.

+: каждая программа имеет свое окно; в окно направляется сообщение; диспетчер сообщений рассылает сообщения программам.

окна описаны оконным классом (название, id, цвет, размер)

Конкуренцию DOS пытались создать на основании MAC (джобс).

В отличии от DOS, где команды вводились в виде слов (текст), идея Джобса – использовать картинки (иконки).

Тем не менее, ни MAC, ни Apple не получили такой прибыли как DOS и Windows.

Наибольшее развитие на современном этапе получили сети Интернет => программные технологии базируются на сети Интернет, включая web и мобильное программирование, облачные технологии и другое.

Интерфейс операционной системы реализован на 2 уровнях:

* командный интерфейс (через консоль компа, через утилиты)
* языковой интерфейс (API).

# ЛК – 2. СИСТЕМА ВВОДА / ВЫВОДА

**Система ввода/вывода** состоит из **физических устройств** (флеш-память, компакт-диски) и **программной подсистемы** (драйвера внешних устройств)и **системных средств** (управление файлами/каталогами, обработки прерываний).

Физический уровень представлен файлами, каталогами и дисками.

Диски:поверхность диска покрыта намагниченным материалом (ферромагнетиком) разбита на **дорожки** (концентричная окружность, на которую записывается и считывается информация с помощью головок чтения/записи.)

Запись и чтение выполняются в соответствии с **законом магнитной индукции.**

- диск вращается с достаточно большой скоростью (10 об/сек)

- магнитная головка наводится на нужную позицию на дорожке

Эти законы были открыты Фарадеем.

Поверхность диска разбита на **сектора** (минимальный физический блок, который можно прочитать). Обычно информация из сектора поступает в буфер (или считывается из нее). **Буферная память** – блок операционной памяти, куда выполняется чтение/запись данных из программы. При заполнении буфер нужно выталкивать. Буфер нужен для согласования работы диска и процессора. При открытии файла можно выставить параметр «без буфера».

**Кластер** – логический блок – небольшие участки блока памяти, использующиеся для операций ввода/вывода.

Файл -> кластер1 //

….. // они мб на различн.дорожках и кластерах

-> кластерz //

Каждый кластер содержит ссылку на следующий.

*Загрузочный сектор* содержит служебные данные (он 1-ый).

Сведения о файлах на диске, каталогах.

## Системы управления файлами:

**\* FAT** – file access table (максимальный размер файла – 4 Гб)

**\* NTFS** – new technology file system

**FAT:**  для каждого имени файла можно найти номер кластера, по нему позицию и дорожку.

**\*** ненадежный (нет резервной копии)

**\*** медленный (нет быстрого поиска на основе дерева)

**\*** таблица кластеров, где для каждого кластера указ.№ следующего

**\*** нет пароля к файлам

Пример: Пусть К = 32746 27534 / 1152 = 23 (1038 - остаток)

Nдор= 1152 23 – номер дорожки

№класт=27534 1038 - № кластера на дорожке

**NTFS:**

\* позволяет вести журнал работы, где определены операции, выполняющиеся над файлами.

\* позволяет восстановить последнее состояние в случае сбоя

\* информация дублируется (системная копия master table)

\* для ускоренного поиска файлов используется дерево

\* неограниченные размеры файлов (чтобы влезли в диск)

\* не различ.по способам доступа каталога и файлами

Все есть файл

\* позволяет навесить пароль на файлы/каталоги

Файл – именованная совокупность записей

(есть имя, расширение, содержание записи)

Файл хранит числа -> запись = число

Файл хранит текст -> запись = строка (может занимать неск кластеров)

На уровне ОС чтение/запись выполняется на основе механизма **прерываний** (interruptions)

Пример: прога выдает команду «Write» -> 2 варианта (сихн/асинх)

**синх:**  пока write не вып, программа будет сидеть и ждать

процессор остан.работу и анализирует № прерывания и по № - **обработчик** (системная программа, соответствующая № прерывания)

поток должен обеспечить возможность ввода/вывода

ввод/вывод на самом нижнем уровне выполняется **драйвером** (выдает сигналы, выставляет данные на шину)

**асинх:** прога не ждет, а работает дальше

параллельно идет работа с файлами

использует определенное средство для коммуникации с программой и устройством ввода/вывода – **порты завершения**

(операция выполнилась -> в порт отправляется сообщение)

требует проверки

выполн.на основе механизма **прямого доступа к памяти,** минуя процессор.

## ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ВВОДА/ВЫВОДА НА С/С++

Файл CreateFile – для создания и открытия.

{LPCTSTR IpFileName …..}

Аргументы: имя файла

способ доступа

режимы совместного использования (общ/монопольный)

атрибуты защиты (зашита от записи)

создание/открытие файла (есть файл – откр, нет - создаем)

флаги и атрибуты (headen)

файл атрибутов

Данная функция возвращает заголовок HANDLE число (о-ошиб)

**Способ доступа:**

\* generic\_read – только для чтения

\* generic\_write – только для записи

\* 0 – приложение может только определять атрибуты

**Режимы совместного использования**

**\*** file\_share\_read

\* file\_share\_write

\* file\_share\_delete

**Атрибуты защиты**

\* null – по умолч (файл открыт всем пользователям)

**Создание открытие файла:**

\* create\_new – создаем новый файл, не дб такого на диске

\* create\_always – созд, старый умрет если был

\* open\_existing – если него нет, неудача

\* open\_always – если не существует, будет создан

\* truncate\_existing – уничтожит содержимое файла, если был

**Атрибуты файла:**

\* file\_atribute\_archive

\* file\_atribute\_encrypted

\* file\_atribute\_headen

\* file\_atribute\_normal

……

\* file\_atribute\_readonly

\* file\_atribute\_system

\* file\_atribute\_temporary

\* file\_atribute\_overlapped

BOOL DeleteFile – удалить файл

{имя файла}

WriteFile – запись данных

(имя файла, указатель на буфер, кол-во байт, кол-во байт, ipDeveloped)

(…………., что запис.., будем записывать, уже есть., режим асинхр)

Пример: int main(){

HANDLE hFile;

hFile = CreateFile(

“C:file.bat” //имя файла

generic\_write, //запись в файл

0, //мнопол.доступ к файлу

null, //защиты нет

create\_new, //созд.new

file\_attribute\_normal, //обычн.ф.

null) //шаблона нет

# ЛК – 3. УПРАВЛЕНИЕ ПАМЯТЬЮ

Оперативная память основана на *полупроводниках* (схемы, на основе вентилей n(-) p(+), которые способны пропускать ток в одном направлении, а в обратном уже не пропускают).

Различают следующие **виды** оперативной памяти:

* **Основная** (где находятся программы, д-е, работает при включении питания)

Память ЭВМ разбита на **сегменты**:

* **CS**(code segment) – сегмент кода

Запис.команды

* **DS**(data segment) – сегмент данных

Записывает данные, которые объявляются или порождаются в процессе работы программы

* **SS**(stack segment) – сегмент стека

\* обеспечивает передачу данных при вызове подпрограмм и возврат результата

\* обеспечивает обработку прерываний

* Доп.сегменты данных – **FS, GS**

Современные ЭВМ мультипрограммные. Каждой программе выделялся размер памяти фиксированного размера. Возникла проблема фрагментации памяти

**Фрагментация памяти** - чередование заполненных и незаполненных фрагментов памяти.

Одним из алгоритмов для управления памятью – **метод Близнецов.**

В этом случае память заранее не разбивается на фиксированные размеры. Считается, что память имеет монолитный единый размер.

Память выделяется блоками, равными степени числа 2.

|  |
| --- |
| 32 |
| 32 |
| 64 |
| 128 |

Пример: 1 прога – 40мб 28=256

2 прога – 10мб

256 / 2 = 2 по 128

128 / 2 = 2 по 64

64 / 2 = 2 по 32

(выбираем больший)

3 прога – 60мб

4 прога – 150мб

Близнецами явл.2 равные половинки памяти.

Когда 1 близнец освобождается, он сливается с другим близнецом и образуется один блок.

+: простая схема управления

-: неэффективность использования

**Метод Фибоначчи** (лучше близнецов)

Сумма 2 предшествующих числа = следующее.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 99, …..

Для 1 проги – 55 (15 не используется) (256 – 55 = 201)

Для 2 проги – 13 (3 не используется) (201 – 13 = 198)

Память используется более эффективно, так как меньше дырки.

**Перекрытие** (overlapping)

Программист сам разбивает программу на секции так, чтобы одна секция могла занять область другой секции (перекрыть).

Поддерживался в языке Pascal.

Большие программы разбивались на секции и грузится кусками. Одна отработала, затем на ее место пришла другая => перекрытие.

**Страничная память.**

Страница играет роль кластера. Получается, что программе выделяются страницы / несколько страниц (размер страницы ~ 8 кб)

Программе на данный момент выделяется сегмент. И в этом сегменте находится множество страниц. Такая организация называется **странично-сегментная**(сейчас)

Особенности:

1. Страницу можно выгружать на диск в файл подкачки. Можно выгружать временно, а на ее место загружать другую страницу, т.е. **обмен страницы (swapping).**

2. Страницы можно выделять в кучи (область динамически выделяемой памяти) (heap).

Благодаря страницам проблема фрагментации исчезла.

**Основные принципы:**

При загрузке программы в память ей выделяется один или несколько сегментов. Размеры сегментов «подстраиваются» под размеры программы. Сегмент разбивается на страницы. Если в процессе работы требуется дополнительная память, то имеется **два варианта:**

* Использовать механизм свопинга
  + Динамически затребовать память (mallock, callock) – выделяется в кучи (heap)

## СТЕК

Файлы, которые используют стек – .exe

Файлы, которые не используют стек – .com (< 64кб).

Принцип стека – первым пришел, последним вышел (FILO).

2 вида передачи параметров в стек:

1. Стандартный (std) – верхушка стека – первый параметр и т.д.
2. Pascal-евый.

Разгрузка стека после завершения подпрограммы – важно (в ассемблере сами).

**Функции** стека:

\*загрузка параметров

\*сохранение состояния программы в момент прерывания

\*использование в компиляторах

Метод для получения машинных команд основан на использовании польской инверсной записи (операторы идут после операндов).

Пример: a = b\*(2+c) => ab2c+\*=

a + b => ab+

c = e => ce=

Пример:

|  |  |  |
| --- | --- | --- |
|  | выход | стек |
| z = a+b\*(c-5) |  |  |
| = a+b\*(c-5) | z |  |
| a+b\*(c-5) | z | = |
| +b\*(c-5) | za | = |
| b\*(c-5) | za | =+ |
| \*(c-5) | zab | =+ |
| (c-5) | zab | =+\* |
| с-5) | zab | =+\*( |
| -5) | zabc | =+\*( |
| 5) | zabc | =+\*(- |
| ) | zabc5 | =+\*(- |
|  | zabc5- | =+\* |
|  | **zabc5-\*+=** |  |

**1 этап** (Польская запись):

1. считывает очередной токен, если это переменная или константа -> отправляем её на выход
2. если это операция -> грузим ее в стек
3. \* имеет более высокий приоритет, чем = и +, -
4. скобки всегда в стек
5. менее приоритетная не может сесть на более приоритетную -> ее на выход; более приоритетная садится на менее приоритетную
6. нулевая операция всегда размещается над скобкой
7. при поступлении закрывающей скобки выталкиваем из стека содержимое вплоть до связанной с ней открывающей скобки
8. скобки удаляются из стека
9. если на входе уже нет символов -> выталкиваем содержимое на выход и получаем окончательный ответ

**zabc5-\*+=**

**2этап** (получение машинных команд):

1. для простоты машинная команда имеет вид:

КОП – код операции *(каждая операция имеет числа)*

|  |  |  |
| --- | --- | --- |
| КОП | 1st операнд | 2nd операнд |

1. читаем токены по порядку

|  |  |
| --- | --- |
| выход | стек |
| **zabc5-\*+=** |  |
| abc5-\*+= | z |
| bc5-\*+= | za |
| c5-\*+= | zab |
| 5-\*+= | zabc |
| -\*+= | zabc5 |

1. из стека выгружаем количество операндов, соответствующее данной операции и обратно загружаем в стек

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| команда | операнды | | результат | стек |
| SUB | 5 | c | w (c-5) |  |
| MUL | w | b | Q (b+w) |  |
| ADD | Q | a | R (a+Q) |  |
| = | z | R | z |  |

## СТРУКТУРА ПРОЦЕССОРА. РЕГИСТРОВАЯ ПАМЯТЬ

Процессор состоит из 3 блоков: арифметико-логический блок (АЛБ), блок управления и память ЗУ. Эти 3 блока взаимодействуют между собой.

(рисунок)

## Арифметико-логический блок:

* сумматор,
* регистр общего назначения (AX (EAX), BX (EBX), CX (ECX), DX (EDX)) – AX и др – 16, EAX и др – 32.
* флажки (признаки операций (ZF – ноль, SF – отриц, OF –переполн, PF – четность)),
* аккумулятор (регистр, куда записывается результат операций).

Наряду с регистрами имеется кеш-память команд и кеш-память данных.

**Кэш** – «блок оперативной памяти» (не регистр)

Кеш-память выполняет функцию расширения регистровой памяти и служит для повышения быстродействия.

Наряду с регистрами общего назначения есть **управляющие** и **специализированные регистры**.

* CS – регистр сегмента кода (хранится адрес начала сегмента)

допускается, что программа занимает более одного сегмента кода

* DS – регистр сегмента данных (полностью доступен программе)
* SS – регистр сегмента стека

SP – stack pointer – указатель верхушки стека,

BP – Bottom Pointer – указатель дна стека

* IC – регистр адреса текущей выполняемой команды (счетчик команд)
* CX0, … CX4 – *управляющие регистры* – хранят информацию о таблице сегментов, таблице страниц – так называемое слово состояния процессора.

Записывается номер прерывания, которое обрабатывается процессором; маски сигнала прерывания, адрес текущей команды и т.д.

## ПЗУ (Постоянное запоминающее устройство - ROM)

ПЗУ != ОЗУ.

В ОЗУ хранится информация пользователей, драйвера, операционная система, диспетчеры и др и закачивается во время загрузки/перезагрузки

В ПЗУ хранятся микропрограммы

**Микропрограммы –** программы, управляющие работой процессора и выполнением команд.

Каждая микропрограмма состоит из микрокоманд. Имеется **2 типа микрокоманд:**

* Операционная команда

Содержит сигналы (0/1), каждый сигнал поступает на определенное место в схеме процессора

ВИД:

Y1 Y2 Y3 Y4 Y5 Y6 ….. Yz

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | … | 1 |

* Условная команда

хранятся условия, которые надо проверить

ВИД:

|  |  |
| --- | --- |
| номер условия | адрес перехода |

ZF ……

# ЛК – 4. УПРАВЛЕНИЕ РЕСУРСАМИ

**Ресурс** – файлы, память, мышка, консоль, принтер и другие средства, подключаемые к ПК.

Система взаимодействует с ресурсами на основе механизма прерываний. Т.е.имеется специально устройство – **контроллер прерываний.** К нему подключается 8 внешних устройств. Если надо больше – строится каскад.

## Как построить каскад?

Выход одного контроллера подключается на вход другого: к одному 8 подключим, ко второму 7 => 15. Выход контроллера идет на вход прерываний (**int** - interrupt) процессора. Этого недостаточно чтобы обработать прерывание. Кроме сигнала прерывания, нужен еще номер прерывания. Он определяется адресом устройства. Устройство подключается к общей шине через адаптер. У каждого устройства свой адаптер. Адаптер регулирует доступ к шине (разрешает выставить адрес на шину или получить команду процессора).

## Как осуществляется взаимодействие внешнего устройства с процессором?

При нажатии на кнопку на клавиатуре или мыши сигнал прерывания поступает на контроллер, который при наличии других прерываний определяет очередность обработки по приоритету. Если нет более приоритетных прерываний –> посылает сигнал на int – > процессор приостанавливает работу –> устройство устанавливает свой адрес.

По выставленному на шину адресу (номер) процессор определяет программу обработки прерывания. Использует для этого специальную таблицу прерываний, которая находится в оперативной памяти. По номеру прерывания определяется адрес программы обработчика. Программа обработки прерывания получает управление и взаимодействует с устройством (подает команды (read/write)) до тех пора пока другое устройство не прервет обработчика более приоритетным прерыванием.

Также отметим возможность **маскирования прерываний**. Маскированное прерывание не наступает (процессор от него защищен).

Самый высокий приоритет – схема питания машины. Устройство связывается с контроллером напрямую (проводами).

## Прерывания

Прерывания бывают аппаратными и программными.Программные прерывания обрабатываются процессором. Программные прерывания можно выдавать по команде **int** в ассемблерных командах, указав номер прерывания. Но в других языках прерывания не поддерживаются.

Управление ресурсами учитывает характер ресурса. Каждый ресурс имеет **дескриптор** – структура, содержащая данные о владельце ресурса, дате создания, правах доступа к этому ресурсу (файл – для чтения/записи) и атрибутах (таких как размер в байтах(для файла)).

## Ресурсы

Ресурсы делятся на **разделяемые** и **монопольно используемые**. В случае разделяемых ресурсов возникает проблема синхронизации. Имеются разные алгоритмы для работы с разделяемыми ресурсами.

## Алгоритм Банкира

Относится к управлению оперативной памятью.

Пусть имеются процессы A,B,C,D.

Процессу А надо 50 МБ, затем 40 МБ.

Процесс B – 80 МБ / 60 МБ.

Процесс D – 20 МБ (однократно)

Процесс С – 60 МБ / 30 МБ.

Допустим, у процессора всего 150 МБ в распоряжении.

Как организовать обработку?

Допустим, сначала обратился процесс В. Ему в совокупности нужно 140. В наличии имеется 150. При любом раскладе можно удовлетворить полностью процесс В. Значит, процесс В получается затребованные 80 МБ. В наличке остается 70 МБ. Когда процесс В освободится он может использовать свободную память для загрузки 60 МБ.

Следующий процесс А. Всего требует 90 МБ. Удовлетворили 50 МБ. Допустим, что процесс А запросил еще 40. Процесс А не может продолжить работу пока не закончится процесс В.

Вот это и есть проблема **узкого места** (deadlock).

Согласно алгоритму банкира требуемый ресурс выделяется только в том случае, если все выполняющиеся процессы с учетом всех их будущих требований суммарно смогут завершиться.

Его минус в том, что экономия расхода ресурса приводит к понижению скорости обработки.

## Алгоритм обслуживания процессов по циклической схеме

В системе работает много процессов и осуществляется цикл обработки, каждому процессу выделяется квант обработки (тета). После окончания времени для одного процесса, осуществляется обработка следующего.

Проблема в том, что процессы разноприоритетные, но все получаются одинаковое время. Вторая проблема – какое тета выделить - > накладные расходы, потеря времени. Важный недостаток – фиксированная очередность.

Тем не менее, с приоритетами ситуацию можно поменять. Если разные приоритеты, то можно поступить двояко:

\* выделять квант времени пропорционально **приоритету** (этого недостаточно). В unix учитывается не только приоритет, но и **время пребывания** процесса в систему (чем больше сидит – тем больше ему надо дать)

\* **метод эстафеты:** процесс получает «эстафетную палочку» и обрабатывается с учетом максимально допустимого времени.

## Управление консолью

**Консоль** = клавиатура + дисплей.

Как и любой ресурс, консоль имеет дескриптор (заголовок), а также имеет структуру, описывающую ее свойства.

**ZeroMemory**– команда API, которая заполняет нулями область памяти.

Для консоли: \*заголовок

\*размеры окна

\*координаты левого верхнего угла

\***буфер экрана** – область память, которая хранит содержимое, отображаемое на консоли (по горизонтали и вертикали)

\*указываем цвет букв и консоли

si.IpTitle = “tis is title”;

si.dxX = 200;

si.dwY = 200;

si.dwXSize = 300

si.dwYSize = 200;

si.dwXCountChars = 100; //размер буфера экрана по горизонтали

si.dwYCountChars = 100; //размер буфера экрана по вертикали

//красные буквы на синем фоне

si.dwFilleAttrubyte = FOREGROUND\_RED | FOREFROUND\_INTENSITY | BACKGROUND\_INTENSITY | BACKGROUND\_BLUE;

//используем параметры, которые установили

si.dwFlags = STARTF\_USECOUNTCHARS | STARTFJJSSEFILEATTRIBUTE| START\_SEROSITION | STARTD\_SESHOWWINDOW | START\_SESIZE;

si.wShowWindow = SW\_SHOWNORMAL;

**createProcess** – запускает процесс с новой цветной консолью.

**CloseHadle** – закрываем дескрипторы процесса.

## Управление курсором

HADLE hStdOut; //дескриптор стандартного вывода

CONSOLE\_CURSOR\_IINFO cci; //информация о курсоре

//читаем дескриптор стандартного вывода

hStdOut = SetStdHandle(STD\_OUTPUT\_HANDLE);

//читаем информацию о курсоре

if(!GetConsoleCursorInfo(hStdOut, &cci)) cout << “Failed”;

//устанавливаем новый размер курсора

if(!setConsoleCursorInfo(hStdOut, &cci))

//делаем невидимый курсор

cci.Visible = FALSE;

# ЛК – 5. УПРАВЛЕНИЕ РЕСУРСАМИ. ПРОДОЛЖЕНИЕ

## Изменение позиции курсора

HANDLE HsTDoUT; //дескриптор стандартного вывода

COORD coord //для позиции курсора

//читаем дескриптор стандартного вывода

hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE)

cout << ‘inout new position << endl;

cout << “X=”

cin >> coord.X;

cout << “Y=”

cin >> coord.Y;

//установить курсор в новую позицию

if(!SetConsoleCursorPosition(hStdOut, coord)) {

Cout <<Set cursot position failed” << endl;

Return GetLastErrorO;

}

cout << “This is a new position” << endl;

cout << “Input any char to exit.”;

## Прокрутка экрана

//для того, чтобы прокрутить экран используем содержимое буфера

CONSOLE\_SCREEN\_BUFFER\_INFO csbinfo;

//область отображения буфера

SMALL\_RECT srctWindow;

//задаем размеры окна в прямоугольник, который объявили

sectWindow = csbindo.srWindow;

if(srctWindow.Top >= iRows){

scrtWindow.Top -=

…….

## Мышь. Клавиатура

Текущая позиция мышки определяется матрицей. И состоит из фотодиодов. Фотодиоды сканируют поверхность стола, на котором находится мышка и формируется матрица позиций. Вычисляется изменение на матрице позиций курсора и проецируется на экран консоли.

INPUT\_RECORD – ТД, который представляет инфу и типе и параметрах события

ReadConcoleInput – читаем соб, связанное с мышью и консолью+в InRec заносим

InRec.EventType – определяем тип события (KEY\_EVENT, MOUSE\_EVENT)

VK\_UP, VK\_DOWN, VK\_LEFT, VK\_RIGTH – виртуальные клавиши

## Изменение позиции курсора

HWND window = FindWindow (“TAskMAnagerWindow”, “Task Manager”);

if(window){

RECT rect = {0};

GetWindowRect(window, &rect);

SetForegroundWindow(window);

SetActiveWindow(window);

SetFocus(window);

Sleep(300);

SetCursorPos(rect.rigth – 200, rect.bottom - 200)

}

return 0;

}

## Окна

Наиболее простой тип окон – диалоговые окна. Можем вывести диалоговое окно, даже не создав окно приложения.

#include <window.h>

INT WINAPI WinMain (….) //аргументы система ставит сама

{

MessageBox (NULL, //определяет куда выводим – тек.прилож

L“Hello, Win”, // заголовок окна

NULL, //текст внутри MessageBox

NULL); //определяет тип диалога (значок иконки) и какие кнопки появляются в диалоге (по умолчанию – 1 кнопка);

return 0;

}

## Расширенный вариант MessageBox:

......

MessageBox(

NULL, //текущее окно

L “Welcome!”, //текст внутри заголовка

L “Hello Again”, //содержимое MessageBox

MB\_YESNOCANCEL | MB\_ICONQUESTION //иконка с «?»

);

return 0;

## Порты

Управление устройствами может быть реализовано не только с помощью функции Windows, но и через порты (COM, USB). К порту COM (общий) можно было подключить любое устройство.

Код, который получает звук из динамика, используя порта (ассемблер):

MODEL SMALL

STACK 256

CODESEG

Start:

mov ax, @data ; установка в ds адреса

mov ds, ax ; сегмента данных

in al,61H ; получить текущее значение (чтение порта)

or al, 02H ; установить второй бит (вывод в порт)

out 61H, all ;вывести значение в порт

mov cx, 9000H ;количество циклов

daley:

loop delay ;ждать

and al, 0fdH ;очистить второй бит

out 61H, al ;вывести в порт

……

## Окна

Концепция Windows означает: каждому приложению отельное окно. За каждым окном своя оконная процедура(WinProc). Смысл оконной процедуры – обрабатывать сообщения различного рода (Messages)

WM\_OnCreate – сообщение посылается когда создается окно

WM\_OnPaint – сообщение подается, когда окно перерисовывается

WM\_OnDestroy – сообщение подается, когда окно закрывается

Таким образом, **Windows** – система, управляемая сообщениями. Сообщения рассылает программа Диспетчер Сообщений. **Диспетчер сообщений** – системная программа, которая посылает сообщения приложения, используя для этого очередь сообщений.

Необходимо создавать самим окна приложений. Эту задачу можно возложить на плечи «мастера».

В следующей программе создается приложение с окном:

HINSTANS hInst;

TCHAR szTitle[MAX\_LOADSTRING];

THCAR szWindowClass[MAX\_LOADSTRING]; //главный Window класс

//процедура обратного вызова (ее вызывает система)

//оконная процедура

LRESULT CALLBACK **WndProc** (HWND hWnd, UINT message, WPARAM wParal lParam){

Int wmId, wmEvent;

PAINTSTRUCT ps;

HDC hdc;

switch (message){

case WM\_COMMAND:

WmId = LOWORD(wParam); //младшие биты

wnEvent = HIWORD(wParam); //старшие биты

switch(wmId){

case IDM\_ABOUT:

DialogBox(….); //команда для вызова диалога

break;

case IDM\_EXIT:

DestroyWindow(hWnd);

break;

defult:

return DefWindowProc (….)

}

break;

case WM\_PAINT:

hdc = BeginPaint();

EndPaint(hWnd, &ps);

case WM\_DESTROY:

PostQuitMessage(0); //закончить программу

break;

default:

return DefWinProc(….);

return 0;

}

//оконный класс опис с помощью процедуры WNDCLASSEX

ATOM **MyRegisterClass**(HISTANCE hInstance){

WNDCLASSEX wcex;

wcex.cbSize = sizeof(WNSCLASSEX);

wcex.style = CS\_HREDRRAW | CS\_VREDRAW;

wcex.lpfnWndProc = WndProc; //имя оконной процедуры

wcex.cbClsExtra = 0; //доп.д-е для класса

wcex.cbWndExtra = 0; //до.д-е для окна

wcex.Instance =hInstance; //заголовок

wcex.Icon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI\_MYWIN3));

wcex.hCursor = LoadCursor(Null, IDC\_ARROW); //тип курсора

wcex.hbrBackground = (HBRUSH)(COLOR\_WINDOW+1);//тип закраски окна

wcex.lpszMenuName = MAKEINTRESOURCE(IDC\_MYWIN3);

wcex.lpszClassName = szWindowClass; //указатель на оконный класс

wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE (IDI\_SMALL));

return ReficterClass(&wcex)

}

//

**int APIENTRY \_tWinMain**(

HINSTANCE hInstance,

HINSTANCE hPrevInstance,

LPTSTR lpCmdLine,

int nCmdShow){

UNREFERENCED\_PARAMETER(hPrevInstance);

UNREFERENCED\_PARAMETER(lpCmdLine);

MSG msg;

HACCEL hAccelTable;

//инициализируем глобальные строки

LoadString(hInstance, IDS\_APP\_TITLE, szTitle, MAX\_LOADSTRING);

LoadString(hInstance, IDC\_MYWIN3, szWindowClass, MAX\_LOADSTRING);

MyRegisterClass(hInstance);

//создаем окно

if (!InitInstance (hInstance, nCmdShow)){

return FALSE;

}

//загружаем таблицу быстрез клавиш

hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC\_MYWIN3));

//организуем цикл диспетчера сообщений и направляем их в соответствующее окно

while (GetMessage(&msg, NULL, 0, 0)) {

if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

return (int) msg.wParam;

}

# ЛК – 6. ПРОЦЕССЫ. ПРОДОЛЖЕНИЕ

Получение информации о всех выполняющихся процессах.

//сделать снимок состояния процессов.

HANDLE hSnapshot = CreateToolhelp32Snapshot (TH32CS\_SNAPPROCES, 0)

//перечисление

PROCESSENTRY32 pe32;

//выбрать процесс

Process32First – выбрать первый процесс

Process32Next – выбрать следующий процесс

Проверка выполнения процесса с указанным именем:

//задаем текстовый буфер, где exe – имя исполняемого процесса

TCHAR buf[] = TEXT (“GetProcessNAme.exe”)

//получение списка процессов

HANDLE hSnapshot = CreateToolhelp32Snapshot (TH32CS\_SNAPPROCES, 0)

//объявляем структуру и получаем первый процесс + проверяем на совпадение имена

# ЛК – 7. ПЛАНИРОВАНИЕ ПРОЦЕССОВ

Существует **2 типа планирования**:

\*статическое – применяли ранее, в системах пакетной обработки

Сейчас используется в системах военного обеспечения

\*динамическое –

Допустим, даны процессы:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| процесс | P1 | P2 | P3 | P4 | P5 |
| Время вып-ния | 10 | 6 | 4 | 12 | 5 |

## **Минимизация среднего времени ожидания**

(с момента постановки в очередь и начала обработки)

**Суммарное время / кол-во процессов = время ожидания**

Имеется точный **алгоритм:**

Для минимизации среднего(суммарного) времени ожидания необходимо выполнять процессы в порядке возрастания времени обработки (самые тяжелые в конец, а самые легкие вперед)

Построим временную диаграмму:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P3 | P5 | P2 | P1 | P4 |
| 0 4 | 9 | 15 | 25 | 37 |

Время ожидания (W)

W3 = 0;

W5 = 4;

W2 = 9;

W1 = 15;

W4 = 25;

Получаем сумму Wi = 53.

Среднее время ожидания (53/5) = 10,6;

Запишем, как было бы в исходном виде

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 |
| 0 10 | 16 | 20 | 32 | 37 |

Время ожидания:

W1 = 0;

W2 = 10;

W3 = 16;

W4 = 20;

W5 = 32;

Сумма Wi = 78;

Среднее время ожидания = 15.6 (превышена в 1.5 раза)

Если о времена **неизвестны**, то обработка выполняется путем выделения кванта времени (квантов), т.е. процессы образуют очередь. Из этой очереди они идут в центральный процессор и обратно.

Квант времени – тау (0 с перегородкой)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | …. |  |  |

O1 = 1;

O2 = 2;

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 | P1 | P2 | P3 | P4 | P5 | ……. |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ……. |
|  | | | | | | | | | | |
| ….. | P1 | P2 | P3 | P4 | P5 | P1 | P2 | P3 | P5 | ….. |
| ….. | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | … |
|  | | | | | | | | | | |
| P1 | P2 | P4 | P1 | P4 | ….. | P1 | P4 | P4 |
| 25 | 26 | 27 | 28 | 29 | …… | 35 | 26 | 37 |

Желтый – закончился.

W1-1=0; W1-2=4; W1-3 =4; W1-4=4; W1-5=3; W1-6=2; W1-7=1; W1-8=1

При квантовании времени минимизируется среднее время ожидания каждого процесса. При этом, чем меньше квант, тем лучше.

Выбор кванта – отдельная самостоятельная задача. В реальных машинах величина кванта – микросекунды.

**Осн.вывод** – при планировании на основе квантования минимизируется среднее время ожидания для каждого процесса.

В системе unix величина кванта определяется **2 факторами:**

\*приоритетом процесса

\*проведенным временем в системе

Этим самым учитывается трудоемкость процесса.

## Планирование при наличии директивных сроков

**Директивный срок** – это время, к которому нужно завершить процесс. С учетом возможности нарушения директивного срока задача может быть переформулирована => спланировать работу, чтобы наибольшее отклонение от директивного срока было наименьшим, т.е. минимизировать наибольшее отклонение.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Процесс | P1 | P2 | P3 | P4 | P5 |
| Ti: Время вып-ния | 10 | 6 | 4 | 12 | 5 |
| Di: Директ.время | 18 | 10 | 8 | 28 | 10 |

Спланировать, чтобы P1 закончился к моменту времени 18 и т.д. Т.к. по ранее проведенным подсчетам заканчивается время в 37, то мы нарушим директивные сроки в любом случае.

Есть точный **метод решения** – назначать процессы в первую очередь с наименьшим директивным временем. Если время одинаковое – не имеет значения какой процесс назначать первее.

Составим расписание:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 10 | 10(+5) | 18(+7) | 28(+9) |
| P3 | P2 | P5 | P1 | P4 |
| 4 | 10 | 15 | 25 | 37 |

P3 и P2 – вложились, P5 – на 5 превысил; P1 – на 7 превысил;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 18 | 10(+6) | 8(+12) | 28(+4) | 10(+27) |
| P1 | P2 | P3 | P4 | P5 |
| 10 | 16 | 20 | 32 | 37 |

На основе этого алгоритма можно построить алгоритм минимизации числа процессов нарушителей.

**Процесс-нарушитель –** процесс, который нарушает директивный срок.

Назначаем процессы по **принципу:** сначала – с наименьшим директивным сроком. Если процесс нарушает директивный срок – то он считается нарушителем, исключается из расписания, а оставшиеся процессы планируем по той же схеме, т.е. очередной процесс должен иметь наименьший директивный срок. Если опять появляется нарушитель, то снова исключаем и его и снова планируем без него.

Таким образом, получим расписание без нарушителей. После этого всех нарушителей, которые были обнаружены помещаем в конец расписания.

## Метод Джонсона (Алгоритм Джонсона)

Метод Джонсона решает задачу минимизации времени завершения пакетной обработки в системе с двумя ресурсами. Обычно первый ресурс – процесс, второй ресурс – устройство ввода/вывода.

Иначе говоря, решает метод 2фазной обработки.

Сначала процесс обрабатывается на процессоре, а затем выполняет вывод на диск, т.е. работает с внешней памятью.

T1 – время обработки на процессоре

T2- время обработки на внешнем устройстве

|  |  |  |
| --- | --- | --- |
|  | T1 | T2 |
| P1 | 2 | 6 |
| P2 | 4 | 1 |
| P3 | 3 | 2 |
| P4 | 5 | 2 |

Находим минимальный элемент в матрице и выделяем его кружком.

Первый минимальный – 1 (2 строка, 2 столбец)

Если минимальный элемент находится во 2 столбце, то назначаем соответствующий процесс (p2) в конец списка (выполняем его последним). И вычеркиваем из списка.

Если минимальный элемент находится во 2 столбце, то снова в конец списка, а если в 1ом столбце – в начало списка.

Список:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | P2 |
| P1 |  |  | P2 |
| P1 |  | P4 | P2 |
| P1 | P3 | P4 | P2 |

Согласно этому алгоритму процесс обработки пойдет следующим образом:

Сначала p1, который требует 2 единицы на процессоре

Процессор:

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P3 | P4 | P2 |
| 2 | 5 | 10 | 14 |

P1 проводит на процессоре 2, и идет на выход. Так и с остальными.

Вывод:

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P3 | P4 | P2 |
| 8 | 10 | 12 | 13 |

## Потоки

В отличии от процессор, потоки работают в адресном пространстве породившего их процесса. А процессы работают в своем собственном адресном пространстве. Это означает, что потокам доступны общие переменные.

Поток создается по команде CreateThread.

В параметрах (3ий аргумент) передается имя функции, которая будет выполнять поток.

4ый параметр – имя параметра, передаваемый функцию потока.

6ой параметр – идентификатор потока (последний)

Т.е.по данной команде создается поток и запускается на выполнение (начинает выполняться параллельно основному потоку)

сounter = (int\*) lpParameter //приведение типа из void к int

Потоки можно приостанавливать(suspend) , возобновлять (resew) и удалять (terminate)

# ЛК – 8. СИНХРОНИЗАЦИЯ

Проблема тупика (deadlock – смертельный замОк) – имеется ресурс, но его недостаточно (палочки)

Проблема синхронизации связана с недостатком общего ресурса.

|  |  |
| --- | --- |
| Процесс1  File1  |  |  |  File2 | Процесс2  File2  |  |  |  File1 |

Данная ситуация наз.**блокировка.**

Имеется **2 способа решения** проблем.

**1** – предупреждение (запускать процессы по одиночке).Система должна так себя вести, чтобы гарантировать не возникновение тупиков.

+: всегда легче, чем наблюдать

-: потеря быстродействия

**2** – наблюдение (мониторинг) и принятие мер в случае тупика. (какой-то процесс снять и дать завершиться другому.)

-: более затратный механизм

**Как предупредить?**

Использовать критические секции.

**Критическая секция** – системная переменная – кусок кода, где процесс захватывает общий ресурс (файл). Не может использоваться более чем 1 потоком. (в табличке – в квадратике)

EnterCriticalSection(&cs) – вход в критическую секцию

LeaveCriticalSection(&cs) – выход из критической секции

InitializeCriticalSection(&cs) – создать критическую секцию

**Семафоры** имеют много общего с критической секцией, но несколько более развиты.

**Семафоры** – переменные (условия), которые разрешают или блокируют вход в критическую секцию. Каждый процесс может устанавливать семафор в запрещенное состояние или сбрасывать его.

В отличии от критических секций семафоры позволяют в общем случае заходить в охраняемый фрагмент кода более чем 1 потоку (процессу).

Если > 0, вычитаем единичку и заходим в охраняемую зону. Если семафор обнулен – доступа нет.

Семафоры в терминологии системного программирования известны как мьютексы (mutex)

**Мониторинг** – если ресурс захвачен, то рисуем стрелку от ресурса к процессу, а если требуется, но не захвачен, то рисуем стрелку в обратном направлении. (картинка в телефоне 4 П и 3 r)

Если данному процессу нужен ресурс, который удерживает другой процесс.

Например, П1 нужен ресурс r3, а он удерживается ресурсом П3. Значит, проведем стрелку от П1 к П3. Далее, п3 нужен р1, а он удерживается П1. И так далее.

Если имеется блокировка, то во втором графе, где нету ресурсов существует замкнутый цикл. Наша задача – обнаружить цикл. (в данному случае П1-П3).

**Как найти цикл математически?**

Алгоритм основан на последовательном разрушении графов.

Что будем делать? Будем выбрасывать вершины до те пор пока это возможно. Если выбросить не будет возможности, а вершины еще есть, то мы говорим, что есть цикл (тупик).

Какой критерий отбрасывания вершин? Критерием отбрасывания вершины является наличие у вершины либо только выходящих дуг, либо только входящих.

В нашем случае, можем выбросить П2 (выходящие только из нее).

Т.к.П2 больше нету – выбрасываем П4 (только выходящая дуга)

Остается П1 и П3 – цикл.

**Механизм предупреждения тупиков для последовательности захватываемых ресурсов**.

Будем представлять каждый процесс в виде последовательности шагов.

П1: R1, R3, R1 (нужны следующие ресурсы)

П2: R3, R2, R1

П3: R2, R4

Неизвестно когда понадобится следующий ресурс – нужен механизм предупреждения блокировок.

(картинка с кружочками)

Если процесс захватывает ресурс, то помечаем внутрь фишку. Наличие фишки = захвачен.

2 типа фишек (1 замазана полностью, 2 – тень (занят))

Механизм:

Назначаем процесс на ресурс только при условии, что тупик не возникает и не возникнет в последующем. Как мы узнаем, что он не возникнет?

Данный алгоритм нуждается в «проводнике» («иван сусанин»). Проводник будет выводит фишки из системы. Если ему удастся вывести фишки – исходная ситуация не заблокирована и к блокировкам не приведет. Если не удастся – исходная ситуация помечается как запрещенная.

Будем применять проводник с принципом работы FIFO (First input, first output – первый зашел, первый вышел).

В первую очередь наш проводник пытается двигать более ранние фишки до тех пор пока это возможно. Как только первая фишка затыкается – двигаем следующую. Вторая затыкается – опять первую.

В нашем случае не было блокировки. Проводник всех вывел. Значит, можно разрешить исходную ситуацию. Следовательно, ситуация нормальная.

Как организовать проводника более эффективно?

Более эффективный проводник всегда «прогоняет» большее количество процессов. Т.е. система должна быть насыщена фишками.

**Механизм почтовых ящиков.**

В этом случае процессы обмениваются сообщениями о возможности освободить тот или иной ресурс, разрешить другому процессу захватывать тот или иной ресурс.

Этот механизм старый и неэффективным – не прижился.

В настоящее время прижились более простые механизмы – критические секции, семафоры.

# ЛК – 9. ВЗАИМОДЕЙСТВИЕ МЕЖДУ ПРОЦЕССАМИ

Имеются разные способы взаимодействия. Мы изучили критические секции. Но есть другие варианты. Например, взаимодействие на основе протоколов обмена. В данном случае будем использовать протокол TCP (Transport Communication Product). В этом случае говорим о взаимодействии между сокетами. Кроме TCP есть протокол HTTP, UDP (User Datagram Protocol) и другие.

Отличие

UDP быстрый, но ненадежный.

TCP медленней, дешевле, надежный.

**Протокол** – порядок (алгоритм) взаимодействия клиента и сервера, включая сюда форматы передаваемых данных и необходимую управляющую информацию.

Как узнать, что идет передача?

Существует **маркер начала сообщения** – 64 единицы подряд означают начало сообщения. Дальше идет адрес приемника, адрес отправителя, длина сообщения, число попыток повторной передачи, блоки данных. Кроме данных передаются **контрольные биты** для проверки ошибок передачи и восстановления искаженных данных. Также может быть **уровень приоритета** сообщения.

**Клиент-сервер**

Для работы в локальной сети нужно создать два раздельных проекта – один для сервера, другой – для клиента. Сервер создается первым и должен быть запущен. Между клиентом и сервером устанавливается соединение. Инициатива соединения всегда на стороне клиента. Сервер переходит в режим прослушивания и ждет подключения клиента.

Подробное описание работы клиента и сервера, основных функций приводится в лекции 3 курса «программирование и безопасность сетевых приложений».

**2-ой способ связи** – соединение **на основе канала** (pipe). В отличии от сокетного соединения pipe ориентирован на связь процессов, работающих на одном процессоре.

Pipe устанавливается между 2 процессами, работающими в системе.

Работа через pipe = работа через файл.

**Именованный канал**

**Именованный канал** – объект ядра ОС, который обеспечивает обмен данными между процессами, выполняющимися на компьютерах в одной локальной сети. Процесс, создающий именованный канал, называется **сервером именованного канала.** Процессы, которые связываются с именованным каналом называются **клиентами именованного канала.**

Процесс, который использует команду CreatePipe можно рассматривать как сервер, а процесс, который использует CreateFile – клиент.

Команда WriteFile позволяет записать текст (последний параметр – длина строки)

Любой канал идентифицируется своим именем, которое задается при создании канала.

**Типы** именованных каналов:

\*дуплексные – данные передаются в обе стороны;

\*полудуплексные – данные передаются только в одну сторону.

Передача данных в именованном канале может осуществлять как потоком, так и сообщениями. Обмен данными в канале может быть **синхронным** и **асинхронным.**

Функции подробно описаны в лекции №4 курса «программирование и безопасность сетевых приложений».

**SendMessage** – команда для отправки сообщений в окно. На ПК открыто любое окно (ворд, блокнот, калькулятор) и с помощью сообщения мы можем закрыть окно (ворд, например). Т.е., с помощью SendMessage мы можешь посылать команды в окна, а не в консольное окно.

WndProc **–** читаем сообщение, которое пришло к нам из другого процесса.

# ЛК – 10. WEB-СОКЕТНАЯ СВЯЗЬ

TCP сокеты можно реализовать для web-соединения. Те же принципы сетевого взаимодействия можно перенести на web-связь. В основе связи – протокол TCP. Над ним надстраивается HTTP.

**Серверная часть.**

Инициализируем библиотеку wsadata;

Проверяем не пустая ли она;

getadrrinfo(“127.0.0.1”, “8000”, &hints, &addr).

Создается сокет с пар-ми (семейство протоколов, тип сокет,

Привязываем сокет к IP-адресу с помощью ф-ции bind

Инициализируем слушающий сокет с помощью listen

Ожидаем входящее соединение – accept.

С помощью функции recv получаем инфу от клиента.

stringstream response; //сюда запис.ответ клиенту

stringstream response\_body; //тело ответа клиенту

Как формируется боди?

Response\_body << <title> Test C++ HTTP Server <\title>

<< <h1>…..<\h1>

<< …….;

Response << “HTTP/1.1 200 OK \r\n”

<< “version: HTTP/1.1\r\n”

<< “Content-Type: text/html; utf-8”

<< “Content-Length: “ << response\_body.str().length()

<< response\_body.str;

**ПРИНЦИПЫ УПРАВЛЕНИЯ**

**Блок управления** процессора реализует функции управления процессом выполнения команд.

Кэш память используется для того, чтобы подгружать команды и данные заранее, а не обращаться каждый раз к шине. Т.е. пока блок управления выполняет команды кэш заполняется следующими.

Имеется 2 способа управления:

1. Микропрограммный
2. Схемный (автоматный)

**Смешанная архитектура** – процессоры RISC (Restricted Instruction Set Computer), т.е процессоры с ограниченным набором команд (часть команд реализуется аппаратно, в часть схемно).

Первоначально было известно 2 операции: умножение (есть суммирование несколько раз) и деление (есть вычитание несколько раз)

Для вычитания числа переводят в дополнительный код.

Как это делают?

Пример1:

6 – 4 = 2;

Ведущий 0 означает, что число положительное.

0110 = 6;

0100 = 4;

Для формирования доп.кода биты заменяют на противоположные и добавили 1 в младший бит.

4 = 1011 + 1 = 1100;

Теперь не вычитаем, а складываем 6 + 4.

0110 + 1100 = 1| 0010; //1 выходит

Пример2:

9 – 5 = 4;

1001 = 9;

0101 = 5;

5 в доп.коде:

1010

+ 1

1011

Получается, 9 + 5:

1001 + 1011 = 1| 0100; //1 выходит

0100 = 4;

Аппаратное управление (тем же сумматором) требует подачи управляющих и синхронизирующих сигналов на различные ходы. Например, чтобы записать операнд на регистр, необходимо подать сигнал записи. Чтобы обнулить регистр – нужно подать сигнал очистки. Т.е. эти управляющие сигналы выдает устройство управления.

Каждая команда (сложение, умножение, деление, сдвиг, логических переход и т.д.) характеризуется своей собственной последовательностью управляющих сигналов. Но эти сигналы выдаются в определенном порядке.

Пример:

y1 – записать 1ый операнд в 1ый регистр

y2 – записать 2ой операнд в регистр 2

y3 – начать сложение

y4 – записать результат в выходной регистр

y5 –назначить адрес для сохранения в памяти

y6 –записать результат в память

Как их нужно вызывать?

|  |  |  |
| --- | --- | --- |
| y1 | y2 | микропрограмма |
|  | |
| y3 | |
|  | |
| y4 | y5 |
|  | |
| y6 | |

Эта схема называется микропрограмма. Прямоугольники – микрокоманда, а стрелочки – микрооперации. Микропрограммы постоянно находятся в ПЗУ (Read Only Memory). Каждый горизонтальный провод в ПЗУ – микрокоманда. Вертикальная– микрооперация.

**Микропрограммное управление** – универсальный вид управления. Главный недостаток – микропрограммное управление медленное.

Команды делятся на 2 типа: операционные и условные (логические).

DC (дешифратор) – на вход подается номер в виде 0101, а на выходе снимается сигнал с того выхода, который соответствует номеру (в нашем случае 5).

# ЛК – 11. СЖАТИЕ И ВОССТАНОВЛЕНИЕ ДАННЫХ

Одной из задач системы является обеспечение сжатия данных и их восстановление в случае ошибок.

Архивация данных (сжатие) – основано на избыточности и использует наиболее часто используемые части кода для чего-то там.

**Метод Хафмана** (сжатие по Хафману)

Возьмем последовательность битов: 100100001000001

Как это сжать? Т.е. записать более короткой последовательностью?

В методе Хафмана исх.последовательность разбивается на триады (тройки)

Получим: 100 100 001 000 001

Если не получается разбить на тройки – слева дописываем нули.

Запишем комбинации:

100

001

000

Посчитаем сколько раз появилась каждая комбинация, т.е. частоты :

100 – 2

001 – 2

000 – 1

Следующий шаг – упорядочивание, т.е. выписываем наши комбинации в порядке убывания частот (можно оставить как есть у нас).

Теперь начинается сам метод:

1. Объединяем 2 последние комбинации

001 – 2

000 – 1

Это будет новая комбинация с суммарной частотой 3

1. Переписываем комбинации в порядке убывания частот.

[001,000] – 3

100 - 2

1. Объединяем 2 последние комбинации

100

001

000

1. 2 фаза: разметка ребер дерева (0 и 1 – в телефоне)
2. 3 фаза: кодировка (двигаемся из конечного узла в ребра комбинационным кодом)

100 => 0

001 => 10

000 => 11

Это и есть результат сжатия – кодировочная таблица.

Теперь можем записать нашу последовательность уже в сжатом в виде с помощью кодировочной таблицы: 0 0 10 11 10

Получилось почти двукратное сжатие.

Далее можно повторно сжать. Но есть риск проиграть или ничего не достичь.

Восстановить исходную последовательность:

1. Заходим в кодировочную таблицу

Первый 0 => заменяем 100

Второй 0 => заменяем 100

Третий 1, но это не однозначно, берем еще один бит 10 => 100

Пятый 1, что сужает область поиска, а шестой бит – 1 => 11 => 000

Седьмой бит 1, восьмой 0 => заменяем 001

**Метод Шеннона.**

Шеннон – создатель теории информации.

Первичная информация – свойство материи.

CMYK формат отображения.

Косинусное преобразование.

Квантование (маленькие числа заменяются нулями; и эти блоки из 0 кодируются 5(0), где 5 – сколько нулей подряд идет (множитель) перед следующей 1).

Теорема Котельникова.

# ЛК – 12 ВОССТАНОВЛЕНИЕ ДАННЫХ

**(Продолжение лк 11)**

**Метод Хэмминга:**

2k = m+k;

где m – число основных битов

k – число контрольных битов

**Циклический код :**

Пришел на смену метода Хэмминга. Применяется при передаче сообщений по сети, т.к.его проще технически реализовать, просчитать + более экономный.

При передаче по сети формат сообщения для TCP:

1. Преамбула (64 подряд стоящие 1)
2. Служебная инфа (адрес получателя, адрес отправителя, длина сообщения в байтах, количество проб передач, время таймаута, флажки)
3. Данные (контрольные биты)
4. Концовка (64 подряд стоящие 1)

x1 x2 x3 x4 p1 p2 p3

1 0 0 1 0 1 1

x3 x2 x1 x0

1\*x3+0\*x2+0\*x1+0\*x0 = x3 + 1;

Рассматривают так называемые **первичные** полиномы – такие многочлены, которые нельзя разложить на множители, кроме его самого и 1 (единицы). Чтобы получить доп.биты необходимо исх.многочлен \* xk, где k – кол-во контр.битов и разделить на первичный полином => получить остаток (этот остаток и есть контрольные биты).

((x3 + 1) \* x3 ) / (x3 + x2 + 1) =

(x6 + x3 ) / (x3 + x2 + 1) =

**x3 + x2+x+1** (ост. (х+1))

х+1 соответствует двоичной последовательность 011.

Если в остатке не 0 – получили ошибку (при исправлении ошибки).

Пока в остатке 2 и более единиц, выполняем циклический сдвиг.

Когда в остатке получится одна единственная 1 (она должна быть на последнем месте) – процесс сдвигов закончен.

Первая фаза закончена. Теперь добавляем единичку в последний разряд (по модулю 2) и делаем сдвиги (4 раза как в прошл).

1011101 + 1 = 1011100

В рез-те сдвигов из 101100 получим 1001011.

**Шифрование**

**Метод RSA**

x = 5;

Y = xe mod M

e – открытый ключ.

M – вспомогательное число, кот.дб взаимно простым с открытым ключом

e = 11, M = 17;

511 mod 17 = 52\*52\*52\*52\*52\*5 mod 17 = 8\*8\*8\*8\*8\*5 mode 17 = 64\*64\*40 mod 17 = 13\*13\*6 mod 17 = 26\*39 mod 17 = 9\*5 mod 17 = 11;

Отправляем 11.

Как преобразовать обратно?

Для этого используется секретный ключ d.

d\*e mod фи(M) = 1;

фи(17) = 17 – 1 = 16;

методом подбора м.опред., что d=3;

X = yd mod M;

113 mod 17 = 11 \* 11 \* 11 mod 17 = 2\*11 mod 17 = 5

Получается x=5 (совпало)

# ЛК – 13. Введение в HTTP