1. Функции ФС. Низкоуровневое и высокоуровневое форматирование. Разделы. Логическое устройство, лог. диск, кластер.

Основные функции любой файловой системы нацелены на решение следующих задач:

- именование файлов;
- программный интерфейс работы с файлами для приложений;
- отображения логической модели ФС на физическую организацию хранилища данных;
- организация устойчивости ФС к сбоям питания, ошибкам аппаратных и программных средств;
- содержание параметров файла, необходимых для правильного его взаимодействия с другими объектами системы (ядро, приложения и пр.).

В многопользовательских системах появляется ещё одна задача: защита файлов одного пользователя от несанкционированного доступа другого пользователя, а также обеспечение совместной работы с файлами, к примеру, при открытии файла одним из пользователей, для других этот же файл временно будет доступен в режиме «только чтение».

Кластер - единица дискового пространства, в которой работает ФС (ОС). Обычно в кластере 1, 2, 4 сектора. Фс выделяет пользователю кластеры, а с ЖД обменивается секторами.

Для начала использования ЖД нужно провести низкоуровневое форматирование: распределить пространства на полезные участки и промежутки, разбиение на секторы. Низкоуровневое форматирование не зависит от ОС, Границы блоков отмечаются специальной информацией.
Затем следуем высокоуровневое форматирование, на котором определяется размер кластера, записывается инфа об ОС и ФС:

доступность пространства
границы областей под файлы и каталоги
инфа о поврежденных областях
загрузчик (компактная программа для инициализации ОС)

Высокоуровневое форматирование зависит от ФС

Низкоуровневое форматирование (Low level format)— операция, в процессе которой на магнитную поверхность диска наносится служебная информация, которая используется для позиционирования головок жёсткого диска. Выполняется в процессе изготовления жёсткого диска, на специальном оборудовании, называемом ***серворайтером***.

Низкоуровневое форматирование — это процесс нанесения информации о позиции треков и секторов, а также запись служебной информации для сервосистемы. Этот процесс иногда называется "настоящим" форматированием, потому что он создает физический формат, который определяет дальнейшее расположение данных. Когда в первый раз запускается процесс низкоуровневого форматирования винчестера, пластины жесткого диска пусты, т.е. не содержат абсолютно никакой информации о секторах, треках и т.п. Это последний момент, когда у жесткого диска абсолютно пустые пластины. Информация, записанная во время этого процесса, больше никогда не будет переписана.

Высокоуровневое форматирование— процесс, который заключается в создании главной загрузочной записи с таблицей разделов и (или) структур пустой файловой системы, установке загрузочного сектора и т.п. В процессе форматирования также проверяется целостность носителя для блокировки дефектных секторов.

Известен также способ без проверки носителя, который называется «*быстрое форматирование*».

После завершения процесса низкоуровневого форматирования винчестера, мы получаем диск с треками и секторами, но содержимое секторов будет заполнено случайной информацией. **Высокоуровневое форматирование** — это процесс записи структуры файловой системы на диск, которая позволяет использовать диск в операционной системе для хранения программ и данных.

Раздел — часть долговременной памяти жёсткого диска, выделенная для удобства работы, и состоящая из смежных блоков. Всего разделов на физическом диске может быть не более 4-х. Из них не более одного расширенного(дополнительного) раздела

Виды логических разделов:

Первичный (основной) раздел - В ранних версиях Windows *первичный раздел* обязательно должен был присутствовать на физическом диске первым. Соответственно, эти ОС могли быть установлены только на первичный раздел. Этот раздел всегда содержит только одну ФС. При использовании MBR (первые 512 байт - устройства хранения данных), на физ. диске может быть до 4 первичных разделов.

Расширенный (дополнительный) раздел - Основная таблица разделов может содержать не более 4 первичных разделов, поэтому был изобретён *расширенный раздел*. Это первичный раздел, который не содержит собственной ФС, а содержит другие логические разделы.

Кластер(cluster)— в некоторых типах ФС логическая единица хранения данных в таблице размещения файлов, объединяющая группу секторов.

2. Функции генерации сигнала (POSIX): имена, аргументы, принцип действия

Функции генерации сигналов. Для генерации сигналов в Unix предусмотрены две функции – kill(2) и raise(3). Первая функция предназначена для передачи сигналов любым процессам, к которым владелец данного процесса имеет доступ, а с помощью второй функции процесс может передать сигнал самому себе. Как это обычно принято в мире Unix, семантика вызова функции kill()совпадает с семантикой одноименной команды ОС. У функции kill()два аргумента –PID процесса-приемника и номер передаваемого сигнала. С помощью функции kill(), как и с помощью одноименной команды можно передавать сообщения не только конкретному процессу, но и группе процессов.

Таблица 2 демонстрирует поведение функции kill()в зависимости от значения PID:

PID > 1	Сигнал посылается процессу с соответствующим PID.
PID == 0	Сигнал посылается всем процессам из той же группы что и процесс-источник.
PID < 0	Сигнал посылается всем процессам, чей идентификатор группы равен абсолютному значению PID.
PID == 1	Сигнал посылается всем процессам системы.

Из методички Душутиной:

Кроме того, сигнал может быть отправлен процессу либо ядром, либо другим процессом с помощью системного вызова kill():

#include int kill(pid_t pid, int sig);

Аргумент pid адресует процесс, которому посылается сигнал.

Аргумент sig определяет тип отправляемого сигнала.

С помощью системного вызова kill() процесс может послать сигнал, как самому себе, так и другому процессу или группе процессов. В этом случае процесс, посылающий сигнал, должен иметь те же реальный и эффективный идентификаторы, что и процесс, которому сигнал отправляется. Разумеется, данное ограничение не распространяется на ядро или процессы, обладающие привилегиями суперпользователя. Они имеют возможность отправлять сигналы любым процессам системы.

Аналогичное действие можно произвести из командной строки в терминальном режиме, используя команду интерпретатора kill pid

К генерации сигнала могут привести различные ситуации:

1. Ядро отправляет процессу (или группе процессов) сигнал при нажатии пользователем определенных клавиш или их комбинаций.

2. Аппаратные особые ситуации, например, деление на 0, обращение недопустимой области памяти и т.д., также вызывают генерацию сигнала. Обычно эти ситуации определяются аппаратурой компьютера, и ядру посылается соответствующее уведомление (например, виде прерывания). Ядро реагирует на это отправкой соответствующего сигнала процессу, который находился в стадии выполнения, когда произошла особая ситуация.

3. Определенные программные состояния системы или ее компонентов также могут вызвать отправку сигнала. В отличие от предыдущего случая, эти условия не связаны с аппаратной частью, а имеют программный характер. В качестве примера можно привести сигнал SIGALRM, отправляемый процессу, когда срабатывает таймер, ранее установленный с помощью вызова alarm(). Процесс может выбрать одно из трех возможных действий при получении сигнала:

1. игнорировать сигнал,

2. перехватить и самостоятельно обработать сигнал,

3. позволить действие по умолчанию. Текущее действие при получении сигнала называется диспозицией сигнала.

3. Обращение к файлам из процесса. Этапы, системные таблицы, идентификация файлов и процессов.

Взаимосвязь системных таблиц иллюстрирует логическая схема управления параллельной обработкой 2-х файлов в 2-х процессах, показанная на следующем рисунке.

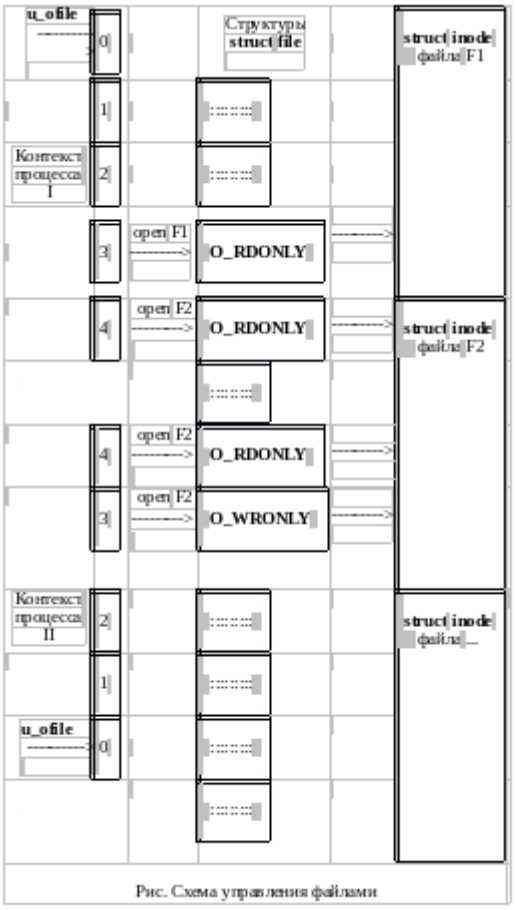


Рис. Схема управления файлами

Процесс I обрабатывает файлы *F2* и *F1*, которые открыты в нем для чтения. Их пользовательские дескрипторы равны **3** и **4** в контексте процесса I. Им соответствуют разные записи в *системной таблице файлов* и различные индексные дескрипторы в резидентной *таблице индексных дескрипторов*. Процесс II обрабатывает только файл *F2*, который открыт в нем отдельно по чтению и записи. Эти варианты доступа индексируются пользовательскими дескрипторами **3** и **4** в контексте процесса II. Им соответствуют разные записи в *системной таблице файлов*, но общий индексный дескриптор, причем тот же, что для файла *F2*, открытого процессом I.

Системная таблица файлов состоит из записей - по одной на каждое открытие файла в одном или различных процессах. Каждая запись системной таблицы файлов формально отражается структурой **struct file**, которая имеет следующие поля:

f_next, f_prev	-	указатели на следующую и предыдущую запись в системной таблице файлов;
f_flag	-	режим открытия файла (запись, дополнение, чтение и т.д.);
f_offset	-	смещение указателей чтение-записи при работе с файлом;
f_count	-	число ссылок на данную запись из таблицы открытых файлов в контекстах процессов;
f_inode	-	ссылка на соответствующий индексный дескриптор в резидентной таблице индексных дескрипторов

Расширение системной таблицы файлов - результат открытия или создания файлов в пользовательских процессах. При завершении обработки любого файла значение поля **f_count** структуры **struct file** в соответствующей записи системной таблицы файлов уменьшается на 1. Запись с нулевым значением поля **f _count** удаляется из системной таблицы файлов и вместе с ней исчезает ссылка (**f_inode**) на соответствующий образ индексного дескриптора в таблице **inode**, уменьшая на 1 значение поля **i_count** в его структуре **struct inode**.

Что касается **идентификации процесса** , то почти во всех ОСях **каждому процессу присваивается числовой идентификатор, который может быть просто индексом в первичной таблице процессов**. В любом случае должно иметься некоторое отображение, позволяющее операционной системе найти по идентификатору процесса соответствующие ему таблицы. Идентификаторы могут использоваться в разных ситуациях.
В частности, они используются для реализации перекрестных ссылок на таблицы процессов из других таблиц, находящихся под управлением ОС. Например, таблицы памяти могут предоставлять информацию об основной памяти с указанием всех областей, выделенных каждому из процессов, указываемому посредством его идентификатора. Аналогичные ссылки могут быть и в таблицах ввода-вывода или таблицах файлов. Если процессы обмениваются между собой информацией, их идентификаторы указывают операционной системе участников такого обмена. При создании нового процесса идентификаторы указывают родительский и дочерние процессы.

4. Принципы организации именованных и неименованных каналов, возможности и ограничения, системные функции, утилиты, примеры применения

Обмен данными между процессами порождает программный канал, обеспечивающий симплексную передачу между двумя процессами (задачами).

Системный вызов: int pipe(int *filedes);

filedes[0] – указатель на запись;

filedes[1] – указатель на чтение;

По сути системный вызов вызывает 2 файловых дескриптора, которые позволяют открыть файл на запись и файл на чтение.

Эти каналы могут использоваться только родственными процессами.

```
#include <stdio.h> <unistd.h>
int main (void)
{
	int fd [2];
	char str[256];
	char pipe_str[256];
	int err = pipe(fd);
	if(err < 0) ОШИБКА СОЗДАНИЯ
	if (!fork()) { СЫН
		printf("Entering SON function\n");
		int input = fopen("input.txt","r");
		if (input <=0) Ошибка
		else {
			printf("opened file: %s\n","input.txt");
			fscanf(input,"%s",str);
			write(fd[1],str,strlen(str));
			printf("wrote data to pipe\n",str);
			close(input);
			close(fd[1]);
		}
		return 0;
		}
	else {
		printf("Entering FATHER function\n");
		read (fd[0],pipe_str,30);
		close(fd[0]);
		printf("read from pipe: %s\n",pipe_str);
	}
return 0;
}
```

FIFO.

В отличие от неименованных каналов (pipe) – возможен обмен данными не только между родственными процессами, так как буферизация происходит в рамках файловой системы с именованием.

Системный вызов: mknode(char *pathname, int mode, int dev);

Правила открытия канала на запись и чтение для независимых процессов:

1) при чтении n байт из N (n<N) системный вызов возвращает n байт, а остальное сохраняет до следующего чтения;

2) при чтении n байт из N (n<N) (n>N) возвращается N и происходит обработка прерывания;

3) если канал пуст, то возвращается 0 байт, если канал открыт на запись, то процесс будет заблокирован до того момента, пока не появится хотя бы 1 байт;

4) если в канал пишут несколько процессов, то информация не смешивается (это достигается путем использования tag’ов);

5) при записи, когда FIFO полон, процесс блокируется до тех пор, пока не появится место для записи.

При использовании FIFO не гарантируется атомарность операций.

При записи FIFO работает, как клиент-серверное приложение.
Функции сервера:

- создание FIFO (mknode());
- открытие FIFO на чтение;
- чтение сообщений;
- закрытие FIFO.

Функции клиента:

- открытие FIFO на запись;
- запись сообщений для сервера в FIFO;
- удаление FIFO.

```
#include <stdio.h>, <stdlib.h>, <unistd.h>, <sys/types.h>, <sys/stat.h>, <fcntl.h>
void server(int input, int output){
        char buf[256];          int num=10;             int count=0;
        char * FileName = "input.txt";
        if( num <0) ОШИБКА ЧТЕНИЯ
        printf("SERVER has read %d bytes\n",num);
        printf("Server has recieved data: %s\n",buf);
}
int main (void){
        int res = mknod ("output",S_IFIFO | 0666,0);
        if (res<0) ОШИБКА СОЗДАНИЯ FIFO
        res = mknod ("input",S_IFIFO | 0666,0);
        if (res<0) ОШИБКА СОЗДАНИЯ FIFO
        int output,input;
        output = open("output",O_WRONLY);
        if (output) OK else НЕ ОК
        sleep(1);
        input = open("input",O_RDONLY);
        if (input) ОК else НЕ ОК
        server(input,output);
        close(output);          close(input);
        if (unlink("input")<0) ОШИБКА
        if (unlink("output")<0) ОШИБКА
}
void client(int input, int output){
        char buf[256];          char FileName[64];
        int num = 10;           int FileID;
        num = read (input,FileName,64);
        printf("recieved file name: %s\n",FileName);
        FileID = fopen(FileName,"r");
        if (FileID<0) ОШИБКА
        int res = fscanf(FileID,"%s",buf);
        fclose(FileID);
        write(output,buf,strlen(buf));
}
int main (void){
        int input, output;
        if ((input = open("output",O_RDONLY))<0) ОШИБКА
        if ((output = open("input",O_WRONLY))<0) ОШИБКА
        client(input,output);
        close(input);
        close(output);
        return 0;
}
```

Различают два типа каналов анонимные (иначе их называют «программные» или **«неименованные»**) и **именованные**. Они по-разному реализованы, но доступ к ним организуется одинаково с помощью обычных функций read и write (унифицированный подход по типу файловой модели).
Одним из свойств программных каналов и FIFO является то, что данные по ним передаются в виде потоков байтов (аналогично соединению TCP). Деление этого потока на самостоятельные записи полностью предоставляется приложению (в

отличие, например, от очередей сообщений, которые автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP).

Неименованные каналы

Программные (неименованные) каналы – однонаправленные, используются только для связи родственных процессов, в принципе могут использоваться и неродственными процессами, если предоставить им возможность передавать друг другу дескрипторы (т.к. имен они не имеют). Неименованный канал создается посредством системного вызова pipe(2), который возвращает 2 файловых дескриптора

/* возвращает 0 в случае успешного завершения. –1 – в случае ошибки:*/

Доступ к дескрипторам канала может получить как процесс, вызвавший pipe() , так и его дочерние процессы. Канал создается одним процессом, может использоваться им единолично (но редко). Как правило, это средство применяется для связи между двумя процессами, следующим образом: процесс создает канал, а затем вызывает fork(), создавая свою копию — дочерний процесс. Затем родительский процесс закрывает открытый для чтения конец канала, а дочерний, в свою очередь, — открытый на запись конец канала. Это обеспечивает одностороннюю передачу данных между процессами. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону.

Именованные каналы

Именованные каналы в Unix функционируют подобно неименованным — они позволяют передавать данные только в одну сторону. Однако в отличие от неименоаванных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO. Аббревиатура FIFO расшифровывается как «first in, first out» — «первым вошел, первым вышел», то есть эти каналы работают как очереди. После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции open, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, fopen). FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись одновременно, поскольку именованные каналы могут быть только односторонними.

5. Принципы организации очередей сообщений, возможности, ограничения

Сообщения обслуживаются ОС, являются разделяемым системным ресурсом, являются частью ядра ОС, образуют очереди, которые образуют список. Сама очередь имеет уникальное имя – идентификатор.

Процессы могут читать сообщения из различных очередей.

Атрибуты сообщений:

- тип сообщения (мультиплексный/немультиплексный);
- длина сообщений (байт);
- сами данные (м.б. структурированы).

Хранится очередь в виде однонаправленного односвязного списка. Для каждой очереди ядро создает свою структуру.

msgid_ds ---------- ipc_perm
msg_perm (permission – права доступа)
msg_cbytes (число байт)
msg_num (число сообщений в очереди)
msg_first (первый объект в списке)
msg_last (последний в списке)

Функции:
msg_snd() – отправка сообщения;
msg_rcv() – прием сообщения;
msg_get()– прием сообщения;
msg_ctl()– прием сообщения.

Очередь сообщений находится в адресном пространстве ядра и имеет ограниченный размер.

В отличие от каналов, которые обладают теми же самыми свойствами, очереди сообщений **сохраняют границы сообщений**. Это значит, что ядро ОС гарантирует, что сообщение, поставленное в очередь, не смешается с предыдущим или следующим сообщением при чтении из очереди. Кроме того, **с каждым сообщением связывается его тип**. Процесс, читающий очередь сообщений, может отбирать только сообщения заданного типа или все сообщения кроме сообщений заданного типа.

Очередь сообщений можно рассматривать как связный список сообщений. Каждое сообщение представляет собой запись, очереди сообщений автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP. Для записи сообщения в очередь не требуется наличия ожидающего его процесса в отличие от неименованных каналов

В результате выполнения программы новое сообщение не записывается, пока предыдущее не прочитается всеми потоками-читателями. Порядок чтения для потоков-читателей произволен.

6. Взаимосвязь обработчиков событий в ОС: при наличии аппаратных событий, внутренних прерываниях, исключениях

Обработка прерываний. Последовательность прог и апп средств для обработки.

1) Прерывание от таймера – вызывает смену задач в мультизадачной ОС;

2) Управление вводом/выводом, потоками данных;

3) Системные вызовы обеспечиваются прерываниями и позволяют менять пользовательский режим на привилегированный.

Прерывания:

- внутренние – генерируются процессором при возникновении аварий в результате выполнения некоторых инструкций, ошибки четности;
- аппаратные (внешние) – прерывания от ПУ, контроллера прерываний:
- *Маскируемые–генерируются контроллером прерываний по запросу ПУ;
- *Немаскируемые (ошибка обращения к памяти, ошибка сети и т.д.) - формируются системной платой.
- программные.

Классификация прерываний зависит не только от процессора, но и от другой аппаратуры (наличие/отсутствие контроллера прерываний).

Программные прерывания - оптимизированные процедуры ОС, вызываемые пользователем или системными программами. Эти модули используются много и часто. Для обработки прерывания необходимо сохранение в стеке значения регистра состояния процессора, а после выхода из прерывания – его восстановление. Программное прерывание не является истинным прерыванием и возникает при вызове соответствующей команды (int – Intel, trap – Motorola…). Команда прерывания зарезервирована в системе команд процессора.

Идея использования программных прерываний:

- использование компактного кода;
- переход из пользовательского режима в привилегированный.

Программные прерывания используются для выполнения ограниченного числа вызовов функций ядра ОС, которые называются <u>системными вызовами.</u>

Приоритеты: относительный и абсолютный.

Аппаратные прерывания обрабатываются драйверами соответствующих устройств.

Исключения (внутренние прерывания) обрабатываются специальными модулями ядра.

Программные прерывания обрабатываются модулями ОС и соответствующими системными вызовами.

Аппаратная поддержка прерываний.

2 способа реализации прерываний на шине:

- векторный;

- голосование - динамический приоритет.

В обоих случаях передается инфа о приоритетах. **С каждым уровнем прерываний может быть связано несколько устройств и несколько обработчиков.** При векторном способе передаются еще и адреса начала программ обработки прерываний. При опросе производится вызов всех обработчиков прерываний данного уровня приоритета, после чего производится ожидание подтверждения тем обработчиком, который соответствует реальному устройству.

Характеристики

<u>Быстродействие</u>: векторный способ работает быстрее при наличии на шине нескольких устройств.

<u>Гибкость</u>: в зависимости от аппаратных платформ.

В реальности используется смешанный способ: соблюдается векторный способ по отношению к процессору и способ голосования по отношению к шине.

Для физической реализуемости такой структуры необходим посредник, который хранил бы адреса подпрограмм обработки прерываний – это контроллер прерываний, может также присутствовать и контроллер ПДП. Каскадное соединение позволяет использовать много

устройств. Шины, использующие опрашивание – ISA, PCI и др. Векторная шина VME – позволяет процессору общаться с ПУ без посредников.
Программная поддержка – прерывания позволяют процессору быстро реагировать на асинхронные по отношению к вычислительному процессу события.
Для упорядочения событий в некоторых ОС применяется механизм приоритетных очередей – в соответствии с приоритетами источников. Для обеспечения работы данной структуры используется специальный программный модуль – диспетчер прерываний. При возникновении прерывания диспетчер прерываний вызывается первым, временно запрещаются все прерывания, диспетчер выясняет источник возникновения прерывания, сравнивает его приоритет с приоритетом текущего прерывания и принимает решение о прерывании или ожидании его завершения работы текущего прерывания.
По отношению к любому потоку прерывания всегда обладают более высоким приоритетом.
Таким образом, используется двухуровневая система прерываний:
1-ый уровень – уровень диспетчера прерываний;
2-ой уровень – уровень диспетчера потоков.
В качестве стратегии планирования используется алгоритм квантования. Также используются различные схемы приоритетов и различные схемы квантования.

7. Преимущества и недостатки клиент-серверной модели. Способы повышения производительности при ее использовании.

Преимущества:

1)Защита глобальных структур данных серверов и других приложений

2)автономность серверов

3)сервер и приложения пользователя не могут менять данные ядра

4)единообразный доступ к ядру

5)разделение между ОС и клиентом

6)любое количество серверов поддерживает множество сред

Недостатки:

1)падение производительности

Способы повышения производительности:

1)пакетизация

2)сокращения количества обращений к серверу за счет реализации на стороне клиента функций API, которые не используют и не модифицируют глобальные данные и в результате все API функции выполняются в dll на стороне клиента.

3)кеширование данных

8. Унифицированная модель драйвера в NT. Многослойные драйверы.

Требования, которые предъявляются для разработчиков драйверов под NT:

- Драйверы пишутся на языках высокого уровня для обеспечения переносимости.
- Операция IO управляется IRP-пакетами. IRP используются многократно по слоям. Система динамически назначает драйверы для управления доп. новыми.
- Драйверы должны синхронизировать доступ к глобальным данным
- Должны позволять вытеснять потоками более высокого приоритета
- Должны быть прерываемы другими потоками
- Код драйвера должен быть способен выполняться на нескольких процессорах.
- Драйверы должны восстанавливаться после сбоя и выполнять нереализованные операции (в NT 4.0 не реализовано)
- драйвер должен быть реентерабельным
- должен поддерживать мультипроцессирование

Драйверы имеют унифицированную модель интерфейса => диспетчер IO не видит их структуру и детали реализации. При взаимодействии различных драйверов диспетчер играет роль посредника.

Драйверы могут быть:

-
- Однослойные (драйверы байт ориентированных устройств, параллельных порт)
-
- Многослойные (для ЗУ большой емкости, блок-ориентированных устройств)

Пример многослойного драйвера – при работе через SCSI-интерфейс драйвер дисков передает запрос на драйвер SCSI, который реализует обработку. Запрос к дискам реализует драйвер SCSI.

Многослойность структуры облегчает решение задач подсистемы ввода-вывода, таких как простота включения новых драйверов, поддержка нескольких файловых систем, динамическая загрузка-выгрузка драйверов и др.

Плюсы:

Использование слоев позволяет отделить вопросы использования высокоуровневых протоколов от вопросов, связанных с управлением собственно оборудованием. Это позволяет осуществлять поддержку аппаратуры от разных производителей без переписывания больших объемов кода. Многослойная архитектура позволяет повысить гибкость системы за счет использования при одном драйвере, реализующем протокол, сразу нескольких драйверов аппаратуры, подключаемых непосредственно во время работы.

Минусы:

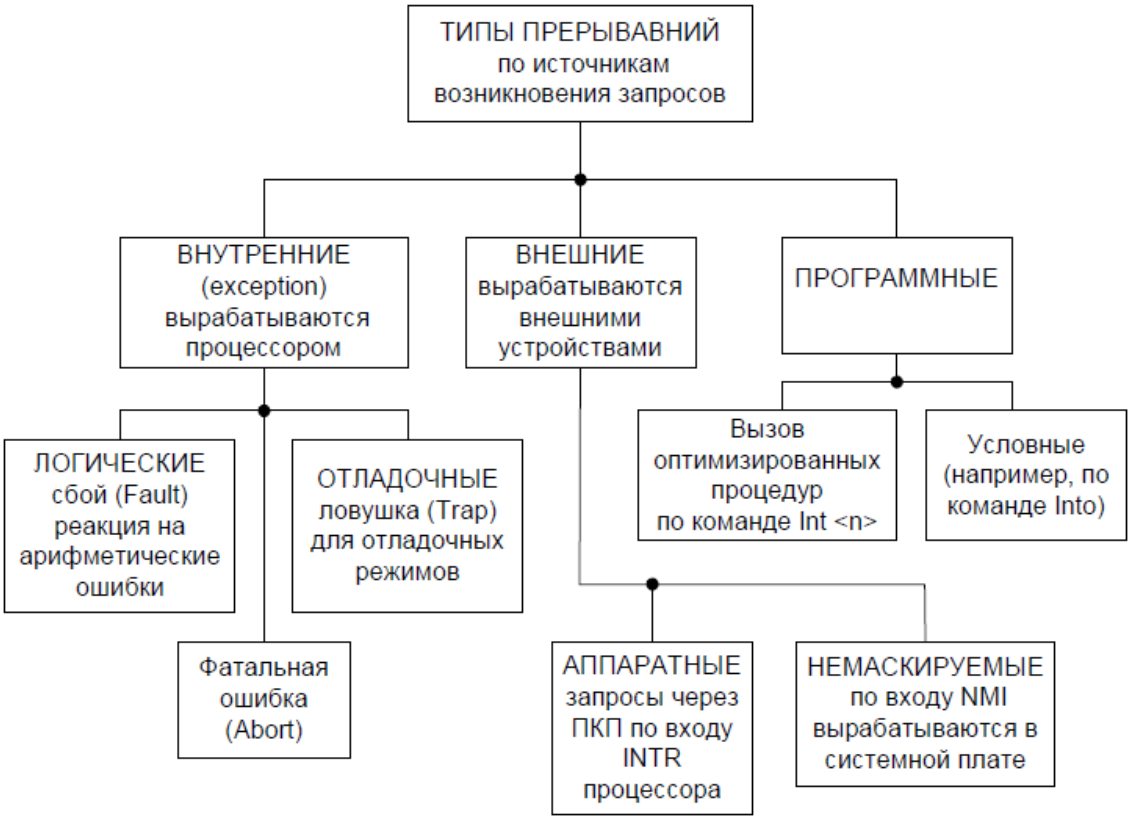
Во-первых, обработка запросов ввода/вывода получает дополнительные накладные расходы, связанные с тем, что пакет IRP пропускается через код Диспетчера ввода/вывода всякий раз, когда он переходит из одного драйвера в другой. В некоторой мере эти затраты могут быть уменьшены путем введения прямого меж-драйверного интерфейса, который будет действовать в обход Диспетчера ввода/вывода.

Потребуются также дополнительные усилия в тестировании для того, чтобы убедиться, что отдельные компоненты получившейся драйверной конфигурации приемлемо стыкуются друг с другом. При отсутствии стандартов это может оказаться весьма болезненным этапом, особенно если драйверы поступают от разных поставщиков. Возникает также не самый простой вопрос совместимости версий разных участников получившейся иерархии.

Наконец, инсталляция многослойных драйверов также становится сложнее, поскольку каждый из них требует для себя соответствующей инсталляционной процедуры. При инсталляции необходимо установить отношения зависимости между разными драйверами в иерархии, чтобы обеспечить их запуск в правильной очередности.

9. ISR, примеры реализации, передача управления стандартному обработчику

ISR = interrupt servicing routine



Контроллер прерываний
Запросы от внешних источников подключены к входам IRQ0-IRQ7 контроллера ; с выхода INT контроллера на вход INTR процессора подается сигнал запроса от наиболее приоритетного источника. При получении ответного сигнала INTA контроллер выдает номер прерывания на шину данных; номер образуется суммой IRQ номера с некоторым базовым значением.
При поступлении заявок от различных источников выполняется заявка с наименьшим номером IRQ.
Имеется возможность выборочно заблокировать заявки от отдельных IRQ входов. Блокировку, или
маскирование заявок, а также выбор заявки с наибольшим приоритетом обеспечивают три байтовых регистра iMR, iRr,iSr.
Виды прерываний: Внешние и внутренние. Внутренние не могут быть замаскированы. Внешние бывают Маскируемые и немаскируемые. Внутренние : Сбой(Fault), Ловушка(Trap), Фатальная
ошибка(Abort).
Примером передачи управления обработчику прерыванй может служит инструкция int x.
Информация об адресе обработчика прерывания хранится в таблице векторов прерывний.

Таблица векторов прерываний используется в архитектуре x86 и служит для определения корректного ответа на прерывания и исключения.

Инструкция int x позволяет напрямую обратиться к процедуре, адрес которой задан в таблице векторов по адресу x.

10. Средства синхронизации Windows

В общем случае все средства синхронизации отличаются как по использованию (только между потоками внутри процесса или еще и между процессами), так и по быстродействию (в работе измерения не проводились).
Самым быстрым средством считается **критическая секция**, т.к. она выполняется в режиме задачи и максимально упрощена.
Мьютексы являются аналогом семафора с двумя возможными значениями. Реализуют взаимное исключение, т.е. одновременно доступ к мьютексу имеет только 1 поток. Мьютексы так же могут быть использованы для синхронизации как потоков, так и процессов.
Семафор позволяет захватить себя нескольким потокам, после чего захват будет невозможен, пока один из ранее захвативших семафор потоков не освободит его. Семафоры применяются для ограничения количества потоков, одновременно работающих с ресурсом (ресурсами). В Windows семафоры и мьютексы могут иметь имя, через которое другие процессы могут получить доступ к объектам. В отличие от синхронизации потоков, принадлежащих одному процессу, при межпроцессной синхронизации в каждом процессе должен быть получен свой дескриптор к одному и тому же объекту ядра.
Критическая секция помогает выделить участок кода, где поток получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. К критической секции одномоментно имеет доступ только 1 поток. Данное средство не может быть применено для синхронизации процессов. Критическая секция не является объектом ядра, и использовать функции семейства Wait() не представляется возможным.
События используются для уведомления ожидающих потоков о наступлении какого-либо события. Существует два вида событий - с ручным и автоматическим сбросом. События с ручным сбросом используются для уведомления сразу нескольких потоков. При использовании события с автоматическим сбросом уведомление получит и продолжит свое выполнение только один ожидающий поток, остальные будут ожидать дальше.
Практически все средства синхронизации (кроме критических секций) используются вместе с ожидающими функциями, которые возвращают управление, если все или часть переданных им объектов свободны.

1) критическая секция – это объект, который принадлежит процессу, а не ядру. А значит, не может синхронизировать потоки из разных процессов. Функции: 1) создание – InitializeCriticalSection(…), 2) удаление – DeleteCriticalSection(…), 3) вход – EnterCriticalSection(…), 4) выход – LeaveCriticalSection(…).
Ограничения: поскольку это не объект ядра, то он не виден другим процессам, то есть можно защищать только потоки своего процесса. Крит. раздел анализирует значение спец. переменной процесса, которая используется как флаг, предотвращающий исполнение некоторого участка кода несколькими потоками одновременно. Самые простые.
2) mutex – объект ядра, у него есть имя, а значит с их помощью можно синхронизировать доступ к общим данным со стороны потоков разных процессов. Ни один другой поток не может завладеть мьютексом, который уже принадлежит одному из потоков. Windows расценивает мьютекс как объект общего доступа, который можно перевести в сигнальное состояние или сбросить. Сигнальное состояние мьютекса говорит о том, что он занят. Потоки

должны самостоятельно анализировать текущее состояние мьютексов. Если требуется, чтобы к мьютексу могли обратиться потоки других процессов, ему надо присвоить имя. Функции: 1) CreateMutex(имя) – создание, 2)OpenMutex(имя) – открытие, 3)WaitForSingleObject(hnd) – ожидание и занятие, 4)ReleaseMutex(hnd) – освобождение, 5)CloseHandle(hnd) – закрытие.
3) семафор – объект ядра “семафор” используются для учёта ресурсов и служат для ограничения одновременного доступа к ресурсу нескольких потоков. Используя семафор, можно организовать работу программы т.о. , что к ресурсу одновременно смогут получить доступ несколько потоков, однако количество этих потоков будет ограничено. Создавая семафор, указывается max количество потоков, которые одновременно смогут работать с ресурсом. Каждый раз, когда программа обращается к семафору, значение счетчика ресурсов семафора уменьшается на единицу. Когда значение счетчика ресурсов становится = 0, семафор недоступен. Функции: 1) создание - CreateSemaphore, 2) открытие - OpenSemaphore, 3) занять - WaitForSingleObject, 4) освобождение - ReleaseSemaphore
4) События - самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и 2 bool переменные: 1ая - тип данного объекта-события, 2ая — его состояние (свободен или занят). Объекты-события используются для уведомления ожидающих нитей о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом. Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких нитей.При использовании события с автосбросом уведомлени получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше.
Функция CreateEvent создает объект-событие, SetEvent - устанавливает событие в сигнальное состояние,
ResetEvent - сбрасывает событие.
5) Условные переменные - само по себе не применяется, а только в сочетании с другими средствами синхронизации (например, с крит. секциями). Условные переменные отсутствуют в Windows 2003 и XP.
Функции: 1) Инициализация — InitializeConditionVariable, 2) ожидания восстановления — SleepConditionVariableCS, 3) Для пробуждения потоков — WakeConditionVariable.

11. Синхронный и асинхронный ввод-вывод в NT. Формат IRP. Проекционный ввод-вывод.

Часть исполнительной системы, получающая запросы от процессов пользователя либо режима пользователя, либо ядра, и передающая эти запросы устройствам I/O в преобразованном виде.
Система ввода/вывода управляется пакетами запроса ввода/вывода (I/O Request Packet, IRP). Каждый запрос ввода/вывода представляется в виде пакета IRP во время его перехода от одной компоненты системы ввода/вывода к другой.
IRP позволяет использовать иерарх. настр. системы I/O и драйверов.
IRP – структура данных, управляющая обработкой выполнения операций на каждой стадии их выполнения.

Синхронный IO - приложение ждет, когда устройство выполнит передачу данных и вернет код статуса по завершении операции ввода-вывода. После этого программа продолжает работу и немедленно использует полученные данные. В таком простейшем варианте Windows-функции *ReadFile* и *WriteFile* выполняются синхронно. Перед возвратом управления они должны завершить операцию ввода-вывода.

Асинхронный ввод-вывод **позволяет приложению выдать запрос на ввод-вывод и продолжить выполнение, не дожидаясь передачи данных устройством.** Этот тип ввода-вывода увеличивает эффективность работы приложения, позволяя заниматься другими задачами, пока выполняется операция ввода-вывода. Инициировав операцию асинхронного ввода-вывода, поток должен соблюдать осторожность и не обращаться к запрошенным данным до их получения от устройства. Следовательно, поток должен синхронизировать свое выполнение с завершением обработки запроса на ввод-вывод, отслеживая описатель синхронизирующего объекта (которым может быть событие, порт завершения ввода-вывода или сам объект «файл»), который по окончании ввода-вывода перейдет в свободное состояние.

Независимо от типа запроса операции ввода-вывода, инициированные драйвером в интересах приложения, выполняются асинхронно, т. е. после выдачи запроса драйвер устройства возвращает управление подсистеме ввода-вывода. А когда она вернет управление приложению, зависит от типа запроса.

Диспетчер I/O – входит в состав системы I/O, управляет передачей запросов I/O в файловую систему, реализует процедуру общего назначения для разных драйверов.
Функции и этапы работы диспетчера: 1) Создание IRP определенного формата 2) Передача пакета соответствующему драйвер 3) Удаление IRP
Функции драйвера: 1) Получение IRP 2) Выполнение операции 3) Возвращение управления I/O, либо реализация завершения
В NT существуют драйверы ФС и драйверы устройств I/O. При любой операции I/O задействованы и драйверы ФС и драйверы устройств. Работают через IRP, контролируются диспетчером. Реализуется общая процедура для различных драйверов. Упрощаются отдельные
драйвера и вносится универсальность.

12. Причины генерации сигнала, примеры, диспозиция. Пример кода (C/C++) с обработкой ошибки деления на ноль.

Причины:

• особые ситуации (например, деление на ноль)
• терминальные прерывания (CTRL + C - вызывает отправление сигнала текущему процессу, связанному с терминалом)
• другие процессы (kill вызывает отправку другому процессу указанного сигнала)
• управление заданиями (Командные интерпретаторы, поддерживающие систему управления заданиями, используют сигналы для манипулирования фоновым и текущими задачами.)
• квоты (Когда процесс превышает выделенную ему квоту вычислительных ресурсов)
• уведомления
• алармы (процесс установил таймер => ему будет отправлен сигнал, когда таймера = 0)
Сигналы не могут быть посланы завершившемуся процессу, находящемуся в состоянии «зомби»

Если pid > 0, это значение трактуется как идентификатор процесса. При нулевом значении pid сигнал посылается всем процессам из той же группы, что и вызывающий. Если значение pid равно 1, адресатами являются все процессы, которым вызывающий имеет право посылать сигналы. При прочих отрицательных значениях pid сигнал посылается группе процессов, чей идентификатор равен абсолютной величине pid

Диспозиции - действия при получении сигнала:

• игнор - signal(номер сигнала, SIG_IGN)
• использование обработчика по умолчанию - (номер сигнала, SIG_DFL)
• использование собственного обработчика – (номер сигнала, имя функции обработчика)
Посылка сигнала — kill (PID, номер сигнала)

Пример кода:

```
#include <stdio.h> #include <fenv.h> #include <signal.h> #include <stdlib.h>
void fpehandler(int sig_num) {
	signal(SIGFPE, fpehandler);
	printf("SIGFPE: floating point exception occured, exiting.\n");
	exit(1);
}
int main() {
	double x, y;
	// ловушка при делении на 0
	int feenableexcept();
	feenableexcept(FE_ALL_EXCEPT);
	signal(SIGFPE, fpehandler);
	x= 0.0;
	y= 0.0;
	x= x/y;
	return 0;
}
```

13. Структура тома NTFS, системные файлы

NTFS делит все полезное место на кластеры — блоки данных,используемые единовременно.
NTFS поддерживает почти любые размеры кластеров - от 512 байт до 64 Кбайт, неким стандартом же считается кластер размером 4 Кбайт.
Диск NTFS условно делится на две части.Первые 12% диска отводятся под MFT зону - пространство, в которое растет метафайл MFT. Запись каких-либо данных в эту область невозможна.MFT-зона всегда держится пустой — это делается для того, чтобы самый главный,служебный файл (MFT) не фрагментировался при своем росте. Остальные 88% диска представляют собой обычное пространство для хранения файлов.
Структуризация: каждый элемент системы представляет собой файл -даже служебная информация. Самый главный файл на NTFS называется MFT, или Master File Table - общая таблица файлов. Именно он размещается в MFT зоне и представляет собой централизованный каталог всех остальных файлов диска. MFT поделен на записи фиксированного размера (обычно 1 Кбайт), и каждая запись соответствует какому-либо файлу.
Первые 16 файлов носят служебный характер и недоступны операционной системе - они называются метафайлами, причем самый первый метафайл -сам MFT. Эти первые 16 элементов MFT - единственная часть диска, имеющая фиксированное положение. Интересно, что вторая копия
первых трех записей, для надежности (они очень важны) хранится ровно посередине диска.
Первые 16 файлов - метафайлы, носят служебный характер. Каждый из них отвечает за какой-либо аспект работы системы. Преимущество настолько модульного подхода заключается в поразительной гибкости - например, на FAT-е физическое повреждение в самой области FAT фатально
для функционирования всего диска, а NTFS может сместить, даже фрагментировать по диску,все свои
служебные области, обойдя любые неисправности поверхности - кроме первых 16 элементов MFT.
Метафайлы находятся корневом каталоге NTFS диска - они начинаются с символа имени"$", хотя
получить какую-либо информацию о них стандартными средствами сложно. Любопытно, что и для
этих файлов указан вполне реальный размер — можно узнать, например, сколько операционная
система тратит на каталогизацию всего вашего диска, посмотрев размер файла $MFT. В следующей
таблице приведены используемые в данный момент метафайлы и их назначение.

- $MFT сам MFT
- $MFTmirr копия первых 16 записей MFT, размещенная посередине диска
- $LogFile файл поддержки журналирования (см. ниже)
- $Volume служебная информация - метка тома, версия файловой системы, т.д.
- $AttrDef список стандартных атрибутов файлов на томе
- $. корневой каталог
- $Bitmap карта свободного места тома
- $Boot загрузочный сектор (если раздел загрузочный)
- $Quota файл, в котором записаны права юзеров на исп. Дискового пространства
- $Upcase файл - таблица соответствия заглавных и прописных букв в имен файлов на текущем томе. Нужен в основном потому, что в NTFS имена файлов записываются в Unicode, что составляет 65 тысяч различных символов, искать большие и малые эквиваленты которых очень нетривиально.

Файлы. Прежде всего, обязательный элемент - запись в MFT, ведь, как было сказано ранее, все файлы диска упоминаются в MFT. В этом месте хранится вся информация о файле, за исключением собственно
данных. Имя файла, размер, положение на диске отдельных фрагментов, и т.д. Если для информации не хватает одной записи MFT, то используются несколько, причем не обязательно подряд.
Каталоги. Каталог на NTFS представляет собой специфический файл, хранящий ссылки на другие файлы и каталоги, создавая иерархическое строение данных на диске. Файл каталога поделен на блоки,
каждый из которых содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который уже
предоставляет полную информацию об элементе каталога. Внутренняя структура каталога представляет собой бинарное дерево.
Поддерживает журналирование и сжатие.

14. Генерация сигнала одним потоком с целью завершения другого потока (POSIX)

```
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
int flag = 0
int t2tokill = 0;
pthread_t t1, t2;
int nit1count = 0;
int nit2count = 0;
void *thread1(void) {
        while(1) {
                printf("\nthread1 - called %d times",nit1count++);
                sleep(5);
                if(flag==0) {
                        pthread_kill(t2, SIGUSR1);
                        flag=1;
                }
        }
}
void *thread2(void) {
        while(!t2tokill) {
        printf("\nthread2 -called %d times",nit2count++);
        sleep(1);
        }
        printf("\nthread2 is killed");
}
void *SigHandlr(int s) {
        t2tokill = 1;
}
int main() {
        signal(SIGUSR1, SigHandlr);
        pthread_create(&t1, NULL, thread1, NULL);
        pthread_create(&t2, NULL, thread2, NULL);
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
}
```

На пятой секунде работы приложения удаляется вторая нить при получении сигнала SIGUSR1.

15. Подробный анализ ресурсоемкости взаимодействия посредством сигналов (на примере IPC POSIX)

Ресурсоёмко, потому что требуется много переключений из пользовательского режима в режим ядра из-за системных вызовов (issig, sendsig).
Для отправки и доставки сигнала требуется системный вызов. Для доставки – прерывание и его обработка. При этом требуется проведение довольно большого числа операций со стеком – копирование пользовательского стека в системную область, извлечение параметров и результатов работы системных вызовов и прерываний. Поскольку объем передаваемой информации при этом способе взаимодействия не велик, а затраты на его реализацию существенны, сигналы считаются одним из самых ресурсоемких способов IPC.
Доставка сигнала происходит после того, как ядро от имени процесса вызывает системную процедуру issig() которая проверяет, существуют ли ожидающие доставки сигналы, адресованные данному процессу.
Функция issig вызывается ядром в трех случаях:
1. Непосредственно перед возвращением из режима ядра в режим задачи после обработки системного вызова или прерывания.
2. перед переходом процесса в состояние сна
3. Сразу же после пробуждения после сна с приоритетом
Если процедура issig() обнаруживает ожидающие доставки сигналы, ядро вызывает функцию доставки сигнала, которая выполняет действия по умолчанию или вызывает спец. функцию sendsig() запускающую обработчик сигнала, зарегистрированный процессом. Функция возвращает процесс в режим задачи, передает управление обработчику сигнала, а затем восстанавливает контекст процесса для продолжения прерванного сигналом выполнения.
Ненадежность (по временному критерию) сигналов.
Сигнал будет доставлен только после выбора процесса планировщиком и выделения ему ресурсов, что нивилирует асинхронный характер сигнала.

17. Способы (низкоуровневые) отключения прерываний на критических участках выполнения кода ОС или приложений

Для отключения прерываний на критических участках кода можно использовать несколько возможностей:
1) замаскировать необходимые прерывания в регистре iMrx контроллера прерываний
IMRx - регистр маски прерываний, служит для хранения, разрешения или запрета (маскирования) прерываний.
2)снять флаг разрешения прерываний процессора(флаг I).

1.Для отключения отдельного устройства в регистр масок контроллера прерываний необходимо записать значения, позволяющее замаскировать порт к которому будет подключено соответствующее устройство. Для этого нужно обратиться к порту контроллера прерываний и записать в него в разряд iMrx единицу для маскирования соответствующего прерывания от устройства.

```
; пример маскирования запросов от КНМГД
mov   al,    01000000b  ; маскирование бита 6
out   21h,   al         ; запись в IMR
    * * *
mov   al,    0          ; очистка IMR в конце
out   21h,   al         ; программы
```

Mov dx, porty//port
mov ax ,08h//маска
out dx, ax//посылаем

Регистр IMR позволяет программно запретить или разрешить прерывания от периферийных устройств (поэтому их иначе называют маскируемыми). Маскирование может быть полным, когда блокируются все прерывания, и частичным, когда маскируются только определенные аппаратные прерывания.

Полное маскирование необходимо в случаях, когда критический участок программы должен быть выполнен целиком прежде, чем компьютер выполнит какое-либо другое действие. Например, при изменении вектора аппаратного прерывания, чтобы избежать выполнения прерывания по не полностью измененному вектору.

Частичное маскирование может потребоваться, когда некоторые прерывания могут взаимодействовать с операциями критичными к временным интервалам. Например, точно рассчитанная процедура ввода/вывода не должна быть прервана длительным дисковым прерыванием.

Следовательно, дисковые прерывания необходимо временно запретить. Такое маскирование часто используют в задачах реального времени.

Регистр IMR обычно используют для маскирования отдельных аппаратных прерываний. Для этого используется вывод в порт с адресом 21h ведущего контроллера прерываний и в порт с адресом A1h ведомого контроллера прерываний, позволяющего маскировать прерывания IRQ8 - IRQ15. По указанному порту устанавливаются в "1" те биты регистра, которые соответствуют номерам запрещаемых вами прерываний.

IMR доступен только по записи. В конце программы его необходимо очистить, иначе обращение к замаскированным вами устройствам будет невозможно и после завершения вашей программы.

2. Для снятия/установки флага I в ассемблере существуют команды cli/sti. Они позволяют заблокировать все внешние прерывания от устройств. Но запрос через дополнительный вход процессора NMI(Non Maskable Interrupt) не блокируется и обслуживается сразу независимо от флага I. Ко входу NMI обычно подключены сигналы фатальных системных ошибок, например сигнал ошибки четности или обращения к памяти.

Чтобы провести полное блокирование аппаратных прерываний обычно используют специальные команды управления битом I в регистре флагов процессора RF: cli (clear interrupt) и sti (set interrupt). Когда флаг I равен 1 - аппаратные прерывания запрещены, когда он равен 0 - разрешены все прерывания, разрешенные в IMR.

Таким образом, управление от процессора более приоритетно по сравнению с управлением от контроллера прерываний .

150

Для очиски (обнуления) I-флага применяют команду STI, а команду CLI для записи в RF 1. При использовании этих команд необходимо соблюдать следующие правила:

- нельзя отключать прерывания на длительный период, так как это влечет за собой нарушение системного времени;
- за командой CLI всегда должна следовать команда STI, иначе не избежно "зависание" компьютера из-за блокировки клавиатуры;
- при создании своих программных прерываний начинайте программу обработки с команды STI, если аппаратные прерывания допустимы.

18. Обработка запроса в ОС при нажатии клавиши, других событиях в системе. Примеры

При нажатии клавиши и отпускании клавиши подается сигнал запроса на вход IRQ1 ведущего контроллера прерываний. Номер прерывания от клавиатуры равен 9. Регистры устройства подключены к микросхеме интерфейса с периферией, в частности код клавиши доступен через порт 060. Чтение принятых данных не влияет на состояние устройства. Причина прерываний при этом не сбрасывается, а данные при повторном чтении остаются теми же. Для начала необходимо заменить стандартный обработчик, с сохранением его адреса. Записать в таблицу векторов прерываний по этому индексу адрес нового обработчика. После завершения работы программы восстановить старый обработчик.

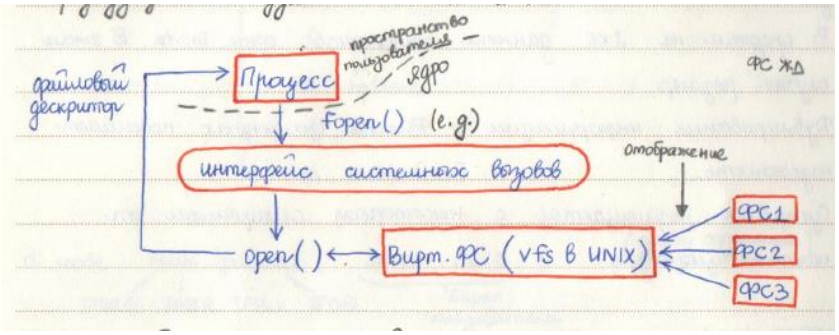
Процедура обработки прерывания должна содержать:

1) определения кода нажатой клавиши
2) выполнение определенного действия в зависимости от кода
3) сброс текущей заявки в контроллер прерываний
4)снятие причины прерывания — посылкой импульса подтверждения в бите 7 порта 061

Пример кода:

```
jmp start:
old_vect            dd       ?
n_vect  equ      9
lea       ax,new
mov dx, cs
mov ds, 0
xchg w [n_vect*4], ax
xchng w [n_vect*4], dx //замена вектора прерывания
mov ds, cs

mov w old_v, ax
mov w old_v, dx

mov ax, w old_v
mov dx, w old_v+2
mov es, 0
es mov w [n_vect *4], ax//воостановление вектора прерывания
es mov w [n_vect*4 +2], dx

mov ax,04c00
int 021

new:
push ax,bx,dx
mov dx, 060
in ax,dx
//обработка в зависимости от кода клавиш
mov dx, 020h//снятие заявки в контроллере прерываний
mov ax,020h
out dx,ax

mov dx, 061h//снятие причины прерывания
mov ax,08h
out dx, ax
pop dx,bx,ax
iret
```

19. Структуры данных ФС UNIX-подобных ОС (vfs, vnode, vfssw): назначение, основной состав



Виртуальная ФС — независимая прослойка между ОС и ФС носителей.
struct vfssw — структура, описывающая имеющийся набор ФС
struct vfs — сама VFS
struct vnode - метаданные файлов

Интеграция ФС устройств в в VFS осуществляется путём монтирования (утилита — mount).

Метаданные всех активных файлов (хотя бы раз упомянутых в каком-либо процессе) представлены в памяти в виде in-core node. В VFS этому соответвсвует vnode.

Структура vnode одинакова для всех файлов независимо от реальной ФС, где хранится этот файл. Vnode содержит инфу для работы ВФС с неизменным хар-ми файла

struct vnode:

- vflag — флаги vnode
- v_count — число ссылок на эту vnode
- filock v_filocks — структура файлов блокировок
- vfs *v_vfsmountedhere - указатель на подключенную ФС
- vfs *v_vfsp — указатель на ФС, где хранится файл
- v_type — тип файла (regular, dir ...)
- v_data — хар-ки реальной ФС
- modeops *v_ops -операции, которые можно производить с файлом

struct vfs:

- vfs *vfs_next - следующая ФС в списке монтирования
- vfsops* vfs_op — допустимые операции применительно к ФС
- vnode* vfs_vnodecovered — ссылка на точку монтирования
- int vfs_flag — флаги (read only, suid, ...)
- int vfs_bsize — размер блока ФС
- caddr vfs_data — хар-ки реальной ФС

Структур vfssw содержит:

- имя типа файла
- адрес на процедуру инициализации
- указатель на внутр. операции ФС
- флаги

Задача vfssw — переключение меджу ФС

20. Генерация сигнала с клавиатуры (нажатием комбинации клавиш, не командная строка) с целью завершения потока в многопоточном приложении (POSIX)

```
#include <stdio.h>	#include <pthread.h>	#include <unistd.h>		#include <signal.h>
#define		N	4
#define OTHER	2
#define COUNT	10
pthread_t threads[N];
int wasJoint = 0;
void handler(){
		puts("^C - signal received");
		signal(SIGINT, SIG_DFL);
		pthread_cancel(threads[OTHER]);
		wasJoint = 1;
}
void * threadUsual (void* ptr) {
		for (int count = 0; count < COUNT; count++) {
			printf("Usual thread # %ld cnt = %d\n", ((long)ptr), count);
			sleep(1);
		}
		return NULL;
}
void* threadOther (void* ptr) {
		for (int count = 0; count < COUNT; count++){
			printf("Other thread  cnt = %d\n", count);
			sleep(2);
		}
		return NULL;
}
int main(void) {
		signal(SIGINT, handler);
		int i;
		for(i = 0; i < N; i++){
			if(i != OTHER)
			{	if(pthread_create(&threads[i], NULL, threadUsual, i))
					exit(i);
				printf("Usual thread created # %i", i);
			} else{
				if(pthread_create(&threads[i], NULL, threadOther, i))
					exit(i);
				printf("Other thread created # %i", i);
			}
		}
		for (i = 0; i < N; ++i)
			if(i != OTHER)  pthread_join (threads[i], NULL);
		if(wasJoint)  pthread_join(threads[OTHER], NULL);
		return 0;
}
```

Идея в следующем: порождаем всего N потоков, каждый поток в консоль выводит раз в секунду свой счетчик, поток с номер OTHER выводит свой счетчик раз в две секунды. По нажатии комбинации клавиш cntr + C - порождении сигнала SIGINT мы завершаем поток с номером OTHER. Каждый поток досчитывает и джоиниться.

21. Объектная модель системы ввода-вывода в NT. Пример открытия файлового объекта.

Система ввода/вывода имеет три особенности:

– объектная модель – управление пакетами;

– унифицированная модель драйвера;

– асинхронная обработка.

Объект – ***драйвер и объект*** – ***устройство.*** Когда поток открывает описатель файлового объекта, диспетчер ввода/вывода по имени этого объекта определяет к какому драйверу он должен обратиться для обработки запроса. Для этого используют следующие объекты:

· Объект-драйвер, представляющий в системе некоторый драйвер и хранящий для диспетчера ввода/вывода адреса вс□ех процедур распредел□ения драйвера (точек входа).

· Объект-устройство, представляющее физическое, логическое или виртуальное устройство, имеющееся в системе.

Объект-драйвер создается диспетчером при загрузке драйвера в систему. После чего диспетчер вызывает процедуру инициализации драйвера, которая заполняет объект точками входа драйвера. Процедура инициализации создает по одному объекту-устройству для каждого устройства, управляемого данным драйвером. Она связывает объекты-устройства с объектом-драйвером. Объект-устройство содержит обратный указатель на объект-драйвер. Часто с объектом – драйвером связано несколько объектов-устройств.

Использование объектов для хранения информации о драйверах избавляет диспетчера ввода/вывода от крайне важности знать детали отдельных драйверов.

В Windows NT программы осуществляют ввод-вывод в виртуальные файлы, манипулируя ими посредством файлового объекта.

Система ввода/вывода управляется пакетами. Каждый запрос ввода/вывода представляется в виде пакета запроса ввода–вывода. Диспетчер не управляет вводом-выводом, он создает пакет запроса ввода/вывода, передает пакет соответствующему драйверу и удаляет его. Диспетчер предоставляет общие для драйверов устройств процедуры. Драйверы обращаются к этим процедурам во время ввода/вывода. Благодаря объедин□ению общих задач в диспетчере драйверы устройств становятся компактными.

В *Windows NT* как и в ОС *UNIX* программы осуществляют ввод-вывод в виртуальные файлы, манипулируя ими посредством файлового объекта.

Все источники или приемники ввода/вывода представлены файловыми объектами.

Потоки пользовательского режима вызывают базовые сервисы файлового объекта *NT* для чтения из файла, записи в файл.

Диспетчер ввода/вывода преобразует эти запросы к виртуальным файлам в запросы к настоящим файлам, физическим устройствам.

Унифицированная модель. Каждый драйвер – это автономный компонент, который может добавляться к ОС и удаляться из нее. Диспетчер определяет модель драйверов I/O:

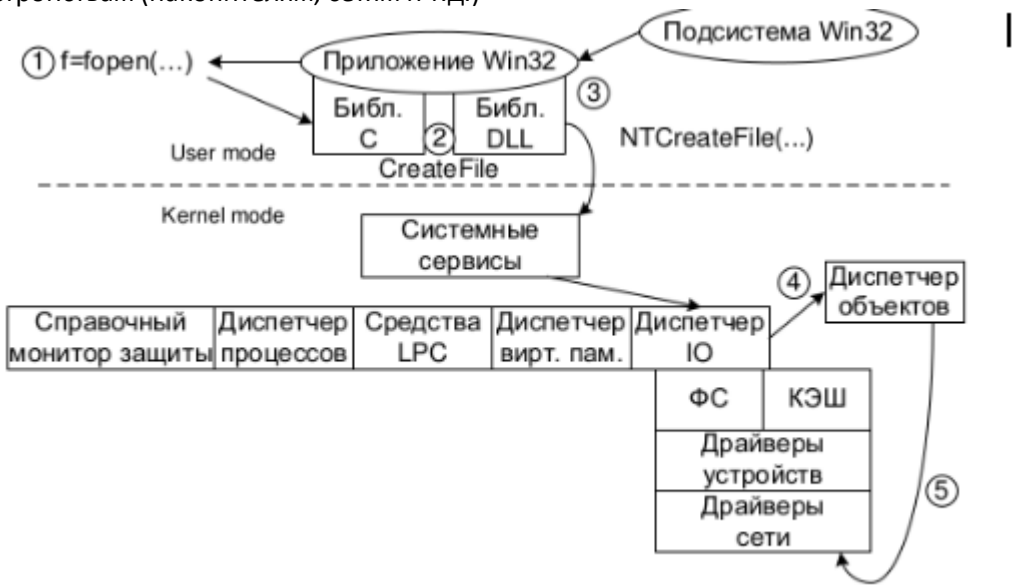
· драйверы переносимы и написаны на языках высокого уровня;

· операции ввода/вывода управляются пакетами запросов ввода/вывода;

· драйверы должны синхронизировать свой доступ к глобальным данным;

· драйверы должны корректно восстанавливаться после сбоя питания и запускать процедуру сбора.

Потоки пользовательского режима вызывают базовые сервисы файла объекта (чтение, запись, открытие). Диспетчер динамически преобразует эти запросы в запросы к реальным физическим
устройствам (накопителям, сетям и т.д.)



22. Реализация ISR при необходимости программного отключения отдельного устройства, при необходимости полной замены обработчика (с учетом управления контроллером прерываний при его наличии).

Для отключения отдельного устройства в регистр масок контроллера прерываний необходимо записать значения, позволяющее замаскировать порт к которуму будет подключено соответствующее устройство. Для этого нужно обратиться к порту PORTy контроллера прерываний и
записать в него в разряд iMrx единицу для маскирования соответствующего прерывания от устройства.

```
Mov dx, porty//port
mov ax ,08h//маска
out dx, ax//посылаем
```

Для полной замены обработчика нужно написать свой обработчик, который помимо всего прочего должен сбрасывать заявку в регистре контроллера прерываний. В таблице прерываний заменить адрес старого обработчика на адрес нового. Для воостановления старого обработчика необходим записать его адрес в таблицу векторов прерываний.

Правила замены стандартного обработчика прерывания

Если действия программы обработки прерывания не устраивают, то ее можно заменить собственной, перенастроив вектор прерывания. Для этого необходимо выполнить следующие действия:

- получить текущее значение вектора прерывания,
- сохранить старое значение вектора в заранее зарезервированных ячейках сегмента данных,
- установить новое значение вектора,
- написать основную программу, инициирующую замененное прерывание
- восстановить старое значение вектора для дальнейшей бесконфликтной работы. В противном случае последующая программа может вызвать данное прерывание и передать управление на то место в памяти, где вашей программы уже нет,
- написать собственную программу обработки прерывания,
- в случае необходимости передать управление стандартному обработчику. Это целесообразно сделать, если собственный обработчик функционально дополняет имеющийся в системе.

Замена обработчика:

```
lea ax,new                  // new – метка нового обработчика
mov dx,cs
mov d, 0                    //устанавливаем нулевой сегмент
xchg w [n_vect*4], ax
xchng w [n_vect*4], dx  //замена вектора прерывания
mov ds, cs                  //восстанавливаем сегмент

mov w old_v, ax             //сохраняем старый обработчик
mov w old_v, dx
```

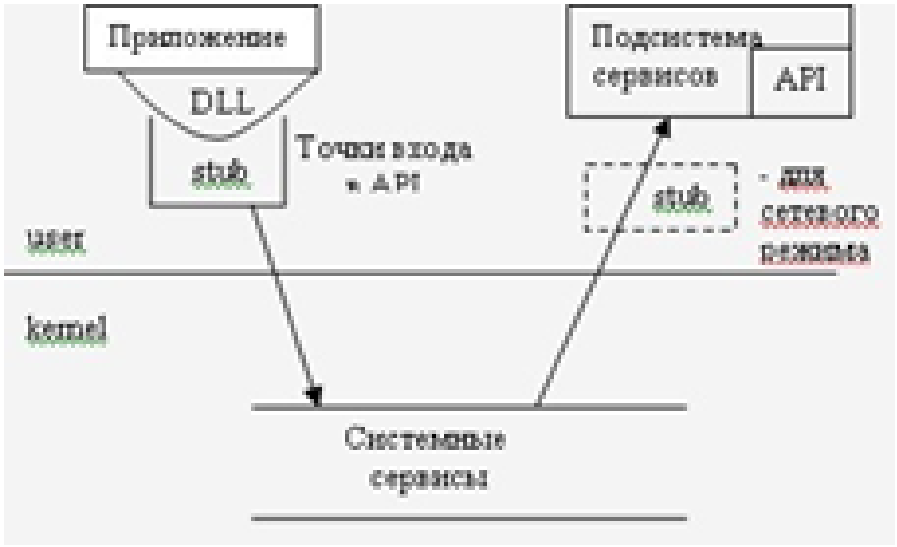
23. Реализация сервисов по запросу от приложений в ОС с различными архитектурами. Два подхода к реализации клиент-серверной модели (1. совместное использование сегментов DLL; 2. модель с защитой памяти подсистем).

В случае клиент-серверной модели:

1) Совместное использование сегментов dll (Windows X, OS/2).

Различные приложения разделяют АП различных динамических библиотек, при этом в Windows делается однократное копирование библиотек в разделяемую память (доступ от всех), а в OS/2 – многократное копирование библиотек в каждый процесс отдельно (в АП процесса). Каждый процесс может модифицировать данные, используемые всеми приложениями. Обращение производится за счет обычных вызовов подпрограмм.

2) с защитой памяти подсистем (Windows NT).



stub – приводит запрос к формату для передачи в подсистему.

Модель защиты (клиент-серверная):

«+»

1. Защита глобальных структур данных
2. Автономность сервисов
3. Сервера для приложения работают в пользовательском режиме, следовательно, не могут влиять на ядро и не могут вызывать внутренние функции ОС. Единственный доступ к ядру – вызов системных сервисов.
4. Обеспечивает разделение сервера – ОС.
5. Возможность одновременной работы любого количества серверов поддерживающих средства API. Повышение степени параллелизма.

DLL-БИБЛИОТЕКИ - РАЗДЕЛЯЕМЫЙ РЕСУРС, ОБЫЧНО СТАРАЮТСЯ НЕ КОПИРОВАТЬ ИХ, А ИСПОЛЬЗОВАТЬ СОВМЕСТНО КАК РАЗДЕЛЯЕМУЮ ПАМЯТЬ.

API - ЭТО ТОЖЕ РАЗДЕЛЯЕМЫЙ РЕСУРС, ПОЭТОМУ ВОЗНИКАЕТ ЭТА ПРОБЛЕМА; ДР. ВАРИАНТ ЕЕ РЕШЕНИЯ: ЧЕРЕЗ LPC:

ДОСТОИНСТВА:

1. ДАЕТ ЗАЩИТУ СЕРВЕРАМ (ПОДСИСТЕМАМ)
2. ДАЕТ АВТОНОМНОСТЬ СЕРВЕРАМ (ПОДСИСТЕМАМ)
3. ПОЗВОЛЯЕТ ЗАЩИТИТЬ ЯДРО, Т.К. ПРИЛОЖЕНИЯ НЕ МОГУТ НАПРЯМУЮ ВЫПОЛНЯТЬ ЕГО ФУНКЦИИ
4. ЕДИНООБРАЗИЕ ОБРАЩЕНИЙ
5. КЛИЕНТ-СЕРВЕРНАЯ МОДЕЛЬ ОБЕСПЕЧИВАЕТ РАЗДЕЛЕНИЕ ФУНКЦИОНАЛЬНОСТ
6. ВОЗМОЖНОСТЬ ПОДДЕРЖКИ $\forall$ ЧИСЛА СРЕД (РАСШИРЯЕМОСТЬ СИСТЕМЫ)

НЕДОСТАТОК:

ПОТЕРЯ ПРОИЗВОДИТЕЛЬНОСТИ: ВЗАИМОДЕЙСТВИЕ ЧЕРЕЗ LPC СЛОЖНО И СРАВНИТЕЛЬНО ДОЛГО (ДЛЯ ОБРАБОТКИ СООБЩЕНИЯ СЕРВЕРОМ НЕОБХОДИМО ПЕРЕКЛЮЧАТЬ КОНТЕКСТ: СОХРАНИТЬ КОНТЕКСТ КЛИЕНТА, ВЫБРАТЬ И ЗАГРУЗИТЬ КОНТЕКСТ СЕРВЕРА, ВЫПОЛНИТЬ Ф-ИЮ API И ВЕРНУТЬСЯ).

НА ПРОИЗВОДИТЕЛЬНОСТЬ ТАКЖЕ ВЛИЯЕТ РЕАЛИЗАЦИЯ LPC И ЧАСТОТА ОБРАЩЕНИЙ.

24. Физическая организация устройств ввода-вывода. Организация ПО ввода-вывода.

Физическая организация устройств ввода-вывода

Устройства IO делятся на два типа: блок-ориентированные устройства и байт-ориентированные устройства. Блок-ориентированные устройства хранят информацию в блоках фикс. размера, каждый из которых имеет свой собственный адрес. Самое распространенное блок-орие. устр-во - диск. Байт-ориентированные устр-ва не адресуемы и не позволяют производить операцию поиска, они генерируют или потребляют послед-ть байтов. Примерами являются терминалы, строчные принтеры, сетевые адаптеры.

Внешнее уст-во обычно состоит из механического и электронного компонента. Электронный компонент называется контроллером устр-ва или адаптером. Механический компонент представляет собственно уст-во. Некоторые контроллеры могут управлять несколькими устр-ми.

ОС обычно имеет дело не с устр-ом, а с контроллером. Контроллер, как правило, выполняе простые функции, например, преобразует поток бит в блоки, состоящие из байт, и осуществляют контроль и исправление ошибок. Каждый контроллер имеет несколько регистров, которые исп. для взаимодействия с цент. процессором. В некоторых компах эти регистры являются частью физ. адресного пространства. В таких компьютерах нет специальных операций IO. В других компьютерах адреса регистров IO, называемых часто портами, образуют собственное адресное пространство за счет введения спец. операций IO.

Организация программного обеспечения ввода-вывода

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

Ключевым принципом является независимость от устройств. Вид программы не должен зависеть от того, читает ли она данные с гибкого диска или с жесткого диска.

Очень близкой к идее независимости от устройств является идея единообразного именования, то есть для именования устройств должны быть приняты единые правила.

Другим важным вопросом для ПО IO является обработка ошибок. Вообще говоря, ошибки следует обрабатывать как можно ближе к аппаратуре. Если контроллер обнаруживает ошибку чтения, то он должен попытаться ее скорректировать. Если же это ему не удается, то исправлением ошибок должен заняться драйвер устройства.

Еще один ключевой вопрос - это использование блокирующих (синхр) и неблокирующих
(асинхр) передач. Большинство операций физического IO выполняется асинхронно - процессор начинает передачу и переходит на другую работу, пока не наступает прерывание.
Последняя проблема состоит в том, что одни устройства являются разделяемыми, а другие - выделенными. Диски - это разделяемые устройства, так как одновременный доступ нескольких пользователей к диску не представляет собой проблему. Принтеры - это выделенные устройства, потому что нельзя смешивать строчки, печатаемые различными пользователями. Наличие выделенных устройств создает для ОС некоторые проблемы.
Для решения поставленных проблем целесообразно разделить ПО IO на 4 слоя :

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

25. Семафоры. Мьютексы. Механизм разделения памяти, алгоритм с минимальным количеством средств синхронизации.

Семафоры – средство синхронизации доступа нескольких процессов к разделяемым ресурсам (функция разрешения/запрещения использования ресурса).
Требования к семафорам:

- должны быть размещены в адресном пространстве ядра;
- операции над семафорами должны быть неделимыми (атомарными), что возможно только при выполнении операций в режиме ядра.

Семафоры – системный ресурс. Операции над семафорами производятся посредством системных вызовов. В UNIX системах под семафором понимается группа семафоров.
Структура для описания семафоров:

- sem_id – идентификатор семафора;
- sem_perm – права доступа;
- sem_first – указатель на первый элемент;
- sem_num – число семафоров в группе;
- sem_oper - последняя операция;
- sem_time – время последнего изменения;

Также в структуре sem хранится имя семафора, значение семафора, идентификаторы процессов, выполнивших последние операции изменения семафора, число процессов, ожидающих
увеличения семафора, число процессов, ожидающих обнуления семафора.

Мьютекс — одноместный семафор, служащий в программировании для синхронизации одновременно выполняющихся потоков.

Мьютексы — это один из вариантов семафорных механизмов для организации взаимного исключения. Они реализованы во многих ОС, их основное назначение — организация взаимного исключения для потоков из одного и того же или из разных процессов.

Мьютексы — это простейшие двоичные семафоры, которые могут находиться в одном из 2х
состояний — открыт и закрыт. Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта mutex, последний переводится в неотмеченное(закрытое) состояние. Если задача освобождает мьютекс, его состояние становится отмеченным(открытым).

Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищенным мьютексом. Если другому потоку будет нужен доступ к переменной, защищенной мьютексом, то этот поток засыпает до тех пор, пока мьютекс не будет освобождён.

Цель использования мьютексов — защита данных от повреждения; однако им порождаются
другие проблемы, такие как взаимная блокировка (клинч) и «соревнование за ресурс».

Рассмотрим несколько вариантов постановки задачи синхронизации доступа к разделяемой памяти.

5.1. Вариант 1. Есть один процесс, выполняющий запись в разделяемую память и один процесс, выполняющий чтение из нее. Под чтением понимается извлечение данных из памяти. Программа должна обеспечить невозможность повторного чтения одних и тех же данных и невозможность перезаписи данных, т.е. новой записи, до тех пор, пока читатель не прочитает предыдущую. В таком варианте задания для синхронизации процессов достаточно двух

семафоров. Покажем, почему не достаточно одного на примере. Так как мы используем один семафор, то алгоритм работы читателя и писателя может быть только таким – захват семафора, выполнение действия (чтение / запись), освобождение семафора. Теперь допустим, что читатель прочитал данные, освободил семафор и еще не до конца использовал квант процессорного времени. Тогда он перейдет на новую итерацию, снова захватит только что освобожденный семафор и снова прочитает данные – ошибка. Теперь покажем, почему достаточно двух семафоров. Придадим одному из них смысл «запись разрешена», т.е. читатель предыдущие данные уже использовал; второму – «чтение разрешено», т.е. писатель уже сгенерировал новые данные, которые нужно прочитать

Рис.4. Организация доступа к разделяемой памяти одного «читателя» и одного «писателя»

Оба семафора бинарные и используют стандартные операции, захват семафора – это ожидание освобождения ресурса (установки семафора в 1) и последующий захват ресурса (установки семафора в 0), освобождение ресурса – это установка семафора в 1. Пару семафоров, использованных таким образом, иногда называют разделенным бинарным семафором, поскольку в любой момент времени только один из них может иметь значение 1. При таком алгоритме работы, оба процесса после выполнения своей задачи и освобождения одного из семафоров, будут ждать освобождения другого семафора, которое произведет другой процесс, но только после выполнения своей работы. Таким образом повторное чтение, или повторная запись стала невозможной.

5.2. Вариант 2. К условиям предыдущей задачи добавляется условие корректной работы нескольких читателей и нескольких писателей одновременно. Как и в предыдущем варианте под чтением понимается извлечение данных из памяти, т. е. одну порцию данных может прочитать только один читатель. Легко понять, что это условие не приводит к необходимости использования дополнительных средств синхронизации. Теперь вместо одного процесса, за каждый семафор будут конкурировать несколько. Но повторная запись или чтение все также невозможно. Так как, чтобы очередной процесс- писатель отработал, нужно освобождение семафора, которое выполняется из процесса-читателя, и наоборот.

5.3. Вариант 3. К условиям предыдущей задачи добавляется наличие не единичного буфера, а буфера некоторого размера. Тип буфера (очередь, стек, кольцевой буфер) не имеет значения. Двух семафоров по прежнему достаточно, но это приведет к вырождению буфера, так как все процессы будут работать только с одной ячейкой. Так как размер буфера не равен единице, то больше нет необходимости в чередовании операций чтения и записи, допустима ситуация нескольких записей подряд, и после этого нескольких чтений. Нужно только следить, чтобы не было записи в уже заполненный буфер и не было чтения из пустого буфера. Для этого выберем

другие типы семафора и придадим им другую семантику. Возьмем два считающих семафора. Максимальное значение обоих – размер буфера. Первый инициализируется нулем и имеет смысл «количество заполненных ячеек», второй инициализируется N, где N – размер буфера и имеет смысл «количество пустых ячеек». Процессы-читатели перед своей работой захватывают семафор «количество заполненных ячеек», т.е. ждут появления хотя бы одной порции данных, а после чтения освобождают семафор «количество пустых ячеек». Процессы-писатели перед записью захватывают семафор «количество пустых ячеек», т.е. ждут появления хотя бы одной пустой ячейки для записи, а после записи освобождают семафор «количество полных 39 ячеек». Таким образом, решается проблема чтения из пустого буфера и запись в полный. Так как семафоры не бинарные, захватить их может сразу несколько процессов, т.е. несколько процессов попадут в секцию записи или чтения. В этом случае, если операция записи или чтения не атомарная (а зачастую так оно и есть), может произойти нарушение нормальной работы программы, к примеру, несколько процессов-писателей попытаются произвести запись в одну и ту же ячейку буфера, или несколько читателей выполнят чтение одной и той же ячейки. Таким образом, операции записи-чтения становятся критическими секциями, доступ к которым также необходимо синхронизировать. Для этого будет достаточно еще одного бинарного семафора, имеющего смысл «доступ к памяти разрешен». Оба типа процессов должны захватывать его при попытке взаимодействия с памятью и освобождать после. В итоге, алгоритм работы процессов-писателей, и процессов-читателей выглядит следующим образом:

Рис.5. Организация доступа к разделяемой памяти многих «читателей» и «писателей»

Важно отметить, что порядок операций освобождения семафоров не важен, в то же время изменение порядка захвата семафоров может привести к взаимной блокировке процессов (dead lock). Взаимная блокировка в частности может произойти в таком случае: процесс-читатель захватил семафор «доступ к памяти разрешен», далее он проверяет, есть ли заполненные ячейки, т.е. пытается захватить семафор «количество заполненных ячеек», предположим, что свободных ячеек не оказалось и процесс переключился в неактивный

режим, ожидая пока какой-нибудь процесс-писатель не запишет данные и не освободит семафор «количество заполненных ячеек». Но, этого никогда не произойдет, так как перед записью данных, процесс-писатель должен захватить семафор «доступ к памяти разрешен», который уже захвачен ожидающим процессом-читателем. Пример решения последнего варианта задачи. В качестве разделяемого ресурса используется массив, находящийся в разделяемой памяти. Ячейка памяти, расположенная за последним элементом массива, интерпретируется как индекс последнего записанного элемента.

26. Преобразование виртуального адреса в физический при странично-сегментной организации памяти (на примере процессора Intel)

Переход от виртуальных адресов к физическим:

1) с помощью перемещающего загрузчика (спец. сист. прога, которая загружает нашу прогу в память, заменяя вирт. адреса на физ.) - наиболее экономичный подход

2) динамически (прога загружается без замены адресов, адреса пересчитываются при каждом обращении) — более гибкий способ, позволяет проге перемещаться в памяти; но постоянные затраты в коде функционирования; постоянно надо преобразовывать адреса

Методы распределения памяти:

1) без использования внешней памяти:

- фикс. разделами
- динам. разделами
- перемещаемыми разделами

2) с использованием внешней памяти

- страничное
- сегментное
- сегментно-страничное

Достоинства сегментации и страничной организации сочетает в себе сегментно-страничный способ распределения памяти, при реализации которого сохраняется содержательное разбиение программы на сегменты, а последние разделены на страницы фиксированной длины.

Блок управления памятью MMU МП Pentium содержит как блок сегментации, так и блок управления страницами и в общем случае может осуществлять многоэтапное преобразование логических адресов в физические. Сегментирование является обязательным этапом преобразования логических адресов (селектора и внутрисегментного смещения), используемых пользователем при написании своих программ. В результате такого преобразования на выходе блока сегментации формируется 32-битный линейный адрес. Если страничный механизм не включен, МП считает, что линейный адрес является физическим и выдает его на внешнюю шину адреса. В противном случае МП с помощью блока управления страницами осуществляет дополнительную трансляцию линейного адреса в физический. Включение-отключение механизма страничного преобразования выполняется программно путем установки-сброса бита PG в управляющем регистре управления CR0.

По сравнению с простым сегментированием при использовании страничного преобразования изменяются как способ формирования физического адреса операнда (выполняется в два этапа), так и размер адресуемой виртуальной памяти. Первый этап преобразования виртуального адреса, связанный с использованием смещения, селектора и дескрипторной таблицы GDT или LDT, полностью совпадает с этапом преобразования при отключенном страничном механизме. Отличительной особенностью этого этапа преобразования является то, что в формировании линейного виртуального адреса участвует базовый адрес сегмента в виртуальной памяти, а не в физической памяти, как в случае простого сегментирования при отключенном страничном механизме. Так как виртуальная память объединяет все сегменты задачи объемом до 4 Гбайт каждый, то в режиме страничного преобразования возможно наложение сегментов, однако процессор не контролирует такие ситуации, оставляя решение этой проблемы ОС. На втором этапе (этапе страничного преобразования) линейный виртуальный адрес преобразуется в адрес операнда физической страницы ОП.

всегда должна находиться в ОП. Разделы загружаются в ОП по мере необходимости. Каталог предназначен для хранения базовых адресов и другой информации, относящейся к адресуемой странице раздела. Разделы представляют собой собственно таблицы соответствия: строки раздела адресуют физические страницы ОП.

Преобразуя линейный адрес, процессор последовательно определяет раздел в каталоге и выбирает страницу с искомым операндом. Выбор элементов каталога, раздела или страницы осуществляется с помощью индексов в соответствующих полях виртуального адреса. Линейный (виртуальный) адрес (32 разряда) рассматривается как совокупность трех полей: TABLE, PAGE и BYTE (рис. 4.27). Разрядность полей TABLE и PAGE составляет 10 бит, а поля BYTE — 12 бит. Поле TABLE указывает относительный адрес (индекс) дескриптора с базовым адресом раздела в каталоге разделов. Поле Page задает относительный адрес дескриптора с базовым адресом физической страницы в выбранном разделе. 12-разрядное поле BYTE содержит смещение искомого операнда в физической странице ОП.

27. Системные функции и идентификация различных IPC. Пространство имен.

Идентификаторы и имена в IPC

Как было показано, отсутствие имен у каналов делает их недоступными для независимых процессов. Этот недостаток устранен у FIFO, которые имеют имена. Другие средства межпроцессного взаимодействия, являющиеся более сложными, требуют дополнительных соглашений по именам и идентификаторам. Множество возможных имен объектов конкретного типа межпроцессного взаимодействия называется *пространством имен* (name space). Имена являются важным компонентом системы межпроцессного взаимодействия для всех объектов, кроме каналов, поскольку позволяют различным процессам получить доступ к общему объекту. Так, именем FIFO является имя файла именованного канала. Используя условленное имя созданного FIFO два процесса могут обращаться к этому объекту для обмена данными.

Для таких объектов IPC, как очереди сообщений, семафоры и разделяемая память, процесс назначения имени является более сложным, чем просто указание имени файла. Имя для этих объектов называется ключом (key) и генерируется функцией *ftok(3C)* из двух компонентов — имени файла и идентификатора проекта:

```
#include <sys/types.h>

#include <sys/ipc.h>

key_t ftok(char* filename, char proj);
```

В качестве `filename` можно использовать имя некоторого файла, известное взаимодействующим процессам. Например, это может быть имя программы-сервера. Важно, чтобы этот файл существовал на момент создания ключа. Также нежелательно использовать имя файла, который создается и удаляется в процессе работы распределенного приложения, поскольку при генерации ключа используется номер inode файла. Вновь созданный файл может иметь другой inode и впоследствии процесс, желающий иметь доступ к объекту, получит неверный ключ.

Пространство имен позволяет создавать и совместно использовать IPC неродственным процессам. Однако для ссылок на уже созданные объекты используются идентификаторы, точно так же, как файловый дескриптор используется для работы с файлом, открытым по имени.

Каждое из перечисленных IPC имеет свой уникальный дескриптор (идентификатор), используемый ОС (ядром) для работы с объектом. Уникальность дескриптора обеспечивается уникальностью дескриптора для каждого из типов объектов (очереди сообщений, семафоры и разделяемая память), т.е. какая-либо очередь сообщений может иметь тот же численный идентификатор, что и разделяемая область памяти (хотя любые две очереди сообщений должны иметь различные идентификаторы).

ОБЪЕКТ IPC	ПРОСТРАНСТВО ИМЕН	ДЕСКРИПТОР
Канал	–	Файловый дескриптор
FIFO	Имя файла	Файловый дескриптор
Очередь сообщений	Ключ	Идентификатор
Объект IPC	Пространство имен	Дескриптор
Семафор	Ключ	Идентификатор
Разделяемая память	Ключ	Идентификатор

Еще какая-то инфа:

Способы генерации и передачи ключа IPC

Все три типа IPC ресурсов создаются в виде структур ядра с уникальным идентификатором, являющимся возвращаемым значением системных вызовов

```
int msgget(key_t key, int msgflg);
int shmget(key_t key, size_t size, int shmflg);
int semget(key_t key, int nsems, int semflg);
```

использующих в качестве первого параметра ключ key – любое целое число, определяющее однозначно значение идентификатора ресурса. То есть, взаимодействующие процессы для получения доступа к общему IPC ресурсу должны использовать одинаковый ключ key. Кроме того, для создания абсолютно нового ресурса процесс-создатель должен использовать уникальный ключ, отличный от ключей создания других IPC ресурсов. Для этого существует несколько способов.

1. Использование предопределенного key значения IPC_PRIVATE. При этом гарантируется, что будет создан новый ресурс, но возникает проблема передачи полученного (от get-операции) id ресурса другим процессам для подключения к ресурсу.
2. Ключ может быть объявлен, например, в заголовочном файле, общем для всех программ , требующих доступ к ресурсу. Но при этом существует вероятность, что какой-либо ресурс уже использовал данное значение ключа — попытка создать ресурс (используя флаг IPC_CREAT для *_flg параметра) с повторно используемым ключом приведет к сообщению об ошибке: errno равно EEXIST.
3. Ключ может генерироваться во всех взаимодействующих процессах с помощью функции ftok/3, преобразующей указанный путь к существующему файлу (pathname - согласованный для всех процессов) и идентификатор проекта (proj_id - число от 0 до 255) в ключ IPC:

```
key_t   ftok(const char *pathname, int proj_id);
```

28. Идентификаторы файлов, файловые дескрипторы, индексные дескрипторы. Файловые ссылки. Укажите назначение и сферу применения, соответствующие каждому понятию.

индексный дескриптор — inode. Содержит информацию о файле , необходимую для обработки данных, т.е. метаданные файла. Используется файловой системой.
файловый дескриптор — индекс в таблице файловых дескрипторов процесса. Представляет собой неотрицательное число, возвращаемое системными вызовами, такими как open(), pipe(). Файловый дискриптор связан с полями u_ofile и u_pofile структуры user и таким образом обеспечивает доступ к соответствующему элементу файловой таблицы(структуре данных file).
Файловые ссылки — 2 вида. Жесткие ссылки и символьные. Жесткие ссылаются на vnode. Символьные — это путь к жесткой ссылке. Для связи vnode и содержимого каталогов ФС.Связь имя файла ->inode.

29. Структура драйвера. Минимальные наборы процедур в различных драйверах (простейший, с прерываниями, многослойный)

драйвер – независимый элемент ОС, который может быть динамически загружен или удален без изменения конфигурации ОС. Драйверы имеют унифицированную модель интерфейса => диспетчер В/В не видит их структуру и детали реализации. При взаимодействии различных драйверов диспетчер играет роль посредника.

Драйверы могут быть:

- Однослойные (для последовательного порта и др.)
- Многослойные (для ЗУ большой емкости)
- с прерываниями

Передача может быть:

- Асинхронной – более быстродействующий вариант и экономичный в плане ресурсов
- Синхронный – проще в плане реализации

Структура драйвера:

- **Процедура инициализации(1)** - Выполняется диспетчером В/В при загрузке ОС. Создаются системные объекты для распознавания устройства и доступа к нему.
- **Набор процедур распределения(2)** - Главные функции, предоставляемые драйверами устройствам Запрос В/В заставляет диспетчер В/В генерировать IRP и обращается к драйверу через процедуру распределения.
- **Процедура запуска В/В(3)** - Драйвер использует эту процедуру для начала передачи данных. Необязательная процедура характерна для физических устройств при передаче больших объемов.
- **ISR Interrupt(4)** - Прерывание от устройства. На момент выполнения приоритет RPL повышается. Сообразно ISR выполняет часть кода обработчиков.
- **Процедура DPC(5) -** Процедура обработки прерывания, которая выполняется при более низком, чем у IRQ приоритете. Выполняет большую часть обработки по обслуживанию прерывания.
- **Процедура завершения(6) -** Является посредником между драйверами нижнего и верхнего уровня. В ней хранится информация о возможных значениях, сбое или необходимости очистки при инициализации.
- **Процедура отмены В/В(7)** - Может быть не одной. Информация об активизации процедуры отмены хранится в IRP.
- **Процедура выгрузки(8)** - Освобождение системных ресурсов, которые используются драйвером. Затем диспетчер В/В удаляет драйвер из памяти. Драйвер может быть загружен/выгружен во время работы.
- **Процедура протоколирования ошибок(9)** - Информирование диспетчера В/В об ошибке, который помещает эту информацию в журнал ошибок.

Минимальный драйвер должен содержать 3 процедуры: 1-2-8
(1), (2), (3), (4), (5), (6), (8) для многослойного драйвера

30. Структуры каталогов и форматы записей каталогов для различных ФС

Каталог FFS. Представлен следующей структурой:
d_ino – номер inode
d_reclen – длина записи
d_namlen – длина имени файла
d_name[] - имя файла

Имя файла имеет переменную длину, дополненную нулями до 4-байтной границы. При удалении
имени файла принадлежащая ему запись присоединяется к предыдущей

В FAT16, как и FAT 12, используются имена 8+3 символа, но в ОС Win32 возможно применение длинных имен, из 255 символов UNICODE (по 2 байта на символ). Первые 8+3 символа хранятся в этом случае также в основной записи каталога а последующие символы – по 13 штук (26 байт) + 6 служебных байт – отдельными порциями за основной записью. Аналогично устроена ФС FAT32.

Каталог на NTFS представляет собой специфический файл, хранящий ссылки на другие файлы и каталоги, создавая иерархическое строение данных на диске. Файл каталога поделен на блоки, каждый из которых содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который уже предоставляет полную информацию об элементе каталога. Внутренняя структура каталога представляет собой бинарное дерево.

31. Система ввода-вывода в NT: компоненты, процедуры диспетчера, функции драйвера.

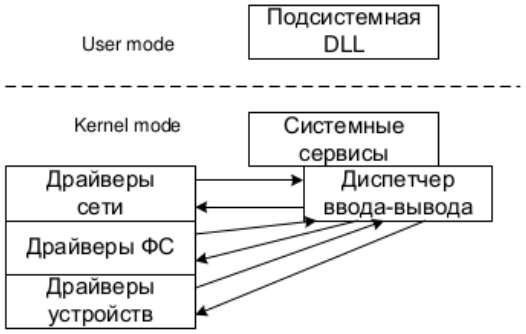
Часть исполнительной системы, получающая запросы от процессов пользователя либо режима пользователя, либо ядра, и передающая эти запросы устройствам ввода-вывода в преобразованном виде.
Система управляется пакетами. Каждый запрос представляет IRP-пакет (Input Request Package). IRP позволяет использовать иерарх. настр. системы В/В и драйверов.
IRP – структура данных, управляющая обработкой выполнения операций на каждой стадии их выполнения.
Диспетчер В/В – входит в состав системы В/В, управляет передачей запросов В/В в файловую систему, реализует процедуру общего назначения для разных драйверов.
Функции и этапы работы диспетчера:

- Создание IRP определенного формата
- Передача пакета соответствующему драйверу
- Удаление IRP

Функции драйвера:

- Получение IRP
- Выполнение операции
- Возвращение управления В/В, либо реализация завершения

В NT существуют драйверы ФС и драйверы устройств В/В. При любой операции В/В задействованы и драйверы ФС и драйверы устройств. Работают через IRP, контролируются диспетчером. Реализуется общая процедура для разли. драйверов. Упрощаются отдельные драйвера и вносится универсальность.



Принцип: виртуальные файлы устройств (как в UNIX) – объектная модель. Работа осуществляется посредством файловых описателей. Описатель ссылается на объект – файл. Потоки пользовательского режима вызывают базовые сервисы файла объекта (чтение, запись, открытие). Диспетчер динамически преобразует эти запросы в запросы к реальным физическим устройствам (накопителям, сетям и т.д.)

32. Вызов подпрограмм и задач в ОС. Действия процессора при вызове через шлюз.

```
#include <stdio.h>
int main(int argc, char* argv[]) {
	FILE *file;
	char a;
	fopen(argv[1], "r");
	for (int i = 0;;i++) {
		if (feof(file)){
			printf("\n");
			break;
		}
		fscanf(file, "%c", &a);
		printf("%c", a);
	}
	fclose(file);
	return 0;
}
```

Содержимое файла /root/text:

You couldn’t see this text

Программе-шлюзу и файл, имеют следующие права доступа:

$ ls -l ./root/shluz ./root/text

-rwSrwxr-x ./root/shluz

-rw------- ./root/text

Читаем файл программой shluz:

$ ./root/shluz ./root/text

You couldn’t see this text

Результат успешен, так как файл ./root/shluz имеет атрибут SUID.

Вызов подпрограмм и задач

Подпроги мб не только пользовательские, но и из библиотек, системные вызовы + возможно переключение задач в рез-те вызова.

Для обеспечения возможностей доступа к подпрограммам и задачам, находящимся в другом кодовом сегменте, нужно: найти соотв. LDT - таблицу и получить к ней доступ.

Два способа вызова подпрограмм:

1) указываем селектор и смещение, прямо в команде JMP или CALL (подходит пользовательских положений)

2) через шлюз вызова

Разрешение вызова - исходя из привилегий и бита подключения.

Существует т.н. вызов подпрограмм по точкам входа (заранее определенных и описанных с помощью спец. дескрипторов - „дескрипторов шлюзов вызовов подпрограмм"). В таком случае в CALL или JMP указывается селектор такого дескриптора (вызывающий код должен иметь права, не меньшие чем у дескриптора, причем этот дескриптор уже может перенаправить вызов в более приоритетный сегмент), при этом смещение не используется, оно заменяется шлюзом - т.о. отдается (на точку входа как раз) контроль со стороны системы.

При обращении адр. пр-ва возникает проблема передачи параметров. Для этого можно исп-ть сегмент TSS.

При вызове кодов с разными привилегиями исп-ют различные сегменты стеков, селекторы этих сегментов хранятся в TSS. В стек целевого сегмента копируется столько слов (из стека задачи), сколько указано в поле счетчика дескриптора шлюза.

В поле команды CALL можно указать непосредственно, если привилегии позволяют, или косвенно - через шлюз вызова задач (тогда достаточно иметь права доступа к шлюзу).

В ТАКОМ СЛУЧАЕ ВЫПОЛНЯЮТСЯ СЛЕД. ДЕЙСТВИЯ:

1) ВЫПОЛНЕНИЕ КОМАНДЫ CALL, СЕЛЕКТОР ДЫЙ УКАЗЫВАЕТ НА СЕГМЕНТ ТИПА TSS.

2) В TSS ТЕКУЩИЕ ЗАДАЧИ СОХРАНЯЮТ РЕГИСТРЫ ПРОЦЕССОРА. В РЕГИСТРЕ TR СОХРАНЯЕТСЯ СЕЛЕКТОР СЕГМЕНТА TSS ЗАДАЧИ, НА ДЫЙ МЫ ХОТИМ ПЕРЕКЛЮЧИТЬСЯ. ИЗ НОВОГО TSS ПЕРЕНОСИТСЯ ЗНАЧЕНИЕ LDTR.

3) ВОССТАНОВЛЕНИЕ ЗНАЧЕНИЙ РЕГИСТРОВ ПРОЦЕССА ИЗ СООТВЕТСТВУЮЩИХ ПОЛЕЙ НОВОГО TSS-СЕГМЕНТА

4) В ПОЛЕ СЕЛЕКТОРА ВОЗВРАТА ЗАНОСИТСЯ СЕЛЕКТОР TSS, ОТНОСЯЩИЙСЯ К СНИМАЕМОЙ ЗАДАЧЕ.

ЧЕРЕЗ ШЛЮЗ ДОБАВЛЯЕТСЯ ПОИСК ДЕСКРИПТОРА СЕГМЕНТА TSS ПО ЗНАЧЕНИЮ СЕГМЕНТА ШЛЮЗА ВЫЗОВА

(ДАННЫЕ ДЛЯ ВОССТАНОВЛЕНИЯ ЗАДАЧИ; ПРОЦЕССОР ПРОИЗВОДИТ АППАРАТНЫЕ ПЕРЕКЛЮЧЕНИЯ TSS)

СТРУКТУРА СЕГМЕНТА TSS ЗАДАЧИ (КОНТЕКСТ):

- ФИКСИРОВАННЫЕ ПОЛЯ РЕГИСТРОВ И Т.П. + CS, SS, DS + РОНы ...
- Input-output privilege level
- КАРТА ВВОДА-ВЫВОДА (65536 ПОРТОВ)
- ЧАСТЬ, ЗАПОЛНЯЕМАЯ СЛУЖЕБНОЙ ИНФОЙ ОС

ФОРМАТ ДЕСКРИПТОРА TSS АНАЛОГИЧЕН ФОРМАТУ ДЕСКРИПТОРА СЕГМЕНТОВ ДАННЫХ.

TSS

ОБЯЗАТЕЛЬНАЯ ЧАСТЬ (104 БАЙТА) (ВСЯ ИНФА, НЕОБХОДИМАЯ ПРОЦЕССУ)

ДОПОЛНИТЕЛЬНАЯ ЧАСТЬ (БИТОВАЯ КАРТА НОМЕРОВ ПРЕРЫВАНИЙ И АДРЕСОВ УСТРОЙСТВ ВВ, К ЧЬИМ ДОСТУПНО ОБРАЩЕНИЕ ПРИ ВЫЗОВЕ ЗАДАЧИ)

ПЕРВЫЕ 2 БАЙТА СЕГМЕНТА TSS ХРАНЯТ СЕЛЕКТОР ПРЕД. ЗАДАЧИ ДЛЯ ОБЕСПЕЧЕНИЯ ВОЗВРАТА – „СЕЛЕКТОР ВОЗВРАТА".

TSS ОБНОВЛЯЕТСЯ ПРИ КАЖДОМ ПЕРЕКЛЮЧЕНИИ ЗАДАЧ

33. Жизненный цикл сигнала (POSIX), описание каждой фазы, системные вызовы в каждой фазе жизненного цикла

Сигналы имеют определенный жизненный цикл. Вначале сигнал создается – высылается процессом или генерируется ядром. Затем сигнал ожидает доставки в процесс-приемник. В конце жизненного цикла сигнала происходит его перехват процессом и выполнение связанных с сигналом действий

Сигналы – способ передачи уведомления о событии, произошедшем либо между процессами, либо между процессом и ядром. Сигналы очень ресурсоемки. Ограничены с точки зрения системных средств. Они малоинформативны. Являются простейшим способом IPC. Используются для генерации простейших команд, уведомлений об ошибке. Обработка сигнала похожа на обработку прерывания. Сигнал имеет собственное имя и уникальный номер. Максимальное количество сигналов – 32.

Причины генерации сигналов:

- Ядро отправляет сигнал процессу при нажатии клавиши;
- Деление на 0;
- Обращение к недоступной области памяти;
- Особые аппаратные ситуации;
- Особые программные ситуации.

Основные действия при приеме сигналов:

- игнорирование;
- действия на данный сигнал, согласно умолчанию;
- обработка собственным обработчиком.

Одно из возможных действий – диспозиция сигнала.

Две фазы существования сигнала: 1) генерация и отправка 2) доставка и обработка

Доставка и обработка - Для каждого сигнала прописана обработка по умолчанию, задаваемая ядром либо обработка из нашего процесса.

Возможные действия при получении сигнала:

- завершение процесса с сохранением образа процесса в ОЗУ;
- завершение процесса без сохранения образа процесса в ОЗУ;
- игнорирование сигнала;
- приостановка процесса;
- возобновление процесса при условии, что он был приостановлен.

Наиболее частый вариант – завершение процесса с сохранением образа в ОЗУ. Можно изменить действия по умолчанию на свой обработчик или указать системный. Можно заблокировать сигнал.

Обработка. Для обработки сигнала необходимо, чтобы на момент его получения процесс выполнялся. Фаза ожидания доставки может быть длительной.

Доставка. Осуществляется ядром. Ядро от имени процесса осуществляет вызов ISSIG().Данный системный вызов выполняется в трех случаях:

- возвращение процесса из режима ядра в пользовательский режим;
- переход процесса в состояние сна;
- выход процесса из состояния сна.

Все это происходит при условии, что приоритет процесса допускает его прерывание сигналом. Системный вызов ISSIG() определяет, есть ли сигналы, ожидающие доставки. Если нет – то процесс просто выполняется.

Ресурсоемкость сигналов. Для отправки-доставки требуется системный вызов. Для доставки – прерывание и его обработка. Для этого необходимо большое число операций со стеком - копирование пользовательского стека в системную область, извлечение параметров системных вызовов, результатов работы системных вызовов и др. **Вывод: затраты слишком велики по сравнению с объемом передаваемой информации.**

34. Структура тома FAT, формат записи каталога, FAT-таблица. Каталог и файловая запись в HPFS

В ФС FAT смежные секторы диска объединяются в единицы, называемые **кластерами**. Количество секторов в кластере может быть равно 1 или степени двойки. Для хранения данных файла отводится целое число кластеров (мин. 1), так что, например, если размер файла составляет 40 байт, а размер кластера 4 кбайт, реально занят информацией файла будет лишь 1% отведенного для него места. Для избежания подобных ситуаций целесообразно уменьшать размер кластеров, а для сокращения объема адресной информации и повышения скорости файловых операций – наоборот. На практике выбирают некоторый компромисс. Так как емкость диска вполне может и не выражаться целым числом кластеров, обычно в конце тома присутствуют т.н. surplus sectors – «остаток» размером менее кластера, который не может отводиться ОС для хранения информации.

Пространство тома FAT32 логически разделено на три смежные области:

Зарезервированная область. Содержит служебные структуры, которые принадлежат загрузочной
записи раздела и используются при инициализации тома;

Область таблицы FAT, содержащая массив индексных указателей ("ячеек"), соответствующих
кластерам области данных.

Область данных, где записано собственно содержимое файлов, а также т.н. метаданные.

В FAT12 и FAT16 также специально выделяется область корневого каталога. Она имеет фиксированное положение и фиксированный размер в секторах.

Каталог FAT является обычным файлом, помеченным специальным атрибутом. Данными такого файла в любой версии FAT является цепочка 32-байтных файловых записей. Директория не может штатно содержать два файла с одинаковым именем. Если программа проверки диска обнаруживает искусственно созданную пару файлов с идентичным именем в одном каталоге, один из них переименовывается.

При создании каталога для него «пожизненно» выставляется DIR_FileSize = 0. Размер содержимого
каталога определяется простым следованием по цепочкам кластеров до метки End Of Chain. Размер самого каталога лимитируется файловой системой в 65 535 32-байтных записей (т.е. записи каталога в таблице FAT не могут занимать более 2Мб).

Это ограничение призвано ускорить операции с файлами и позволить различным служебным программам использовать 16 битное целое (WORD) для подсчета количества записей в директории. Каталогу отводится один кластер области данных и полям DIR_FstClusHI / DIR_FstClusLO присваивается значение номера этого кластера. В таблицу FAT для записи, соответствующей этому кластеру, помещается метка EOC, а сам кластер забивается нулями.

Далее создаются два специальных файла, без которых директория FAT считается поврежденной – файлы нулевого размера “dot” (идентификатор каталога) и “dotdot” (указатель на родительский каталог) с именами “.” (точка) и “..” (две точки) соотв. Штампы даты-времени этих файлов приравниваются значениям для самого каталога на момент создания и не обновляются при изменениях каталога.

35. Преобразование виртуального адреса в физический при страничной организации

Страничная организация памяти. Для решения проблемы фрагментации разработчики ВМ предложили распределять память блоками фиксированного размера — страницами. Подобная модель виртуальной памяти получила название памяти со страничной организацией. Размер страниц обычно принимается равным 4 Кбайт (реже 4 Мбайт), а сами страницы часто называют страничными кадрами (page frame). Фиксированная длина страниц значительно упрощает распределение памяти. При необходимости загрузки новой страницы в ОП ее можно либо поместить в незанятую страницу, либо вытеснить другую страницу, чтобы освободить требуемое место. В любом случае не требуется по-новому располагать другие страницы в ОП. Благодаря этому обеспечивается более эффективное использование физической памяти и устраняется фрагментация памяти. В отличие от сегментации, требующей загрузки сегмента целиком, страничная организация позволяет сократить объем передаваемой информации между ВЗУ и ОП за счет того, что страницы программы могут не загружаться в ОП, пока они действительно не понадобятся.

Система виртуальной памяти со страничной организацией функционирует аналогично системе сегментированной памяти. Сначала в ОП загружается начальная страница программы и ей передается управление. Если в процессе выполнения программы потребуется выборка операндов из другой страницы, процессор обнаружит ее отсутствие и передаст управление ОС, которая загрузит отсутствующую страницу. Особенно заметны преимущества страничной организации памяти при реализации мультизадачных систем: при загрузке новой задачи ее страницы могут быть направлены в любые свободные в дан-

ный момент физические страницы ОП. Именно этим и объясняется широкое применение страничной организации памяти для систем виртуальной памяти с подкачкой страниц по запросу.

Соответствие между виртуальными и физическими страницами устанавливается ОС с помощью таблицы дескрипторов страниц. Операционная система выполняет заполнение таблицы в процессе распределения памяти. Количество адресуемых страниц определяется размером этой таблицы. Каждая строка таблицы (страничный дескриптор) содержит базовый адрес страницы в ОП, биты присутствия P, обращения A, модификации D и биты защиты страниц. Биты присутствия P и обращения A используются для управления виртуальной памятью. Их назначение аналогично функциям, выполняемым одноименными битами дескрипторов сегментов. При обращении к странице с установленным битом P ($P = 1$) из таблицы соответствия извлекается базовый адрес страницы с искомым операндом. Физический адрес операнда образуется при суммировании базового адреса с внутристраничным смещением, указываемым в логическом (виртуальном) адресе операнда. При обращении к странице, отсутствующей в ОП (при $P = 0$), процессор формирует особый случай неприсутствия страницы и передает управление ОС. Последняя, обрабатывая этот особый случай, считывает из ВЗУ отсутствующую страницу в ОП, после чего управление возвращается процессору, который повторно выполняет команду обращения к памяти.

Признак модификации D (Dirty) устанавливается в страничном дескрипторе при записи в определяемую дескриптором страницу. Операционная система использует значение бита D для исключения необязательных свопингов страницы: при $D = 0$ к странице не было обращений для записи и передавать страницу из ОП в ВЗУ нет необходимости.

В отличие от сегментации страницы не имеют прямой связи с логической структурой программ. Трансляция виртуального адреса в физический осуществляется автоматически.

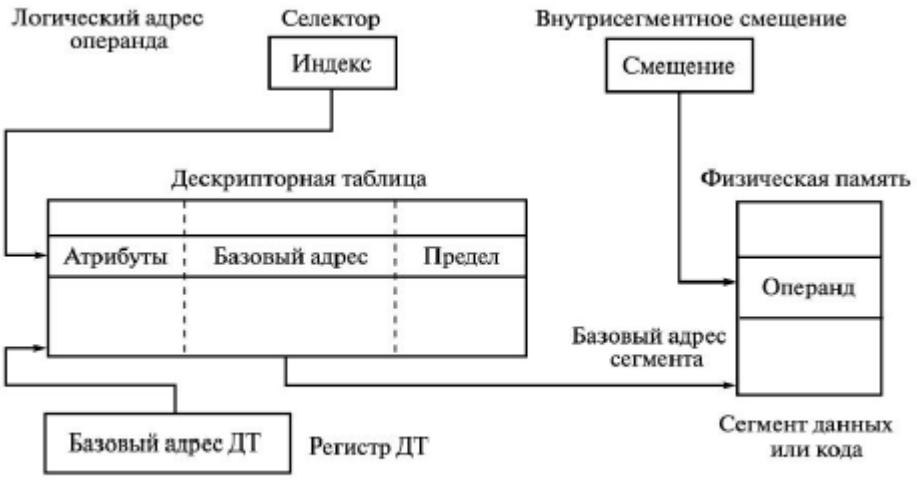


Рис. 4.30. Формирование физического адреса операнда сегментированной памяти