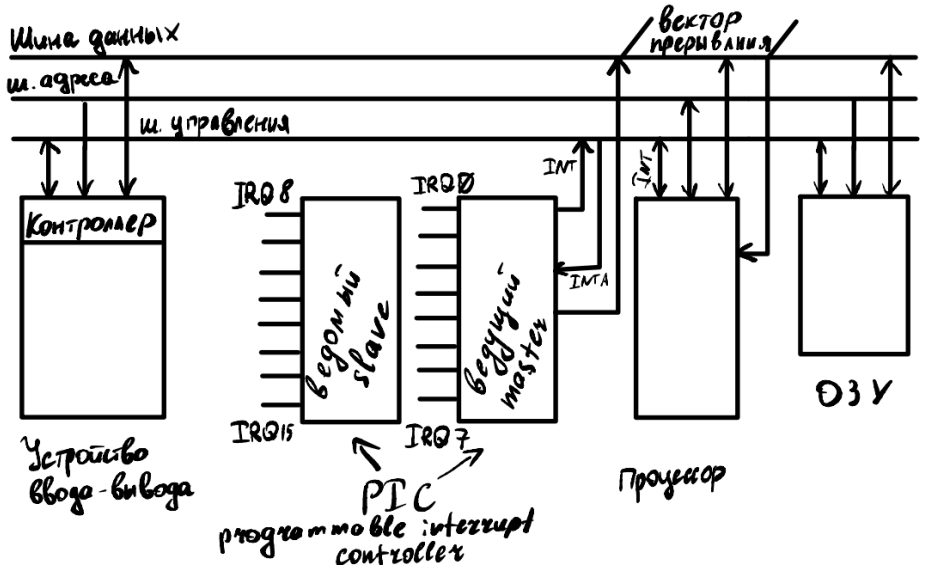
Оглавление

1 Ликбез по 1 семестру

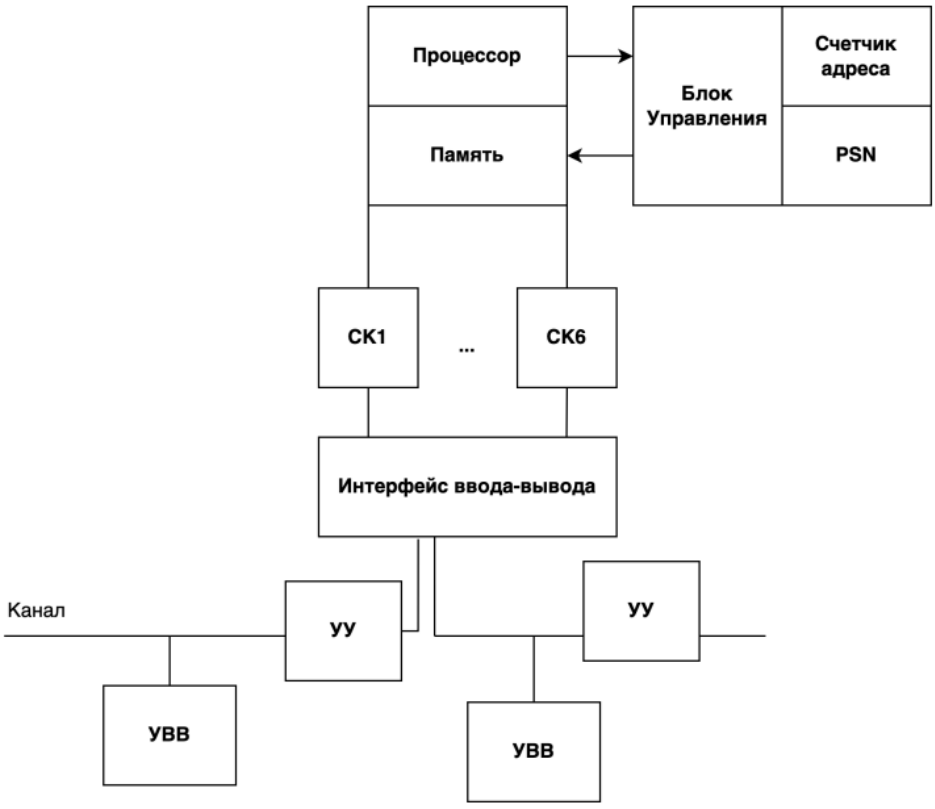
1.1 Концептуальная трехшинная архитектура

В компьютерах передаются сигналы трех типов: данные (данные и команды), адреса, сигналы управления.



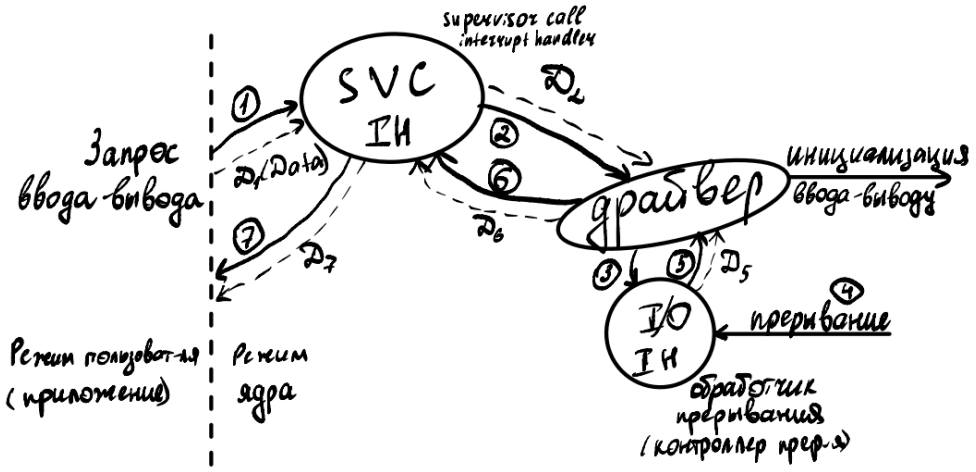
На линию IRQ приходит сигнал. Когда на контроллер приходит прерывание, он формирует сигнал INT который по шине управления поступает на выделенную ножку процессора. В конце цикла выполнения команды процессор проверяет наличие сигнала на выделенной ножке, если пришел сигнал прерывания то контроллер посылает INTA. Получив INTA контроллер по шине данных выставляет вектор прерывания (базовый вектор + номер линии IRQ). В реальном режиме процессор считывает вектор прерывания и переходит на его выполнение, переведя его смещение по таблице и считав адрес обработчика.

1.2 Канальная архитектура



Канал – программно-управляемое устройство – получает канальную программу от процессора. Быстрые устройства подключаются к селекторным каналам. Медленные устройства к мультиплексорным.

1.3 Прерывания в последовательности ввода-вывода



В современных системах для отдельно стоящей машины блокирующий ввод-вывод — основная модель IO (запросив IO, приложение блокируется, в система переходит в режим ядра и запрос обслуживается).

В этом виде ввода/вывода в полной мере реализуется идея распараллеливания функций (3-е поколение ЭВМ), которая базируется на стремлении освободить процессор от непроизводительных действий управления медленными внешними устройствами. Пока медленное внешнее устройство выполняет задачу ввода данных, процессор может перейти на выполнение другой работы.

В результате выполнения функций ядра будет вызван драйвер устройства, он формирует команды и данные и передает их по шине данных УВВ. Далее управление передается каналу или контроллеру. Контроллер сохраняет данные и возвращает их приложению.

1.4 Классификация прерываний

- Системные вызовы (программные прерывания) – процесс переключается в режим ядра и выполняет реентерабельный код ядра, после переключается в режим пользователя
- Исключения. Синхронные. 2 типа
  - Исправимые. страничное прерывание.
  - Неисправимые
- Аппаратные. Асинхронные. 3 типа
  - От системного таймера. Является единственным периодическим событием в системе. прерывание.
  - От устройств ввода/вывода.
  - От действий оператора

2 Возможно стоит знать

2.1 Архитектуры

Сетевое взаимодействие связано с внешним устройством – сетевой картой. Внешние устройства адресуются портами ввода-вывода. АП данных и УВВ начинаются с нуля, возникает наложение. Начиная с PowerPC введены два моста – северный, южный и bus mastering (захват шины). Шина PCI – между внешними устройства (вытеснило ISA шину). Мост определяет кому направить данные – внешнему устройству или памяти. Адресация УВВ и памяти контролируется с помощью захвата шины.

DMA (direct memory access) – передача данных не через регистры процессора. Контроллер прямого доступа и памяти позволяет передавать данные минуя процессор, не загружая данные из ОП и наоборот.

В SMP архитектуре все процессоры равноправны. Процессоры подключаются к общей памяти. NUMA альтернатива к SMP. NUMA (non-uniform memory access) – все процессоры имеют прямой доступ к памяти определенного объема, и могут обращаться к памяти других процессоров. Это позволяет процессорам работать с памятью параллельно.

В шинах PCI не используются физические линии прерываний, прерывания доставляют как MSI

2.2 Мультиплексирование

– select/poll. Устаревшее API
– epoll

Существует два режима наблюдения
– Level-triggered – файловый дескриптор возвращается, если остались непрочитанные/записанные данные
– Edge-triggered – файловый дескриптор с событием возвращается, только если с момента последнего возврата произошли новые события (например, пришли новые данные)

epoll instance – ключевой концепт epoll API, структура данных содержащая список файловых дескрипторов для мониторинга и список готовых дескрипторов для работы.

set_ nonblocking(fd) – переводит сокет в неблокирующий режим.

epoll_ create1(0) – создаёт структуру данных (epoll instance), с которой в дальнейшем идёт работа. Структура одна для всех файловых дескрипторов, за которыми идёт наблюдение. Функция возвращает файловый дескриптор, который в дальнейшем передаётся во все остальные вызовы epoll API.

epoll_ ctl(epollfd, op, fd, event) – используется для управления epoll instance, в частности, позволяет выполнять операции EPOLL_CTL_ADD (добавление файлового дескриптора к наблюдению), EPOLL_CTL_DEL (удаление файлового дескриптора из наблюдения), EPOLL_CTL_MOD (изменение параметров наблюдения), EPOLL_CTL_DISABLE – для безопасного отключения наблюдения за файловым дескриптором в многопоточных приложениях. EPOLLET устанавливает edge-triggered режим.

epoll_ wait(epollfd, events, maxevents, timeout) – возвращает количество (один или более) файловых дескрипторов из списка наблюдения, у которых поменялось состояние (которые готовы к вводу-выводу).

События, за которыми можно наблюдать с помощью epoll:
– EPOLLIN — новые данные (для чтения) в файловом дескрипторе
– EPOLLOUT — файловый дескриптор готов продолжить принимать данные (для записи)
– EPOLLERR — в файловом дескрипторе произошла ошибка
– EPOLLHUP — закрытие файлового дескриптора

2.3 Монитор Хоара

2 типа процессов – читатели, могут читать параллельно; писатели, работают в режиме монопольного доступа. 5 семафора: 3 считающих: активные читатели, очередь ждущих писателей, очередь ждущих читателей и 2 бинарный: активный писатель и захват критической секции.

В мониторе определены 4 процедуры: start_read, stop_read, start_write, stop_write.

Для того, чтобы читатель начал читать, он вызывает start_read. Читатель может читать, если нет активного писателя и очередь ждущих писателей пуста. В противном случае процесс блокирован на

переменной типа условие may_read. Увеличивается число активных читателей и посылается сигнал, что можно читать – (каждый читатель активизирует следующего). Когда читатель завершает чтение он вызывает stop_read. Уменьшается число активных читателей. Если нет активных читателей, то посылается сигнал «можно писать».

Для того, чтобы писатель начал писать, он вызывает start_write. Писатель может писать, если нет активного писателя и активных читателей. В противном случае процесс блокирован на переменной типа условие may_write. Когда процесс заканчивает писать – вызывает stop_write. Есть в stop есть ждущие читатели, дальше читают они, иначе писатель начинает писать.

В мониторе исключается бесконечное откладывание читателей/писателей.

2.4 Семафоры

Набор семафоров, чтобы одной неделимой операцией изменять состояние над частью семафоров набора

Семафоры предложил Дейкстра для того, чтобы убрать активное ожидание (на проверку в цикле тратится процессорное время). Семафор – не отрицательная защищенная переменная, на которой определены 2 неделимые операции: P(декремент) и V(инкремент). Если семафор принимает только знач. 0,1, то он называется бинарным. Если может принимать неотрицательные целые - считающий. Операция P(S) при $S = 0$ заблокирует процесс, чтобы остаться в области целых неотрицательные чисел. Операция V(S) при $S = 0$ разблокирует процесс, ожидающий семафор. За экономией времени платим переходом в режим ядра (P и V - системные вызовы). Считающие семафоры контролирует значения величин, бинарные – монопольный доступ.

semop – операции на семафоре. Для semop определена структура struct sembuf – номер семафора (наборы семафоров представлены как массивы семафоров), в наборе каждый семафор достается по номеру; semop – операция; semflg – флаги. semop получает semid, указатель на массив структур и intlen – количество семафоров, на которых будет выполнен semop (неделимая операция). semctl производит операции управления.

2.5 Системные вызовы

exec – переводит процесс на новое адресное пространство программы, путь к которой передаётся в качестве аргумента.

fork – создает новый процесс потомок, который является копией процесса предка, в том смысле, что он наследует от предка дескрипторы открытых файлов, сигнальную маску и переменные окружения.

2.6 Способы взаимоисключения

- программный
- аппаратный
- с помощью семафоров
- с использованием мониторов

3 struct task_struct

Каждый процесс в системе описывается структурой struct task_struct. В загружаемом модуле можно напрямую получить доступ к полям структуры struct task_struct, которая содержит информацию о процессе. В ядре определен специальный символ current, который является указателем на структуру struct task_struct для текущего процесса.

```
1 static int __init my_init(void)
2 {
3         struct task_struct *task;
4         do {
5                 printk(KERN_INFO "%d %s %d %d", task->pid, task->comm, task->ppid,
    task->prio);
6         } while(task = next_task(task) != &init_task);
7         return 0;
8 }
```

Флаги

- PF_NOFREEZE - – не может быть блокирован.
- PF_FORKNOEXEC – не может вызвать exec(), то есть не может сменить адресное пространство по адресное пространство программы, передаваемой в качестве фактического параметра exec(). (не позволяет внедрить вредоносный код в прогу).
- PF_NOSETAFFINITY – запрещает пользователю менять cpu_mask
- PF_KTHREAD – поток ядра

Приоритет: чем меньше значение, тем выше приоритет. Приоритет 0 является максимальным. Для RT-процессов (реального времени) диапазон: 1–99. Для обычных процессов: 100–139.

- SCHED_NORMAL – используется для обычных процессов в системе. Выделяется процессорное время на основе приоритета - процесс с более высоким приоритетом будет выполняться до тех пор, пока не истечет его квота или пока система не переключится на другой процесс.
- SCHED_FIFO – процессы организуются в FIFO-очередь. Он будет выполняться, пока не завершит свою задачу или не будет прерван. Является политикой планирования без переключения
- SCHED_RR – процессы организуются в очередь. Если процесс исчерпал квант, но не успел завершиться, то он помещается в конец очереди. Является политикой планирования с переключением:
- SCHED_BATCH – предназначена для выполнения «пакетных» задач, которые могут длиться долго и не требуют быстрых откликов. Пакетные процессы выполняются, когда в системе нет активных высокоприоритетных задач.
- SCHED_IDLE – фоновый (холостой) процесс. Предназначена для процессов, которые выполняются в фоновом режиме и имеют самый низкий приоритет.

Справедливость SHED_NORMAL в том, чтобы в итоге все процессы получили процессорное время и по возможности в равных количествах.

Состояния

- TASK_RUNNING – Задача выполняется или готова к выполнению
- TASK_INTERRUPTIBLE – Задача спит, можно разбудить сигналом
- TASK_UNINTERRUPTIBLE – Задача спит, нельзя разбудить сигналом

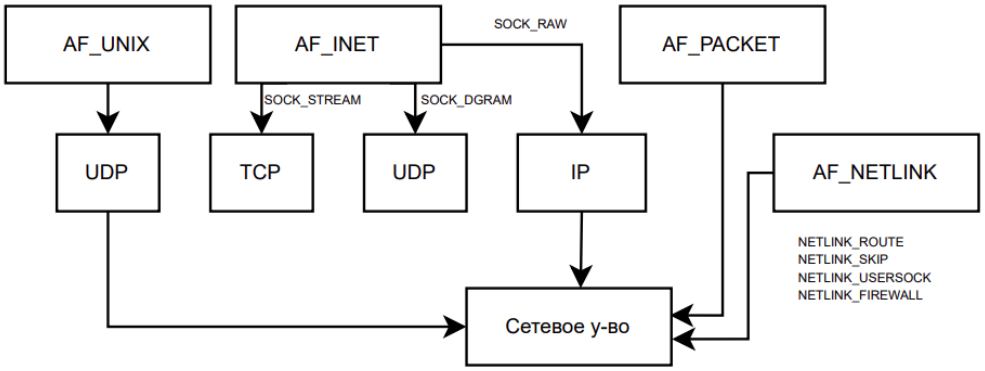
Процесс миграции, например, migration/0, является специальным процессом, который помогает системе управлять процессами, которые необходимо перенести с одного ядра на другое.

4 Сокеты

Средства взаимодействия параллельных процессов. Универсальность сокетов – позволяют организовать взаимодействие и на отдельной машине и в распределенной системе.

Сокет – абстракция конечной точки взаимодействия. Создание сокета: *int socket(int family, int type, int protocol)*, family – семейство (сокеты могут принадлежать различным семействам), тип и протокол.

Если сокеты используются для взаимодействия процессов на отдельной машине, то по сути адреса – специальные файлы. В распределенных системах используются сокеты AF_INET.



Сетевой устройство имеет драйвер и когда на карту приходит пакеты приходит аппаратное прерывание.

Ядро Linux предоставляет для работы с сокетами системный вызов

```
asmlinkage long sys_socketcall(int call, unsigned long *args) {
        unsigned long a[6];
        int err;
        if (copy_from_user(a, args, nargs[call]))
                return -EFAULT;
        unsigned long a0=a[0];
        unsigned long a1=a[1];
        switch(call)
        {
                case SYS_SOCKET: err = sys_socket(a0,a1,a[2]); break;
                case SYS_BIND: err = sys_bind(a0, (struct sockaddr *)a1, a[2]); break;
                case SYS_CONNECT: err = sys_connect(a0, (struct sockaddr *)a1, a[2]);
   break
                ...
                default: err = -EINVAL; break;
        }
}
```

В switch перечисляют средства все функции сетевого стека. Взаимодействие на сокетах всегда клиент-сервер, даже на отдельных машинах.

```
asmlinkage long sys_socket(int family, int typ, int protocol) {
        struct socket *sock;
        ...
        int retval = sock_create(family, type, protocol, &sock);
        ...
        return retval;
}
```

```
struct socket {
        socket_state state;
        short type;
        unsigned long flags;
        const struct proto_ops *ops; // протокол
        struct fasync_struct *fasync_list; // список асинхронного запуска
        struct file *file; // дескриптор файла
```

```
        struct sock *sk;
        wait_queue_head_t wait;
};
```

У сокета различают 5 состояний

– SS_FREE - свободный сокет, с которым можно соединяться;
– SS_UNCONNECTED - несоединенный сокет;
– SS_CONNECTING - сокет находится в состоянии соединения;
– SS_CONNECTED - соединенный сокет;
– SS_DISCONNECTING - сокет разъединяется в данный момент

Дизайн сокетов следует парадигме UNIX: в идеале отобразить все объекты которые осуществляют чтение/запись на файлы, чтобы с ними можно было работать используя обычные функции чтения/записи из/в файлы.

Семейства

– AF_UNIX – для взаимодействия на локальной машине;
– AF_INET – сокеты семейства протоколов TCP, IP, UDP, основанные на протоколе IPv4;
– AF_INET6 – IPv6;
– AF_IPX – протокол IPX;
– AF_UNSPEC – неопределенный домен.

Тип

– SOCK_STREAM – потоковые сокеты, ориентированные на потоки. Надежное, упорядоченное, полнодуплексное логическое соединение;
– SOCK_DGRAM – — определяет ненадежную службу без установки логического соединения, где пакеты могут передаваться без сохранения порядка;
– SOCK_RAW – — низкоуровневый сокет, обеспечивает прямой доступ к сетевому протоколу.

Протокол

– 0 – если AF_INET и SOCK_STREAM – по умолчанию TCP, AF_UNIX и SOCK_DGRAM – по умолчанию UPD;
– явное указание IPPROTO_* (например, IPPROTO_TCP)

Системный вызов socket не определяет, что с чем соединять, то есть не определяет адрес

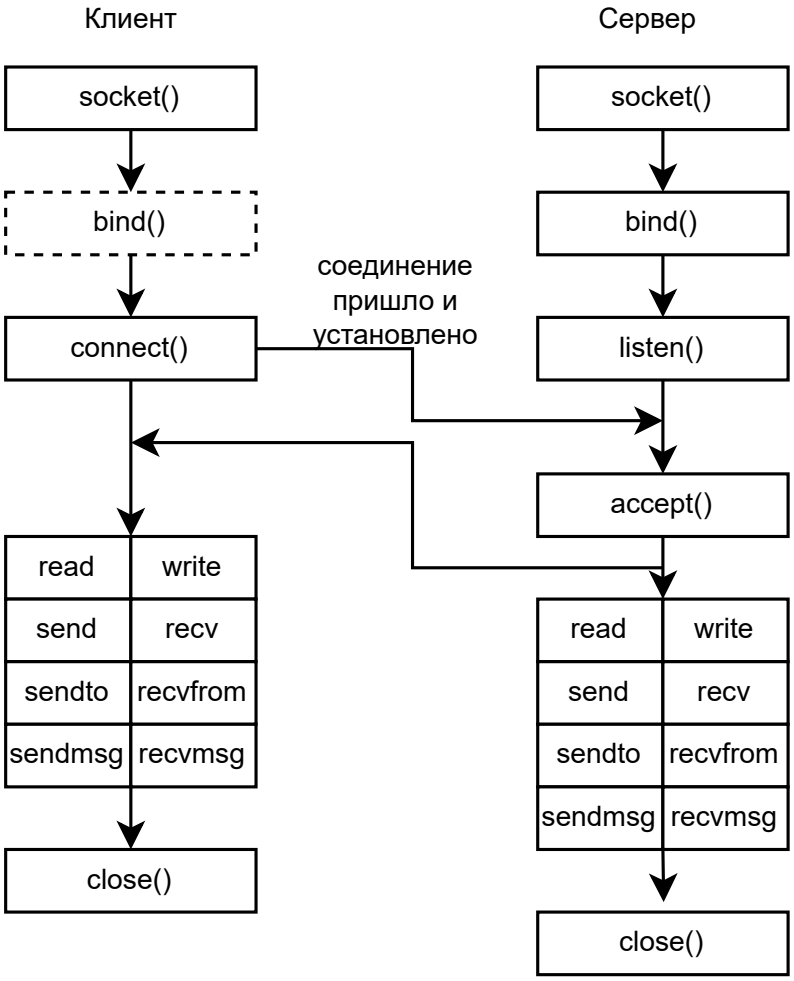
```
struct sockaddr
{
        sa_family_t sa_family;
        char sa_data[14];
}
```

Парные сокеты *int socketpair(int domain, int type, int protocol, int sv[2]);* - альтернатива программным каналом, обеспечивают дуплексную связь, но не реализовано взаимоисключающие. sockaddr – базовый адрес в работе с сокетами. Для сокетов AF_UNIX в массиве имя файла и семейство. AF_UNIX реализуется в пространстве файловых имен. Для сетевых сокетов необходима детализация и используется sockaddr_in.

На адресацию в сети 14 байт не хватит, введена структура для адресации по сети:

```
struct in_addr {
        _u32 s_addr;
};

struct sockaddr_in
{
        short int sin_family; // Семейство
        unsigned short int sin_port; // порт
        struct in_addr sin_addr; // IP-адрес
        unsigned char sin_zero[8]; // выравнивание
};
```

Модель клиент-сервер позволяет создать и сервер и клиент и настроить взаимодействие на одной машине. На сокетах определены системные вызовы, образующие сетевой стек. Стеком является только потому, что функции вызываются в определенном порядке.



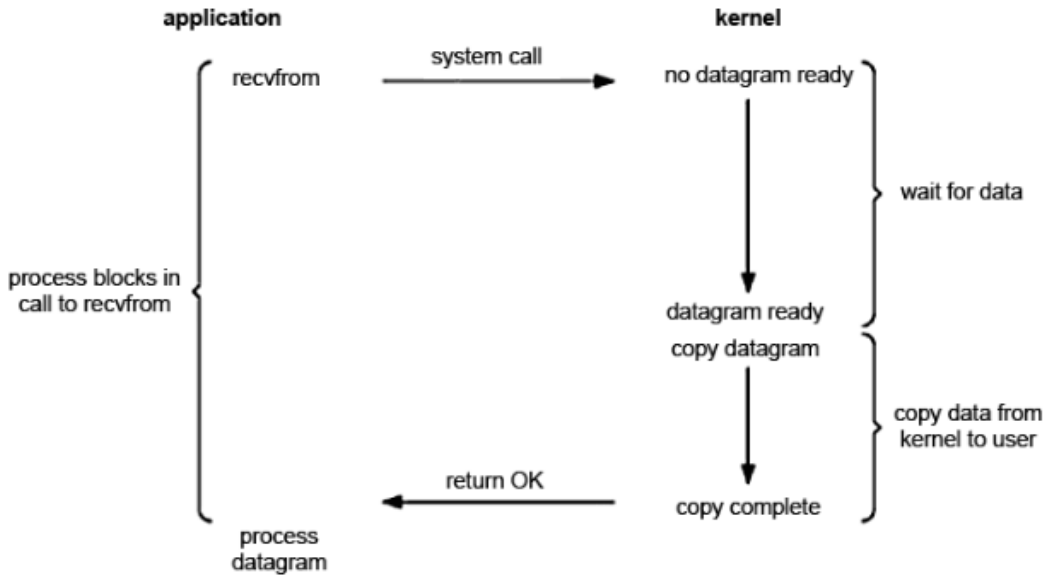
- *int socket(int family, int type, int protocol)* – возвращает файловый дескриптор, который нужно связать с адресом.
- *int bind(int sockfd, struct sockaddr *addr, int addrlen);*
  Связывает сокет с адресом. На стороне клиента нужен не всегда. После связывания переходит в режим пассивного ожидания соединения.
- *int connect(int sockfd, struct sockaddr *serv_ addr, int ad drlen);*
  Клиенты присоединяются к серверу пытаясь установить связь. Клиент знает адрес сервера
- *int listen(int sockfd, int backlog);*
  Вызывается сервером для получения сообщение от клиентов. Сервер сообщает ядру о переходе в состояние пассивного ожидания запросов клиента.
- *int accept(int sockfd, void *addr, int *addrlen);*
  Выполняется на стороне сервера для того, что принять соединение. Если пришел запрос на соединение, то вызывается accept. Создает копию исходного сокета, возвращает файловый дескриптор исходного дескриптора, чтобы исходный сокет оставался в режиме прослушивания соединений, а копия находилась в состоянии соединения.

Протокол TCP реализован в виде тройного рукопажатия, чтобы обеспечить надежную связь клиента и сервера.

Два типа порядка байтов: Прямой (host byte order) от старших к младшим разрядам и обратный, сетевой (network byte order) наоборот. Функции для конвертации *htons*, *ntohs* и htonl, ntohl.

5 Модели ввода-вывода

5.1 Модель блокирующего ввода-вывода



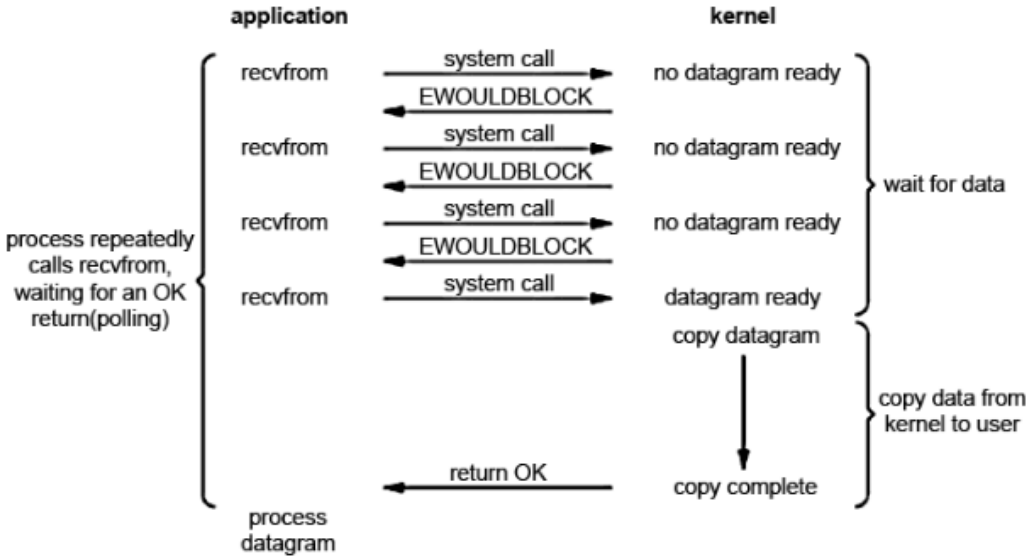
При вводе-выводе, система должна инициировать обращение к внешнему устройству. Получить сразу по запросу приложение данные не может. Обращение к внешним устройствам — длительное действие.

Запросив ввод-вывод, процесс будет блокирован (на диаграмме — no datagram ready).

Когда внешнее устройство завершает операцию ввода-вывода, он формирует сигнал прерывания, по которому формируется вектор прерывания. По этому вектору процессор адресует обработчик прерывания.

Данные из буфера устройства записываются в буфер ядра и процесс должен быть разблокирован.

5.2 Модель неблокирующего ввода-вывода



Исторически более ранняя модель, процессор тратит время на опрос о готовности данных, нет прерываний.

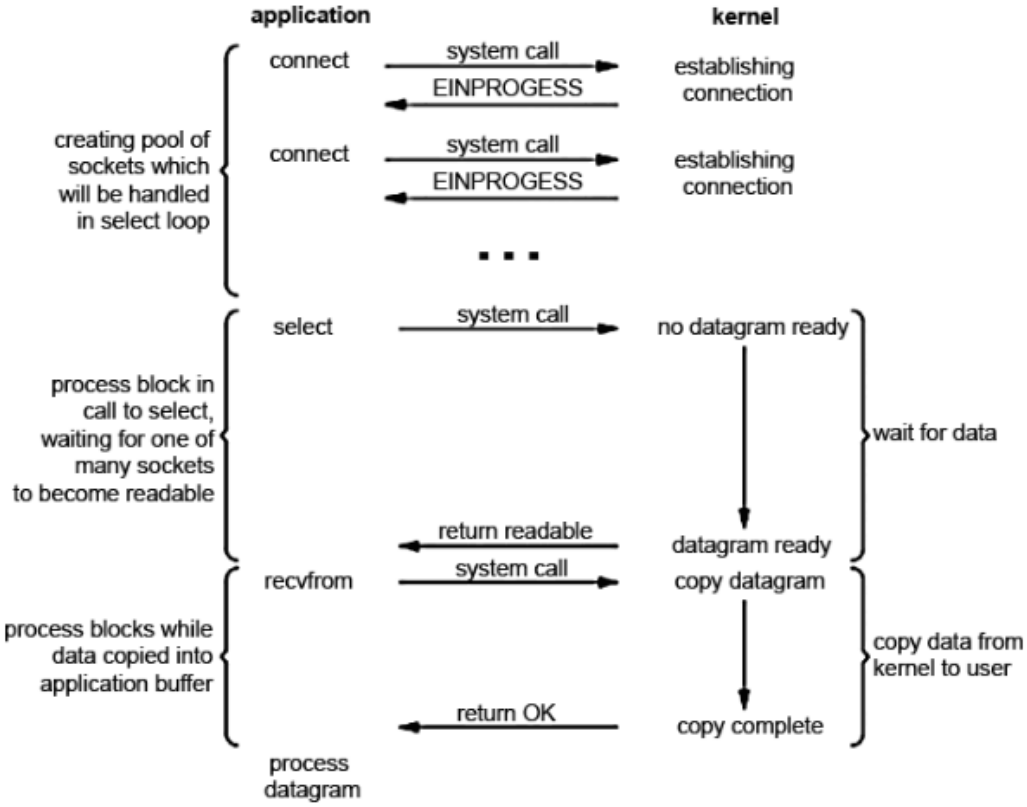
В какой-то момент пакеты данных начинают приходить с большой частотой на сетевую карту – много аппаратных прерываний. Обслуживание аппаратных прерываний затратно. NAPI (new api), переход в режим поллинга (опроса), что быстрее АП.

5.3 Мультиплексирование ввода-вывода

Для такой реализации нужно системные вызов *poll()*, *select()*, *epoll()*, *pselect()*.

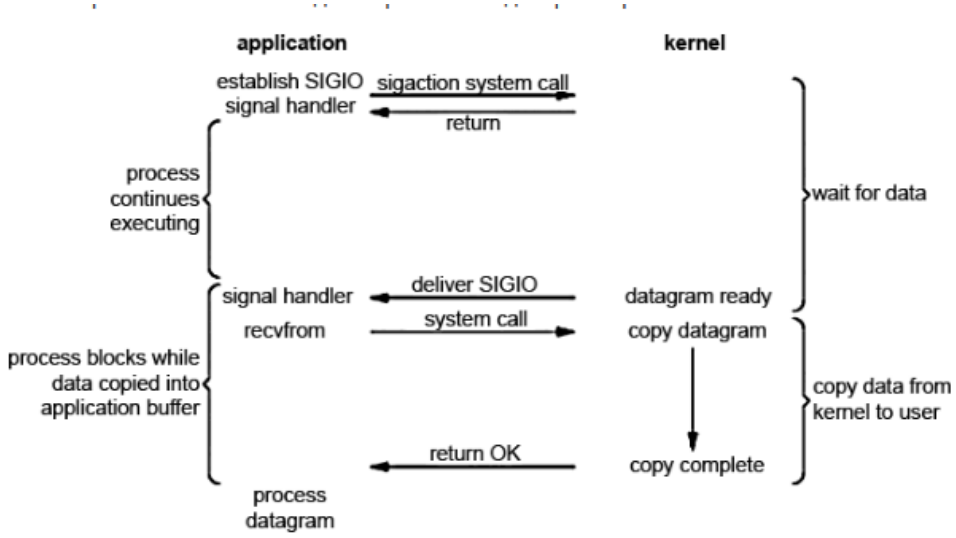
Мультиплексор вызывается до accept(), после listen(). Ядро выполняет мультиплексирование, начиная опрашивать готовность сокетов. Блокировка по сути разделена на две. Первая блокировка связана с select() и закончится когда любой из сокетов станет доступным для взаимодействия. В результате будет установлено соединение и начнется обмен данных. Процесс будет блокироваться так как необходимо получить данные в результате receive()

Мультиплексор объединяет информацию из несколько входов и выдает ее по одному входному каналу.



Преимущество мультиплексирования – сокращение времени блокировки.

5.4 Управляемый сигналом ввод-вывод

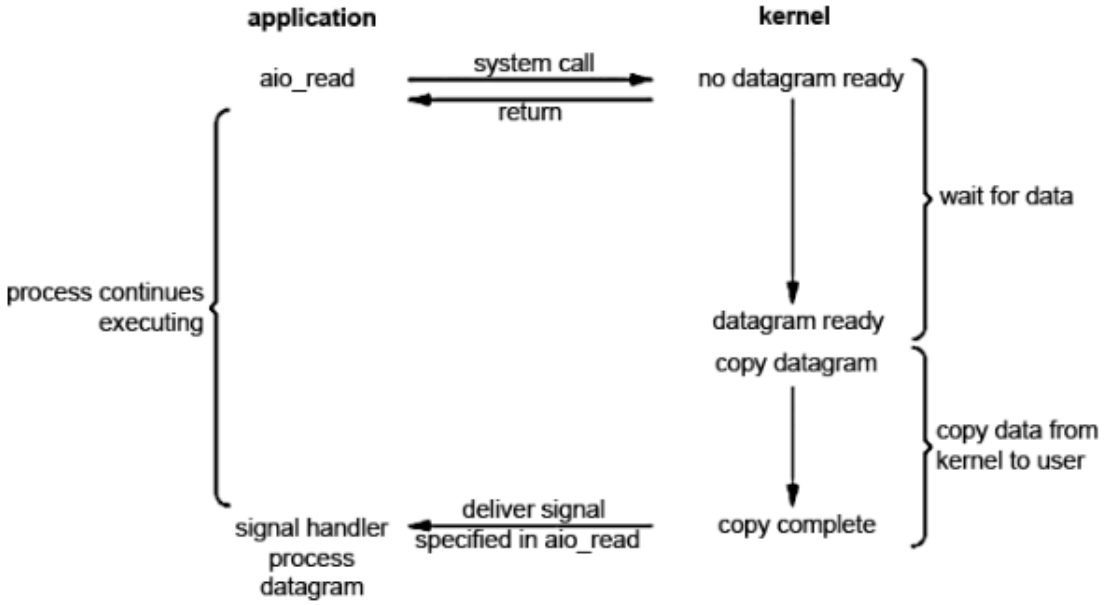


Ядро отслеживает когда данные будут готовы, оно посылает сигнал и вызывается обработчик,

так что приложение не нужно блокировать. Запросив данные может выполняться другое действие не связанное с этим.

5.5 Асинхронный ввод-вывод

Ядро должно предоставлять пользователю специальные системные вызовы асинхронного ввода-вывода.



Проблема как и в прошлой модели – найти работу, которую может выполнять процесс пока данные не готовы.

	Блокирующий	Неблокирующий
Синхронный	read, write	read, write (O_NONBLOCK)
Асинхронный	I/O мультиплексирование (select/poll) + сигналы	AIO

6 Файлы

Файл – средство долговременного хранения данных, любая поименованная совокупность данных, может быть бессмысленная, во вторичной памяти. В UNIX все файл.

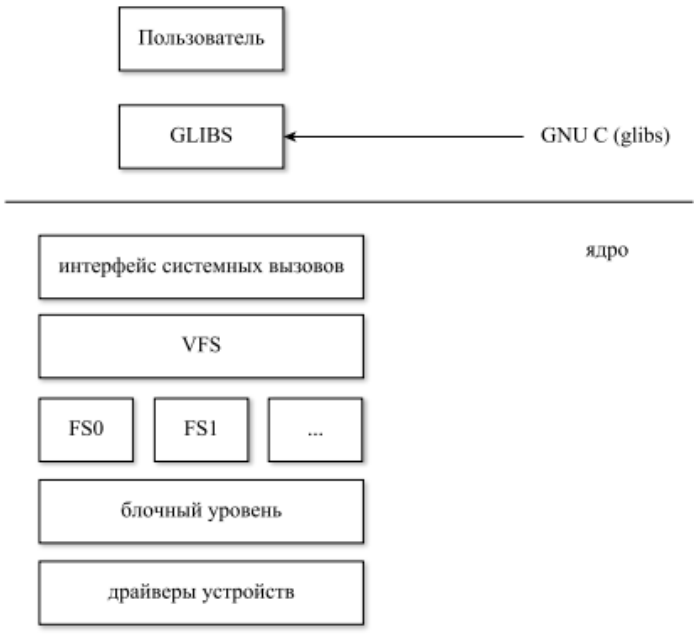
Файловая система – часть ОС, которая определяет способ организации, хранения и доступа к файлам.

UNIX поддерживает большое количество различных файловых систем. ФС UNIX организована через интерфейс VFS/vnode. В Linux не определена vnode, только VFS. Задача vnode – чтобы ОС без изменений ядра могла поддерживать большое кол-во самых разных ФС.

Ядро UNIX имеет иерархическую структуру по отношению к аппаратному обеспечению. Принято выделять следующие уровни ФС:

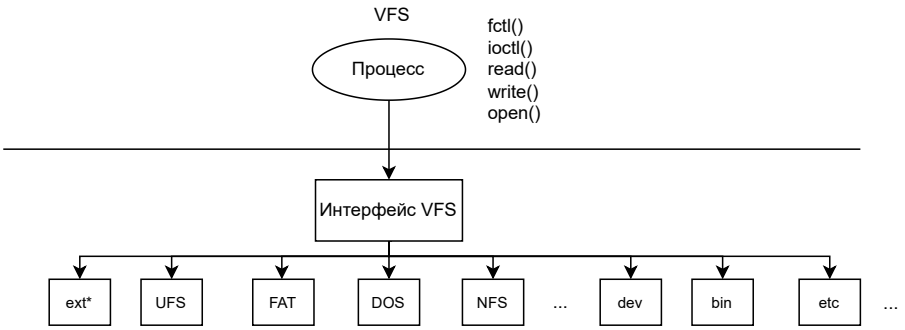
Символьный – именование файлов, строки символов. Не являются идентификатором. Имя файла – это полное имя файла начиная с корневого каталога, содержит все элементы пути и завершается shortname. inode – базовый уровень – уникальный идентификатор файла. Любой файл в логическом представлении начинается с нулевого смещения и его АП непрерывно. Файлы бывают текстовые и бинарные (байтоориентированные). Linux поддерживает FHS (File system standart). FHS определяет структуру и содержимое каталогов. Существует корневая ФС – «\». Корневой каталог и его ветви должны составлять единую ФС, расположенную на одном разделе диске. Существует 2 типа устройств – символьные и блочные.

Структура VFS: четыре структуры для представления VFS – super_block, inode, dentry и file.

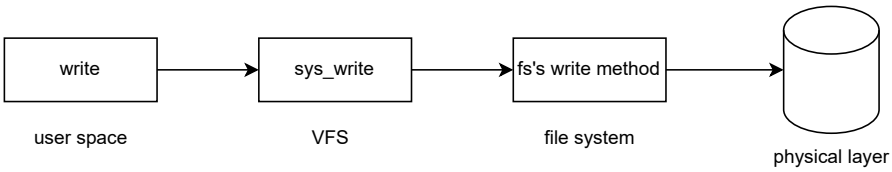


Разделенные на независимые порции потоки данных диска называются секторами. Сектор диска

– участок дорожки магнитного диска представляющий собой наименьший размер порции данных, которая может быть изменена в результате форматирования (номер дорожки + номер сектора). На диске вся информация адресуется, имеет уникальный адрес. Раздел – выделенная на диске АП, характеризуется группами блоков.



Операции над файлами в любой ФС будут подобны



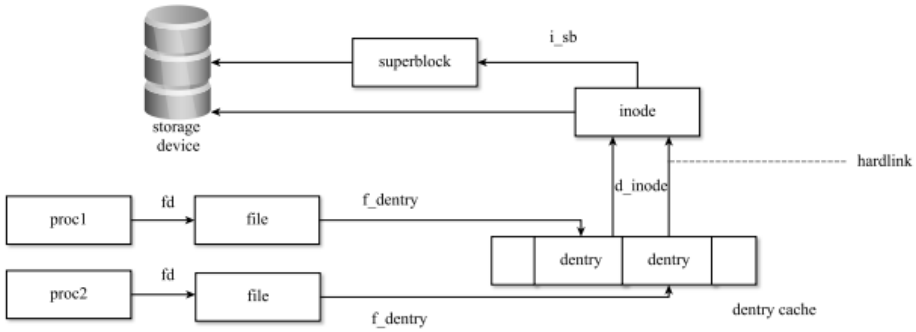
Монтирование ФС – например подключение флешки (у нее своя ФС).

Дерево файлов и каталогов Linux формируется из ветвей которые могут соответствовать различным физическим носителям (дерево формируется из отдельных ФС).

ФС – большое дерево с корневой директорией «\». Монтирование корневой ФС – часть процесса инициализации. Монтирование – последовательность действий в результате которой ФС становится доступной.

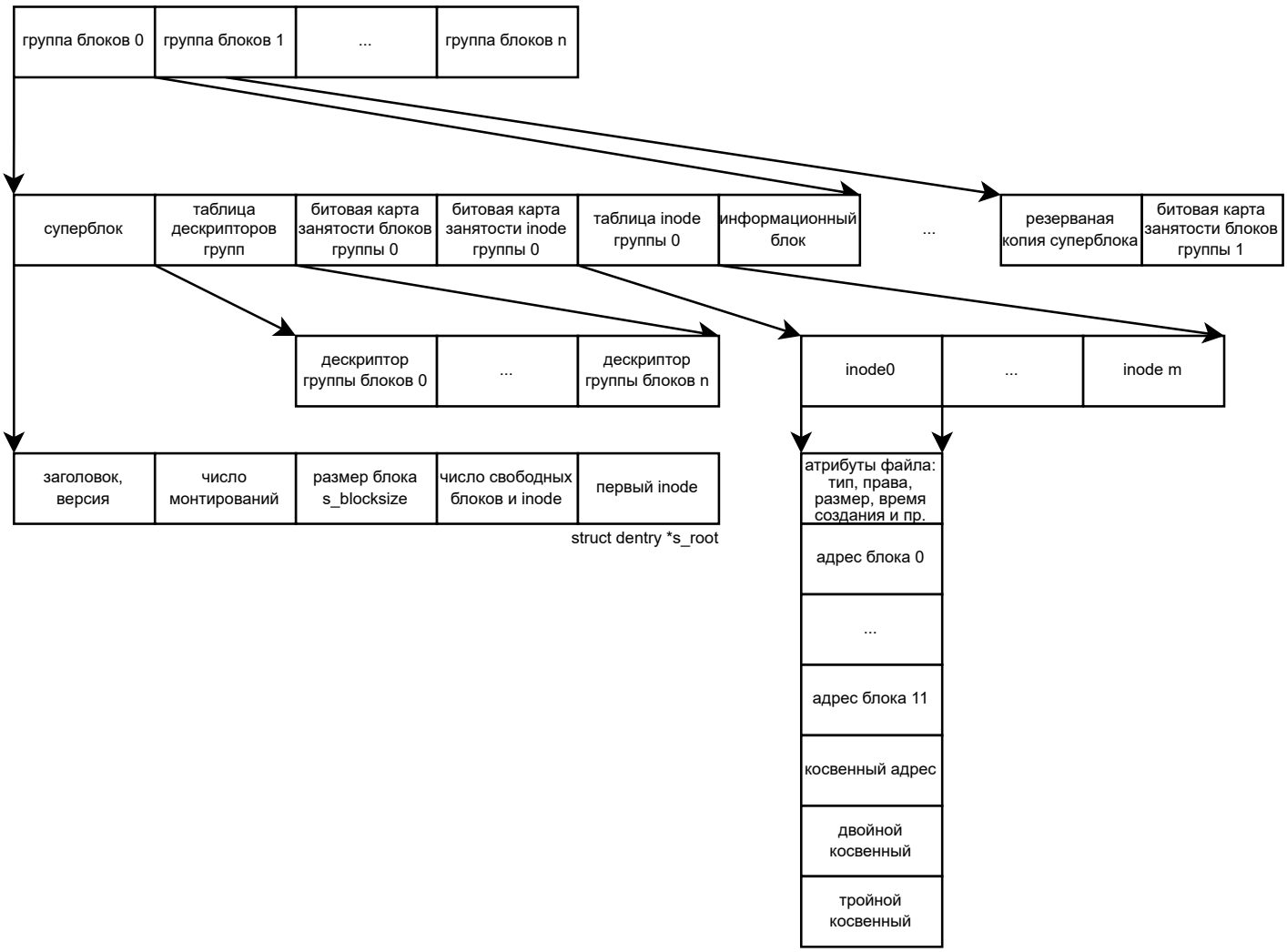
Когда ФС монтируется в существующую директорию все файлы и директорий этой ФС становятся файлами и директориями точки монтирования. Если эти точки содержат какие-то файлы или директории, то они становятся невидимыми.

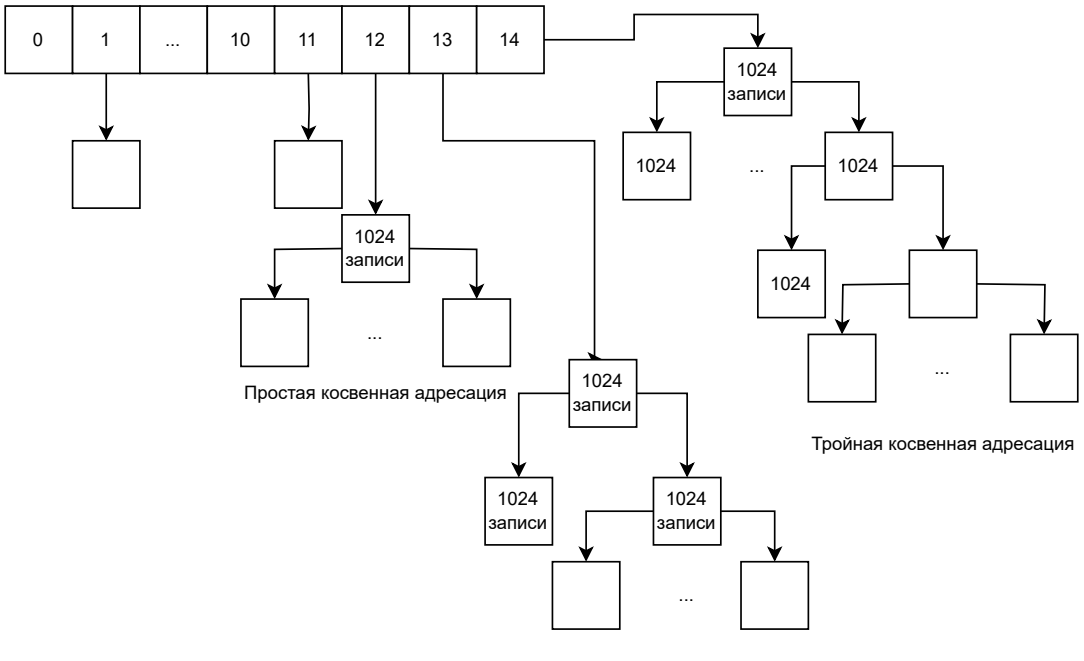
mount <ключи> -t <тип ФС> -o <опции ФС> <имя УВВ/Другого устройства содержащего ФС> <точки монтирования>



Суперблок – структура, содержащая информацию необходимую для монтирования и для доступа к файлам. В каждой ФС ровно 1 суперблок. Инициализированный суперблок располагается в начале раздела. Все АП делится на разделы, в начале суперблок, содержащий всю необходимую информацию для доступа к физическим файлам которые будут хранится в рамках ФС.

struct superblock – структура, описывающая смонтированную ФС. Содержит метаданные для обращения к файлам ФС. АП диска делится на разделы. Один раздел занимает одна ФС.


группа блоков 0
группа блоков 1
...
группа блоков n
суперблок
таблица дескрипторов групп
битовая карта занятости блоков группы 0
битовая карта занятости inode группы 0
таблица inode группы 0
информационный блок
...
резерваная копия суперблока
битовая карта занятости блоков группы 1
дескриптор группы блоков 0
...
дескриптор группы блоков n
inode0
...
inode m
заголовок, версия
число монтирований
размер блока s_blocksize
число свободных блоков и inode
первый inode
struct dentry *s_root
атрибуты файла: тип, права, размер, время создания и пр.
адрес блока 0
...
адрес блока 11
косвенный адрес
двойной косвенный
тройной косвенный


0
1
...
10
11
12
13
14
1024 записи
1024
...
1024
1024 записи
...
Простая косвенная адресация
1024
1024 записи
1024 записи
...
Тройная косвенная адресация
Двойная косвенная адресация

Таким образом можно хранить огромные файлы, благодаря тому что они хранятся по частям в свободных блоках диска.

```
struct superblock {
        struct list_head s_list;                // обеспечивает организацию связного сп
    иска
        dev_t s_dev; // информация об устройстве, на котором находится файловая система
        unsigned long s_block_size;         // размер блока в байтах
        loff_t s_maxbytes;                      // максимально возможный размер файла
        struct file_system_types *s_type;       // тип файловой системы
        const struct super_operations *s_op;    // все операции, определенные на суперб
    локе
        const struct export_operations *s_export_op; // операции, не учтенные ядром
        ...
        unsigned long s_flags;                  // флаги монтирования
        ...
        unsigned long s_magic;                  // магическое число для доступа к супер
    блоку
        struct dentry *s_root;                  // точка монтирования
        ...
        int s_count;                            // счетчик ссылок на суперблок
        struct list_head s_mounts;              // список монтирований
        struct block_device *s_bdev;            // связанное блочное устройство
        char s_id[32];                          // символьное имя
        ...
        fmode_t s_mode;                         // права доступа для монтирования
        const struct dentry_operations *s_d_op; // в этом суперблоке операции над
    dentry
        struct list_head s_inodes;              // список всех inode'ов
        ...
}
```

```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*free_inode)(struct inode *);
        void (*dirty_inode) (struct inode *, int flags); // вызывается VFS, если в
    inode были внесены изменения, чтобы не потерять изменения и отразить их в дисковом
    inode
        int (*write_inode) (struct inode *, struct writeback_control *wbc); // записыва
    ет указанный inode на диск
        int (*drop_inode) (struct inode *); // вызывается VFS, когда удаляется последня
    я ссылка на inode.
        void (*put_super) (struct super_block *); // вызывается VFS при размонтировании
     файловой системы
        int (*statfs)(struct dentry *, struct kstatfs *);
};
```

В перечне функций super_operations нет функции создания суперблока, но такая функция есть.

```
static struct super_block* alloc_super(struct file_system_type *type, int flags,
    struct user_name_space *user_ns) {
        struct super_block* s = kzalloc(sizeof(struct super_block), GFP_USER);
        static const struct super_operations default_op;
        ...
        INIT_LIST_HEAD(&s->s_mounts);
        ...
        INIT_LIST_HEAD(&s->s_inodes);
}
```

Для удаления суперблока используются функции kill_block_super, kill_anon_super, kill_litter_super.
struct dentry – определяет элемент пути в имени файла (короткое имя + путь к файлу, то есть все директории с корневого каталога).

```
struct dentry {
        unsigned int d_flags;
```

```
        struct hlist_bl_node d_hash;
        struct dentry *d_parent; // Объект dentry родительского каталога
        struct qstr d_name; // Имя dentry
        struct inode *d_inode; // Связанный inode
        unsigned int d_count; //
        spinlock_t d_lock;
        unsigned long d_time;
        void *d_fsdata;
        unsigned char d_iname[DNAME_INLINE_LEN]; // короткое имя файла
        const struct dentry_operations *d_op;
        struct super_block *d_sb;
        struct list_head d_lru; // указатель на список lru
        struct list_head d_children;
        ...
        struct list_node d_hash; // объекты dentry кэшируются
        int d_mounted; // Является ли объект точкой монтирования
};
```

Состояния dentry: используется (in use), не используется (unused), отрицательный – не связанный с допустимым индексным узлом (d_inode == NULL). Если in_use, то данные dentry соответствует допустимому inode, у такого dentry один или несколько пользователей. Если d_count == 0, то состояние unused, то есть dentry соответствует допустимому inode, но d_count == 0. Такой dentry все еще хранится в кэше. Если dentry отрицательный, он все равно сохраняется так как какие-то запросы могут к нему обращаться, но при необходимости можно его и удалить. Для отрицательного dentry фактически нет информации на диске.

Кеш объектов dentry: сам каталог dentry является файлом специального типа. Чтобы каждый раз не обращаться к диску за информацией о dentry, информация заносится в кэш. При проходе по пути к файлу система сначала смотрит на кэш.

3 части кэша: список актуальных dentry, связанные с определенным индексным узлом (может быть несколько dentry с одним и тем же inode). Двусвязный список dentry_lru – этот список отсортирован по времени обращения к dentry. Каждому объекту dentry присваивается временная метка и список перестраивается при обращении к этому объекту. Если ядру нужно освободить память, элемент удаляется из хвоста списка. Хэш-таблица и хэш-функция для обращения к указанному файлу: таблица в виде массива dentry_hashtable в котором каждый элемент имеет указатель на dentry. Размер таблицы зависит от объема памяти, зачение хэша вычисляется по функции d_hash(). Поиск по таблице с помощью d_lookup(), которая возвращает или NULL (если хэш не найден) или указатель если найден.

dentry не соответсвует како-либо структуре данных на диске, он создается на лету ФС. Так как чтение и запись каталога с диска и создание соответсвующего объекта dentry требует достаточного времени имеет смысл хранить объекты dentry в памяти, даже если работа с ними закончена, чтобы обратиться к элементу пути повторно при обращении к тому же файлу.

/home/mdir/cf/my_server.c: VFS при каждом обращение должна пройти по всем dentry данного файла начиная с корневого каталога. Если бы не было кэша то каждый раз нужно было бы обращаться к диску

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, unsigned int);
        int (*d_hash)(const struct dentry *, struct qstr *);
        int (*d_compare)(const struct dentry *, unsigned int, const
        int (*d_init)(struct dentry *);
        void (*d_release)(struct dentry *);
        char *(*d_dname)(struct dentry *, char *, int);
};
```

d_hash создает значения хэша, d_compare – вызывается когда сравниваются два имени файла, d_del вызывается при d_count == 0, d_release – когда объект dentry теряет связь с inode.

Существует 2 структуры inode: дисковый – описывающий файл и представляющий информацию об адресах блоков в которых находится информация из файла. inode ядра – описывает файл в ядре для ускорения доступа к дисковым inode.

struct inode – описывает конкретный физический файл: в Linux нет понятия vnode в отличии от Unix. Но все еще 2 типа inode: ядра и диска. inode ядра содержит информацию о файле, что делает получение информации быстрее. inode диска содержит адреса блоков диска с данными файла.

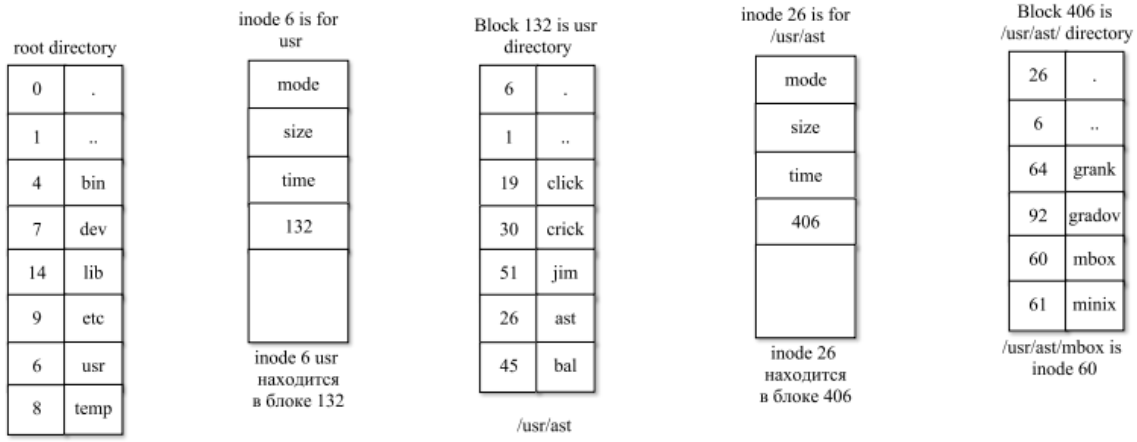
```
struct inode
{
        umode_t i_mode; //права доступа
        unsigned short i_opflags;
        kuid_t i_uid;
        kgid_t i_gid;
        unsigned int i_flags;
        ...
        const struct inode_operations *i_op; /* указатель на ту таблицу операций с инде
   ксом которые определены в конкретной системе для работы с этим индексом.
        Такая структура создаётся для конкретной файловой системы */
        struct super_block *i_sb; /*указатель на связаный суперблок, так как именно он
   хранит информацию об inode данной файловой системы. */
        struct address_space *i_mapping;
        ...
        unsigned long i_ino; // индекс inode
        union
        {
                const unsigned int i_nlink;
                unsigned int _i_nlink;
        };
        ...
        struct list_head i_lru; // список inode построенный по LRU.
        ...
        union{
                struct hlist_head i_dentry;
                struct rcu_head i_rcu;
        };
        ...
        atomic_t i_count; /* счётчик ссылок */
        ...
        atomic_t i_writecount; /*счётчик использования для записи */
        ...
        count struct file_operations *i_top;
        struct address_space i_data;
        struct list_head i_devices; /* список блочных устройств */
        ...
}
```

```
struct inode_operations {
        struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int); // вы
   полняет поиск inode в указанном каталоге
    ...
        int (*create) (struct inode *,struct dentry *, umode_t, bool); // создает дескр
   иптор файла
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,umode_t);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
        int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry
   *, unsigned int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);
    ...
}
```

Шаги при доступе к файлу */usr/est/mbox*

ФС ищет компонент пути usr в корневом каталоге, чтобы найти номер inode файла /usr. По номеру inode система может найти адрес блока. Получив адрес блока система получает содержимое

Имя	Номер inode
.	3470036
..	3470017
folder1	3470031
file1	3470043
file2	3470023
folder2	3470024
file3	3470065



директории usr, у нее есть inode. Далее в этой директории система ищет каталог ast, по номеру inode находит адрес следующего блока. Получив адрес блока система получает содержимое директории ast. Из этого inode система нашла файл mbox. struct file – структура описывающая открытый файл (с которым работает процесс).

Структура, описывающая файловую систему

```
struct file_system_type {
        const char* name; /* Название файловой системы */
        int fs_flags; /* Флаги типа файловой системы */
        #define FS_REQUIRES_DEV 1
        #define FS_USERNS_MOUNT 8
        struct dentry* (*mount)(struct file_system_type*, int, const char *, void *);
        void (*kill_sb)(struct super_block*); // используется для прекращения доступа к
    суперблоку
        struct module* owner; // Модуль, владеющий файловой системой
        struct file_system_type* next; // Следующая файловая система
        struct hlist_head fs_supers; // Список объектов типа суперблок
        ...
};
```

Функции монтирования:

```
typedef int (*fill_super_t)(struct super_block *, void *, int); // Передаваемая функции
    mount() функция fill_super() заполняет поля структуры struct super_block.
struct dentry *mount_bdev(struct file_system_type *fs_type, int flags, const char
   *dev_name, void *data, fill_super_t fill_super); // для монтирования ФС, находящейся
    на блочном устройстве,
struct dentry *mount_single(struct file_system_type *fs_type, int flags, void *data,
   fill_super_t fill_super); //  для монтирования ФС, точки монтирования которой раздел
   яют один единственный экземпляр ФС.
struct dentry *mount_nodev(struct file_system_type *fs_type, int flags, void *data,
   fill_super_t fill_super); //  для монтирования ФС, не связанной ни с каким устройств
   ом,
```

В этих функциях вызывается функция fill_super, которая выполняет основные действия по инициализации полей struct super_block (размер, смещение, магическое число и super_operations). Эти функции возвращают объект dentry для корневого каталога. Для ФС необходимо создать корневой каталог, что позволяет выполнить монтирование ФС. Чтобы создать корневой каталог, необходимо создать inode.

```
struct file_system_type my_fs_type =
{
        .owner = THIS_MODULE,
        .name = "my_fs",
        .mount = my_mount,
        .kill_sb = my_kill_sb,
        .fs_flags = FS_REQUIRES_DEV
};
static int __init my_init(void) {
        ...
        return register_filesystem(&my_fs_type);
}
static int __init my_init(void) {
        ...
        return unregister_filesystem(&my_fs_type);
}
```

При инициализации структуры file_system_type можно воспользоваться предописанными структурами ядра

```
void generic_shutdown_super(struct super_block *sb) // удалить суперблок
int simple_fill_super(struct super_block *s, unsigned long magic,
const struct tree_descr *files); // инициализировать поля суперблока
```

Для ФС необходимо определить, какие функции для работы с файлами необходимо реализовать. Для этого существует simple_dir_operations.

```
struct inode *new_inode(struct super_block *sb)
{
        struct inode *inode;
        spin_lock_prefetch(&inode_sb_list_lock);
        inode = new_inode_pseudo(sb);
        if (inode)
                inode_sb_list_add(inode);
        return inode;
}
EXPORT_SYMBOL(new_inode);

struct inode *new_inode_pseudo(struct super_block *sb)
{
        struct inode *inode = alloc_inode(sb);

        if (inode) {
                spin_lock(&inode->i_lock);
                inode->i_state = 0;
                spin_unlock(&inode->i_lock);
                INIT_LIST_HEAD(&inode->i_sb_list);
        }
        return inode;
}
```

6.1 Загружаемые модули ядра

UNIX имеет монолитное ядро – единая программа из структур и функций. Загружаемые модули ядра позволяет вносить в ядро собственную функциональность.

У загружаемого модуля ядра обязательно две точки входа, устанавливаются макросами MODULE_INIT и MODULE_EXIT.

Точки входа – места с которых начинает выполняться код.

```
1 static int __init my_init(void);
2 MODULE_INIT(my_init);
3
4 static void __exit my_exit(void);
5 MODULE_EXIT(my_exit);
```

init вызывается insmod. exit вызывается rmmod.

В ядре нельзя обращаться к функциям стандартных библиотек. В ядре могут использоваться только функции, определенные в ядре. Нельзя читать и писать сразу в ядро: нарушается безопасность ядра. Для передачи информации из ядра и в ядро используются специальные библиотеки и функции.

Макрос MODULE_LICENSE является обязательным. Макрос MODULE_LICENSE используется для того, чтобы сообщить ядру, под какой лицензией распространяется исходный код модуля, что влияет на то, к каким функциям и переменным (символы ядра) он может получить доступ в ядре.

Для протоколирования используется функция printk(). В функции указывают уровень протоколирования: KERN_INFO, KERN_ERR, KERN_DEBUG (всего 8 уровней). Чем ниже значение макроса, тем выше приоритет. printk() пишет в системный журнал, для обращения к нему используют команду dmesg.

```
1 printk(KERN_INFO "module loaded");
```

Взаимодействие модулей заключается в том, что один модуль может использовать данные, объявленные и описанные в другом модуле.

В одном модуле переменная объявлена как public (доступная другим модуле), в другом, чтобы прошла компиляцию, объявлена как extern. Так один модуль может получать доступ к данным другого модуля. Макрос EXPORT_SYMBOL – позволяет экспортировать данные модуля и теперь им могут пользоваться другие модули ядра. Модуль может использовать только те имена, которые явно экспортированы.

Модуль связывается с экспортируемым именем по прямому абсолютному адресу. Абсолютный адрес — физический адрес.

Загружаемый модуль становится часть ОС, он загружается в физическую память. Коды ядра работают с физическими адресами. Модуль может обращаться к данным другого модуля только по физическим адресам – имеет значение последовательность загрузки. У модулей есть счетчик ссылок на него, выгрузить его можно когда счетчик равен нулю.

7 proc

proc – виртуальная файловая система. Файлы и каталоги такой VFS не хранятся во вторичной памяти и создаются на лету.

proc создавался разработчиками Linux, чтобы предоставить разработчикам ПО в режиме пользователя информацию из режима ядра о разработанном ПО для анализа с точки зрения потребляемых ресурсов во время выполнения.

В proc у каждого процесса есть директория его идетификатора, в ней находятся файлы, символьные ссылки, поддиректории с информацией о каждом процессе */proc/<pid>*. Можно использовать /proc/self для получении информации текущего процесса

Элемент	**Тип**	**Описание**
cmdline	файл	Содержит командную строку процесса, которая была набрана
cwd	символическая ссылка	Указывает на директорию процесса
environ	файл	Содержит список переменных окружения процесса
exe	символическая ссылка	Указывает на исполняемый файл процесса
fd	директория	Содержит ссылки на открытые процессом файлы
root	символическая ссылка	Указывает на корень файловой системы процесса
stat	файл	Содержит информацию о процессе
maps	файл	Содержит текущие регионы ВАП процесса
pagemap	файл	Содержит по одному 64-битному значению на каждую виртуальную страницу памяти процесса
mem	файл	информация о страницах памяти процесса
interrupts	файл	информация о об источниках прерываний и о частоте возникновения этих прерываний.
task/<tid>	директория	информация о потоке процесса

```
* MAPS:
       address                  perms  offset   dev   inode
1      5f17aaa8a000-5f17aaa8b000 r--p 00000000 08:03 3932858
   /home/Desktop/task-03/s.out
1      5f17aaa8b000-5f17aaa8c000 r-xp 00001000 08:03 3932858
   /home/Desktop/task-03/s.out
1      5f17aaa8c000-5f17aaa8d000 r--p 00002000 08:03 3932858
   /home/Desktop/task-03/s.out
1      5f17aaa8d000-5f17aaa8e000 r--p 00002000 08:03 3932858
   /home/Desktop/task-03/s.out
1      5f17aaa8e000-5f17aaa8f000 rw-p 00003000 08:03 3932858
   /home/Desktop/task-03/s.out
33     5f17aae6d000-5f17aae8e000 rw-p 00000000 00:00 0         [heap]
33     7e67a0000000-7e67a0021000 rw-p 0        00:00 0
33     7e67ac000000-7e67ac021000 rw-p 0        00:00 0
33     7fffee537000-7fffee558000 rw-p 00000000 00:00 0         [stack]
4      7fffee5b0000-7fffee5b4000 r--p 00000000 00:00 0         [vvar]
2      7fffee5b4000-7fffee5b6000 r-xp 00000000 00:00 0         [vdso]
1      ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

```
* PAGEMAP
addr : pfn soft-dirty file/shared swapped present
0x5f17aaa8a000     : 59927            1          1          0          1
0x5f17aaa8b000     : 2d274            1          1          0          1
0x5f17aaa8c000     : 51c00            1          1          0          1
0x5f17aaa8d000     : 0            1          0          0          1
0x5f17aaa8e000     : 0            1          0          0          1
```

pfn (Page Frame Number) - Номер физической страницы. soft-dirty – страница была модифицирована. file/shared – является файловой или разделяемой, 0 если анонимная страница. swapped – выгружена в swap. present – страница доступна в физической памяти.

```
stat
pid  17387
comm  myapp
state  S,
ppid  7980,
flags  4194304
utime  78074,
stime  8573
priority  20,
nice  0,
num_threads  3
vsize  2118049792
rss 100
startcode  104829120306400
endcode  104829128962048
startstack  140735838567920
exit_signal  17
processor  4
rt_priority  0
policy  0
start_data  104829129340584
end_data    104829129350435
```

```
struct proc_dir_entry {
        atomic_t in_use;
        const struct inode_operations *proc_iops;
        union {
                const struct proc_ops *proc_ops;
                const struct file_operations *proc_dir_ops;
        };
        const struct dentry_operations *proc_dops;
        union {
                const struct seq_operations *seq_ops;
                int (*single_show)(struct seq_file *, void *);
        };
        proc_write_t write;
        void *data;
        unsigned int low_ino;
        nlink_t nlink;
        kuid_t uid;
        kgid_t gid;
        loff_t size;
        struct proc_dir_entry *parent;
        char *name;
        u8 flags;
};
```

Раньше в proc_dir_entry был только file_operations. struct proc_ops — определяет операции над файлами proc, нововведение. Функции read/write в struct file_operations используются интенсивно, а использование их для ВФС proc, которая носит справочный характер, вносит дополнительную нагрузку. Поэтому разработчики Linux ввели proc_ops, чтобы разгрузить функции.

```
struct proc_ops {
        unsigned int proc_flags;
        int (*proc_open)(struct inode *, struct file *);
        ssize_t (*proc_read)(struct file *, char __user *, size_
        t, loff_t *);
        ssize_t (*proc_write)(struct file *, const char __user *,
        size_t, loff_t *);
        int (*proc_release)(struct inode *, struct file *);
};
```

Для создания файла proc используется системный вызов proc_create. proc_create — обертка proc_create_data.

```
proc_dir_entry* proc_create_data(const char* name, umode_t mode,struct proc_dir_entry*
    parent, const struct file_operations* proc_fops, void* data);

static inline struct proc_dir_entry* proc_create(const char* name, umode_t mode,
    struct proc_dir_entry* parent, const struct file_operations* proc_fops) {
        return proc_create_data(name, mode, parent, proc_fops, NULL);
}
```

Создавать каталоги в файловой системе /proc можно используя proc_mkdir(), а также символические ссылки с proc_symlink().

```
extern struct proc_dir_entry *proc_symlink(const char *, struct proc_dir_entry *,
    const char *);
extern struct proc_dir_entry *proc_mkdir(const char *, struct proc_dir_entry *);
```

Для передачи данных из пространства ядра имеется 2 функции

```
unsigned long __copy_to_user(void __user* to, const void *
from, unsigned long n);

unsigned long __copy_from_user(void *to, const void __user
*from, unsigned long n);
```

Ядро находится в физической памяти, а процесс памяти не имеет, имеет виртуальное адресное пространство. Физические страницы выделяются, когда процесс обращается к данным соответствующей виртуальной страницы. Если при обращении функции ядра к буферу пользователя нужная станица выгружена, то система должна загрузить соответствующие страницы страничным прерыванием. Это прерывание обрабатывается ядром. Происходит переключение между кодами ядра для обработки страничного прерывания и возникнет паника ядра.

Для предотвращения этой ситуации введены эти специальные функции.

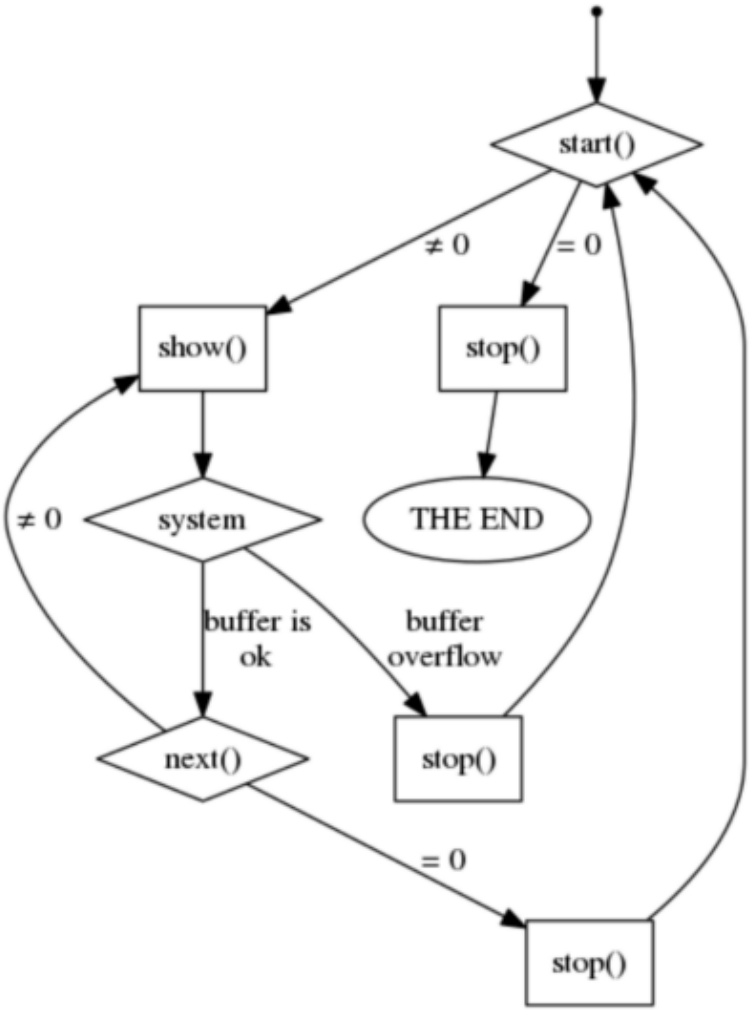
```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5,6,0)
#define HAVE_PROC_OPS
#endif

#ifdef HAVE_PROC_OPS
static struct proc_ops fileops = {
        .proc_read = fortune_read,
        .proc_write = fortune_write,
        .proc_open = fortune_open,
        .proc_release = fortune_release,
};
#else
static struct file_operations fileops = {
        .owner = THIS_MODULE,
        .read = fortune_read,
        .write = fortune_write,
        .open = fortune_open,
        .release = fortune_release,
};
#endif

```

Sequence файлы: второй интерфейс для работы с proc файлами. Для сиквенсов определена структура struct seq_file и структура struct seq_operations. В union proc_dir_entry два поля: seq_ops и single_open. Для seq файлов определены два способа работы: полный и упрощенный.

```
struct seq_file {
        char *buf;
        size_t size;
        size_t from;
        size_t count;
        size_t pad_until;
        loff_t index;
```

```
        loff_t read_pos;
        struct mutex lock;
        const struct seq_operations *op;
        int poll_event;
        const struct file *file;
        void *private;
};
```

```
struct seq_operations {
        void * (*start) (struct seq_file *m, loff_t *pos);
        void (*stop) (struct seq_file *m, void *v);
        void * (*next) (struct seq_file *m, void *v, loff_t *pos);
        int (*show) (struct seq_file *m, void *v);
};
```



Функция stop вызывается в трех случаях:

- Вызов stop сразу после start означает окончание передачи данных
- stop после show означает, что после последней передачи данных буфер, в который передаются данные, заполнен и передача данных завершается.
- вызов после next выполняется, если нужно закончить передачу в буфер данных и завершить работу.

В функции show вызывается seq_printf. Sequence файлы обеспечивают более надежную передачу информации из пространства ядра в пространство пользователя, за счет того что ядро контролирует указатель на буфер с данными.

```
int seq_open(struct file * file, const struct seq_operations *op);
ssize_t seq_read(struct file* file, char __user* buf, size_t size, loff_t* ppos);
int seq_release ( struct inode * inode, struct file * file);
```

```
void seq_printf(struct seq_file *m, const char *fmt, ...);
int seq_write(struct seq_file *seq, const void *data, size_t len);
loff_t seq_lseek(struct file *, loff_t, int);
int seq_release(struct inode *, struct file *);
```

Упрощенный интерфейс single файлов позволяет не писать итератор, так как внутри функции single_open инициализируются поля структуры seq_operations.

```
int single_open(struct file *file, int (*show)(struct seq_file*, void*), void *data){
        struct seq_operations *op = kmalloc(sizeof(*op), GFP_KERNEL_ACCOUNT);
        int res = -ENOMEM;
        if(op) {
                op->start = single_start;
                op->next = single_next;
                op->stop = single_stop;
                op->show = show;
                res = seq_open(file, op);
                if(!res) {
                        ((struct seq_file*)file->private_data)->private =
                        data;
                }
                else {
                        kfree(op);
                }
        }
        return res;
}
EXPORT_SYMBOL(sigle_open);
```

Для single_open размер передаваемых данных ограничен 64 КБ (16 страниц).

8 Открытые файлы

Любой процесс может открыть много файлов, причем один файл может быть открыт много раз одним или несколькими процессами.

В дескрипторе процесс есть указатель на открытые процессом файлы и указатель на ФС, которая относится к образу процесса.

```
struct task_struct
{
        ...
        struct fs_struct *fs;
        struct files_struct *files;
        ...
}
```

struct file – дескриптор открытого файла.

Системный вызов *int open(const char *pathname, int flags, mode_t mode);* или *int open(const char *pathname, int flags);* Флаги:

- O_CREATE – создать новый файл
- O_EXCL – (обязательно с O_CREATE) – заставить систему проверить существование файла, чтобы не потерять информацию ранее записанную в файл
- O_EXEC
- O_RDONLY
- O_RDWR
- O_APPEND

Информация о файловой системе, к которой относится процесс

```
struct fs_struct
{
        atomic_t count;
        ...
        struct dentry *root; /* Корневой каталог */
        struct dentry pwd; /* Текущий рабочий каталог */
        ...
        struct vfsmount *rootmnt;
        ...
}
```

В этой структуре хранится вся информация об открытых процессом файлах и файловых дескрипторах.

```
struct files_struct {
        atomic_t count; // счетчик ссылок
        ...
        unsigned int next_fd; // число открытых процессом файлов
        unsigned long close_on_exec_init[1];
        unsigned long open_fds_init[1];
        unsigned long full_fds_bits_init[1];
        struct file __rcu * fd_array[NR_OPEN_DEFAULT]; // массив файловых дескрипторов
   файлов, открытых процессом
};
```

Каждый процесс имеет собственную таблицу страниц с указателем на системные дескрипторы открытых файлов. Когда создается процесс для него автоматически открываются три файла: stdin, stdout, stderr – 0, 1, 2. Если процесс в результате exec не унаследовал открытые файлы, то первый номер открытого файла будет 3. close_on_exec_init – какие ФД будут закрыты при exec.

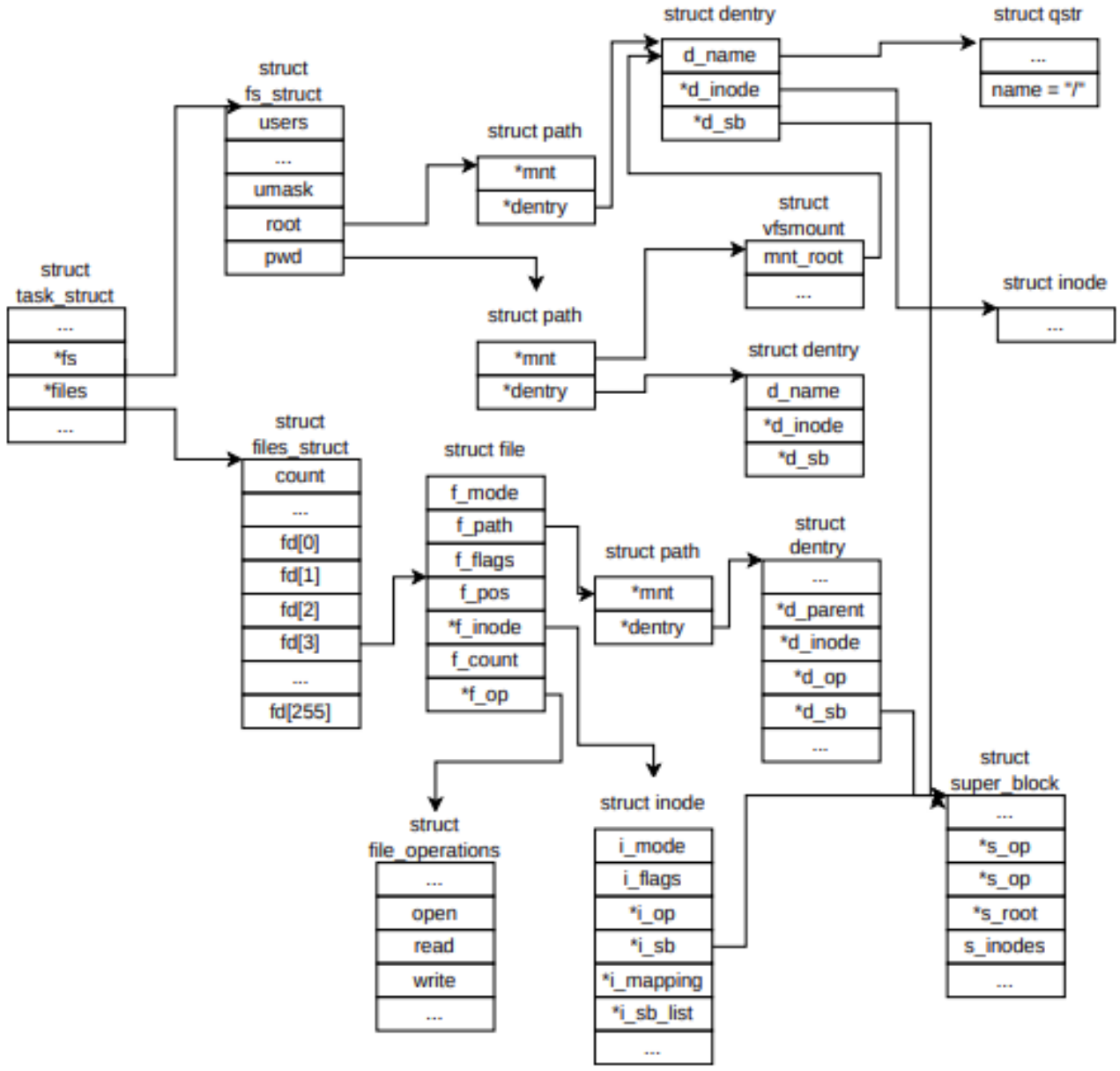
```
struct file {
        f_mode_t mode;
        struct path f_path;
        struct inode *f_inode;
        const struct file_operations *f_op;
        atomic_long_t f_count;
        ...
        loff_t f_pos; // отражает как работать с логической структурой файла
```

```
        ...
}
struct path {
        struct vfsmount *mnt;
        struct dentry *dentry;
};
```

struct file работает с логической структурой файла. Любой файл начинается с 0 адреса и любой файл считает, что его смещение начинается с 0.

Каждая операция чтения или записи сдвигает указатель файла на соответствующее количество байт.

Связь структур ядра: stdio – библиотека буферизированного ввода-вывода.



fopen() – открывает файл и возвращает связанный с ним указатель. Функция возвращает указатель на структуру FILE, которая называется файловым потоком.

fdopen() принимает дескриптор на открытый файл и возвращает файловый поток.

```
struct _IO_FILE {
        int _flags;             /* High-order word is _IO_MAGIC; rest is flags. */
```

```
        /* The following pointers correspond to the C++ streambuf protocol. */
        char *_IO_read_ptr;     /* Current read pointer */
        char *_IO_read_end;     /* End of get area. */
        char *_IO_read_base;    /* Start of putback+get area. */
        char *_IO_write_base;   /* Start of put area. */
        char *_IO_write_ptr;    /* Current put pointer. */
        char *_IO_write_end;    /* End of put area. */
        char *_IO_buf_base;     /* Start of reserve area. */
        char *_IO_buf_end;      /* End of reserve area. */

        /* The following fields are used to support backing up and undo. */
        char *_IO_save_base; /* Pointer to start of non-current get area. */
        char *_IO_backup_base;  /* Pointer to first valid character of backup area */
        char *_IO_save_end; /* Pointer to end of non-current get area. */

        struct _IO_marker *_markers;

        struct _IO_FILE *_chain;

        int _fileno;
        int _flags2;
        __off_t _old_offset; /* This used to be _offset but it's too small.  */

        /* 1+column number of pbase(); 0 is unknown. */
        unsigned short _cur_column;
        signed char _vtable_offset;
        char _shortbuf[1];

        _IO_lock_t *_lock;
        #ifdef _IO_USE_OLD_IO_FILE
};
```

Сначала информация пишется в буфер и оттуда выбирается для обработки. 3 причины перезаписи в файл: буфер заполнен, fflush и fclose().

Пример: файл открывается два раза системным вызовом open() для записи и в него последовательно записывается строка «aaaaaaaaaaaa» по первому дескриптору и затем строка «bbbb» по второму дескриптору, затем файл закрывается два раза. Показать, что будет записано в файл и пояснить результат.

```
int main(void)
{
        int fd1 = open("file.txt", O_CREAT | O_EXCL | O_WRONLY);
        int fd2 = open("file.txt", O_WRONLY);

        char buf1[] = "aaaaaaaaaaaa";
        char buf2[] = "bbbb";
        write(fd1, buf1, sizeof(buf1) - 1);
        write(fd2, buf2, sizeof(buf2) - 1);

        close(fd1);
        close(fd2);

        return 0;
}
```

В ядре есть две таблицы: PFT — process file table и SFT — system file table, в которой находится проинициализированная структура struct file. Процесс может создать только 512 открытых файлов. Базовое значение в ядре: 256. Если надо еще – создается дополнительная таблица еще на 256.

Файл дважды открывается на запись с помощью open(), при этом создаются два экземпляра структуры struct file, которые описывают один физический файл. У каждой из структур инициализируется поле f_pos = 0. Начиная с нулевой позиции первого открытого файла в файл будет записано aaaaaaaaaa. После чего начиная с нулевой позиции будет записано bbbb. Результатом будет bbbbaaaaaaaa.

9 Слабы

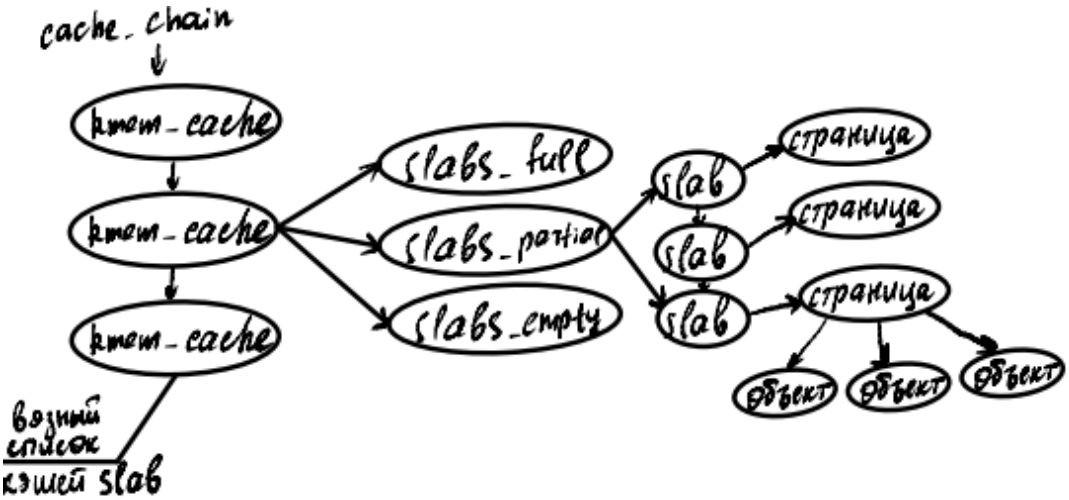
Когда создается объект, инициализируются поля структуры. Если объект удаляется и он понадобится снова его поля нужно будет заново инициализировать. В ядре используются не так много структур, которые описывают базовые объекты. Одни и те же структуры используются многократно для создания одних и тех же объектов. Инициализация полей структур занимает время. Когда нужна в объекте исчезает, объект не удаляется, а остается в слабе для повторного использования.

Побочный эффект слабов – устранение скрытой фрагментации (страница никогда не заполняется полностью). Поскольку память выделяется страницами, скрытая фрагментация присутствует. Но кэш слабов позволяет записывать в страницу несколько слабов одного и того же размера.

```
1 // Создание нового кеша
2 struct kmem_cache* kmem_cache_create(const char* name, unsigned int size, unsigned int
     align, slab_flags_t flags, void (*ctor)(void*));
3 // Для удаления кеша
4 int kmem_cache_destroy(kmem_cache_t *cache);
5 // получение адресного пространства для объекта
6 void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
7 void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

Слабы работают только со структурами ядра. Каждый slab может находиться в одном из трех состояний:

- slab full — полностью распределенный. В таком слабе нет свободных объектов, поскольку все они распределены и используются.
- slab partial — частично распределенный. В таком слабе есть как выделенные, так и свободные объекты.
- slab empty — пустой, незаполненный. В таком слабе нет ни одного распределенного объекта, все его объекты свободны для последующего использования.



Как только из некоторой части ядра поступает запрос на новый объект, он выделяется из частично распределенного блока, если таковой имеется. В противном случае объект выделяется из пустого блока. Если больше не осталось пустых слабов, они будут созданы по мере необходимости.

Inode кеш имеет следующую структуру:

- Глобальная хеш таблица inode_hashtable, в которой каждый inode адресуется по указателю на суперблок и 32-битному номеру inode.
- Глобальный список inode_in_use — inode в состоянии in_use. В этом списке содержится inode с полями i_count > 0 и i_nlink > 0. Только что созданный inode добавляется как раз в этот список.
- Глобальный список inode_unused, который содержит допустимые inode с i_count $= 0$.
- Когда inode помечается как "грязный он добавляется к списку sb->s_dirty при условии, что он в хэш-таблице.

Через поле inode->i_list с inode вставляется в список определенного типа, через поле inode->i_hash . Каждый inode может входить в список i_hash и в один и только в один список типа (in_use, unused или dirty).

10 Аппаратные прерывания

Аппаратная поддержка аппаратных прерываний развивается. В персональных компьютерах реализована шинная архитектура. Управление внешний устройств было передано контроллерам (входят в состав внешнего устройства) и адаптерам (на материнской плате).

В 16 разрядных машинах был реализован PIC. В 32х разрядных машинах APIC (advanced PIC), может работать и с 16 разрядными машинами.

В современных системах применяется реализация прерываний от внешних устройств в виде сообщений. MSI — message signaled interrupts — прерывание, сигнализируемое сообщением. Устройства сигнализируют о прерывании, отправляя/записывая сообщение по определенному адресу в памяти.

Рассмотрим не аппаратную составляющую, а ПО позволяющее обрабатывать эти аппаратные прерывания.

Управление внешними устройствами осуществляется драйверами, для каждого устройство пишется драйвер. В составе драйвера входит обработчик прерывания.

В ядре предоставляется возможность зарегистрировать собственный обработчик прерываний.

```
typedef irq_return_t(*irq_handler_t)(int, void *);
extern int __must_check request_threaded_irq(unsigned int irq, irq_handler_t handler,
   irq_handler_t thread_fn, unsigned long flags, const char *name, void *dev);

static inline int __must_check request_irq(unsigned int irq, irq_handler_t handler,
   unsigned long flags, const char *name, void *dev) {
        return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
void free_irq(unsigned int irq, void *dev_id); // освободить линию
```

При прерывании по линии IRQ вызывается обработчик прерывания. Когда возникает прерывание, система через соответствующее аппаратное обеспечение переходит на соответствующий обработчик прерывания.

Линии прерываний уже заняты основными обработчиками прерываний. Используются флаги

- IRQF_SHARED – разделять линии прерывания несколькими обработчиками прерываний;
- IRQF_PROBE_SHARED
- IRQF_NO_THREAD – не может выполняться на разных ядрах (не может быть распараллелен)
- __IRQF_TIMER – от системного таймера
- IRQF_TIMER (__IRQF_TIMER | IRQF_NO_SUSPEND | IRQF_NO_THREAD)

```
struct irqaction {
        irq_handler_t           handler;
        void                    *dev_id;
        void __percpu           *percpu_dev_id;
        struct irqaction        *next;
        irq_handler_t           thread_fn;
        struct task_struct      *thread; // система выполняет каждый драйвер как поток
        struct irqaction        *secondary;
        unsigned int            irq;
        unsigned int            flags;
        unsigned long           thread_flags;
        unsigned long           thread_mask;
        const char              *name;
        struct proc_dir_entry   *dir; // /proc/irq/<N>/nameentry
} ____cacheline_internodealigned_in_smp;
```

Быстрые и медленные прерывания: в Linux одно быстрое прерывание – от системного таймера.

Проблема: в любом обработчик прерывания выполняется на высочайшем уровне приоритета. В результате никакая другая работа в система не может выполнятся, пока не завершится обработчик. Поэтому все действия в обработчиках – отложенные – через постановку в очередь на выполнение.

Обработчик прерывания выполняется на одном из ядер, при этом на этом ядре запрещаются другие прерывания. А на других ядрах запрещены прерывания по этой линии IRQ.

Медленные делятся на 2 части – top half и bottom half. Верхняя половина нужна, чтобы завершать быстрее в обработчике прерывания минимальный необходимый набор действий. Задача – сохранить данные в буфере ядра

Нижняя половина делится на 3 части: softirq – гибкие прерывания, тасклеты и очереди работ. softirq – отложенные действия, которые определяются статически во время компиляции ядра.

```
struct softirq_action {
        void(*action)(struct softirq_action*); /* ф-я которая должна выполняться */
};
```

Кроме этой структуры в ядре определен массив из 10 экземпляров данной структуры:

```
static struct softirq_action softirq_vec[NR_SOFTIRQS]
```

Индекс	**Приоритет**	**Значение**
HI_SOFTIRQ	0	высоко приоритетные softirq
TIMER_SOFTIRQ	1	программируемые таймеры, которые нужны в ПО для контроллеров
NET_TX_SOFTIRQ	2	отправка сетевых пакетов
NET_RX_SOFTIRQ	3	прием сетевых пакетов
BLOCK_SOFTIRQ	4	блочные устройства
BLOCK_IOPOLL_SOFTIRQ	5	
TASKLET_SOFTIRQ	6	таслкеты
SCHED_SOFTIRQ	7	планировщик
HR_TIMER_SOFTIRQ	8	не используются
RCU_SOFTIRQ	9	сохраняется для сохранение нумерации // обязательно последним должен быть

```
const char *const softirq_to_name[NR_SOFTIRQS] = {
        "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "IRQ_POLL",
        "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

Меньшие номера имеют более высокий приоритет.

```
void open_softirq(int nr, void (*action)(struct softirq_action*))
{
        softirq_vec[nr].action = action;
}
irq_return_t my_interrupt(int irq, void *dev_id) {
        raise_softirq(MY_SOFT_IRQ);
        ...
        return IRQ_HANDLER;
}
void my_soft_handler(struct softirq_action *) {...}
static int __init my_init(void)
{
        /*....*/
        request_irq(irq, my_interrupt, 0, "my", NULL); //регистрируем свой обработчик
        open_softirq(MY_SOFT_IRQ, my_soft_handler);
}
```

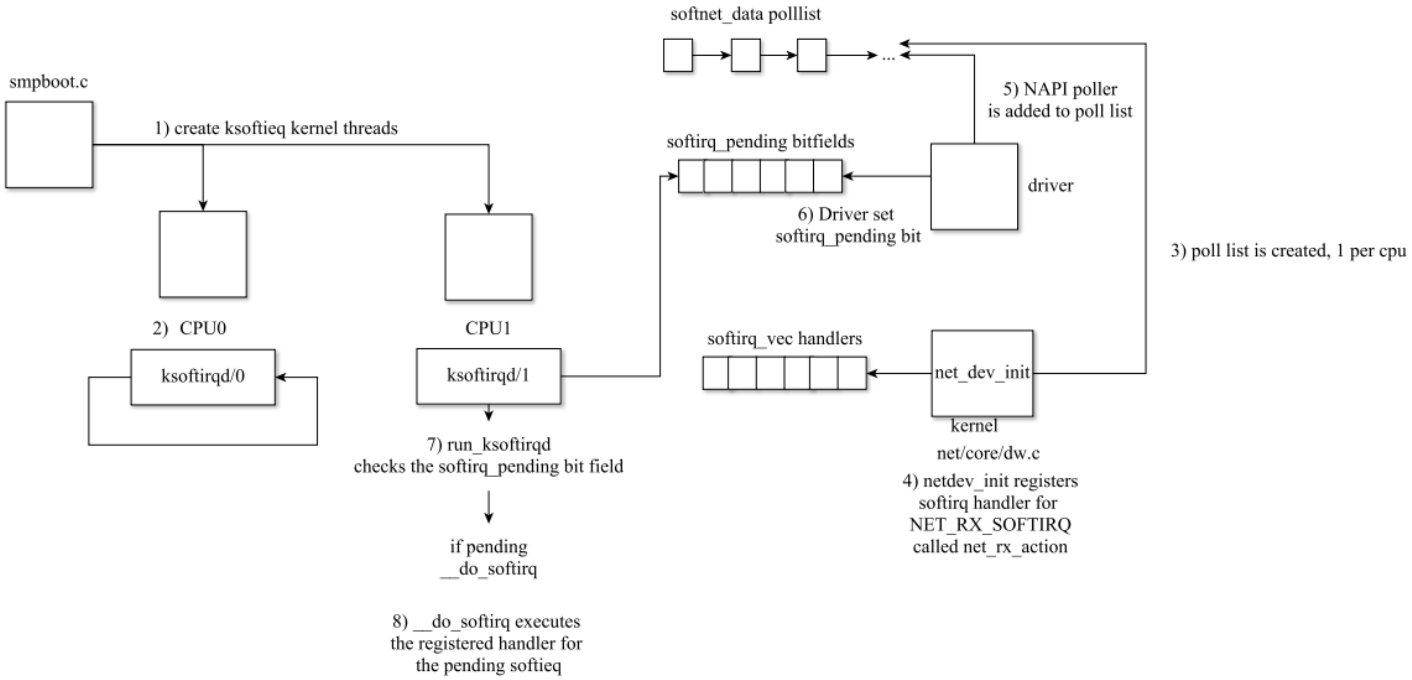
На каждое ядро при загрузке системы создается и инициализируется ksoftirq deamon Задача ksoftirq daemon – обработка отложенных действий, работает с softirq.

Первое ядро находится в цикле выполнения. run_ksoftirq проверяет очередь событий для их обработки.

Если каждый входящий пакет приходит на сетевую карту, то каждый пакет нужно обработать – прерывание, что не эффективно.

NAPI – замена прерываний опросом. Основная цель NAPI — сократить количество прерываний, генерируемых при получении пакетов. Создаются poll list, по одному на ядро.

Демон ksoftirq работает с битовой очередью, обозначающая события, требующие обработки. У сетевой карты есть драйвер. Когда пакет приходит, драйвер устанавливает соответствующий бит очереди, демон его анализирует и понимает, что надо обработать пакет.



Если бит выставлен, то пакет пришел и его надо обработать и вызывается функция do_softirq. Функция do_softirq выполняет зарегистрированный обработчик.

Сетевая подсистема Linux работает по примеру стека BSD. Прием и передача данных с помощью интерфейса сокетов. NAPI – приходит пакет и вызывается rx_scheduler, что гарантирует обработку пакетов в дальнейшем. Тасклеты: частный случай softirq. Но softirq можно сериализовывать (один и тот же тип softirq может выполняться параллельно), так что нужны средства взаимоисключающий.

Тасклеты же не сериализуются, один тип тасклета может выполняться только на одном процессоре, так что средства взаимоисключения не требуются. Разные тасклеты могут выполняться параллельно.

В силу этого тасклеты не используются для обработки нагружающих действий. Тасклет – компромисс между производительностью и простотой использования. Тасклеты могут быть зарегистрированы и статически и динамически.

```
struct tasklet_struct {
        struct tasklet_struct *next; /* Указатель на следующий тасклет в списке */
        unsigned long state;
        atomic_t count;
        bool use_callback;
        void (*func)(unsigned long data); /* Функция-обработчик тасклета */
        unsigned long data; /* Аргумент функции-обработчика тасклета */
};
```

Состояниями тасклета могут быть 0, TASKLET_STATE_SCHED, TASKLET_STATE_RUN. TASKLET_STATE_RUN имеет смысл только в smp-архитектуре, так как если тасклет выполняется, то такой тасклет на другом процессоре выполняться не может.

Тасклет можно выполняться только если счетчик ссылок count == 0.

Для статического создания тасклета используется макрос. Он создает экземплеяр структуры тасклета с именем и функцией.

```
DECLARE_TASKLET(name, func);
DECLARE_TASKLE_DISABLEDT(name, func); // count = 1 (тасклет не может выполняться)
```

Динамически тасклет создается с помощью функции ядра

```
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long
    data)
```

```
{
        t->next = NULL;
        t->state = 0;
        atomic_set(&t->count, 0);
        t->func = func;
        t->data = data;
}
```

Для того чтобы запланировать тасклет используются функции. В зависимости от функции тасклет помещается в разные списки: или tasklet_vec или tasklet_hi_vec.

```
extern void __tasklet_schedule(struct tasklet_struct *);
extern void __tasklet_hi_schedule(struct tasklet_struct *);
```

Особенность тасклетов – не могут блокироваться, можно использовать только спинлоки (tasklet_trylock(), tasklet_unlock())

Тасклеты могут использоваться только 1 раз. Они строго сериализуются по отношению к самому себе, но не по отношению к другим. Очереди работ: альтернатива тасклетам. Задачи идентичные, обе предоставляют возможность реализовать отложенные действия. Тасклет выполняется в контексте прерывания, код тасклета неделим. Работа выполняется в контексте специального потока ядра и может быть заблокирована.

Тасклет всегда выполняется на процессоре, на котором завершилась обработка прерывания, поставившая тасклет в очередь на выполнение. Работа по умолчанию тоже на том же ядре, но можно и поменять процессор. Также работу можно отложить на некоторое время.

```
struct workqueue_struct *alloc_workqueue(const char fmt, unsigned int flags, int
   max_active);
```

```
struct workqueue_struct {
        struct list_head pwqs;
        struct list_head list;
        struct worker *rescuer;
        char name[WQ_NAME_LEN];
        struct pool_workqueue __percpu *cpu_pwqs; // pool workqueues на процессор.
        struct pool_workqueue __rcu *numa_pwq_tbl[];// для новыхсистем - numa
};
```

Структура представляет некоторую работу, которую решает некий обработчик нижней половины, экземпляр структуры помещается в очередь.

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
};
```

Инициализация полей структуры статически:

```
DECLARE_WORK(name, void (*func)(void*));
```

Динамически:

```
INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*func)(void *), void *data); // повторно
```

Для того чтобы отложить работу на выполнение используются функции:

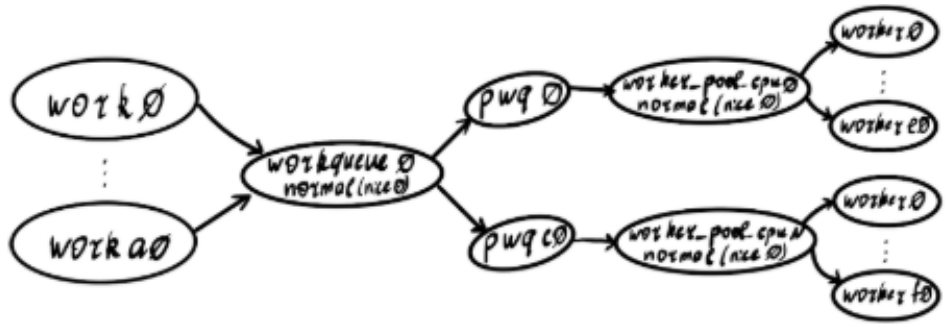
```
int queue_work(struct workqueue_struct *wq, struct work_struct *work);
// на конкретный cpu
int queue_work_on(int cpu, struct workqueue_struct *wq, struct work_struct *work);
// с задержкой
int queue_delayed_work(struct workqueue_struct *wq, struct work_struct *work, unsigned
   long delay);
```

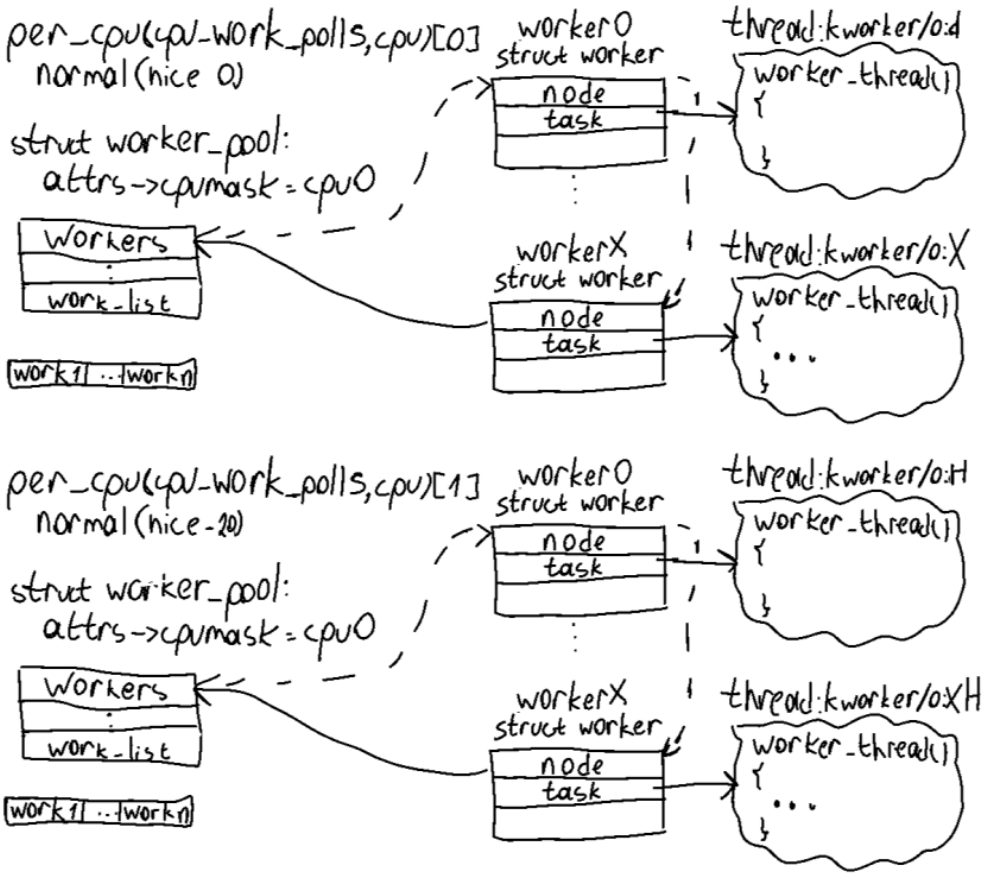
```
1 void destroy_workqueue(struct workqueue_struct *wq);
```

Основная идея с которой очереди работ были реализованы: одна параллельная множественная работа, в которой работы не должны блокировать друг друга. Механизм строится с расчетом на экономию потоков, выделенной памяти и планирования.

CMWQ – councurrency managed workqueue.

- Работа – work
- Очереди работ – workqueue. Очередь и работа представлены отношением 1:многие.
- Worker – рабочий. worker соответствует потоку ядра work_thread.
- PWQ (pool workqueue) – определяется как посредник, отвечающий за установку отношения между work_pool и work_queue; work_queue и pwq представлены отношением 1:многие. work_pool и pwq представлены отношением 1:многие.





Сначала выполняются потоки worker повышенного приоритета. Тасклет планируется на том же процессе, что и прерывание и как следствие обычно выполняется сразу, поэтому он быстрее очереди работ.

Флаги:

- WQ_UNBOUND. Очереди работ были разработаны в расчете на выполнение на одном и том же процессоре, что улучает работу с кэшем.
- WQ_FREEABLE. Может быть заморожен.

11 Специальные файлы устройств

Символьные и блочные устройства. Байт-ориентированные (или символьные) устройства передают данные посимвольно, как непрерывный поток байтов. Блок-ориентированные (или блочные) устройства. Все запоминающие устройства относятся к блочным. Сделаны для обеспечения работы с внешними устройствами как с обычными файлами. При r/w информации с внешнего устройства используются системные вызовы read, write, универсальный механизм.

Файлы устройств могут быть открыты, закрыты, можно читать и писать.

ОС каждому устройству в соответствие ставит 1 специальный файл, обычно в /dev.

Принята система major/minor номеров. Старший номер – идентификатор драйвера устройства. Младший номер – для различия устройства.

В ядре определен тип dev_t – 32 разрядное число. POSIX определяет существование типа, но не формат полей. Код не должен знать какие-либо предположения об организации номеров внутри числа. Нужно использовать макросы.

```
MAJOR(dev_t dev) -- получить старший номер
MINOR(dev_t dev) -- получить младший номер

MKDEV(int major, int minor) -- преобразовать в dev_t
```

Linux поддерживает огромное количество внешних устройств. Старший номер выбирается динамически, функция возвращает его одновременно с первым младшим номером:

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char
   *name) {
        struct char_device_struct *cd;
        cd = register_chrdev_region(0, baseminor, count, name);
        if (IS_ERR(cd)) return PTR_ERR(cd);
        *dev = MKDEV(cd->major, cd->baseminor);
        return 0;
}
```

Реализация драйвера символьного устройства

```
#define IRQ_NO 11
static DECLARE_WORK(work, workqueue_fn);
static struct file_operations fops = {
        .owner = THIS_MODULE,
        .read = ext_read,
        .write = ext_write,
        .open = ext_open,
        .release = ext_release,
};
static int __init ext_device_init(void) {
        int ret;
        if (alloc_chrdev_region(&dev, 0, 1, "ext_dev") < 0) {
                printk(KERN_ERR "Can't allocate device numbers\n");
                return -1;
        }
        printk(KERN_INFO "major=%d, minor=%d\n", MAJOR(dev), MINOR(dev));
        /* Creating cdev structure */
        cdev_init(&ext_dev, &fops);
        if (cdev_add(&ext_dev, dev, 1) < 0) {
                printk(KERN_ERR "Can't add cdev\n");
                unregister_chrdev_region(dev, 1);
                cdev_del(&ext_cdev);
                return -1;
        }
        /* Creating struct class */
        dev_class = class_create(THIS_MODULE, "ext_class");
        if (IS_ERR(dev_class)) {
                ...
        }
        /* Creating device */
        if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "ext_device"))) {
```

```
                ...
        }
        /* Register IRQ handler */
        if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "ext_device", NULL)) {
                ...
        }
        return 0;
}
```

3 типа драйверов устройств

- встроенные в ядро, соответсвующие устройства автоматически обнаружатся системой и станут доступными приложениям
- драйверы реализованные загружаемыми модулями ядра.
  Обычно драйверы управляют звуковыми и сетевыми картами и SCSI-адаптерами. Обычно при инсталляции системы устанавливаются перечень автоматически подключаемых модулей.
- поделенные между ядром и утилитой
  Принтер: система отвечает за взаимодействие с портом, отвечающим за формирование управляющих сигналов

Часто подсистему для работы с внешними устройствами называют подсистемой вводы-вывода, она занимает относительно большой процент кода ядра.

```
struct device {
        struct kobject kobj;
        struct device *parent;
        const char *init_name; /* initial name of the device*/
        const struct device_type *type;
        const struct bus_type *bus; /* type of bus device is on */
        struct device_driver *driver; /* which driver has allocated this device */
        u64 *dma_mask; /* dma mask (if dma'able device) */
        u64 bus_dma_limit; /* upstream dma constraint */
        #ifdef CONFIG_NUMA
        int numa_node; /* NUMA node this device is close to */
        #endif
        dev_t devt; /* dev_t, creates the sysfs "dev" */
};
```

Выделяют драйвера нижнего уровня, они пишутся разработчиками внешних устройств.

Современные ОС (WIN/Linux) позволяют управлять внешними устройствами с помощью контроллеров.

Общая структура описания драйверов:

```
struct device_driver
{
        const char *name;
        struct bus_type *bus; // тип шины
        struct module_t *owner;
        ...
        // точки входа
        int (*probe)(struct device *dev);
        ...
        int (*remove)(struct device *dev);
        void (*shutdown)(struct device *dev);
        int (*suspend)(struct device *dev);
        int (*resume)(struct device *dev);
        ...
```

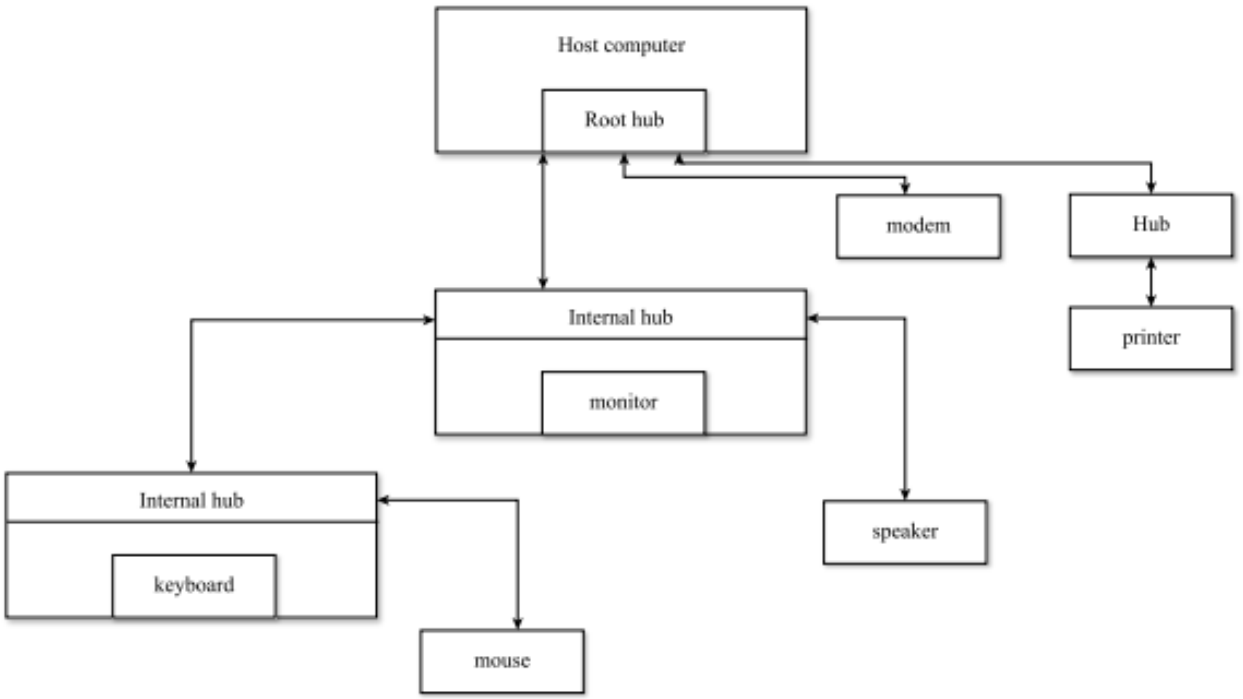
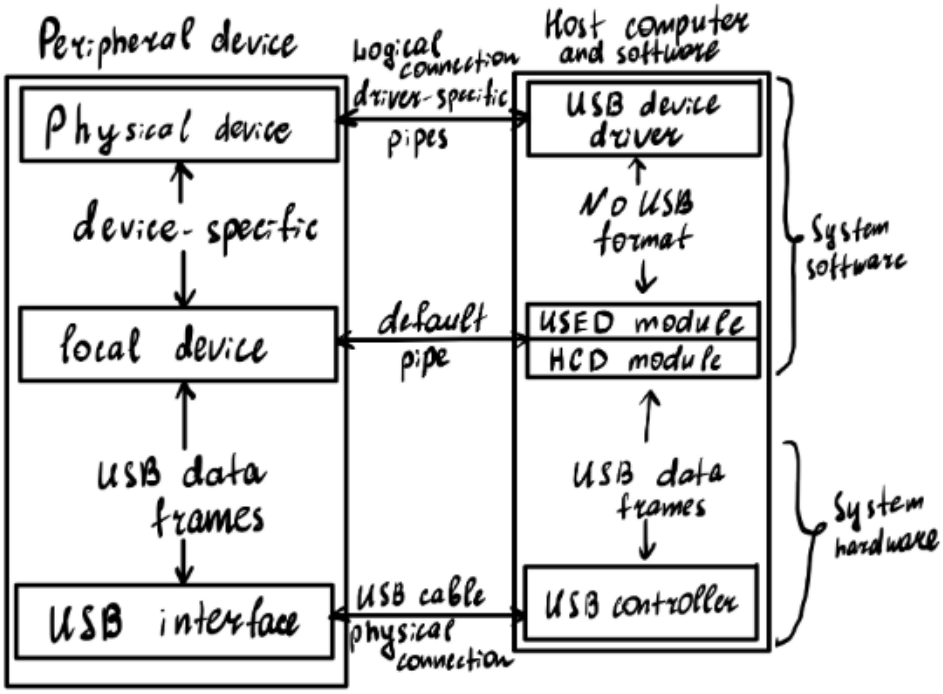


Рис. 12.1: USB топология

Хаб – сетевой концентратор или повторитель, через который соединены узлы сети. Host computer (главный) - корень USB-дерева, который содержит неявные узлы (пока не подключенные). Все USB-устройства работают по принципу горячего подключения: подключил и работает.

Потребность в usb возникла из-за увеличения количества устройств и повышения их быстродействия.

Хаб нужен для распределения электрического сигнала. Такая структура позволяет подключать большое количество устройств ввода/вывода.



В процессе передачи данных

– USB устройство инициирует передачу используя функции интерфейса USB драйвера. Работает по принципу опроса
– USB драйвер отсылает запросы HCD-модулю (host controller driver module)
– HCD делит запросы на отдельные транзакции учитывая возможности шины и характеристики USB-устройства, и планирует транзакции по шине.
– Аппаратная часть выполняет или завершает транзакции.

Транзакции определяются шиной и устройство полностью зависимо.

Базовым понятием при передачи между устройством и шиной является URB (USB request block)

```
struct urb {
        struct kref kref; // reference count of the URB
        struct list_head urb_list; /* list head for use by the urb's current owner */
        struct usb_device *dev; /* (in) pointer to associated device */
        struct usb_host_endpoint *ep; /* (internal) pointer to end point */
        unsigned int pipe; /* (in) pipe information */
}
```

Каждый пакет каждой транзакции содержит номер конечной точки. Драйверы считывает с устройства список конечных точек и создает структуры. Совокупность структур данных в ядре и конечных точек называется каналом (pipe).

4 режима передачи данных

– Control
  Двунаправленная передача данных для обмена короткими пакетами вида вопрос/ответ. Обычно осуществляется конечной точкой 0, но могут быть и другие. Позволяет прочитать информацию об устройстве PRODUCT_ID и VENDOR_ID
– Isochronous
  Гарантирует пропускную способность пакетов за 1 период шины. Также могут быть заданы задержки во время передачи. Данные передаются без подтверждения приёма, используется для устройств реального времени (видео/аудио)
– Interrupt
  Короткая, до 64 байт на полной скорости и до 8 байт на низкой. Характерны для ввода символов/координат. Имеют спонтанный характер и должны обслуживаться не медленнее, чем того требует устройство.
– Bulk
  Сплошная (поточная) передача данных. Используются устройства, принимающие/отправляющие большое количество данных, но не требующие определенной пропускной способности и времени задержек. Данные имеют самый низкий приоритет и занимают всю свободную полосу пропускания шины.

```
struct usb_driver
{
        const char *name;
        int (*probe)(struct usb_interface *intf,  const struct usb_device_id *id);
        void (*disconnect)(struct usb_interface *intf);
        ...
        int (*suspend)(struct usb_interface *intf, pm_message_t message);
        int (*resume)(struct usb_interface *intf);
        ...
        const struct usb_device_id *id_table; // идентификация устройства, обычно
    PRODUCT_ID и VENDOR_ID
        ...
        struct device_driver driver;
};
```

```
struct usb_device_driver {
        const char *name;
        bool (*match) (struct usb_device *udev);
        int (*probe) (struct usb_device *udev);
        void (*disconnect) (struct usb_device *udev);
        struct device_driver driver;
```

```
        const struct usb_device_id *id_table;
};
```

```
static int __init my_module_init(void)
{
        return usb_register(&my_drivers);
}

static void __exit my_module_exit(void)
{
        usb_deregister(&my_driver);
}
static struct usb_driver my_driver =
{
        .name = DEV_NAME,
        .probe = my_probe,
        .disconnect = my_remove,
        .id_table = my_table
};
static const struct usb_device_id my_table[] = {{USB_DEVICE(VID, PID)}, {}};

static int my_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
        struct usb_endpoint_descriptor *int_in;
        ....
        device_urb_interrupt = usb_alloc_usb(0, GPF_KERNEL);
        int res = usb_register_dev(interface, &my_class);
        ...
        usb_fill_int_urb(device.urb_interrupt, device.udev,
   usb_revintpipe(device_ndev, device_endpoint_address), device.data,
   device.buffer_size, proccess_key_press, NULL, device.time);
        ...
}
```

Зачем введены структуры USB? Видимо, вследствие широкого распространения usb и необходимости разработчиков ядра оптимизировать работу с ними. (но это не точно)