



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени
Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Конспект лекций и семинаров
по курсу
«Операционные системы», 2 семестр

Студент Сироткина П.Ю.

Группа ИУ7-66Б

Преподаватель Рязанова Н.Ю.

Содержание

1	Семинар №1. 10 февраля 2022	4
2	Лекция №1. 19 февраля 2022	5
2.1	Файловая подсистема Linux	5
3	Семинар №2. 24 февраля 2022	9
4	Лекция №2. 5 марта 2022	10
5	Семинар №3. 10 марта 2022	11
6	Лекция №3. 19 марта 2022	12
7	Семинар №4. 24 марта 2022	13
8	Лекция №4. 2 апреля 2022	14
9	Семинар 7 апреля 2022	15
10	Лекция 16 апреля 2022	16
10.1	Структура dentry	16
10.2	Кэш inode (слаб)	17
10.3	Структура FILE	19
11	Семинар 21 апреля 2022	20
11.1	Буфферизованный/небуфферизованный ввод-вывод (открытые файла)	20
12	Лекция 30 апреля 2022	24
12.1	Прерывания	24
12.2	Soft irq	27
13	Семинар 5 мая 2022	29
13.1	Системный вызов open (продолжение)	29
13.2	Разработка Virtual File System (VFS)	29
14	Лекция 14 мая 2022	31
14.1	Прерывания. Механизмы обслуживания прерываний в системе	31

14.2 Tasklet-ы	32
14.3 Workqueue (Очереди работ)	34
15 Семинар 19 мая 2022	37
15.1 Сокеты	37
16 Лекция 28 мая 2022	43
16.1 Прерывания. Механизмы обслуживания прерываний в системе. Очереди работ. .	43
16.2 5 моделей ввода-вывода	44
17 Семинар 2 июня 2022	50
17.1 Сокеты в файловом пространстве имен	50
17.2 Мультиплексирование ввода-вывода	53
18 Лекция 4 июня 2022	54
18.1 Специальные файлы устройств.	54

1. СЕМИНАР №1. 10 ФЕВРАЛЯ 2022

2. ЛЕКЦИЯ №1. 19 ФЕВРАЛЯ 2022

2.1 ФАЙЛОВАЯ ПОДСИСТЕМА LINUX

Файловая система предназначена для обеспечения возможности хранения и доступа к файлам в системе.

Файл (рабочий) - информация, хранимая во вторичной памяти или во вспомогательном ЗУ с целью ее сохранения после завершения отдельного задания или преодоления ограничений, связанных с объемом основного ЗУ.

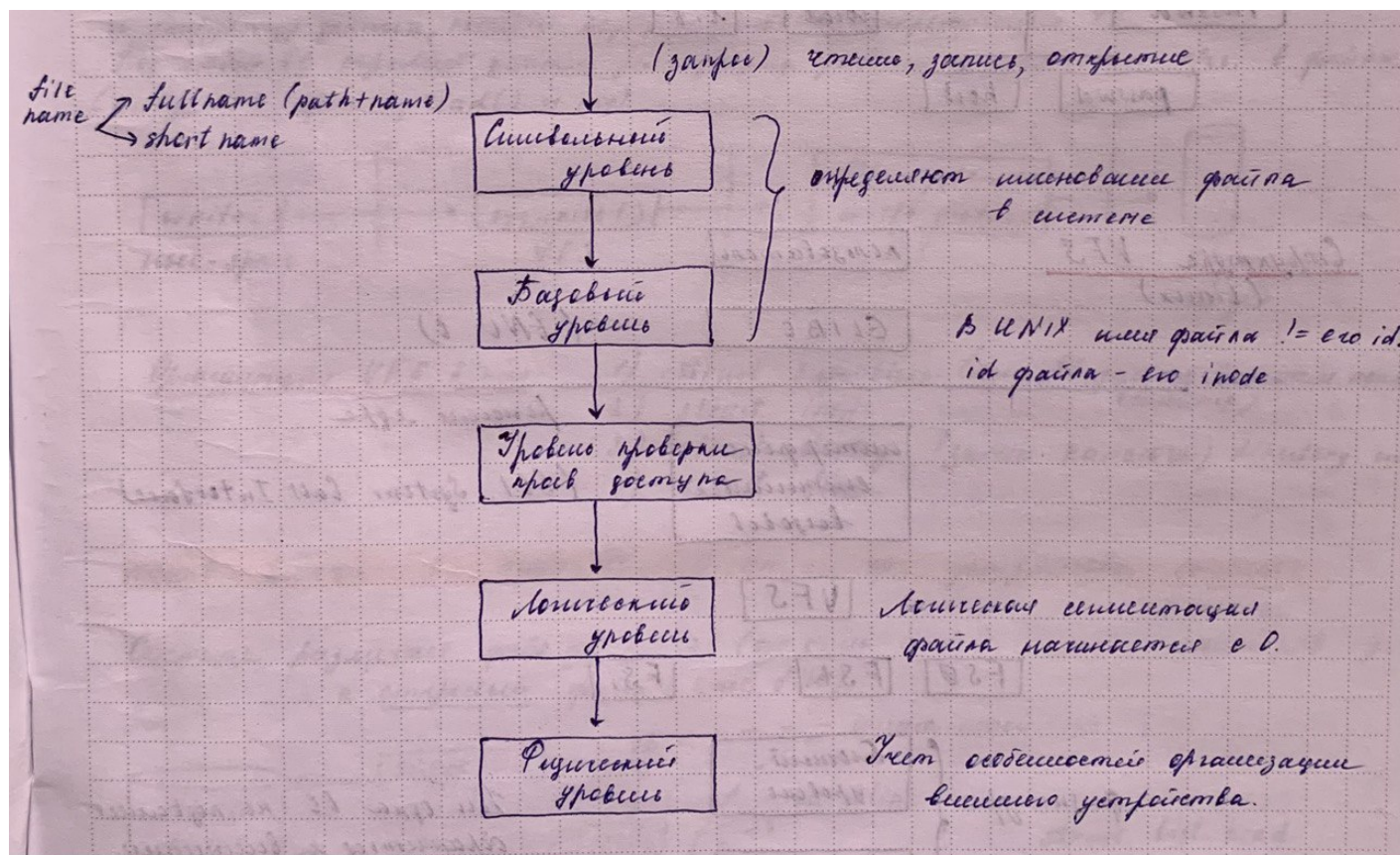
Файл - любая поименованная совокупность данных, которая хранится во вторичной памяти.

Файловая система - порядок, определяющий способ организации, хранения, именования и доступа к данным на вторичных носителях информации.

Любая ФС имеет иерархическую структуру, это связано с разными уровнями ФС, т.к. различные задачи, которые ей решаются, выполняются на разных уровнях ОС. Именованное файлов (символьный уровень) - самый высокий уровень ФС. Он позволяет пользователю в удобной форме задавать имена файлов и искать их в каталогах

Обобщенная модель ФС: файл хранится на физическом устройстве, причем в системе такое устройство внешнее, значит на нижнем уровне доступ к файлу осуществляется через подсистему ввода-вывода, т.е. для того, чтобы считать файл необходимо обратиться к внешнему устройству.

Иерархическая структура ФС:



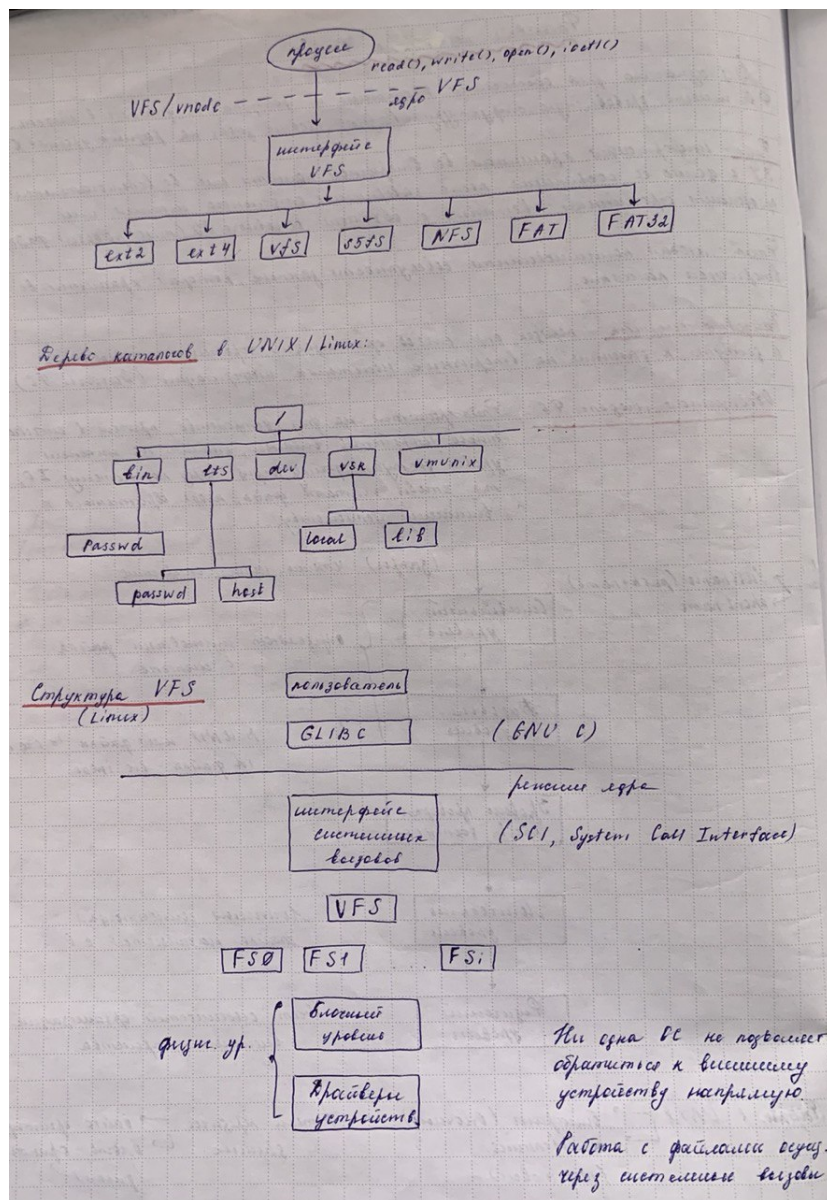
Полное имя = путь к файлу + имя файла. В UNIX путь начинается с корневого каталога. Права доступа RWX.

В данном случае файл похож на программу. Любая программа считает, что она начинается с 0-ого адреса, т.е. в программе находится смещение. Логическая организация файла начинается с 0.

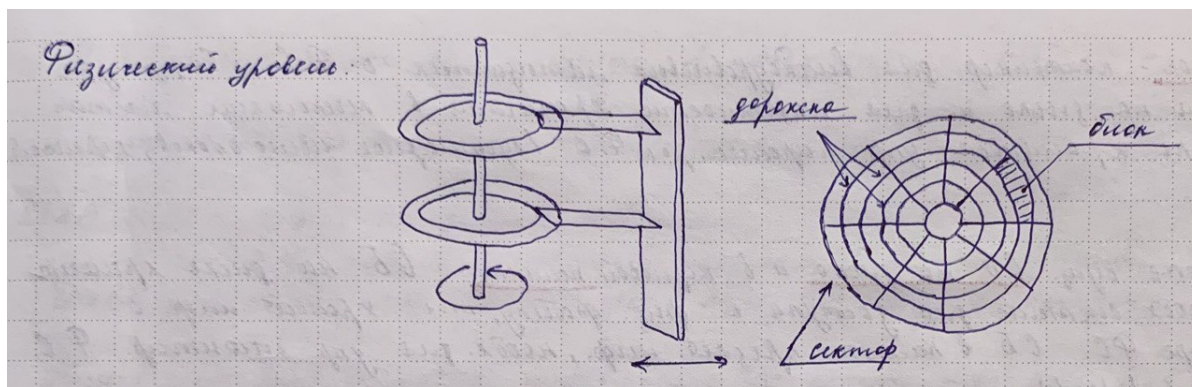
Бинарный файл - это блочный файл. Текстовый файл - это символьный файл. Типы файлов характеризуют операции над ними. Для блочного файла характерна возможность обращения к некому блоку (fseek), для текстового - наличие символа конца файла.

Это связано с тем, что ОС поддерживает 2 типа устройств (символьные и блочные). 2 типа потоков передачи данных: байто-ориентированные и блочно-ориентированные. Дисковые устройства и флэш-память - единственные блочные устройства. Все остальные устройства - символьные.

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



Демонстрация особенностей физического уровня:



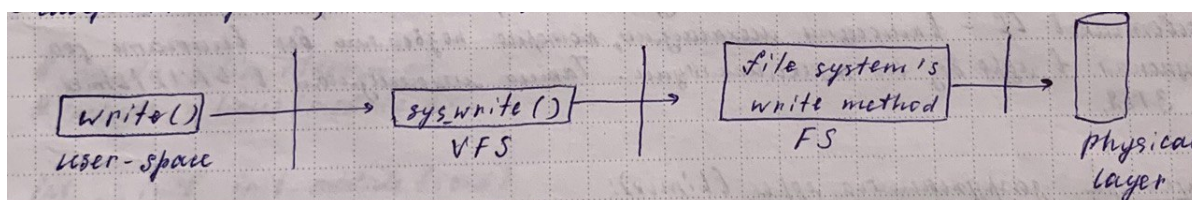
В дисках дорожки - концентрические окружности. Совокупность дорожек одного радиуса составляет цилиндр. Также они разделены на сектора. При этом мы знаем, что жесткий диск постоянно вращается и имеется считывающая/записывающая головка, которая перемещается поступательно, для того чтобы получить доступ к блоку на определенной дорожке.

Современные ФС поддерживают т.н. очень большие файлы. Проводится аналогия с управлением памятью страницами по запросу. Это означает, что любая страница адресного пространства процесса может быть загружена в любую страницу физической памяти. Для того, чтобы управлять этим, существуют таблицы страниц. Аналогично современные ФС обеспечивают хранение файлов вразброс, т.е. файл не занимает на диске непрерывную последовательность адресов. Это обеспечивается хранением адресов блоков. «Родная ФС» - ext2.

Возможность поддержки большого количества разных ФС реализуется за счет того, что в состав системы входит слой абстракции над собственным низкоуровневым интерфейсом ФС. Для этого ВФС предоставляет общую файловую модель, которая способна отображать общие возможности и поведение любой возможной ФС.

Такой уровень абстракции работает на основе базовых концептуальных интерфейсов и структур данных, которые поддерживаются конкретными ФС. Фактически код любой ФС скрывает детали реализации непосредственной работы с данными, организованными в файлы, а именно предоставляют пользователю набор API, таких как открыть файл, прочить, удалить и т.д. В любом варианте любая ФС поддерживает такие понятия как файл, каталог, и действия, определенные над ними.

Это можно представить следующей абстракцией:



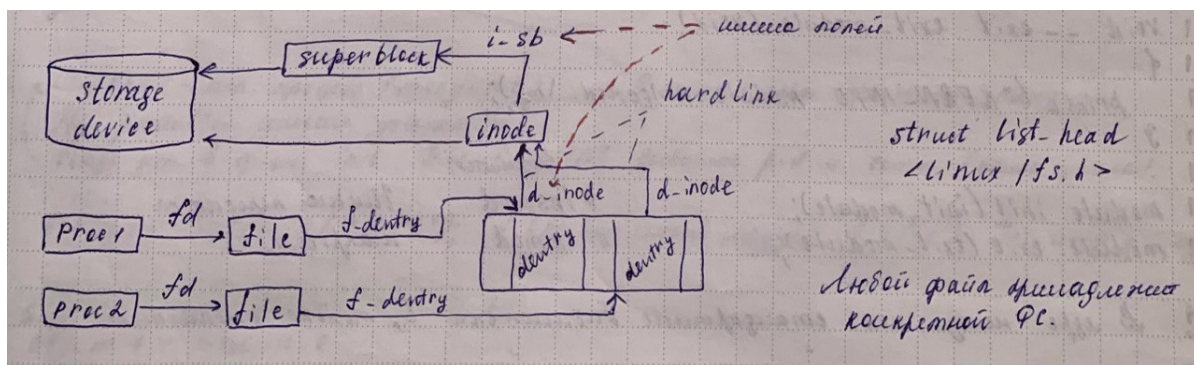
ВФС предоставляет общую файловую модель, которая наследует ФС, реализуя действия для различных posix api.

Как видно, системный вызов `write()` сначала обрабатывается общим системным вызовом `sys_write()`, которая определяет фактический способ записи файлов, характерный для такой

ФС, на которой находится файл. Затем общий системный вызов `sys_write()` вызывает метод конкретной ФС, чтобы выполнить запись данных на физический носитель. Т.е. ВФС скрывает особенности работы с конкретным физическим устройством.

Внутренняя организация VFS Linux:

- struct superblock (описывает ФС, расположенную на внешнем носителе);
- struct inode (индексный узел);
- struct dentry (запись каталога, dentry = directory entry);
- struct file.



Два процесса открывают один и тот же файл. Чтобы обеспечить к нему доступ, имеется `dentry` (обеспечивает работу с каталогами).

Система различает файл, находящийся на диске (его описывает struct inode) и открытый файл (его описывает struct file). i_sb, d_inode - не идентификаторы, а имена полей соответствующих структур.

Структура `superblock` в ВФС описывает конкретную подмонтированную ФС, т.е. ФС, которая располагается на внешнем носителе.

Суперблок - контейнер для высокоуровневых метаданных о ФС. Структура находится на диске и для надежности хранится в нескольких местах. В структуре хранятся управляющие параметры ФС (такие как суммарное число блоков, свободное число блоков, корневой inode и т.д.).

В системе существует суперблок на диске и суперблок в оперативной памяти. Суперблок на диске хранит информацию, необходимую системе для доступа к физическому файлу, т.е. хранит информацию о структуре ФС. Суперблок в памяти предоставляет информацию, необходимую для управления смонтированной ФС (`struct list_head`, связный список суперблоков, `<linux/fs.h>`).

Мониторинг - система действий, в результате которой ФС устройства становится доступной, т.е. можно получить доступ к информации, которая хранится на устройстве.

```
1 mount keys -t fs type -o options fs device catalog
```


3. СЕМИНАР №2. 24 ФЕВРАЛЯ 2022

4. ЛЕКЦИЯ №2. 5 МАРТА 2022

5. СЕМИНАР №3. 10 МАРТА 2022

6. ЛЕКЦИЯ №3. 19 МАРТА 2022

7. СЕМИНАР №4. 24 МАРТА 2022

8. ЛЕКЦИЯ №4. 2 АПРЕЛЯ 2022

9. СЕМИНАР 7 АПРЕЛЯ 2022

10. ЛЕКЦИЯ 16 АПРЕЛЯ 2022

10.1 СТРУКТУРА DENTRY

Ранее рассмотрели структуру dentry, которая описывает элемент пути, начиная с корневого каталога (со слэша).

Эта структура хранится в библиотеке `<linux/dcache.h>`. При этом в этом кэше кэшируются inode-ы, которые представляют элементы пути. Любой элемент пути/директория - это файл. Кэширование делается для того, чтобы сократить время доступа к файлам. Все данные, к которым мы обращаемся, кэшируются.

Если человек обратился к какому-либо файлу, наиболее вероятно, что он обратится к нему еще раз. На этом принципе (LRU) строится кэширование. Пользователь не может ждать неопределенно долго. Время ответа системы не должно превышать 3 секунд.

Обратимся к следующему файлу: `/home/dracula/src/foo.c`. Для того, чтобы система обратилась к файлу `foo.c`, необходимо, чтобы она «спустилась» по всем элементам пути.

Каждый такой элемент каталога - специальный файл, иначе невозможно хранить информацию об дереве каталогов, которое позволяет получить доступ к каталоговым файлам. Т.е. каждый такой элемент каталога будет иметь реальный inode, но эти элементы каталога с точки зрения структуры dentry создаются на лету.

Кэш объектов dentry состоит из:

1. Список используемых объектов dentry, которые связаны с определенным inode. В struct inode находится поле `i_dentr`. Поскольку один и тот же inode может иметь несколько ссылок, соответствующих директориям, то это поле должно представлять из себя связный список.
2. Двусвязный список неиспользуемых и негативных объектов dentry по алгоритму LRU, т.е. в списке находятся объекты dentry, к которым были последние обращения. Добавление в этот список нового объекта dentry должно выполняться по значению времени. Если происходит обращение в объекту dentry, он переносится в хвост. Новый объект dentry записывается в хвост. Удаление элементов выполняется из головы как наиболее долго находящийся в списке, к нему наиболее долго не было обращение.

LRU здесь используется в чистом виде, в отличие от LRU для страниц. Обращение к странице выполняется очень часто, на каждой команде, а то и несколько раз, причем в этой команде м.б. косвенная адресация, значит надо было бы каждый раз редактировать список LRU. Это очень большие накладные расходы. В случае файлов нет такого дикого кол-ва обращений.

3. Хэш-таблица (`dentry_hash_table`) и хэш-функция, которые позволяют преобразовать заданный путь в объект dentry. Каждый элемент этой таблицы (массива) является указателем на список тех объектов dentry, которые соответствуют какому-то ключу. Значение

ключа определяется функцией `d_hash()`, что позволяет для каждой ФС реализовать свою хэш-функцию. Поиск в хэш-таблице осуществляется функцией `d_lookup()`.

Приведенный пример показывает, что сначала идет поиск в `dentry_cache`, если поиск приводит к тому, что какого-то элемента каталога нет в этом кэше, то тогда производится обращение таким образом, который мы рассмотрели. В результате найденный объект `dentry` (его `inode`) будет помещен в кэш.

Структура `dentry` имеет указатель на `dentry_operations`, рассмотрим эту структуру.

```
1 struct dentry_operations
2 {
3     ...
4     int (*d_hash)(const struct dentry *, struct qstr *);
5     int (*d_compare)(const struct dentry *, const struct dentry *,
6                     unsigned int, const char *, const struct qstr *);
7     int (*d_delete)(const struct dentry *);
8     ...
9     struct vfsmount *(*d_automount)(struct path *);
10    ...
11 };
```

10.2 КЭШ INODE (СЛАБ)

Есть `dentry cache`, но `dentry` не исчерпывает всех потребностей. В `linux inode cache` находится в файле `fs/inode.c`.

`Inode cache` представляет из себя:

1. Глобальный хэш-массив `inode_hashtable()`, в котором каждый `inode` хэшируется по значению указателя на суперблок и номеру `inode`.

В случае отсутствия суперблока (`inode_i_sb == NULL`) вместо хэш-массива `inode` добавляется к двусвязному списку `anon_hash_chain`. Пример таких анонимных `inode`: сокеты, которые создаются функцией `sock_alloc()` (`<net/socket.c>`), которая вызывает функцию `get_empty_inode()` из `<fs/inode.c>`.

2. Глобальный список `inode_in_use`, содержит `inode`, у которых `i_count > 0` и `i_nlink > 0`. `Inode`-ы, созданные с помощью вызова функций `get_empty_inode()` и `get_new_inode()`, добавляются в этот список.
3. Глобальный список `inode_unused`, содержит правильные/допустимые `inode`, для которых `i_count = 0`.
4. Также для каждого суперблока имеется `sb->dirty` - список грязных `inode` (`i_count > 0` и `i_nlink > 0`, но они помечены как модифицированные, т.е. как `i_dirty`). Такой грязный

inode добавляется в список `sb→dirty`, если он кэширован. Это позволяет синхронизировать работу с inode.

5. Slab cache, который называется `inode_cacher`. Динамический (slab) кэш, в нем inode могут освобождаться, создаваться, вставляться и изыматься.

Все перечисленные списки защищены спин-локом `inode_lock()`.

Слаб - «брусек», имеющий определенный размер. Подход был введен для OS SUN.

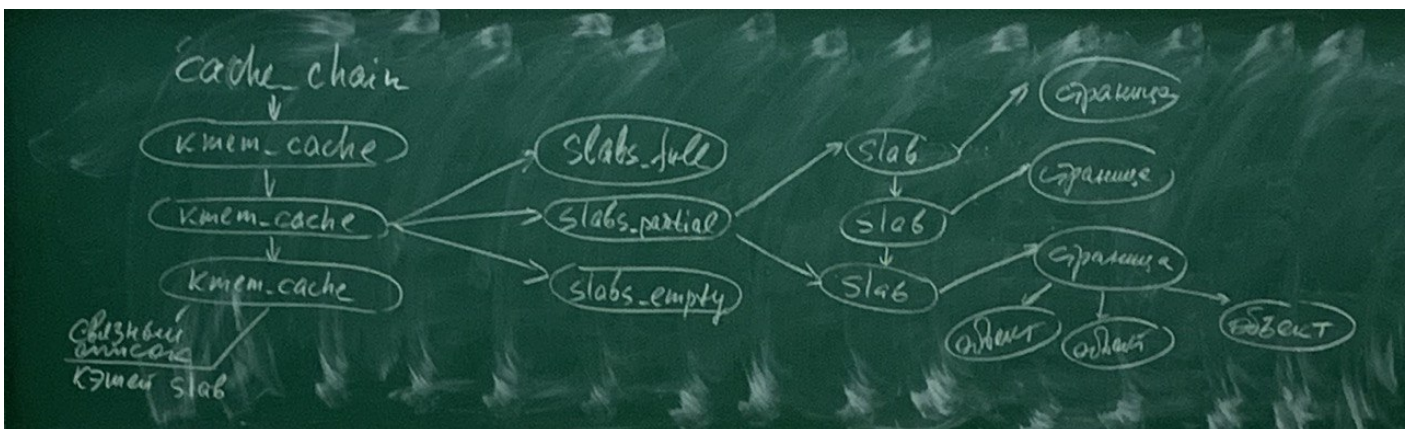
Идея: в ядре значительные объемы памяти выделяются на ограниченный и определенный набор объектов, таких как дескрипторы файлов, inode и т.д. При этом время для создания каждого такого объекта по меркам системы является значительным.

В результате для каждого такого объекты выполняется выделение памяти; когда объект уже становится не нужным, то выполняется освобождение объекта. При интенсивной работе в системе эти объекты постоянно создаются и освобождаются - это трата времени.

Поэтому, если такой объект был и создан и в нем на некоторое время отпала нужда, тем не менее не удалять его из памяти, то есть оставить соответствующий объем памяти, занимаемый объектом, причем оставить в проинициализированном состоянии, для того, чтобы в последующем этот проинициализированный объект использовать для объектов этого же типа.

(Спрашивает на экзамене про семафоры и мьютексы по отношению к слаб кэшу).

Главные структуры слаб-распределителя:



Связный список нужен для алгоритма `best_fit` - самый подходящий, т.е. ищет кэш более всего соответствующий размеру нужного рапсделения.

Виды слабов:

- o `slab_full` - распределен полностью;
- o `slab_partial` - распределен частично;
- o `slab_empty` - пустой, основной кандидат на rearing(возвращение для дальнейшего использования).

Объекты - основные элементы, которые выделяются из соответствующего кэша и возвращаются в него.

В реализации slabов участки памяти, подходящие для размещения объектов данных определенного размера и типа, определяются заранее. Распределитель (аллокатор) slabов хранит информацию о размещении этих участков, которые также известны как кэш. В результате, если возникает запрос на выделение памяти определенного размера, то он удовлетворяется быстро.

Существует библиотека `<linux/slab.h>`. ВФС `proc` предоставляет информацию о слабе: `cat /proc/slabinfo`.

Функции для работы со слабом:

```
1 struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t
    offset, unsigned long flags, void (*ctor)(void *));
2
3 int kmem_cache_destroy(kmem_cache_t *cache);
4
5 void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
6
7 void *kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

10.3 СТРУКТУРА FILE

Эта структура описывает т.н. открытые файлы, т.е. структура предоставляет информацию системе о файлах, которые были открыты процессами. Это одна единственная таблица, недоступная пользователю.

Для процесса открытия файла в различных структурах, даже в структурах ядра, имеется функция `open`. Мы говорим об обычных (regular) открытых файлах.

Работа этой структуры в системе начинается, когда какой-то файл открывается. Чтобы открыть файл в своих программах, мы вызываем системный вызов `open`. Сейчас мы говорим о режиме пользователя.

Библиотека `<fcntl.h>`:

```
1 int open(const char *pathname, int oflags, .../* mode_t *mode */);
2
3 fd = open("myfile", O_RDWR | O_CREATE | O_EXCL, 0644);
```

Флаг `O_CREATE` предписывает, что если файл не существует, то необходимо его создать. И этот флаг требует в обязательном порядке наличие следующего аргумента: `mode`.

Флаг `O_EXCL` предписывает, что если файл (файл?) уже существует, но задан флаг `O_CREATE`, возникла ошибка.

Такая комбинация флагов позволяет выполнять проверку существования файла и его создание атомарно.

Системный вызов `open` возвращает файловый дескриптор.

11. СЕМИНАР 21 АПРЕЛЯ 2022

11.1 БУФФЕРИЗОВАННЫЙ/НЕБУФФЕРИЗОВАННЫЙ ВВОД-ВЫВОД (ОТКРЫТЫЕ ФАЙЛА)

ЛР дает понимание, с какими функциями мы имеем дело, поскольку функции ввода-вывода могут быть буфферизованные (библиотека `stdio`), а функции `read`, `write`, `open` - функции небуфферизованного ввода-вывода. Работа в режиме пользователя.

Именно обычные файлы предназначены для долговременного хранения информации.

Традиционно программирование изучается под UNIX.

Начнем со структуры `task_struct`, поскольку файлы открывают процессы.

```
1 struct task_struct
2 {
3     ...
4     /* File System Information */
5     struct fs_struct *fs;
6     ...
7     /* Open File Information */
8     struct files_struct *files;
9     ...
10    /* namespaces */
11    struct nsproxy *nsproxy;
12 }
13
14 struct nsproxy
15 {
16     atomic_t count;
17     struct uts_namespace *uts_ns;
18     struct ipc_namespace *ipc_ns;
19     struct mnt_namespace *mnt_ns; //Previously only it
20     struct pid_namespace *pid_ns;
21     struct net *net_ns;
22 }
```

В современных UNIX-системах 5 namespace-ов. Дальше эту тему мы не развиваем.

Любой процесс когда-то был файлом, пока мы его не запустили на выполнение. Любой файл принадлежит определенной ФС. Когда файл открывается (системный вызов `open` или библиотечная функция `fopen`), в системе создается т.н. открытый файл, т.е. создается структура `struct_file` для этого файла в одной единственной таблице открытых файлов, содержащей дескрипторы всех открытых файлов в системе, в том числе файлов, открытых в ядре. Это таблица принадлежит ядру и не доступна пользователю.

Любая структура - это таблица. Соответственно, `struct_file` является дескриптором открытого файла.

Рассмотрим структуры:

```
1 struct fs_struct
2 {
3     int users;
4     spinlock_t lock;
5     seg_amount seg;
6     int umask;
7     int in_exec;
8     struct path, root, pwd; //Root and current dir
9 }
10
11 struct path
12 {
13     struct ufsmount *mnt;
14     struct dentry *dentry;
15 }
```

Следующая структура описывает файлы, открытые процессом (не более 64 файлов).

```
1 struct files_struct
2 {
3     ...
4     unsigned int next_fd; //amount of open by process files , next
5     unsigned long close_on_exec_init();
6     unsigned long open_fds_init();
7     struct file rcu *fd_array[NR_OPEN_DEFAULT]; //Constant - amount of open
           files
8 }
```

Поле `close_on_exec_init()`: указывает Число файлов, которые должны быть закрыты системным вызовом `exec`

Поле `open_fds_init()`: определяет число открытых файлов.

RCU - Real Compute Unit.

Поле `fd_array` - двумерный массив файловых объектов (массив указателей на `struct file`). Принято созданные системой открытые файлы по аналогии с другими структурами называть объектами, т.е. это экземпляр структуры `struct_file`, открытый файл.

Ни Unix, ни Linux не имеют отношения к ООП, они написаны на C.

Важно: `exec` - переводит процесс на новое адресное пространство программы, которое передано `exec`-у в качестве параметра. Процесс при этом не создается, он тот же, но в результате начинает выполняться другая программа, и ей не нужны файлы, которые были открыты до того, как она начала выполняться.

```

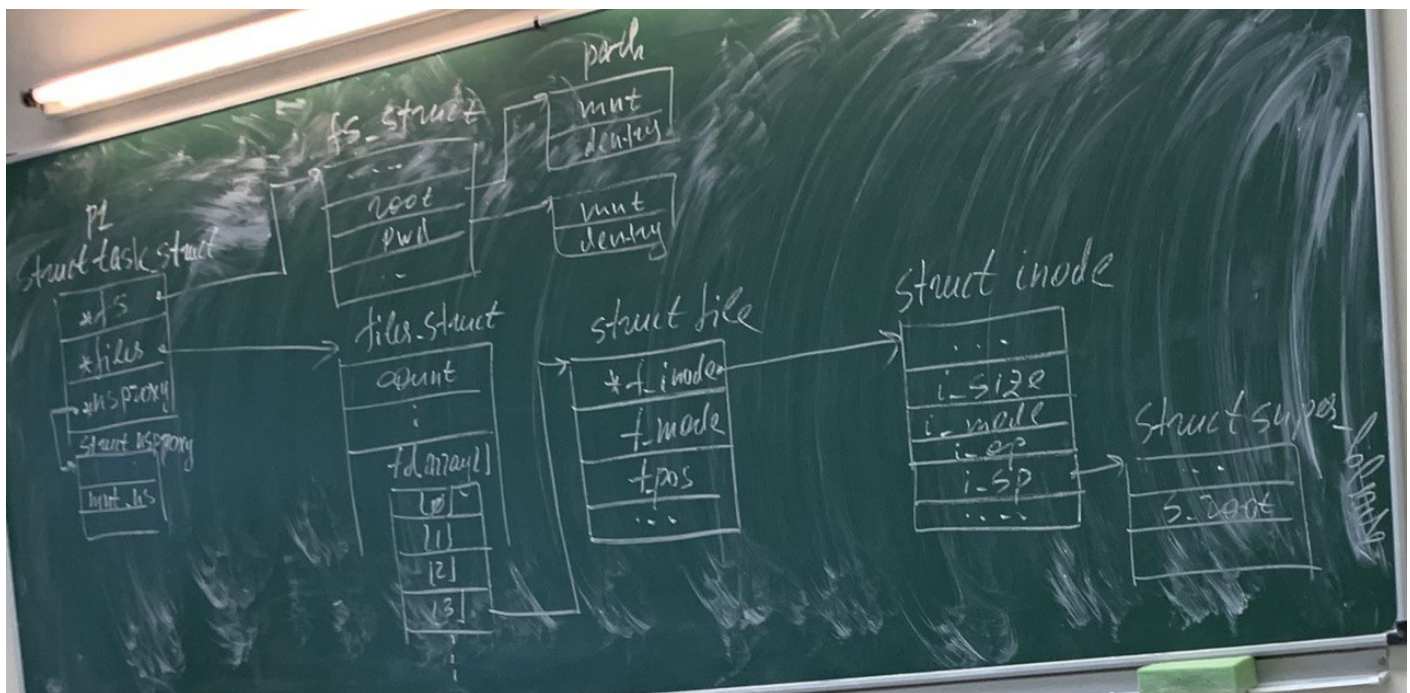
1 struct file
2 {
3     ...
4     struct path f_path;
5     struct inode *f_inode;
6     const struct file_operations *f_ops; //Defines ops on concrete open file.
7     atomic_long_t f_count;
8     unsigned int f_flags;
9     fmode_t f_mode; //Access mode
10    ...
11    loff_t f_pos; //Important
12 }

```

Когда процесс открывает файл, $f_pos = 0$ (нулевое смещение, начало файла).

Демонстрируемая проблема: разные процессы могут открывать один и тот же файл с одним и тем же inode, но они могут иметь разные имена. Один и тот же файл открывается несколько раз. В этом случае необходимо понимать, что может возникнуть при неправильной работе с данными. Также рисуем схему взаимодействия структур ядра.

Свяжем эти структуры:



Для любого процесса будут открыты потоки `stdin`, `stdout`, `stderr` (0, 1, 2 на схеме). Файлы, открытые программистом, будут начинаться с индекса 3. Каждый элемент этого массива ссылается на `struct file`. Нас интересуют файлы, открываемые `open` и `fork`.

Фактически данная связь структур отличает программы, использующие `open`. С `fork` добавляется еще одна структура. Чтобы получить дескриптор, определяем `struct __IO_FILE` (`<stdio.h>`). В этой структуре определены поля, связанные с буфером.

(Эта структура есть в отчете к ЛР)

```
1 struct __IO_FILE
2 {
3     ...
4     char * __IO_buf_base;
5     char * __IO_buf_end;
6     ...
7     int __fileno;
8 }
```

Поля: `fileno` - указатель на дескриптор открытого файла процесса, индекс в массиве дескрипторов открытых файлов процесса.

Библиотека `stdio` - библиотека буферизованного ввода-вывода. Если мы осуществляем запись в файл, то сначала информация записывается в буфер вывода. Если мы читаем информацию из файла, то сначала информация записывается в буфер чтения. Это создает дополнительные проблемы при работе с файлами, особенно когда один и тот же файл открывается несколько раз в одно и то же время.

Ситуации, при которых содержимое буфера переписывается в файл:

- Буфер заполнился;
- Вызов функции `fflush`;
- Вызов функции `fclose`.

Система автоматически закрывает часть файлов, если мы не указали `fclose`.

Одновременный доступ разных процессов к одному и тому же файлу является источником проблем, которые получили название `raise condition` (гонки).

Рассмотрим следующую ситуацию: пусть 2 процесса открывают один и тот же файл в режиме `read/write`. Один процесс добавляет запись в конец файла, для этого он сначала вызывает `fseek`, но вызывать функцию `write` не успевает, т.к. переходит в состояние сна, т.е. теряет квант. Второй процесс делает аналогичные действия, чтобы записать информацию в конец файла. Ему это удастся и он переходит к другой работе. Первый процесс пробуждается, продолжает выполнять указанные действия, т.е. он уже получил позицию, в которой должен записать данные в файл. Он выполняет запись, в результате он запишет свои данные поверх данных, которые записал второй процесс. Т.е. данные потеряны.

Решение: необходимо использовать неделимый (`atomic`) системный вызов, которым является системный вызов в режиме `O_APPEND` для `open`. Если этот режим установлен, то каждой операции `append` гарантируется неделимость.

Следующая ЛР: связана с системным вызовом `open`, предполагает создание алгоритма выполнения системного вызова `open`, т.е. это работа с функциями и структурами ядра. В `open` всегда вызывается `sys_open`, который реализован в виде `switch`. Т.е. в результате вызова система переходит из режима пользователя в режим ядра. Вызов `open` может открыть существующий

файл, а может создать новый. Если создается новый, должен быть создан inode. При одних флагах идет обращение к файлу, а при других создание inode, это целый ряд вызовов функций ядра.

12. ЛЕКЦИЯ 30 АПРЕЛЯ 2022

12.1 ПРЕРЫВАНИЯ

Основа работы ОС - система прерываний. В монолитном ядре все построено на прерываниях. В настоящее время принято выделять:

1. Системные вызовы.
2. Исключения.
3. Аппаратные прерывания.

Когда говорят о прерываниях(interrupts), то речь идет об аппаратных прерываниях.

Основную группу прерываний составляют прерывания от устройств ввода-вывода. Без этих устройств пользоваться вычислительной системой невозможно, т.к. именно эти устройства предназначены для взаимодействия с пользователем.

При этом аппаратные прерывания от внешних устройств возникают, когда внешнее устройство *завершило* операцию ввода-вывода (в соответствии с распараллеливанием функций в вычислительных системах).

Процессор не управляет внешними устройствами, ими управляют специальные устройства. В канальной архитектуре это каналы, в шинной архитектуре - контроллеры или адаптеры. Контроллеры, как правило, входят в состав устройства ввода-вывода, адаптеры находятся на материнской плате. В любом случае, это программно управляемое устройство.

Они получают команду, которая формируется драйвером устройства, когда у него имеется квант процессорного времени, поэтому часто говорят, что процессор посылает команду. Получив по шине данных такую команду, контроллер устройства переходит к ее выполнению. Процессор под управлением внешнего устройства выключается, т.е. переходит на выполнение другой работы. Процессор должен быть проинформирован о завершении ввода-вывода, в данном случае процесса, потому что процессор управляет всей работой, выполняемой вычислительной системой.

Для информирования процессора предназначены прерывания.

Поскольку процессор выполняет какую-то другую работу, то его работа организована следующим образом: в цикле выполнения каждой команды процессор проверяет наличие сигнала прерывания на своей выделенной ножке. Если сигнал прерывания пришел, то процессор переходит на обработку этого прерывания. (На экзамене спрашивают про адресацию прерываний).

Итог ЛР из прошлого семестра: адресация обработчика прерывания, и процессор переходит на выполнение соотв. обработчика прерывания.

Аппаратные прерывания выполняются на высочайших уровнях приоритета в ядре. Обработчики прерываний являются одной из точек входа драйвера. Один драйвер может иметь 1

обработчик прерывания. Обработчик прерывания выполняется на высочайшем уровне приоритета, они находятся выше DPC Dispatch. UNIX/Linux - числовой интервал приоритета. Уровень привилегий - другая величина!

Когда выполняются аппаратные прерывания, никакая другая работа в системе выполняться не может, но SMP (Symmetric Multi Processing, равноправные процессоры, которые работают с общей памятью) архитектура внесла в этот тезис некоторые изменения. Поскольку у нас несколько процессоров, которые выполняют работу параллельно/одновременно, причем разную, возникает ситуация: на том процессоре, который выполняет обработчик возникшего прерывания, запрещены все прерывания. Для остальных процессоров запрещены линии прерывания по этой линии IRQ (Interrupt Request).

Это критическая ситуация для системы, поэтому обработчики аппаратных прерываний должны завершаться как можно быстрее. Если они будут выполняться длительное время, это скажется на отзывчивости, быстродействии системы, поэтому обработчики АП выполняют минимально необходимый набор действий. Например, такой обработчик прерывания для устройства ввода должен получить данные от устройства и поместить их в буфер ядра.

Обработчики АП делятся на 2 части: верхняя и нижняя половины. Кроме сохранения пришедших данных в буфере ядра, АП, которым является top half, также инициализирует выполнение т.н. отложенных действий для того, чтобы система могла завершить обработку ввода-вывода.

В современных UNIX/Linux различаются 3 типа нижних половин:

- soft irq's - гибкие прерывания;
- tasklet;
- work queue - очередь работ.

Обработчик прерывания - одна из точек входа драйвера, соотв. любой драйвер может зарегистрировать в системе собственный обработчик прерываний. Для этого исп. специальные функции. Основная библиотека - `<linux/interrupts.h>`.

```
1  typedef irqreturn_t (*irq_handler_t)(int, void *);
2
3  int request_irq(unsigned int irq, void irq_handler_t handler,
4  unsigned long flags, const char *name, void *dev);
5
6  typedef int irqreturn_t;
7
8  #define IRQ_NONE (0)
9  #define IRQ_HANDLED (1)
10 #define IRQ_RETVAL(x) ((x) != 0)
```

Отсюда следует, что результат работы обработчика прерывания может возвращать или IRQ_NONE, или IRQ_HANDLED. Если прерывание обработано, необходимо возвращать IRQ_HANDLED, если не удалось выполнить обработчик - IRQ_NONE.

Деление действий, связанных с обслуживанием работы устройств ввода-вывода на аппаратные прерывания и отложенные действия связано с уровнем приоритетов, на которых должны выполняться АП и необходимостью завершить обработку операций ввода-вывода, и в итоге передать полученные данные (данные получаются в любом варианте), и когда процесс запрашивает операцию ввода, и когда процесс запрашивает операцию вывода, в любом случае процесс получает дополнительно информацию о выполненной операции (выполнена или нет).

В ядре различаются 2 вида АП: быстрые и медленные. В современных Linux-системах к быстрым прерываниям относится только прерывание от системного таймера. В версии 2.6.19 все флаги, связанные с прерыванием, были радикально изменены. В старой версии флаги имели приставку `sa`, она заменена на `irqf` (`sa_interrupt => irqf_timer`).

Несколько флагов и их 16-ричные значения:

1	<code>irqf_timer</code>	<code>0x00000200</code>
2	<code>irqf_shared</code>	<code>0x00000060</code>
3	<code>irqf_probe_shared</code>	<code>0x00000100</code>
4	<code>irqf_percpu</code>	<code>0x00000400</code>

Флаг `irqf_shared` устанавливается абонентами (теми, кто вызывает), для того, чтобы разрешить разделение линии IRQ разными устройствами. Устройством управляет драйвер. Разные драйвера устройств могут быть заинтересованы в использовании одной и той же линии IRQ, поэтому существует этот флаг, при этом даже одно устройство может иметь несколько обработчиков прерываний.

Флаг `irqf_probe_shared` устанавливается, если предполагается возможность наличия проблем при совместном использовании линии IRQ.

Флаг `irqf_percpu` предполагает, что выполнение обработчика прерывания будет закреплено за определенным процессором, т.е. монопольное выполнение.

Линиями прерывания можно управлять. Макросы управления линиями IRQ предельны в `<linux/irqflags.h>`.

Для локального процессора: `__local_irq_disable()`, для заперта одной линии прерывания `__local_irq_enable()`.

Для одной линии IRQ: `__void disable_irq(unsigned int irq)`, `__void disable_irq_nosync(unsigned int irq)`, `__void enable_irq(unsigned int irq)`.

Функция `void synchronize_irq(unsigned int irq)` предназначена для ожидания завершения обработчика прерывания по линии IRQ, если он выполняется. Например, обработчик прерывания от сетевого адаптера просто скопирует пришедший пакет в ядро. В ядре пакет ставится в буферную очередь, в которой ожидает обработчики соотв. потоком ядра.

В любом случае, перед завершением обработчик АП, который наз. `top half` в нашем обсуждении, инициализирует выполнение своей нижней половины (`bottom half`), отложенного действия.

После завершения выполнения обработчика АП (т.е. когда выполнена команда `return`), завершается взаимодействие с контроллером прерываний, т.е. код аппаратного прерывания выполнен, восстанавливаются локальные прерывания (т.е. разрешаются) на том процессоре, на котором выполнялся обратчик АП, восстанавливается старая маска прерываний (команда `iret`).

Продолжая мысль об обработчике прерываний сетевого адаптера, который копирует в ядро пришедший пакет, прерывание инициализирует выполнение отложенного действия и нижняя половина, инициализированная АП, завершит обработку получения пакета, при этом смысл деления на половины заключается в том, что нижние половины выполняются при разрешенных прерываниях.

12.2 SOFT IRQ

В системе существует перечисление определенных в системе гибких прерываний, они определяются статически при компиляции ядра.

<linux/interrupt.h>: определена структура:

```
1 struct softirq_action
2 {
3     void (*action)(struct softirq_action *);
4 }
```

Когда ядро выполняет обработчик отложенного прерывания типа `softirq`, функция `action` вызывается с указателем на структуру `softirq_action` в качестве параметра.

Количество типов `softirq` (имеет числовой индекс) статически определено в системе. Перечислим их. Существует 10 обработчиков.

Основная задача сетевой системы - прием и отправка пакетов. Она не ограничивается одним пакетом.

```
1 const char *const softirq_to_name[NR_SOFTIRQS] = {"HI", "TIMER", "NET_TX", "
    NET_RX", "BLOCK", "BLOCK_IO", "TASKLET", "SHED", "HRTIMER", "RCU"}.
```

`/proc/softirqs` - здесь можно увидеть эту информацию, т.е. все перечисленные имена.

Определена функция, которая заполняет вектор `softirq_vec` заданным типом `softirq_action`:

```
1 void open_softirq(int nr, void (*action)(struct softirq_action *))
2 {
3     softirq_vec[nr].action = action;
4 }
5
6 void raise_softirq(unsigned int nr);
```

Для того, чтобы зарегистрированное отложенное прерывание было поставлено в очередь на выполнение, необходимо вызывать функцию `raise_softirq`.

Добавить новый уровень `softirq` можно только путем перекомпиляции ядра. Т.е. число обработчиков `softirq` не может быть изменено динамически. При этом смысл имеет только новое `softirq`, имеющее индекс на 1 меньше индекса `tasklet`, т.к. нет смысла переопределять кол-во `softirq`, т.к. можно использовать `tasklet`.

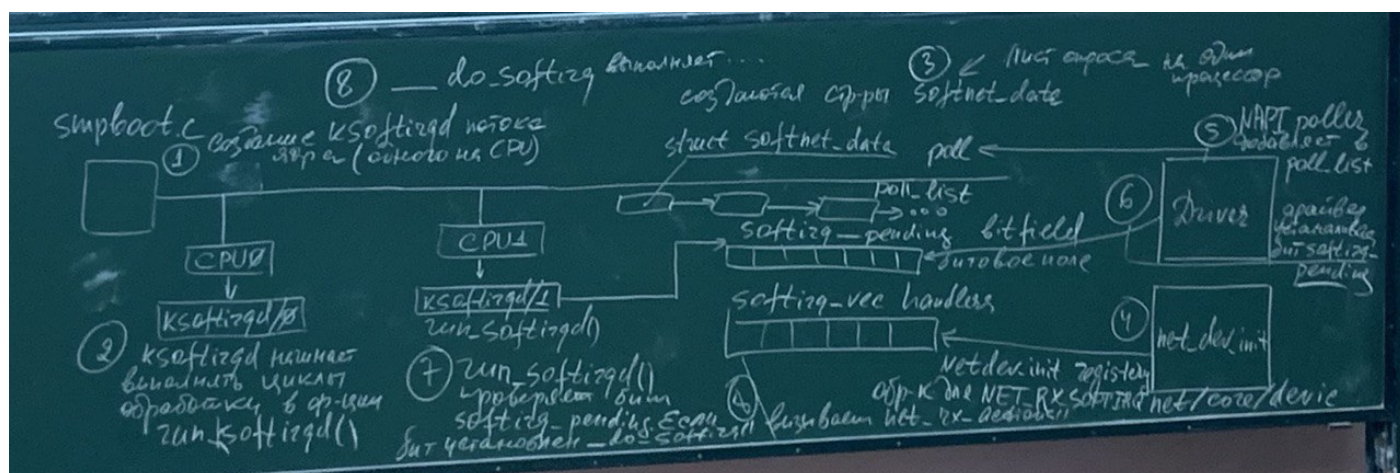
Из перечисления видно, что `tasklet` является одним из типов `softirq`.

Функция `raise_softirq()` с указанием конкретного индекса (`softirq`) д.б. вызвана из ОАП, т.е. ОАП должен отметить конкретное `softirq` как то, которое должно быть поставлено в очередь на выполнение.

Выполнение `softirq`. Проверка ожидающих выполнения обработчиков отложенных действий типа `softirq` вып. в следующих случаях:

1. При возврате из аппаратного прерывания.
2. В контексте потока ядра `ksoftirqd`.
3. В любом коде ядра, в котором явно проверяются и запускаются ожидающие выполнения обработчики `softirq` (например, как это делается в сетевой подсистеме).

Независимо от способа вызова `softirq`, его выполнение осуществляется функцией `do_softirq()`. Эта функция ункция в цикле роверяет наличие отложенных прерываний, т.е. каждый поток ядра `ksoftirqd` выполняет функцию `run_ksoftirqd()`, которая проверяет наличие отложенных действий типа `softirq`, и в зависимости от результата вызывает соотв. функцию.



13. СЕМИНАР 5 МАЯ 2022

13.1 СИСТЕМНЫЙ ВЫЗОВ OPEN (ПРОДОЛЖЕНИЕ)

Лабораторная по **open** проводилась с целью демонстрации того, как работает ядро ОС. Ядро предоставляет приложению интерфейс, который называется API (это системные вызовы). Приложение использует API, чтобы получить обслуживание запросов приложения, т.е. сервис. Ни одна ОС не позволяет приложениям напрямую обращаться к внешним устройствам.

13.2 РАЗРАБОТКА VIRTUAL FILE SYSTEM (VFS)

В ядре Linux имеется структура `file_system_type`.

```
1 struct file_system_type
2 {
3     const char *name;
4     int fs_flags;
5     #define FS_REQUIRES_DEV    1
6     ...
7     #define FS_USERNS_MOUNT    8 /* Can be mounted by usersns root */
8     struct dentry *(*mount)(struct file_system_type *,int,const char *, void *);
9     void (*kill_sb) (struct super_block *);
10    struct module *owner;
11    struct file_system_type *next;
12    ...
13 };
```

ФС реализовывается в виде загружаемого модуля ядра. ФС становится доступной, если она подмонтирована. Функции суперблока выполняются при вызове `mount`, но описана также `kill_sb`. В ядре имеется список таких структур (поле `next`). В системе может существовать только один тип ФС, например, `ext2`. В это же время одна и та же ФС может быть подмонтирована много раз, причем к разным директориям, которые будут для нее корневыми.

Рассмотрим пример объявления собственной ФС.

```
1 struct file_system_type fuse_fs_type =
2 {
3     .owner = THIS_MODULE;
4     .name = "fuse";
5     .fs_flags = FS_HAS_SUBTYPE | FS_USERNS_MOUNT;
6     .mount = fuse_mount;
7     .kill_sb = fuse_kill_sb_anon;
8 }
```

Поле `owner` определяет кол-во ссылок на модуль. Он нужен для того, чтобы модуль не был выгружен, если ФС подмонтирована, т.к. это может привести к краху. Счетчик ссылок не 0 - выгрузить не удастся.

После объявления ФС, ее нужно зарегистрировать: `register_filesystem()` в функции `init`, `unregister_filesystem()` в функции `exit`.

Прежде чем мы сможем выполнить монтирование ФС, необходимо заполнить поля `struct superblock` и определить операции на суперблоке.

```
1 static struct dentry *fuse_mount(struct file_system_type *fuse_fs_type, int
   flags, char const *dev, void *data)
2 {
3     struct dentry *entry = mount_bdev(fuse_fs_type, flags, dev, data, fill_sb);
4 }
```

Нужно четко понимать, где мы регистрируем свою функцию, а где регистрируем функцию, описанную в ядре (`generic`).

14. ЛЕКЦИЯ 14 МАЯ 2022

14.1 ПЕРЕРЫВАНИЯ. МЕХАНИЗМЫ ОБСЛУЖИВАНИЯ ПЕРЕРЫВАНИЙ В СИСТЕМЕ

Простейшую схему получения процессором сигналов прерывания мы рассматривали ранее. В современных системах такая схема не удовлетворяет никаким потребностям, и информирование процессора о возникающих аппаратных прерываниях выполняется с помощью сигналов MSI (Message Signal Interrupt). Это серьезная поддержка аппаратной составляющей системы, но мы не будем ее рассматривать.

В системе аппаратные прерывания делятся на быстрые и медленные с точки зрения их обслуживания в системе. Существует одно быстрое прерывание - от системного таймера. *Быстрые прерывания* имеют следующую особенность: их обработчики выполняются полностью, от начала до конца. *Медленные прерывания* - все остальные, от внешних устройств, при этом в вычислительных системах все устройства - внешние. Такие обработчики реализуются в виде двух частей: верхняя и нижняя половина. Верхняя половина выполняется на высочайшем уровне приоритета.

Когда выполняется обработчик АП в SMP-архитектуре, на том процессоре, на котором выполняется обработчик, запрещены все прерывания, а для других процессоров запрещены прерывания по этой линии IRQ. Обработчик АП должен завершаться как можно быстрее, поэтому он выполняет только самый необходимый объем действий, фактически его действия заключаются в получении данных с соотв. порта, к которому подключено внешнее устройство, и сохранение этих данных в буфере ядра.

Кроме этого, обработчик АП должен инициализировать работу отложенного действия второй половины. Вторая половина может быть реализована в виде softirq, tasklet или work queue. Каждый из перечисленных типов отложенных действий по обработке АП обладает особенностями, которые необходимо учитывать при выборе механизма реализации отложенных действий.

Ранее рассмотрели softirq, которые определяются статически, в системе определено ограниченное количество softirq, при этом tasklet - вид softirq. Мы рассмотрели схему, которая демонстрирует как устроено отложенное действие для сетевой подсистемы компьютера, поскольку именно сетевая подсистема способна воспринимать большое количество приходящих пакетов, при этом возникают соотв. АП, их может быть большое количество, и это довольно объемная обработка, связанная с очередью соотв. сообщений.

Демон ksoftirqd - поток ядра каждого процессора, задача которого - планирование и запуск на выполнение softirq. Когда машина нагружена гибкими прерываниями soft_interrupts, которые обслуживаются при завершении АП, при этом таких прерываний в единицу времени может возникать много, каждое из этих АП инициализирует свой обработчик отложенного действия softirq, в результате в системе для каждого процессора создается очередь softirq, сюда же входят и tasklet. При этом, например, по размеру очереди или по времени работы ksoftirqd, можно

определить степень давления на систему обращений от сетевого устройства. Сетевое устройство обращается к системе за обслуживанием. Это признак нагрузочной атаки.

Чтобы управлять выполнением отложенных действий типа softirq, предназначены такие демоны.

14.2 TASKLET-Ы

Tasklet - частный случай реализации softirq, но в отличие от softirq, которые пишутся таким образом, чтобы одновременно в системе могло выполняться некоторое количество одних и тех же softirq (в полном объеме реализовано взаимное исключение), на tasklet накладывается следующее ограничение: обработчик тасклета в каждый конкретный момент времени может выполняться только на одном процессоре, т.е. один и тот же тасклет не может выполняться параллельно, в отличие от softirq.

Поэтому в системах, в которых требуется быстрая реакция, предпочтение отдается softirq. Реализация тасклета намного проще, чем реализация softirq ввиду этого ограничения. Тасклеты - хороший компромисс между производительностью системы и простотой использования.

В отличие от softirq, которые регистрируются при компиляции системы и их количество определено в системе, тасклеты могут быть зарегистрированы как статически, так и динамически.

Тасклеты как объекты ядра описываются соотв. структурой.

```
1 struct tasklet_struct
2 {
3     struct tasklet_struct *next;
4     unsigned long state;
5     atomic_t count;
6     bool use_callback;
7     union
8     {
9         void (*func)(unsigned long data);
10        void (*callback)(struct tasklet_struct *t);
11    };
12    unsigned long data;
13 };
14
15 enum
16 {
17     TASKLET_STATE_SCHED, // Scheduled
18     TASKLET_STATE_RUN // Running (executing)
19 }
```

В системе имеется связный список обработчиков тасклет. Поле state определяется по enum.

Таслет может быть объявлен статически или динамически:

```
1  \\Static
2  #include <linux/interrupts.h>
3
4  DECLARE_TASKLET(name, func, data);
5  DECLARE_TASKLET_DISABLED(name, func, data);
6
7  \\Dynamic
8  #define DECLARE_TASKLET(name, __callback)
9  struct tasklet_struct name =
10 {
11     .count = ATOMIC_INIT(0);
12     .callback = __callback;
13     .use_callback = true;
14 };
15
16 extern void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
17                         unsigned long data);
18
19 extern void __tasklet_schedule(struct tasklet_struct *t);
20
21 static inline void tasklet_schedule(struct tasklet_struct *t)
22 {
23     if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
24         __tasklet_schedule(t);
25 }
26
27 //High priority tasklet
28 extern void __tasklet_hischedule(struct tasklet_struct *t);
29
30 static inline void tasklet_hischedule(struct tasklet_struct *t)
31 {
32     if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
33         __tasklet_hischedule(t);
34 }
35
36 ... tasklet_disable(...);
37 ... tasklet_disable_nosync(...);
38 ... tasklet_enable(...)
39
40 extern void tasklet_kill(struct tasklet_struct *t);
41 extern void tasklet_kill_immediate(struct tasklet_struct *t, unsigned int cpu);
42
```

```

43 void tasklet_handler(unsigned long data);
44
45 static inline tasklet_trylock(struct tasklet_struct *t)
46 {
47     return !test_and_set_bit(TASKLET_STATE_RUN, &t->state);
48 }

```

Обработчик АП инициализирует отложенное действие, чтобы его выполнить, необходимо запланировать тасклет.

Ограничение тасклета: таскет не может блокироваться, т.е. в таскетах нельзя использовать блокирующие примитивы. Если в таскете используются общие с обработчиком прерывания или каким-то другим таскетом данные, то необходимо использовать спинлоки.

Свойства тасклетов:

- Если вызывается функция `tasklet_schedule()`, то таскет гарантировано будет выполняться на каком-то процессоре, хотя бы один раз после этого.
- Если таскет уже запланирован, но его выполнение еще не началось, то он обязательно будет один раз выполнен через какое-то время.
- Если таскет уже выполняется на другом процессоре или планирование тасклета вызвано из самого кода тасклета, то его перепланирование будет отложено.
- Если один и тот же таскет выполняется несколько раз (он может даже сам себя запланировать), то для монопольного доступа к разделяемым данным необходимо использовать спинлоки.
- В системе имеется определенная на таскетах функция блокировки: `tasklet_trylock()`.

14.3 WORKQUEUE (ОЧЕРЕДИ РАБОТ)

Очереди работ, в отличие от тасклетов, могут блокироваться, но между ними значительно больше отличий. Главное отличие - гибкими прерываниями, таскетами «руководит» `ksoftirqd`, а очереди работ в системе реализованы совершенно по-другому.

Наибольшие изменения произошли с очередями работ в связи с управлением параллельным выполнением очередей работ, и это связано с SMP-архитектурой.

Несколько концепций очередей работ `workqueue` представляют собой связанные с работой структуры данных, которые легко перепутать, т.е. в ядре в отличие от тасклетов, над которыми определена одна структура, над очередями определено несколько структур, каждая из которых играет важную роль.

Работа - та работа, которую должен выполнить обработчик отложенного действия. работы ставятся в некоторую очередь, при этом в одну очередь работ может быть поставлено много работ. Работа связывается с конкретной очередью. В системе определен т.н. `worker` (рабочий) -

поток ядра (work thread). В системе имеется worker pull - множество worker, один worker может принадлежать одному pull.

Посредник, ответственный за установку отношений между workqueue и worker pull(?).

Существенные **отличия** между тасклетами и очередями работ:

1. Тасклеты выполняются в контексте прерывания, в результате чего код тасклета должен быть неделимым(atomic). В отличие от тасклетов, workqueue выполняются в контексте специальных потоков ядра, и как результат имеют большую свободу действий, и в частности очереди работ могут блокироваться или засыпать.
2. Тасклеты всегда выполняются на процессоре, на котором выполнялось АП, запланировавшее данный таскет. Очереди по умолчанию также выполняются на том же процессоре, но могут выполняться и на других процессорах.
3. Код ядра требует, чтобы выполнение функций очередей работ откладывалось на определенный интервал времени.
4. Ключевое отличие - тасклеты выполняются за короткий период времени после того, как были запланированы, а очереди работ имеют значительно большие задержки и необязательно должны быть атомарными, неделимыми.

Каждый из перечисленных механизмов обладает существенными отличиями, поэтому выбор соотв. механизма должен быть осмысленным и обоснованным.

```
1 //linux 3.18.22
2 struct workqueue_struct
3 {
4     struct list_head pwqs;
5     struct list_head list;
6     struct mutex mutex;
7     ...
8     char name[WQ_NAME_LEN];
9     struct rcu_head rcu;
10    ...
11 };
12
13 struct work_struct
14 {
15     atomic_long_t data;
16     struct list_head entry;
17     work_func_t func;
18     #ifndef CONFIG_LOCKDEP
19     struct lockdep_map lockdep_map;
20     #endif
21 };
```

List_head - список всех очередей работ, но на конкретное CPU существует свой список rcu_head, чтобы не перебирать все очереди работ.

Так же как для тасклетов, работу (задачу как отложенное действие) можно поместить в очередь работ как статически, так и динамически.

```
1 //static
2 DECLARE_WORK(name, void (*func)(void*)); //name = work_struct instance
3
4 //dynamic
5 #ifndef CONFIG_LOCKDEP
6 #define __INIT__WORK(__work, __func, __onstack)
7     do
8     {
9         static struct lock_class_key __key;
10        __init_work((__work), __onstack);
11        (__work)->data = (atomic_long_t)WORK_DATA_INIT();
12        INIT_LIST_HEAD(&(__work)->entry);
13        (__work)->func = (__func);
14    }
15    while(0);
16
17 #define INIT_WORK(__work, __func)
18    __INIT__WORK((__work), (__func), 0)
19
20 typedef void (*work_func_t)(struct work_struct *work);
21
22 PREPARE_WORK();
23
24 int alloc_workqueue(char *name, unsigned int flags, int max_active);
25
26 //Flags
27 enum {
28     WQ_UNBOUND = 1 << 1;
29     WQ_FREZABLE = 1 << 2;
30 }
```

Если имеется хоть какая-то возможность, что структура уже инициализирована, лучше вместо init_work() использовать prepare_work().

Для создания очереди работ, до 2.6.36, использовалась create_workqueue(), в более современных версиях - alloc_workqueue(). Flags определяет как очередь работ будет выполняться, max_active - ограничивает число задач, которые одновременно будут стоять в очереди к спу.

15. СЕМИНАР 19 МАЯ 2022

15.1 СОКЕТЫ

Сокеты - универсальное средство взаимодействия параллельных процессов. Нужно сравнивать со средствами взаимодействия из `systemfive`.

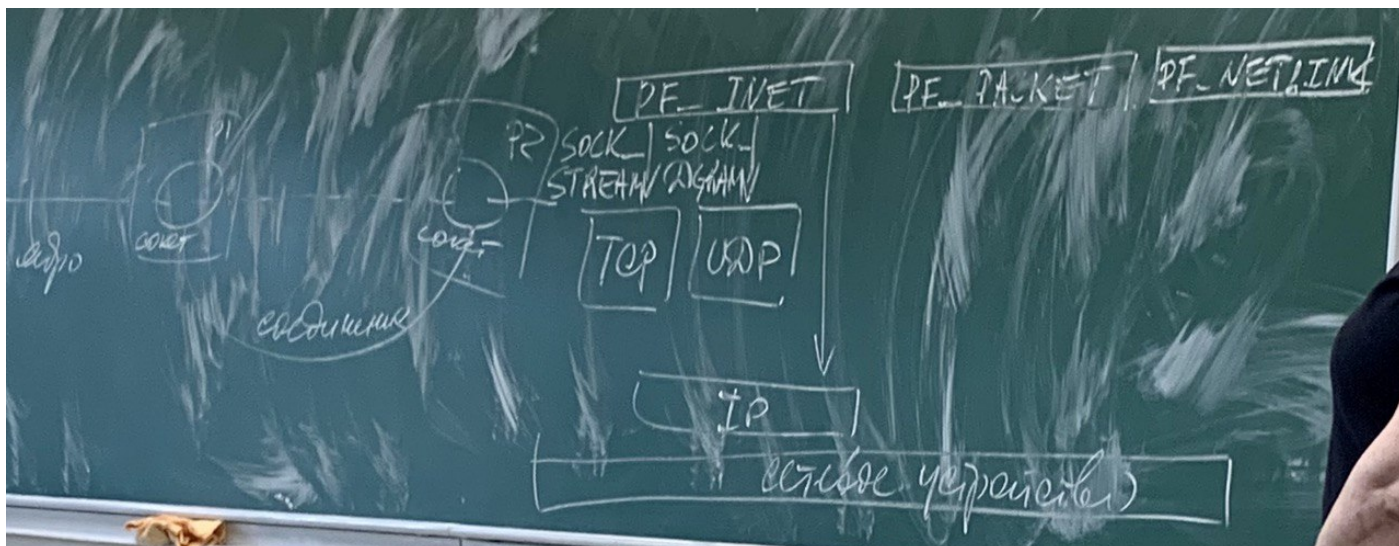
Универсальность заключается в том, что сокеты используются как и на локальной машине, так и в распределенной системе (сети), в отличие от, например, разделяемой памяти, которая применима только на отдельно стоящей машине.

Различают 3 типа сокетов:

- Парные - альтернатива `pipe`, но в отличие от них обеспечивают дуплексную связь;
- Сокеты домена UNIX - предназначены для взаимодействия параллельных процессов на локальной машине по модели клиент-сервер;
- Сетевые (`af net`) сокеты - предназначены для взаимодействия параллельных процессов в распределенной сети по модели клиент-сервер.

Сокет - абстракция конечной точки взаимодействия, потому что для разных систем сокет поддерживается в системе разными средствами. Система предоставляет пользователю соответствующее API.

```
1 int socket(int family, int type, int protocol);
```



Здесь показаны сокеты в UNIX BSD. Самым важным моментом здесь является `SOCK_STREAM TCP` и `SOCK_DGRAM UDP`.

`PF_PACKET` - сокеты для профессиональной деятельности, созданные для непосредственного доступа приложения к сетевым устройствам.

PF_NETLINK - группа сокетов, созданных для обмена информацией между частями ядра, еще один интерфейс ядра, которая позволяет получить информацию из ядра в пользовательское пространство.

Функция socket в результате приведет к вызову sys_socket, но в Linux имеется один единственный вызов и он работает как switch.

```
1 asmlinkage long sys_socketcall(int call, unsigned long *args)
2 {
3     unsigned long a[6];
4     unsigned long a0,a1;
5     int err;
6
7     if(call < 1 || call > SYS_RECVMSG)
8         return -EINVAL;
9
10    /* copy_from_user should be SMP safe. */
11    if (copy_from_user(a, args, nargs[call]))
12        return -EFAULT;
13
14    a0=a[0];
15    a1=a[1];
16
17    switch(call)
18    {
19        case SYS_SOCKET: err = sys_socket(a0,a1,a[2]); break;
20        case SYS_BIND: err = sys_bind(a0,(struct sockaddr *)a1, a[2]); break;
21        case SYS_CONNECT: err = sys_connect(a0, (struct sockaddr *)a1, a[2]);
22            .....
23        default: err = -EINVAL; break;
24    }
25    return err;
26 }
```

В switch перечисляются все функции, определенные на сокетах. Каждая из функций реализуется в ядре. Например:

```
1 asmlinkage long sys_socket(int family, int type, int protocol)
2 {
3     int retval;
4     struct socket *sock;
5
6     retval = sock_create(family, type, protocol, &sock);
7     if (retval < 0)
8         goto out;
9 }
```

```

10  retval = sock_map_fd(sock);
11  if (retval < 0)
12      goto out_release;
13
14  out:
15  /* It may be already another descriptor 8) Not kernel problem. */
16  return retval;
17
18  out_release:
19  sock_release(sock);
20  return retval;
21 }

```

Функция `sock_create()` инициализирует поля структуры `socket`. Рассмотрим эту структуру.

```

1 struct socket
2 {
3     socket_state      state;
4     short             type;
5     unsigned long     flags;
6     struct socket_wq __rcu *wq;
7     struct file       *file;
8     struct sock       *sk;
9     const struct proto_ops *ops;
10 }

```

У сокета различают 5 состояний, 4 из которых - стадии соединения:

- `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.

Флаги используются для синхронизации доступа к сокетам.

Важнейшим полем является структура `proto_ops` (от слов `protocol operations`). Операции на сокетах зависят от типа сокета, но протокол зависит от типа сокета.

Первым параметром является `family` (или `domain`). `AF` = `address family`, можно перевести как пространство имен.

- `AF_UNIX` - эти сокеты предназначены для межпроцессорного взаимодействия на локальном компьютере;

- AF_INET - эти сокеты предназначены для взаимодействия параллельных процессов в распределенных системах, а именно в интернете версии 4, т.е. доменом является любая ipv4-сеть, также это сокеты семейства протоколов tcp/ip;
- AF_INET6 - ipv6;
- AF_IPX - семейство протоколов ipx;
- AF_UNSPEC - неопределенный домен.

Поле type содержит тип сокета. Это второй параметр в функции socket. Он может иметь значение:

- SOCK_STREAM - сокеты потоков, они определяют ориентированное на потоки надежное упорядоченное полнодуплексное логическое соединение между двумя сокетами(point to point);
- SOCK_DGRAM - определяют ненадежную службу DATAGRAM без установления логического соединения, причем пакеты могут передаваться без сохранения порядка (т.н. широковещательная передача данных). Альтернатива - 1-ый тип;
- SOCK_RAW - низкоуровневый интерфейс DATAGRAM по протоколу IP, причем необязательно posix1, сокет взаимодействует с сетевым устройством напрямую;
- SOCK_RDM;
- SOCK_SEGPACKET;
- SOCK_PACKET - не используется.

Основными типами сокетов являются SOCK_STREAM, SOCK_DGRAM, SOCK_RAW.

Третий параметр - протокол. Протоколы выбираются в зависимости от семейства и типа сокета. Обычно этот параметр имеет значение 0, в этом случае протокол выбирается по умолчанию.

Для протоколов также устанавливаются обозначения. Это IPPROTO_*, например IPPROTO_TCP, IPPROTO_DGRAM, IPPROTO_UDP.

Для семейства AF_INET для типа сокета SOCK_STREAM всегда будет выбираться IPPROTO_TCP.

Для семейства AF_INET для типа сокета SOCK_DGRAM всегда будет выбираться IPPROTO_UDP.

Поскольку сокеты - абстракция конечных точек соединения, то необходимо указать адреса этих точек соединения. Поэтому в системе определена структура sockaddr. Она определена в общем виде, потому что существуют разные типы, семейства и протоколы сокетов.

```

1 struct sockaddr
2 {
3     unsigned short  sa_family; // Address family , i.e. AF_XXX
4     char            sa_data[14]; // 14 bytes for adress storing
5 };

```

Сокеты в файловом пространстве имен: сокеты семейства AF_UNIX. После создания такого сокета его можно увидеть в ФС командой ls как специальный файл, обозначаемый буквой s.

Сетевые сокеты мы здесь не увидим. Сетевые сокеты мы можем увидеть только с помощью специальных команд, посмотрев номер соответствующего порта.

Рассмотрим фрагмент программы для сокетов семейства AF_UNIX, демонстрирующий каким образом для этих сокетов заполняются поля структуры sockaddr.

```

1 sock = socket(AF_UNIX, SOCK_DGRAM, 0);
2 if (sock < 0)
3 {
4     perror("socket failed");
5     return EXIT_FAILURE;
6 }
7 srvr_name.sa_family = AF_UNIX;
8 strcpy(srvr_name.sa_data, "socket.soc");
9 if (bind(sock, &srvr_name, strlen(srvr_name.sa_data) +
10         sizeof(srvr_name.sa_family)) < 0)
11 {
12     perror("bind failed");
13     return EXIT_FAILURE;
14 }

```

Здесь мы видим, что установлено семейство AF_UNIX и тип SOCK_DGRAM. Если использовать другой тип, ошибки не будет, но в этом не будет смысла, потому что, как мы видим, у нас как раз второй параметр (sa_data) определяет имя специального файла, и взаимодействие сокетов AF_UNIX выполняется через файловую подсистему (через специальные файлы), поэтому никакой потери данных не будет.

Для UNIX для сокетов AF_INET такой общий адрес struct sockaddr не подходит. Для взаимодействия процессов в сети необходимо указывать адреса и номер порта, поэтому структура не подходит. Поэтому для интернета была определена другая структура:

```

1 struct sockaddr_in
2 {
3     short int sin_family; // AF_INET
4     unsigned short int sin_port; // Port number
5     struct in_addr sin_addr; // IP-address
6     unsigned char sin_zero[8]; // Offsetting to the size of struct sockaddr
7 };

```

```
1 struct in_addr {  
2     unsigned long s_addr;  
3 };
```

Порядок байтов: little (аппаратный) endian и big (сетевой) endian. Поскольку речь идет о значениях адресов, то этот порядок имеет значение. Чтобы не возникало неоднозначности при получении адресов, используются 4 функции.

```
1 uint16_t htons(uint16_t hostint16);  
2 uint32_t htonl(uint32_t hostint32);  
3  
4 uint16_t ntohs(uint16_t hostint16);  
5 uint32_t ntohl(uint32_t hostint32);
```

Эти функции надо использовать всегда. Если сетевой и аппаратный порядок байтов совпадают, то эти функции не будут выполняться.

16. ЛЕКЦИЯ 28 МАЯ 2022

16.1 ПРЕРЫВАНИЯ. МЕХАНИЗМЫ ОБСЛУЖИВАНИЯ ПРЕРЫВАНИЙ В СИСТЕМЕ. ОЧЕРЕДИ РАБОТ.

Ранее перечислили объекты, которые определены в системе для работы с очередями работ: work, workqueue, worker, worker pull, pwq (pull worker queue).

Флаги (на прошлой лекции еще 2 флага было):

```
1 WQ_MEM_RECLAIM
2 WQ_HIGHPRI
3 WQ_SYSFS
4 WQ_MAX_ACTIVE
```

Очереди высокого приоритета (WQ_HIGHPRI) выполняются в первую очередь. Есть очереди нормального приоритета.

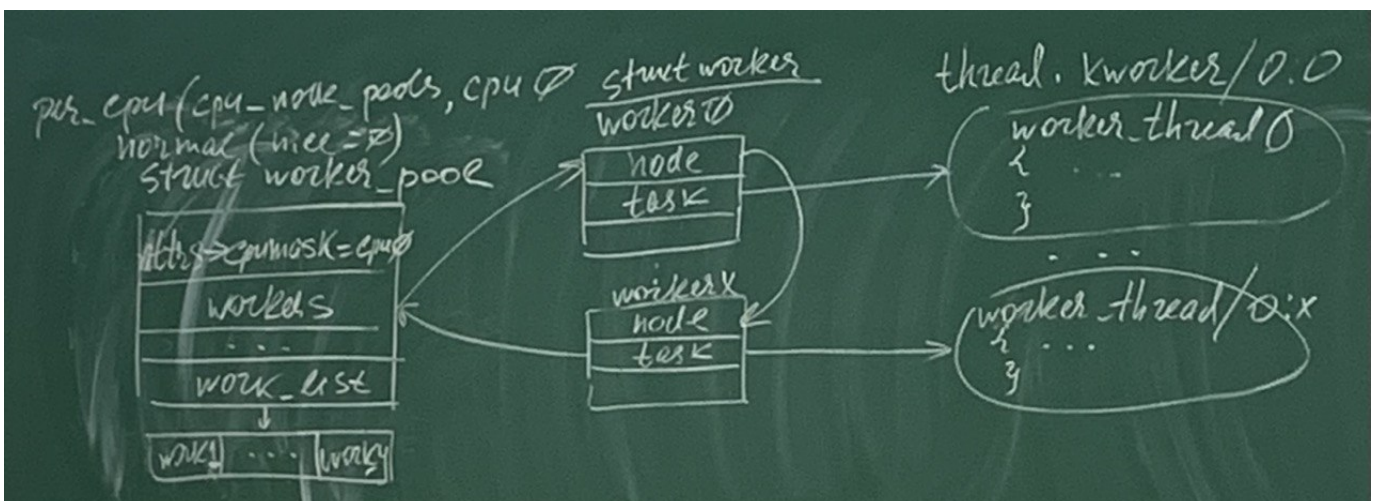
Флаг WQ_MAX_ACTIVE имеет смысл только для привязанных очередей. С этим флагом очереди могут потреблять большое кол-во процессорного времени.

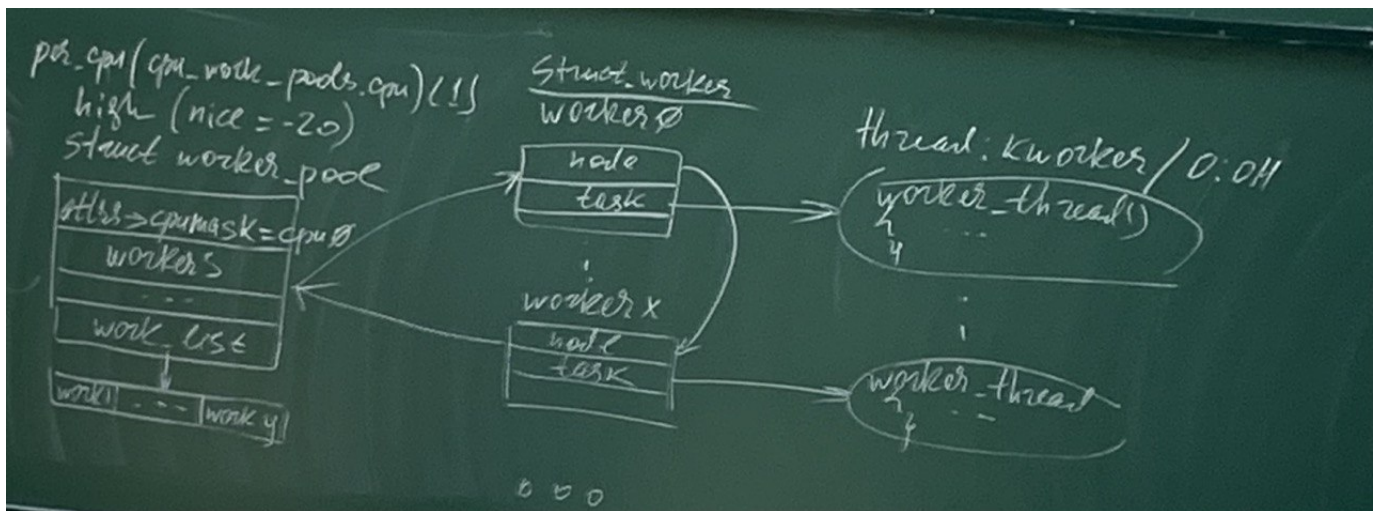
В соответствии с этим флагом (см. прошлую лекцию) очереди делятся на привязанные (normal) и непривязанные (unbound). Для нормальных очередей характерно то, что они выполняются на том же ядре, на котором выполнялось аппаратное прерывание, также как и тасклет. Это полезно для системы, т.к. в этом случае более оптимально использование кэша, но с другой стороны привязанные очереди повышают энергопотребление.

Флаг FREEZABLE: свойство очередей, их важное отличие от тасклетов, они могут переходить в состояние блокировки. Если в тасклете реализуется доступ к критическим секциям, там могут использоваться только спинлоки, а очереди работ могут блокироваться.

Для того, чтобы свести воедино все перечисленные объекты, рассмотрим следующую схему.

На каждое ядро имеется 2 типа workerpool: нормального (normal) и высокого (high) приоритета.





Ставим многоточие для каждого процессора на схеме.

Если ставится задача обработки информации в ядре, то ядро предоставляет механизм очередей работ, при этом в современных ПК это потоки ядра. В структуре есть поле task, речь идет о thread. Это особенность UNIX/Linux: не различаются процессы и потоки. В ядре есть функция clone(), по современным представлениям у нас многопоточное программирование, но UNIX изначально писался для параллельного программирования. Мы это прорабатывали, используя fork() и create_thread().

Для процесса, созданного с помощью fork, характерно все то, что характерно процессу. Именно поэтому называется, что в Linux существует task_struct, будь то процесс или поток. Разница может заключаться в том, что потоки могут не все наследовать, для этого надо указывать соотв. флаги. Нет смысла наследовать все для потока от процесса.

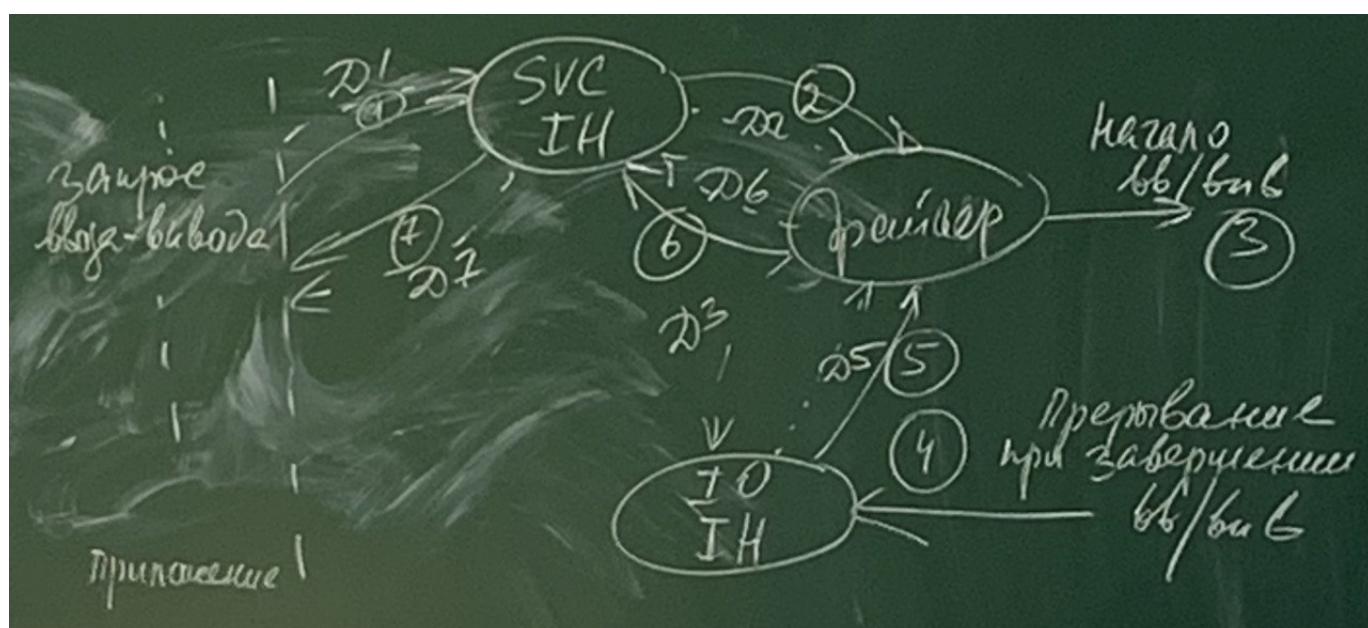
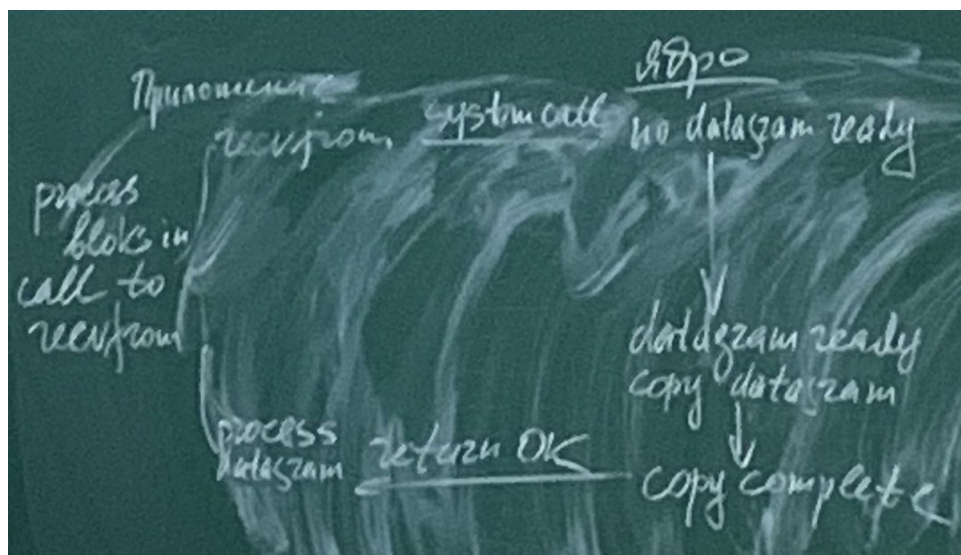
16.2 5 МОДЕЛЕЙ ВВОДА-ВЫВОДА

Блокирующий ввод-вывод (синхронный)

Это то, что мы изучали ранее все время. Асинхронный ввод-вывод невозможен для обычных файлов, т.е. на обычных файлах реализовывать соотв. флаги нельзя, т.е. при работе с обычными файлами всегда будут блокировки.

Блокировки - зло, они занимают время, причем это время нельзя предсказать, они случайные, и проблема очевидная: если не блокироваться, то может возникнуть ситуация, как здесь нарисовано - datagram not ready (данные не готовы) - и тогда что мы считаем? В этом случае системный вызов должен вернуть ошибку, а процесс должен снова запросить данные. Здесь это невозможно. Не рекомендуется делать блокировки там, где это не необходимо.

Также приведена картинка из книги Шоу - Логическое проектирование ОС. Древняя диаграмма, она очень полезная, показывающая, что когда приложение запрашивает ввод/вывод, оно блокируется.

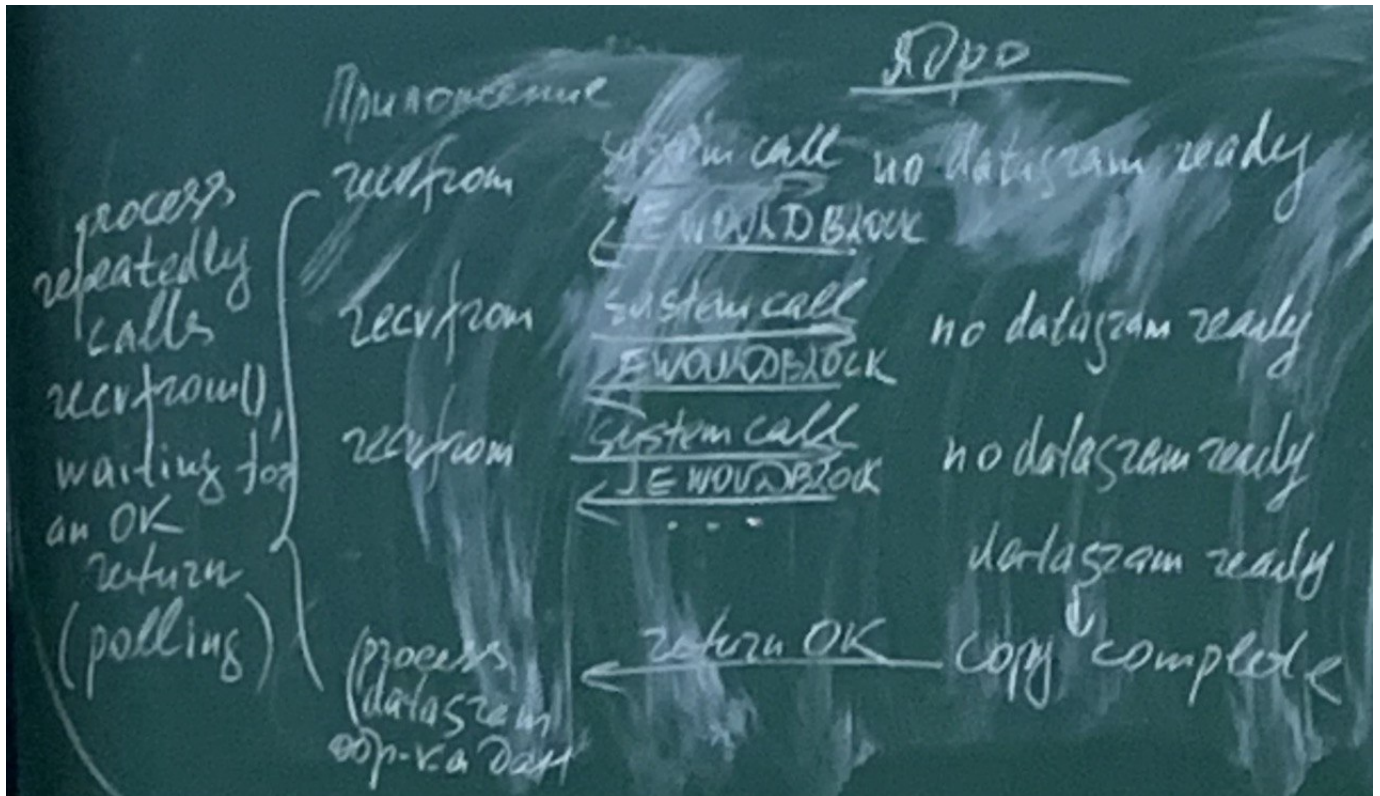


Процессор не управляет операцией ввода-вывода, он выключается и переходит на выполнение другой работы именно в рамках архитектуры, в которой реализовано распараллеливание функций. Соответственно, запросив ввод-вывод, приложение блокируется и оно пробуждается, когда внешнее устройство завершает операцию ввода-вывода, формирует сигнал прерывания, который приходит на контроллер прерывания и этот сигнал с контроллера прерывания в самой простейшей схеме поступает на выделенную ножку процессора в конце выполнения каждой команды процессор проверяет наличие сигнала прерывания на этой ножке, если он пришел - процессор переходит на обработку, для этого он должен адресовать прерывание, для этого в системе есть IDT.

Даже если при выполнении операции ввода-вывода не удалось прочитать или записать данные, все равно приложение получит сообщение об ошибке. В любом случае приложение получит некоторую информацию.

НЕБЛОКИРУЮЩИЙ ВВОД-ВЫВОД (POOLING, ОПРОС)

Везде используем `recvfrom()`, а не `read/write`, это показатель того, что речь идет не об обычных файлах, например может идти о пайпах, передаче сообщений, о сокетах.



В данном случае мы видим, что процесс не блокируется, но постоянно запрашивает данные. Получив ошибку, что данные не готовы, он опять выполняет запрос. Это крайне затратная схема. Процессор контролирует процесс ввода-вывода, постоянно опрашивая готовность внешнего устройства, соотв. готовность устанавливается устройством с помощью флагов, т.е. процессор постоянно опрашивает флаги готовности внешнего устройства.

Когда устройство готово, данные аппаратным прерыванием будут записаны в буфер ядра, а потом каким-то отложенным действием они будут в дальнейшем дообработаны и доведены до приложения.

Этот опрос - самая первая схема, которая была реализована в вычислительных машинах. Прерывания полноценно появились лишь в 3 поколении ЭВМ, когда была реализована идея распараллеливания функций. Раз процессор отключается от управления внешним устройством, его нужно проинформировать, что операция ввода-вывода завершена.

Для этой модели характерны большие накладные расходы.

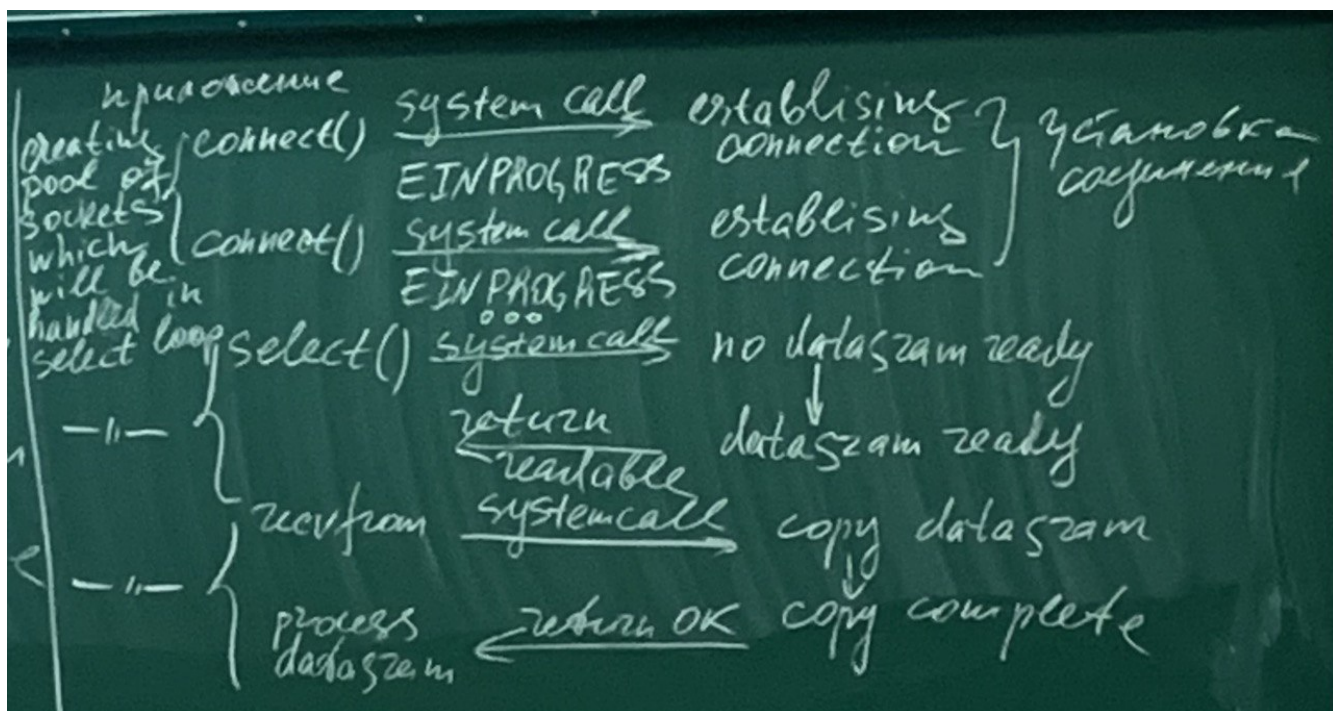
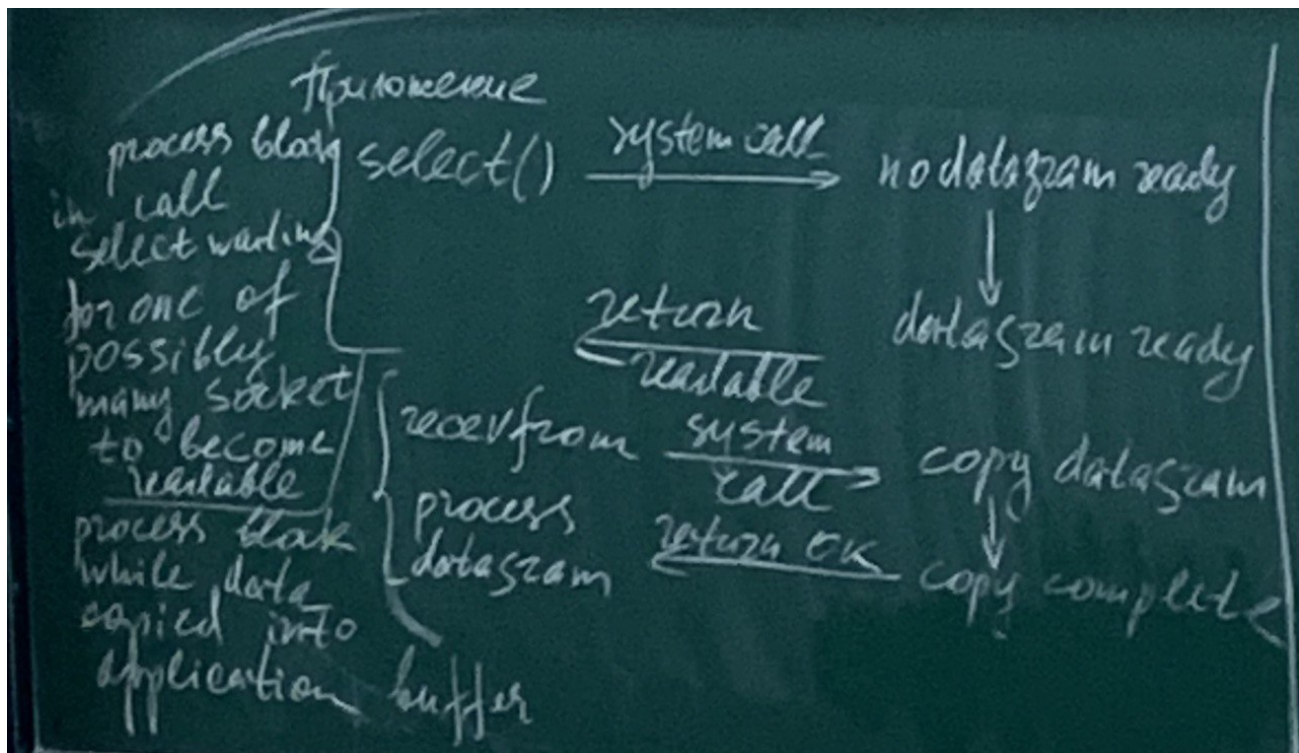
МУЛЬТИПЛЕКСИРОВАНИЕ ВВОДА-ВЫВОДА

При реализации этой модели используется один из мультиплексоров: `select`, `pselect`, `poll`, `epoll`.

Пишут, что select неэффективен, pselect практически как select, более эффективным мультиплексором является poll и его более современная версия epoll.

Мультиплексер (коммутатор) - устройство, которое объединяет информацию, поступающую по нескольким каналам ввода, и выдает ее по одному выходному каналу. Процесс совмещения нескольких сообщений, передаваемых одновременно, в одной физической или логической среде.

Существует 2 основных вида мультиплексирования: временное и частотное.



Более подробная диаграмма позволяет получить более детальную информацию о том, что происходит.

Приложение-клиент обращается к сокету, для этого вызывает системный вызов connect, чтобы установить связь с приложением-сервером. Здесь соединены запросы клиентов и работа на стороне сервера. Системный вызов select или poll выполняется на стороне сервера, клиенты устанавливают соединение.

Смысл мультиплексирования: на мультиплексере все равно происходит блокировка, но на вопрос, какая блокировка будет занимать меньше времени: блокировка, где сокеты опрашиваются строго по порядку, или когда с помощью соотв. переключателя опрашивается весь пул сокетов и начинает обрабатываться сразу первый готовый сокет.

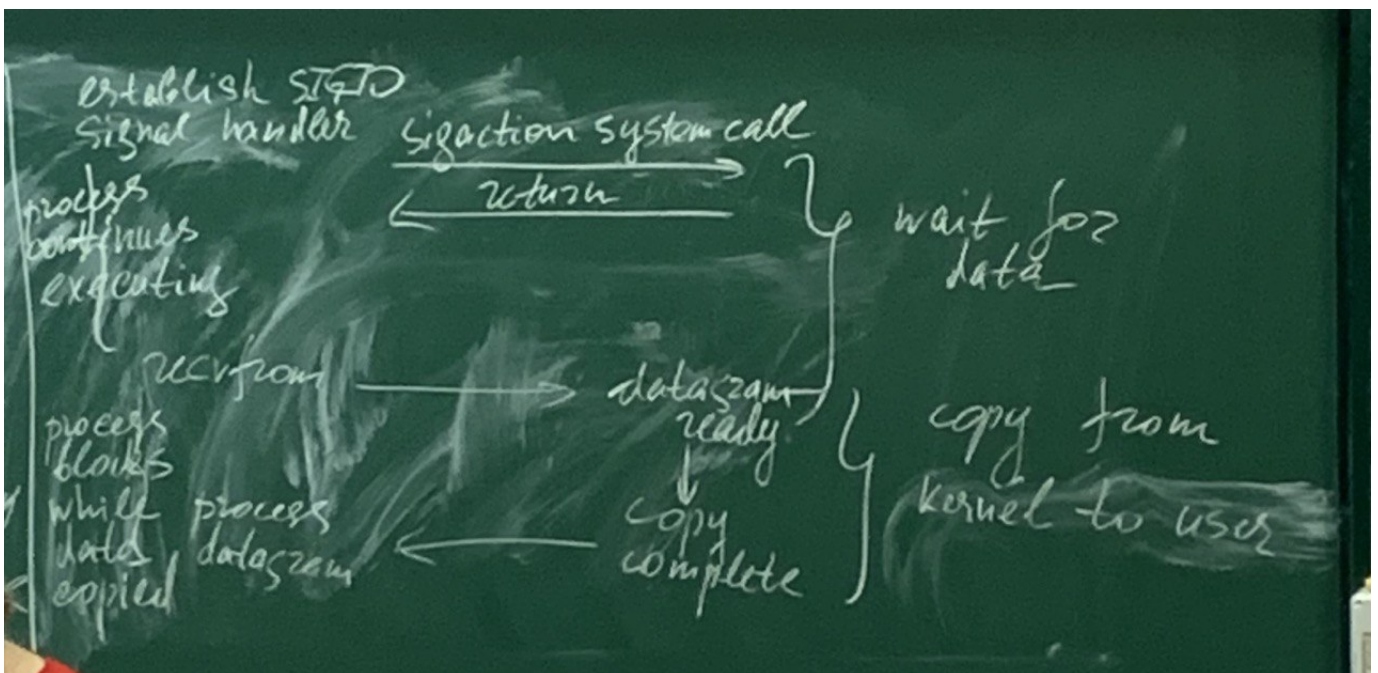
Ведь возможна такая ситуация, когда 1 и 2 сокет не готовы, а 3 готов, но система будет блокирована на 1 сокете, ожидая его готовности. В это время можно было бы начать обрабатывать информацию от другого сокета, для этого надо использовать мультиплексера.

Блокировка с использованием мультиплексера будет занимать меньше времени. Т.е. при мультиплексировании будет обрабатываться первый готовый сокет.

Вариантом реализации такого подхода является многопоточность, но следует иметь в виду, что в Linux очень «дорогие» потоки, потому что при создании UNIX была создана система, в которой реализована идея fork, создания дочерних процессов, которая оказалась крайне эффективной, и вроде как потоки не внесли новые рассуждения. В UNIX ничего не стоит создать параллельные процессы. Игра идет на флагах, которые устанавливают уровень наследования.

Python: GIL (Global Interpreter Lock), в каждом процессе может выполняться только один поток. Это связано с тем, что в питоне используется GIL - глобальная блокировка, которая накладывает ограничения на потоки, т.е. нельзя одновременно использовать несколько процессов.

ВВОД-ВЫВОД, УПРАВЛЯЕМЫЙ СИГНАЛАМИ (SIGNAL-DRIVEN IO)



Сигнал SIG_IO должен быть определен в системе, соотв. функция sigaction может установить собственный обработчик этого сигнала или можно использовать обработчик по умолчанию.

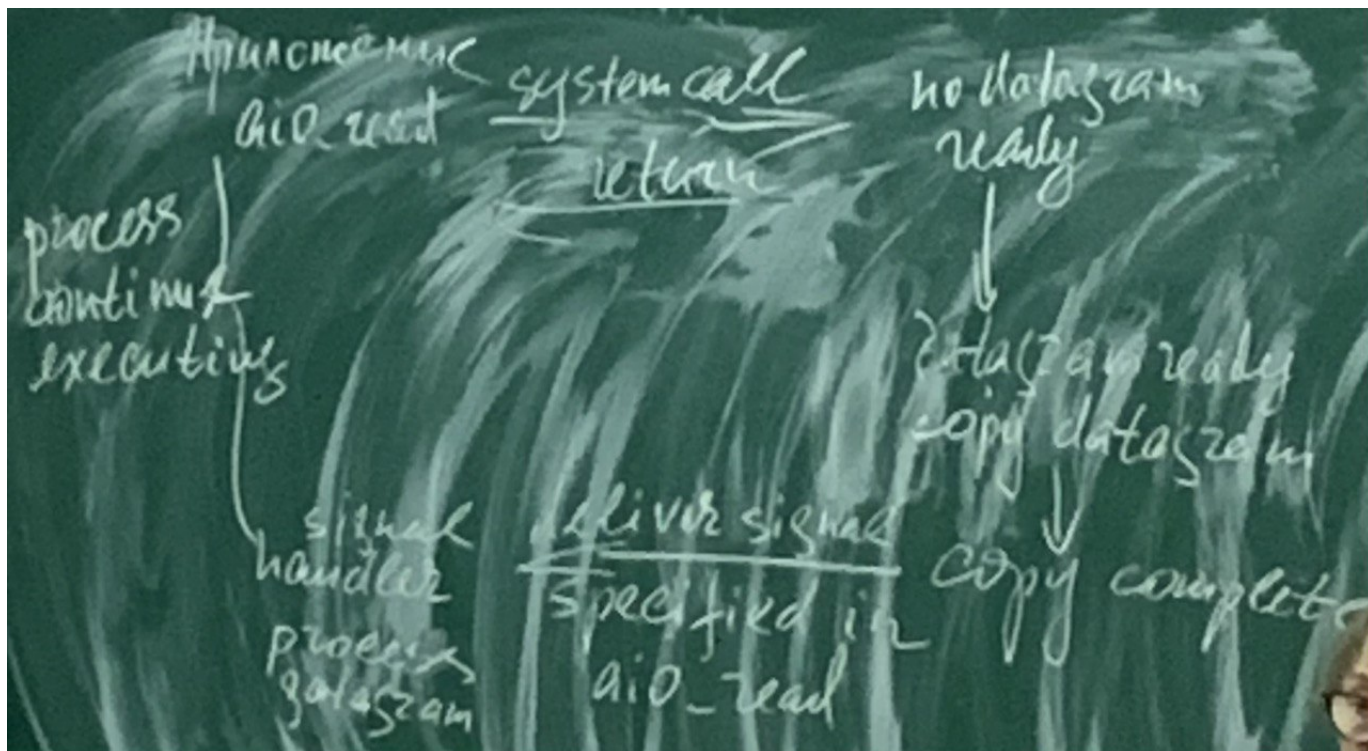
Это уже асинхронный ввод-вывод, и процесс продолжает выполняться. Он блокируется только для того, чтобы дождаться получения данных.

Для того, чтобы реализовать такой асинхронный ввод-вывод, ядро должно взять на себя всю работу, у сигнала SIG_UO есть обработчик, который ждет возникновения сигнала, работу по посылке этого сигнала выполняет ядро, которое отслеживает готовность данных. Когда данные готовы, ядро пошлет сигнал SIG_IO, в результате будет вызван обработчик этого сигнала, при этом функцию `resfrom` можно вызывать либо в обработчике сигнала, либо в основном потоке приложения.

Сигнал типа SIG_IO для каждого процесса может быть только один. В результате, используя сигнал SIG_IO, можно работать только с одним файловым дескриптором.

АСИНХРОННЫЙ ВВОД-ВЫВОД, ИСПОЛЬЗУЮЩИЙ СПЕЦИАЛЬНЫЕ КОМАНДЫ, КОТОРЫЕ НАЗЫВАЮТСЯ

Для такого ввода-вывода используются специальные команды, которые называются `aio_read`, `aio_write`, и тд.



Проблема асинхронного ввода: определить, что может делать приложение, не получив данные? Время выполнения действий соотв. приложением будет меньше, т.е. отзывчивость такой системы может быть меньше при правильном написании.

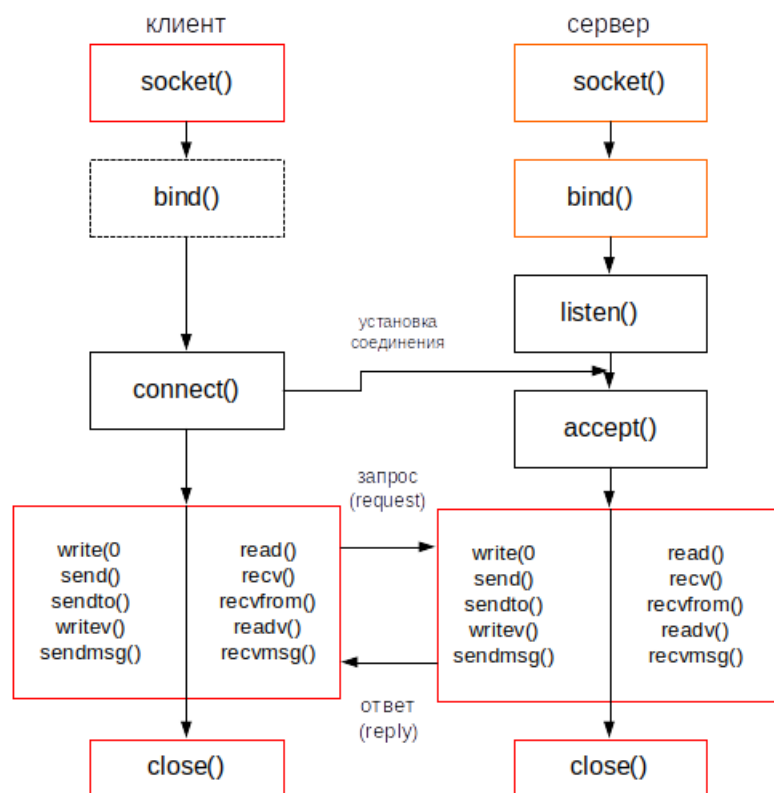
17. СЕМИНАР 2 ИЮНЯ 2022

17.1 СОКЕТЫ В ФАЙЛОВОМ ПРОСТРАНСТВЕ ИМЕН

Название темы подчеркивает особенности этих сокетов. Именно такие сокет мы можем увидеть в файловой подсистеме.

Что значит увидеть сокет в файловой подсистеме? С помощью команды `ls -ial`, увидим, что у него и `inode` есть, он помечен как специальный файл(s). Механизм передачи - *скорее всего* буфер, который определен в файловой подсистеме. Второй тип сокетов - `AF_INET` (сетевые).

Все сокет (AF_UNIX, AF_INET) взаимодействуют по модели клиент-сервер: это означает, что есть сервер, по своему названию это сервис, т.е. он предоставляет обслуживание, а клиенты запрашивают это обслуживание, передавая серверу запросы в сообщениях, т.е. взаимодействие клиент-сервер предполагает как получение сообщений сервером от клиентов, так и посылка сервером ответов клиентам.



Обычно такое взаимодействие выполняется по некоторому протоколу. В сокетах `AF_UNIX` взаимодействие ведется через специальный файл, там протокол `UDP DEGRAM`.

Протокол - некоторое соглашение, определяющее последовательность действий при взаимодействии, например сервер, получив сообщение, может послать сообщение-подтверждение.

Наша задача - реализовать взаимодействие процессов на отдельно стоящей машине и в распределенной системе. Система с сокетами позволяет моделировать взаимодействие процессов в распределенной системе на отдельно стоящей машине, и мы этим будем пользоваться в нашей ЛР.

В своей ЛР необходимо запустить несколько процессов-клиентов, и это не fork. Не меньше 3-х клиентов, которые посылают сообщения, например свой ID серверу. Сервер, получив ID, например, выводит его на экран. Нужно ответить еще что-то клиенту. Важно продемонстрировать параллельность, поэтому запускаем несколько клиентов. Работа ЛР - установление связи и обеспечение обмена сообщениями.

На сокетах определен т.н. сетевой стек. Поскольку у нас две стороны (клиент и сервер), на каждой стороне вызываются свои системные вызовы.

Что-то про стрелочку возле accept. Block until connection from client (accept).

На стороне сервера будет выполняться больше действий. Соединение устанавливается клиентом вызовом connect.

Вызов bind() используется для назначения сокету локального адреса. Для сокета интернета этот адрес состоит из ip-адреса сетевого интерфейса локальной системы и номера порта. Клиенты могут не вызывать bind(), т.к. их точный адрес часто не играет никакой роли, при этом адрес назначается им автоматически.

Важный момент: здесь указывается общая структура sockaddr. Поскольку сокеты декларированы как универсальное средство взаимодействия параллельных процессов, работа на одной локальной машине и работа в сети выполняется именно через абстракции конечных точек соединения. На сокетах определено несколько доменов, типов и протоколов, и поэтому на сокетах в самом общем виде определена структура sockaddr.

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3
4 int bind(int sockfd, struct sockaddr *addr, int addrlen);
5
6 int listen(int sockfd, int backlog);
7
8 int connect(int sockfd, struct sockaddr *srvr_addr, int addrlen);
9
10 int accept(int sockfd, struct sockaddr *cli_addr, int addrlen);
```

Для того, чтобы можно было обеспечивать взаимодействие параллельных процессов в распределенной системе, определена структура sockaddr_in, но они сопоставлены друг с другом. И bind, и connect, и accept будут использовать sockaddr, но в программе, если это сетевой сокет, будут инициализироваться поля структуры sockaddr_in, соотв. функция listen используется сервером чтобы информировать ОС, что процесс-сервер переходит в состояние пассивного ожидания. Соотв. в этом системном вызове 2 параметра - файловый дескриптор, который возвращает socket(), и кол-во прослушиваемых соединений.

Вызов функции listen() имеет смысл только для соединений типа STREAM для протокола TCP.

Функция connect() создает активное соединения клиента с сервером. Должен быть указан адрес сервера.

Вызов `accept()` - важное действие, соединение может быть принято, если поступил запрос на соединение, иначе принимать нечего. `Block until connection from client` - процесс будет блокирован на `accept()`, т.е. очень большую работу выполняет ядро системы.

Функция `accept()` используется сервером для принятия соединения при условии, что ранее сервер получил запрос соединения. В противном случае `accept()` блокируется до получения запроса соединения. При поступлении запроса на соединение, `accept()` начинает выполняться и создает копию исходного сокета, т.е. получит дескриптор сокета, который был создан функцией `socket()`, а вернет дескриптор нового сокета, который является копией исходного, при этом исходный сокет останется в состоянии `listen` (будет продолжать прослушивать возникающие соединения), а новый сокет будет находиться в состоянии `connected` (`struct socket.state`).

UNIX/Linux имеют очень тяжелые потоки. Все создается внутри ядра функцией `clone()`, все зависит от наследования, но между процессами и потоками имеется очень существенная разница - поток выполняется в адресном пространстве процесса, но для UNIX эта декларация весьма условна: системный вызов `fork()` создает копию процесса-предка, в том смысле, что процесс-потомок наследует код предка.

Мультиплексирование - альтернатива запуска на стороне сервера потоков, каждый из которых обрабатывал бы отдельное соединение.

Пример из книги Стивена, глава 5:

```
1 int main(int argc, char *argv[])
2 {
3     int listenfd, connfd;
4     pid_t childpid;
5     socklen_t clilen;
6     struct sockaddr_in clientaddr, servaddr;
7     listenfd = socket(AF_INET, SOCK_STREAM, 0);
8     ...
9 }
```

Все системные вызовы нужно проверять на -1.

Адрес апорта получаем из командной строке при запуске на выполнение сервера, но можно присвоить какое-то значение. Номер порта может быть любой. *Порт - это адрес.*

В наших системах 2 пересекающихся адресных пространства: АП физической памяти и АП портов ввода-вывода, при этом размер АП портов ввода-вывода ограничен 64 Кб. Эти АП пересекаются, потому что оба АП начинаются с нулевого адреса.

А поскольку в системе есть физическая память и есть внешние устройства, которые адресуются через порты, возникает неоднозначность адреса. Чей это адрес - памяти или порта? Поэтому чтобы различать, чей это адрес, исп. т.н. `path_mustering`, необходимая составляющая правильной работы ПК (южный и северной мосты).

Многие технологии реализуются программно.

АП портов ввода-вывода не превышает 64Кб - использует *host to network short*.

Адрес сокета, номер порта `serv_port` - 9877, заранее известен, причем номер порта берется

больше, чем 1023, т.к. в этом случае зарезервированный код (порт?) не нужен, порт 9877 для избежания конфликта с динамическими портами. пассивное прослушивание - основную работу на себя берет система. Сервер блокируется на вызове ассерт в ожидании соединения с клиентом. Для каждого клиента функция `fork` создает процесс-потомок для обслуживания клиента, при этом процесс-потомок закрывает прослушивающий сокет, а в процессе-предке сокет остается прослушивающим. Так можно сделать, потому что это другой процесс. Затем вызывается функция `str_echo` для обработки запроса клиента. Можно сделать на коленке: `recvfrom`, `sendto`.

Пример - показать, альтернативой чему является мультиплексирование. Каждое соединение обслуживается отдельным процессом.

17.2 МУЛЬТИПЛЕКСИРОВАНИЕ ВВОДА-ВЫВОДА

2 основных мультиплексера. Родным для UNIX BDS является `select`, для `system5` - `multiplexing pool`. Есть `select`, есть `pselect`, но если последний параметр (`sigmask`) `pselect` установить в `NULL`, будет тот же самый `select`. Очень хорошо описан `select`, `pool` описан хуже, но он лучше. Каждый из мультиплексеров определен в системе.

Рассмотрим пример: сервер обслуживает какое-то кол-во соединений и ожидает сообщение от любого соединения. В этом случае нельзя использовать блокирующую операцию чтения, т.к. чтение одного дескриптора приведет к блокировке программы, в ожидании когда данные смогут быть прочитаны, в это же время данные могут появиться на других соединениях (на другом дескрипторе).

Для решения этой проблемы используется мультиплексирование ввода-вывода. Для этого необходимо создать список (массив) дескрипторов и вызвать мультиплексер, который мультиплексирует соединения (мультиплексер блокирует процесс до появления *любого* первого соединения) и вернет управление. При выходе из функции мультиплексирования сервер получит информацию о готовых к вводу-выводу дескрипторах.

Мультиплексер `select` определен в `posix.1`, поэтому он лучше описан.

Если мы работаем с мультиплексером, необходимо объявить массив дескрипторов, потому что этот массив будет заполнен и в последствии работа будет выполняться с массивом дескрипторов.

```
1 int select(int h, fd_set *readfds, fd_set *write_fds, fd_set *exceptfds ,  
    struct timeval *timeout);  
2  
3 //functions defined on SELECT  
4 FD_CLR(int fd, fd_set *set);  
5  
6 FD_ISSET(int fd, fd_set *set); // do descr is a part of select?
```

Выставляется `timeout`, чтобы не ждать бесконечно.

18. ЛЕКЦИЯ 4 ИЮНЯ 2022

18.1 СПЕЦИАЛЬНЫЕ ФАЙЛЫ УСТРОЙСТВ.

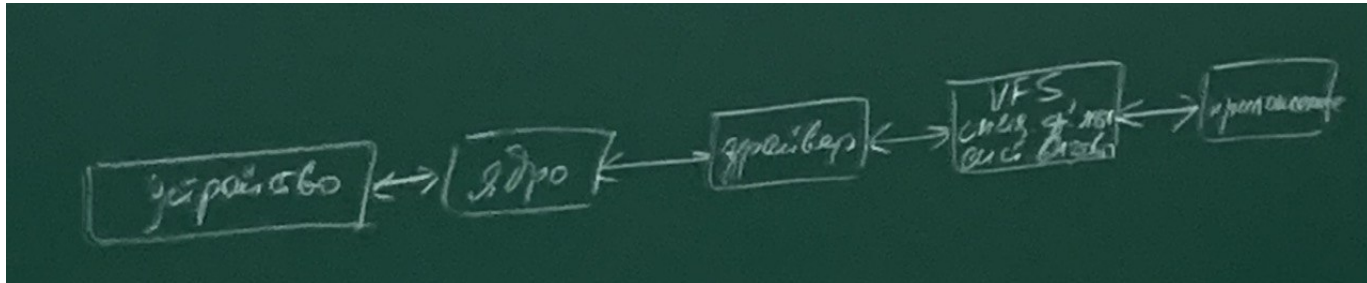
Эти файлы обеспечивают унифицированный доступ к внешним устройствам или периферии. Эти файлы обеспечивают связь файловой системы и драйверов устройств.

Такая интерпретация специальных файлов устройств обеспечивает доступ к внешним устройствам как к обычным файлам. Также как и обычные файлы, файл устройства может быть открыт, закрыт, из него можно читать или в него можно писать.

Каждому внешнему устройству UNIX и Linux ставит в соответствие как минимум один специальный файл. Обычно эти файлы можно увидеть в каталоге `/dev` (первая буква `c` - character, `b` - block) корневой файловой системы. Подкаталог `/dev/fd` содержит файлы с именами `0`, `1`, `2`, но в некоторых системах имеются файлы с именами `/dev/stdin`, и соответственно `/dev/stdout`, `/dev/stderr`.

Система поддерживает 2 типа специальных файлов устройства: *символьный* (небуферизуемый, `non buffered`) и *блочный* (буферизуемый). В UNIX/Linux связь имени специального файла с конкретным внешним устройством обеспечивает `inode`, или индексный дескриптор.

Взаимодействие прикладных программ с аппаратной частью ПК под управлением ОС UNIX/Linux осуществляется по следующей схеме:



Драйвер - часть кода ядра, которая предназначена для управления конкретным устройством. Драйверы в любой системе пишутся по правилам этой системы на основании структур ядра, определенных в системе. Такие структуры перечисляют точки входа в драйвер и другие параметры, т.е. драйвер - не произвольно написанный код, он написан по жестким правилам системы, но задача драйвера - управлять внешним устройством.

Драйвер должен преобразовывать данные поступающие на устройство, получаемые от устройства. Если драйверу нужно передать данные устройству, их нужно передать в формате данных, определенном на устройстве. Когда данные получаются, в итоге формат данных, полученный от устройства, должен быть преобразован в формат, понятный приложению.

В Windows определен т.н. стек драйверов. Внести функциональность в Windows можно только с помощью драйвера. Только разработчик устройства знает все передаваемые параметры.

Драйверы устройств в UNIX/Linux бывают 3 типов:

- о Встроенные драйверы - их выполнение инициализируется при запуске системы. Пример

таких устройств: VGA-контроллер, контроллеры IDE, материнская плата, последовательные и параллельные порты;

- о Драйвера, реализованные как загружаемые модули ядра - часто используются для управления SCSI-адаптерами, звуковыми и сетевыми картами. Файлы модуля ядра располагаются в подкаталогах каталога `/lib/modules`. Обычно при инсталляции системы задается перечень модулей, которые будут автоматически загружаться при этапе загрузки. Список загружаемых модулей хранится в `/etc/modules`, в файле `/etc/modules.conf` находится перечень опций. Для доступа к ним существуют специальные скрипты типа `update-modules`. Для подключения или отключения модулей в работающей системе имеются специальные утилиты (команды) - `lsmod`, `insmod`, `rmmod`, `modprop` (автоматически загружает модули; чтобы отобразить текущую конфигурацию всех модулей нужно воспользоваться командой `modprop -c`);
- о Код драйверов 3 типа поделен между ядром и специальной утилитой, например у драйвера принтера ядро отвечает за взаимодействие с параллельным портом, а формирование управляющих сигналов для принтера осуществляет демон печати `lpd`, который для этого использует специальную программу-фильтр.

Хит-драйвера (хьюман интерфейс) - мышь, клавиатура, обеспечивают взаимодействие с пользователем.

В системе имеются старший (`major`, основной) или младший (`minor`, дополнительный) номера драйвера устройств. Это общий подход к идентификации как символьных, так и блочных устройств. Для примера рассмотрим символьные устройства. Блочными устройствами системы являются внешние запоминающие устройства, остальные - символьные (последовательный ввод).

Рассмотрим пример.

```
1 $ cd dev
2 $ ls -l
3 crw-rw-rw- 1 root root 1, 3 <data> null
4 crw----- 1 root root 10, 1 <data> psaux
5 crw----- 1 root root 4, 1 <data> tty1
6 crw-rw-rw- 1 root tty 4, 64 <data> tty0
7 ...
8 crw-rw-rw- 1 root root 1, 5 <data> zero
```

2 числа через запятую - номера устройств. Традиционно старший и младший номер идентифицируют драйвер, который связан с устройством, например `/dev/null` и `/dev/zero` управляются драйвером 1, а виртуальные консоли и последовательные терминалы управляются драйвером 4.

Современное ядро Linux позволяет множеству драйверов разделять старшие номера, но большинство устройств, которые можно увидеть в `/dev`, все еще организованы по принципу один

старший - один драйвер. Младшие номера используются ядром для определения конкретного устройства.

Внутреннее представление номеров устройств: в ядре существует тип `dev_t`.

Стандарт `posix.1` определяет существование этого типа, но не определяет формат полей. Тип определен в `<linux/types.h>`. Начиная с версии ядра 2.6.0 `dev_t` - 32-разрядное число, в котором 12 бит отведены для старшего номера и 20 для младшего.

Это важно при написании собственного драйвера устройства, при этом код драйвера никак не должен интерпретировать эти значения, он должен их просто использовать. Для этого необходимо использовать набор макросов из `<linux/kdev_t.h>`. Эти макросы позволяют получить старший и младший номера устройств.

```
1 MAJOR(dev_t dev);
2 MINOR(dev_t dev);
```

Также возможны обратные действия - преобразования номеров в `dev_t`:

```
1 MKDEV(int major, int minor)
```

Исходя из формата, можно сказать, что начиная с ядра 2.6, система может поддерживать огромное число номеров устройств, в отличие от ядер предыдущего поколения, которые имели ограничения для старших и младших номеров по 255 штук.

Выделение и освобождение номеров устройств: одно из первых действий драйвера, когда устанавливается символично устройство - получение одного и более номеров устройств.

Для того, чтобы драйвер получил старший и младший номера:

```
1 #include <linux/fs.h>
2
3 int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

Часть `minor` часто устанавливается в 0, но никаких требований нет. `Count` - количество номеров устройств, которые запрашиваются. Если `count` больше, чем диапазон, который запрашивается, то может случиться переход через диапазон на следующий старший номер, но все будет работать нормально до тех пор, пока будет доступен запрашиваемый диапазон номеров. Имя устройства, которое связывается с диапазоном номеров, можно увидеть в `/proc/devices` и `/sys/fs`.

Данная функция хорошо работает, если заранее известно конкретное устройство, которому нужен номер, однако часто неизвестно, какой старший номер использует ваше устройство, поэтому разработчики ядра особо подчеркивают, что для динамического выделения старшего номера нужно использовать следующую функцию:

```
1 int alloc_chrdev_region(dev_t dev, unsigned int firstminor,
2                          unsigned int count, char *name);
```

В этой функции параметр `dev` - только выходной, и он будет содержать первый номер в выделенном диапазоне при успешном завершении вызова функции. `Firstminor` - первый за-

прошенный младший номер, обычно равен 0. Count и name - параметры, аналогичные первой функции.

Несмотря на то, как выполняется распределение номеров устройств в вашем драйвере, необходимо их освободить, когда они больше не используются. Для этого вызывается следующая функция:

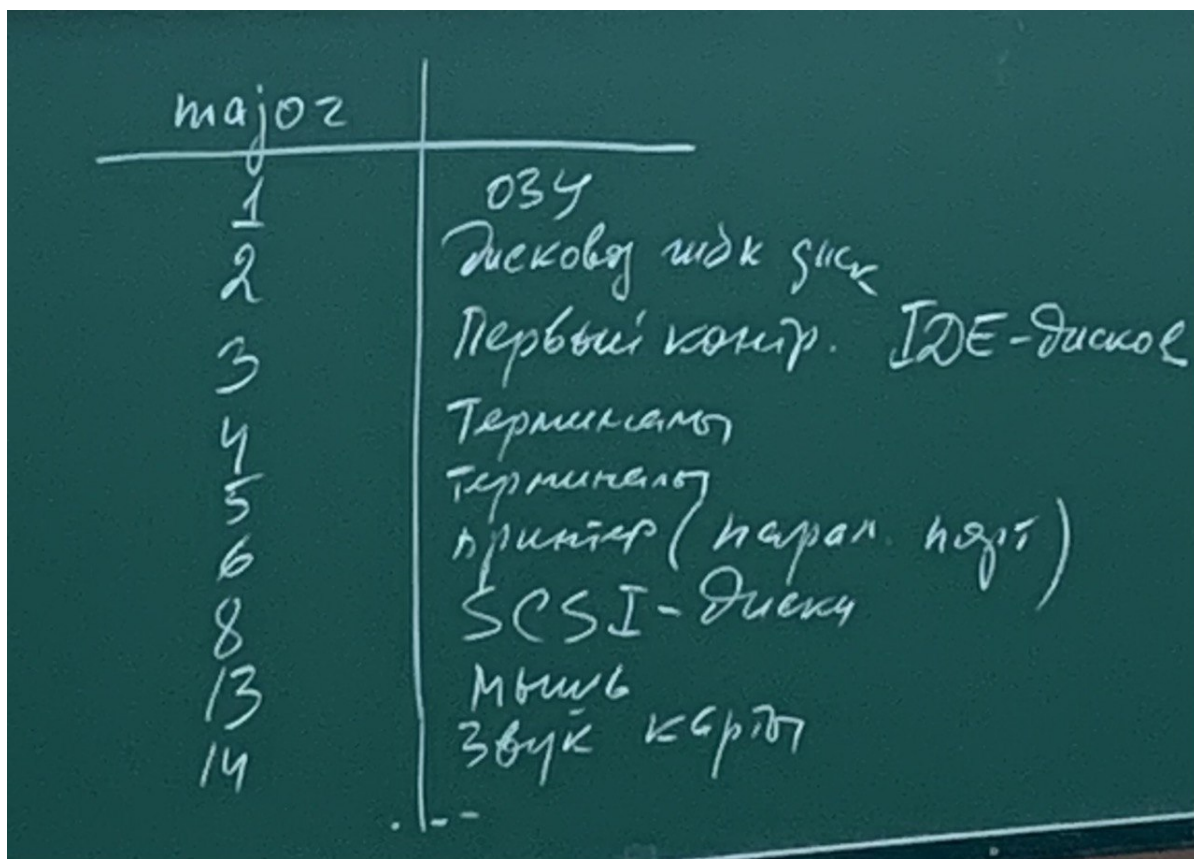
```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Обычно эта функция вызывается exit() или cleanup(), alloc - в init().

Динамическое выделение старшего номера: некоторые старшие номера статически назначаются большинству обычных устройств. Для того, чтобы посмотреть такие старшие номера, уже используемые в текущей реализации Linux, надо посмотреть /proc/devices.

С помощью команды `ls -l /dev/ | grep "c"` мы получим список символьных файлов устройств в системе.

Обычно в системе существует уже хорошо известный набор старших драйверов(?58м):



major	
0	034
1	Дисковая подсистема
2	Первый комп. IDE-диск
3	Терминал
4	Терминал
5	Принтер (парал. порт)
6	SCSI-диск
8	Мышь
13	Звук
14	Кернел

Ориентируясь на выделенные в системе старшие номера, можно выбрать старший номер для своего драйвера. Кроме того, могут быть назначены случайные статические номера, но при этом новые номера не присваиваются. В результате разработчик драйвера должен сделать вывод: можно просто подобрать номер, который кажется неиспользуемым, или выделить старшие номера динамически.

Подобранный номер может работать так долго, как долго вы сами будете пользоваться собственным драйвером. Как только ваш драйвер получит более широкое распространение, случайно выбранный старший номер скорее всего приведет к конфликтам и неисправности. Таким

образом, разработчики ядра настоятельно рекомендуют использовать динамическое получение старших номеров устройства.

У динамического выделения имеется недостаток: нельзя заранее создать узлы устройства, т.к. главный номер будет получен в последствии, т.е. он неизвестен. Существуют рекомендации, одна из них - при загрузке драйвера с динамическим назначением старшего номера вызов `insmod` следует заменить соответствующим скриптом, который после вызова `insmod` будет читать `/proc/devices/`, чтобы создать специальный файл/файлы.

Рекомендуется в скрипте использовать инструмент типа `awk`. Команда `awk` читает документ по строкам и выполняет указанные разработчиком действия, результат выводит на `stdin`.

```
1 $ awk options 'condition{action}' //Example
2 major = $(awk "\\$2 == \"$module\" {print \\$1}")proc/devices.
```

Функция `register_chrdev_region()` используется для статического выделения. Для динамического надо использовать `alloc_chrdev_region()` и соотв. скрипт.

Драйвер устройства содержит такие функции, как `open`, `close`, `read`, `write`, и эти функции должны быть определены для вашего драйвера. Здесь работает все, что было рассмотрено в курсе 2 семестра: миедди, передача данных UK-KU.

Основная часть драйверов в качестве одной из точек входа имеет обработчик аппаратного прерывания. Если нужно дополнительно обработать данные, поступающие в результате работы обработчика прерывания, используются тасклеты и очереди работ.

В ядре имеется набор структур, определенных для работы с внешними устройствами. Основной структурой является `struct device`. Эта структура нижнего уровня, но поля этой структуры инициализируются при включении конкретного устройства. Рассмотрим некоторые поля.

```
1 struct device {
2     struct device *parent;
3     ...
4     const char *init_name;
5     ...
6     struct bus_type *bus;
7     struct device_driver *driver;
8     ...
9     struct dma_coherent_mem *dma_mem;
10    ...
11 };
```

Устройство может не иметь родительского устройства. Такое устройство - `top level`. Первоначальное имя устройства - `init_name`. Для любого устройства важно, к какой шине оно подключается.

`Direct memory access` - способ освобождения процессора от рутинной перекачки данных от устройства в оперативную память. (1.19м)

Установка поля `coherent DMA mapping API` - установка устойчивых соединений многократно используемых буферов. Нет необходимости предварительно выделять буфер `dma`.

dev_t поле для создания в <sys/fs> устройства. u32 id - экземпляр устройства.
driver - основная структура, описывающая драйвер.

```
1 struct driver {
2     const char *name;
3     struct bus_type *bus;
4     struct module *owner;
5     ...
6     int (*probe)(struct device *dev);
7     int (*remove)(struct device *dev);
8     ...
9     int (*resume)(struct device *dev);
10    ...
11 };
```