

Лекция 1

Файловые системы

1.1 Носители

1.1.1 HDD

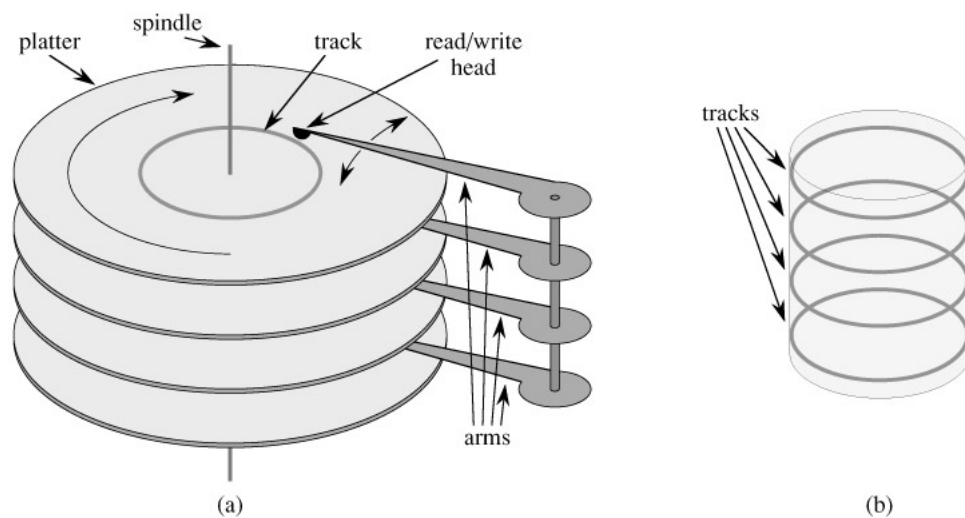


Рис. 1.1: Жесткий диск

- Обороты в минуту(O) — 5400, 7200, 10000, ...
- $\frac{1}{2*O}$ — минимальное время доступа (случайное чтение)
- В мире Unix не существует дефрагментации (ОС должна сама заботиться)
- Время отказа (**MTBF** — min time before failure) — условное количество циклов наработки до отказа
- На **server** — сутки, **desktop** — часы (разница примерно в 3 раза, если одно и то же число циклов)
- Плюсы: стоимость, объем
- Минусы: время доступа, надежность

1.1.2 SSD

- **SATA** и **NVME** — протоколы для дисков
- **NVME** — новомодная штука для **SSD**
- Плюсы: время доступа
- Минусы: надежность, стоимость, объем

1.1.3 Общее

- **IOPS** — input/output operations per second

Показатель применяется для сравнения, например, какого-нибудь **HDD** с **SSD**

- **seek** — рандомное чтение (512 байт)
- Минимум информации: сектор — 512 байт -> 4096 байт
- Чтение одного байта равносильно чтению всего сектора с этим байтом
- Запись одного байта — считать один сектор, заменить байт и записать один сектор
- Аналогия — процессор-память — **cacheline**
(кэшируется линиями, а на диск записывается и считывается секторами)

1.2 Быстродействие

1.2.1 Интересные числа

Числа, которые должен знать каждый программист

| | |
|----------------------------------|--------|
| Cycle | 1 ns |
| Main memory reference | 100 ns |
| Read 4K randomly from SSD | 150 us |
| Read 1 MB sequentially from SSD | 1 ms |
| Disk seek | 10 ms |
| Read 1 MB sequentially from disk | 20 ms |

1.2.2 Выводы для HDD

- Читать нужно последовательно
- Обращения к диску следует минимизировать
- Стоимость доступа сильно дороже передачи данных

1.3 Structure packaging

Сколько будет занимать памяти следующая структура?

hole1.c

```
struct hole {  
    uint64_t a;  
    uint32_t b;  
    uint64_t c;  
    uint32_t d;  
}
```

Ответ: 32 байта, так как *b* и *d* будут выравнены по MAX_ALIGNMENT
Очевидное решение проблемы:

hole2.c

```
struct hole {  
    uint64_t a;  
    uint32_t b;  
    uint32_t d;  
    uint64_t c;  
}
```

Данная структура будет занимать 24 байта на x86_64.

1.4 Алгоритмы элеватора

[Ссылка на презентацию](#)

1. SLIDE 6

Алгоритмы элеватора обрабатывают последовательности запросов к диску (перепорядочивают их)

2. SLIDE 7

FCFS (FIFO)

Самый простой и медленный

3. SLIDE 8-9

SSTF (Shortest Seek Time First)

Сортировка (очередной запрос определяется наименьшим временем seek)

4. SLIDE 10 - ...

Различные способы упорядочивания (**SCAN**)

1.5 Файл

- Абстракция для данных (для Kernel space)
- Последовательность байтов (для User space)
- Формат не определен
- **Unix** — все есть файл (абстракция-интерфейс внутри ядра)
- Типы файлов
 - regular
 - directory
 - symlink
 - socket, fifo
 - character device, block device

1.6 Директория

- Содержит имена находящихся в ней файлов
- `.` — ссылка на текущую
- `..` — ссылка на родителя
- `$ cd` — сменить директорию
- `$ pwd` — текущая директория
- `$ ls` — формирование дерева
- `$ find` — поиск
- *filename vs pathname*: `$ realpath`

1.6.1 Права — просто числа

- `$ view /etc/passwd`
- `$ view /etc/group`
- `$ id` - показывает идентификаторы того, кто ее вызывал
- `$ execute` — search
- `$ read` — directory listing
- `$ write` — changing directory

- Темные директории (переход в директорию внутри директории, для который ты не можешь посмотреть все файлы)
- Права `rx` (read, write, execute)
- **\$ chmod** — меняет права доступа
\$ chmod 123 — 1 - user, 2 - group, 3 - other
- У процесса есть информация о том, кто его запустил
- **SGID** (Set Group ID up on execution)
 Специальный тип прав, который временно выдается запускающему (у него теперь права группы на файл/директорию)

1.6.2 sticky bit

- Изменение поведения при создании нового файла
- `/tmp`
- Создаешь директорию со *sticky bit* и все, кто создают файлы в этой директории имеют на них права

1.7 Иерархия

- /
 - bin/
 - dev/
 - etc/
 - sbin/
 - home/
 - var/
 - usr/
 - * bin/
 - * sbin/
 - tmp

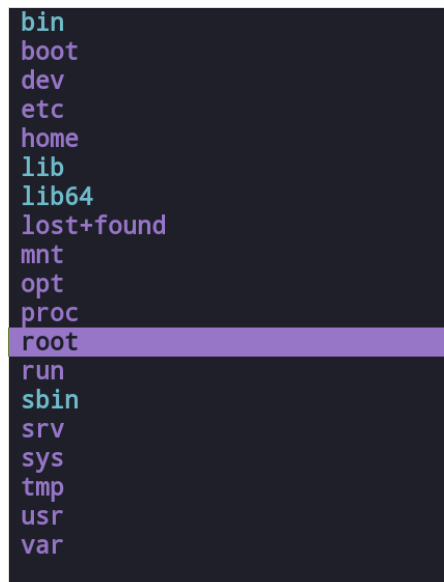


Рис. 1.2: Типичный вид корня в Linux

1.8 Монтирование

- Есть корень и есть узлы, в которые можно монтировать другие файловые системы (часть из них виртуальная)
- **\$ mount**
- Для / обычно используется **ext4** (использует журналирование)
- Для /boot может использоваться **ext2** — так как это более проверено временем (на Ubuntu)
- Файловая система для узла — это не константа, ее можно менять
- **\$ df -h** , **\$ du -hs**

1.9 Inode

- Директория задает mapping имени файла в его inode
- **\$ ln**
- Hardlink — существует в рамках одной файловой системы
- Softlink(symlink) — text string
- **\$ stat** — информация о файле
- *atime* — время последнего доступа
- *ctime* — изменение мета-информации
- *mtime* — изменение содержимого файла
- inode корневой файловой системы фиксирован — 2

1.10 Проход по пути

- Рекурсивный процесс (увеличиваем индекс при проходе в глубину)
- Количество seek по диску зависит от длины пути
- `namei` (name-innode) — lru-cache (файл \leftrightarrow номер inode)

1.11 Атрибуты процесса

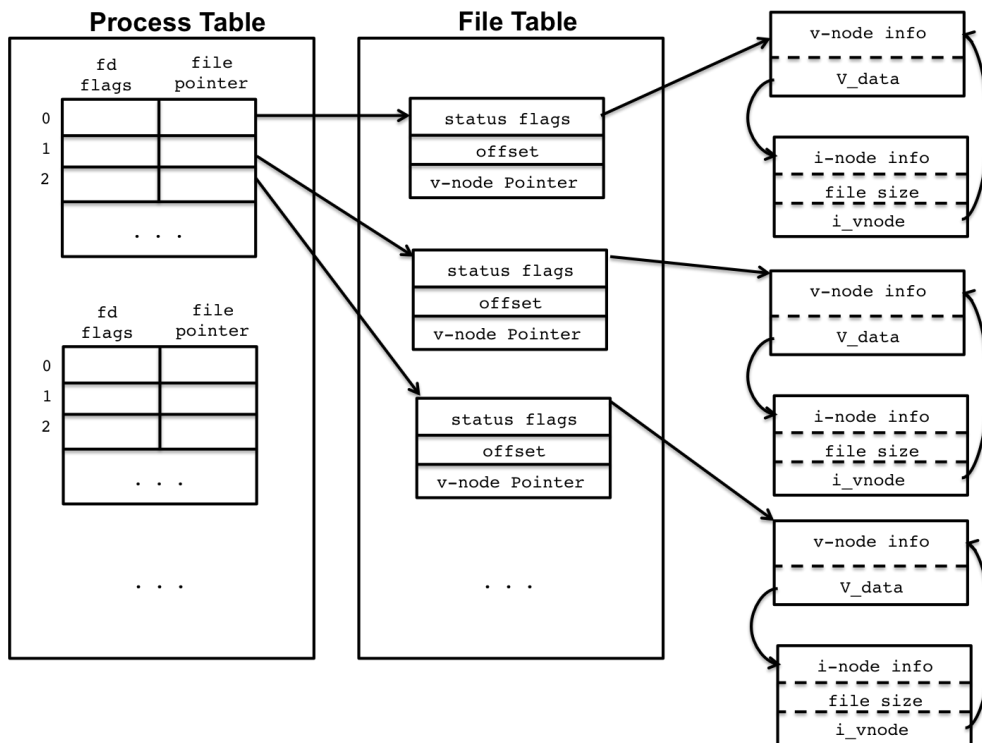


Рис. 1.3: Structures of Kernel

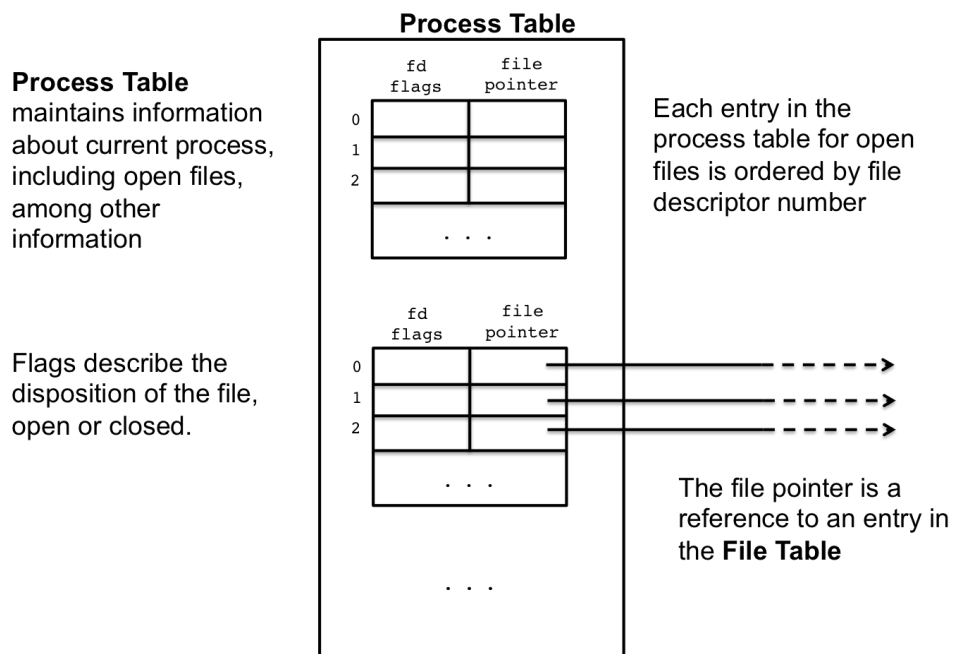


Рис. 1.4: Processes

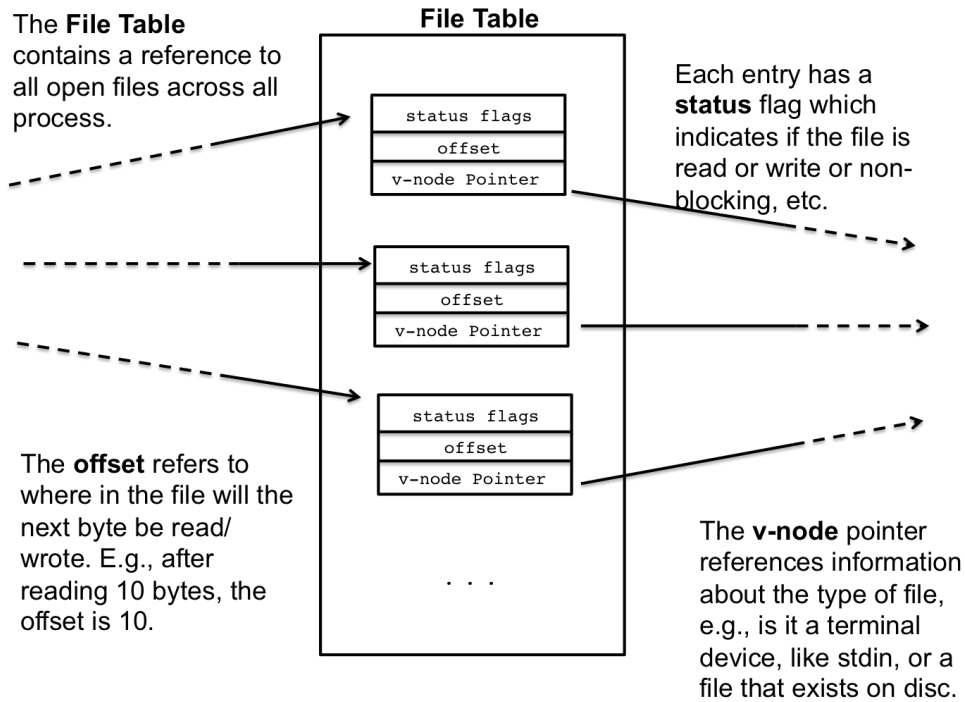


Рис. 1.5: Files

V-node and i-node

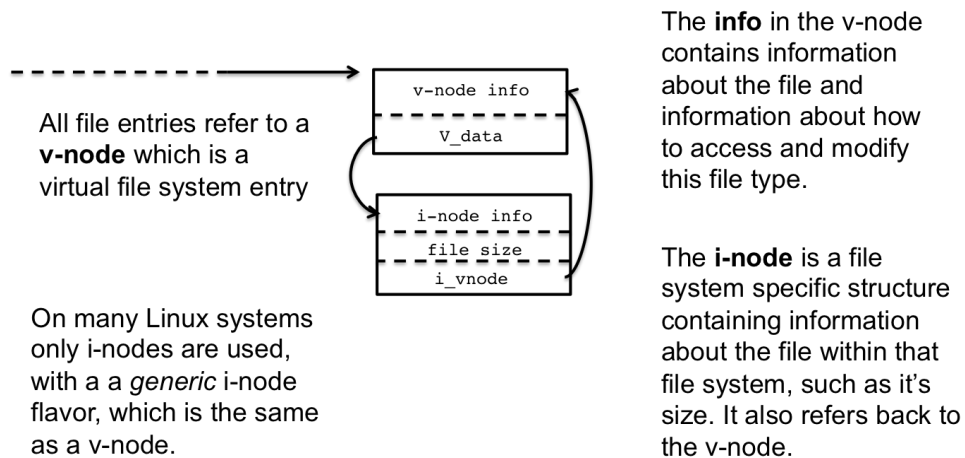
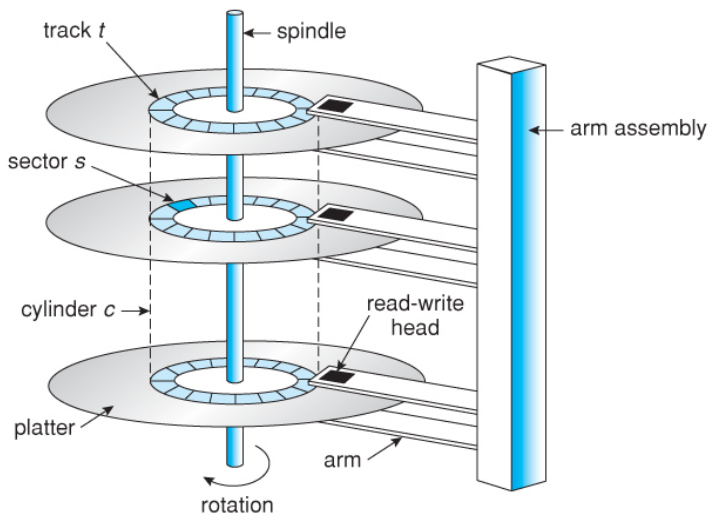


Рис. 1.6: Vnode and Inode

TODO What is this?

1.12 Диски



Устройство диска

- сектор
 - header: метаданные для контроллера диска
 - данные
 - trailer: ECC
- цилиндр
- пластина
- трэк
- шпиндель

Рис. 1.7: Устройство диска

- При записи данных на диск в сектора считаем и записываем **ECC**, при чтении считаем и затем сверяем (пытаемся исправить, если не сошлось)
- **CLV** — Constant Linear Velocity(**CDROM**)
- **CAV** — Constant Angular Velocity(**HDD**)
- На внешних цилиндрах больше секторов, чем на внутренних => чем ближе к центру тем меньше скорость нужна (CD)
- На жестких дисках — постоянная угловая скорость (в центре больше плотность)
- *Partitioning* — разделение диска на несколько логических частей (партиции, на каждой своя файловая система), они трактуются как "отдельные" диски
- Существует другой подход - "собственная" файловая система на "сыром" диске (MySQL)
- Современный контроллер жесткого диска может находить механически поврежденные блоки (bad blocks) и делать remap их на некоторые запасные (sector sparing: replace bad sectors with spare)
- **\$ man 1 badblocks**
- Bootblock — bootstrap program at fixed location
- **MBR** — master boot record — boot code + partition table

1.13 RAID

Redundant Arrays of Independent Disks (Избыточный массив независимых дисков)

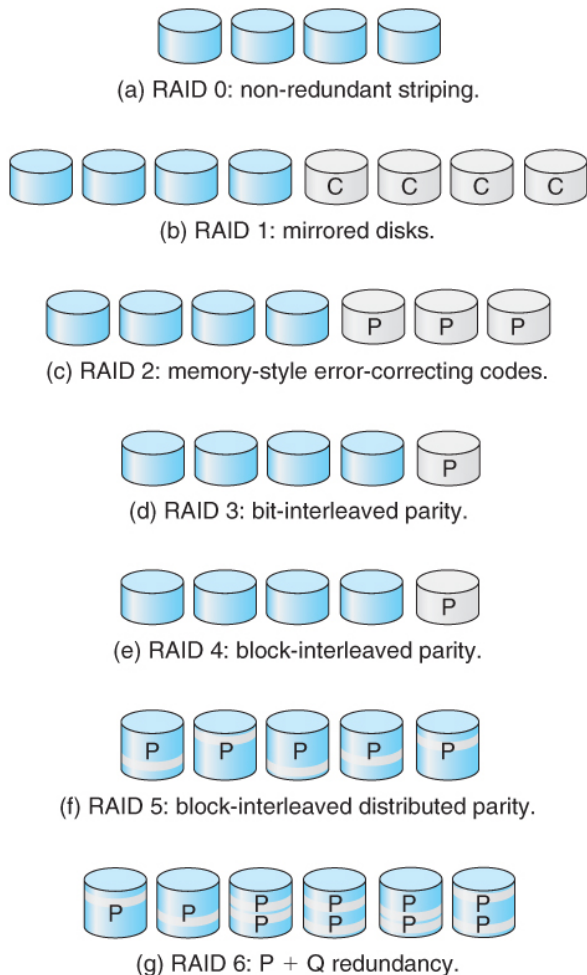


Рис. 1.8: RAID levels

- Reliability (надежность, hacks for more long time of complex usage)
- Performance (striping, суммирование IOPS)
- Levels:
 - **0** — pure striping (1 блок на 1 диске, 2 блок на 2 диске и т.д. — один диск вышел из строя — fail)
 - **1** — pure mirroring (пара дисков, данные продублированы)
 - **0 + 1, 1 + 0**
 - **2, 3, 4, 5** — используются не так часто (хранение доп. данных)
- Rebuild — падает производительность
- Hardware RAID — проблемы: "заложенность" на производителе (vendor lock in), драйвера, как правило, не очень
- Software RAID — гипотетически медленно, но на практике нужная производительность достигается
- У аппаратных RAID — есть батарейка, которая "улучшает" производительность (сначала на батарейку, потом на диск, когда будет удобно)
- **TODO** Байка про SpaceWeb

1.14 Организация файловых систем

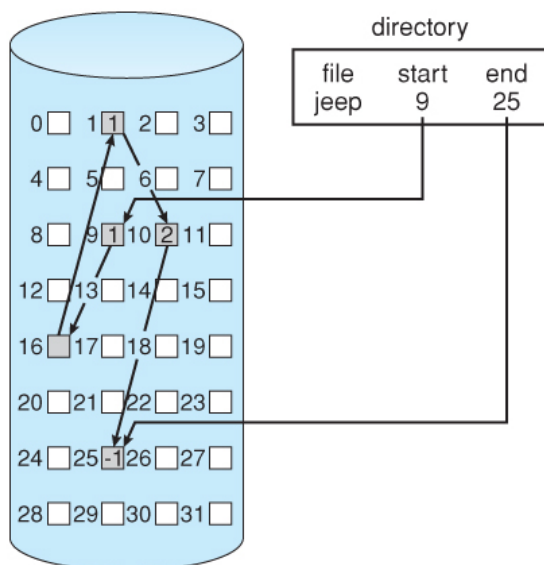
Структура директорий: связный список или хэш-таблица

smart (`$ smartctl`) — оценка диска на практике

Свободные сектора

1. Bit Vector — fast, space usage
2. Список

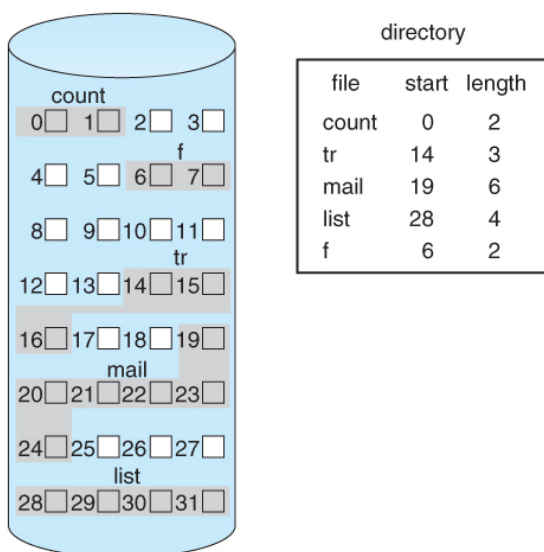
Выделение памяти (allocation)



Линейное

- Объект задается началом и концом (здесь возникают проблемы внешней и внутренней фрагментации)
- Линейное чтение, меньше обращений
- Performance: sequential, random

Рис. 1.9: Linked allocation

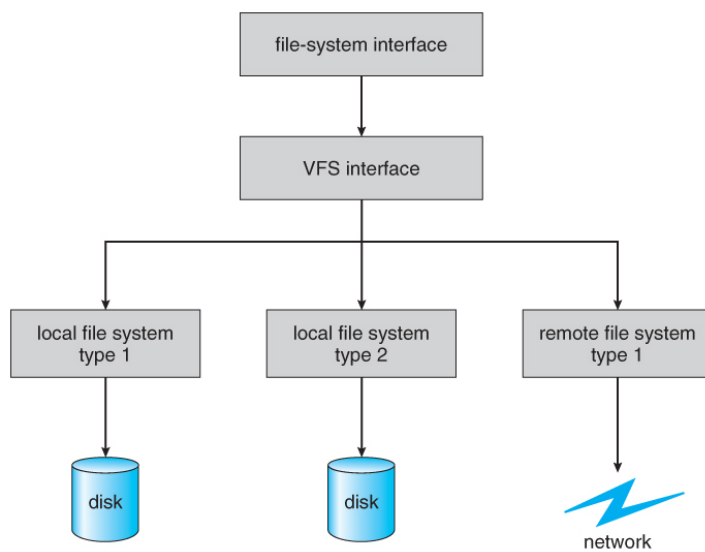


Список

- В каждом "блоке" указатель на следующий
- Плюсы: решает проблему внешней фрагментации
- Минусы: надежность, прыгаем по памяти
- Performance: sequential, awful random

Рис. 1.10: Contiguous allocation

1.15 Файловые системы



- VFS
- Сетевые
- Виртуальные
- На диске
- В памяти

Рис. 1.14: VFS

1.16 Операции с файлами

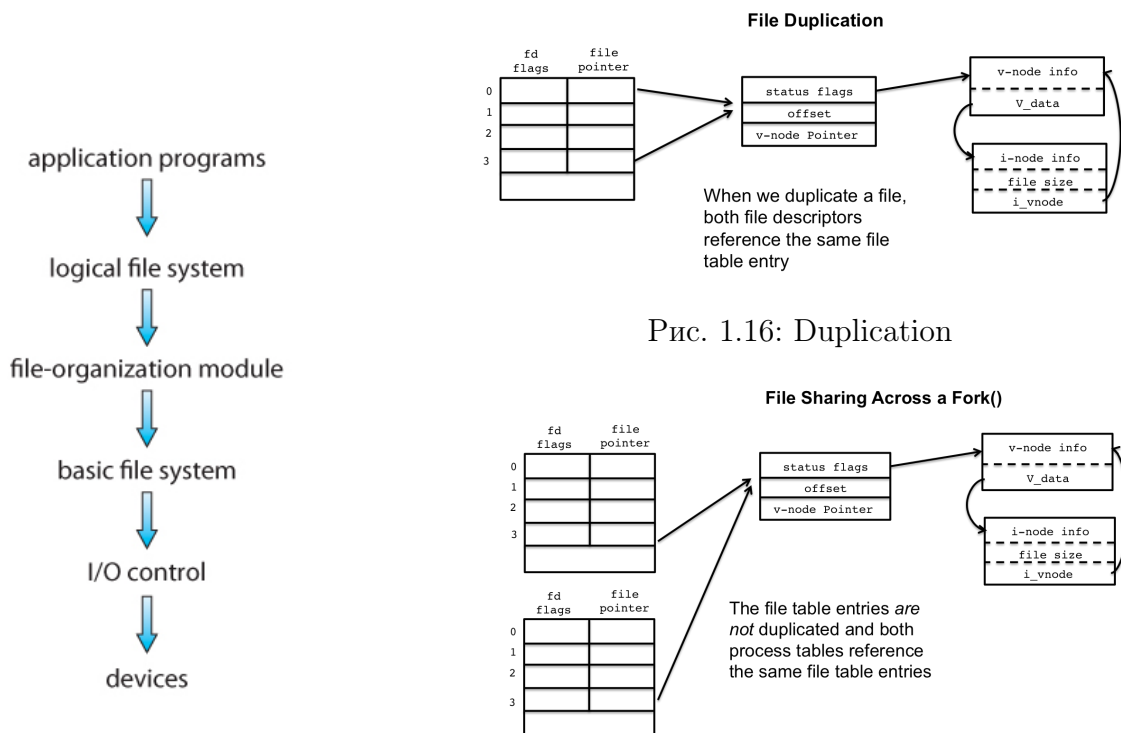


Рис. 1.16: Duplication

Рис. 1.17: Sharing

Рис. 1.15: Layered

1.17 Системные вызовы

1.17.1 Действия с файлами

- Создание, удаление
- Открытие, закрытие
- Позиционирование
- Чтение, запись
- Обрезание
- Блокировки
- Перемещение по файловой иерархии

1.17.2 Общее

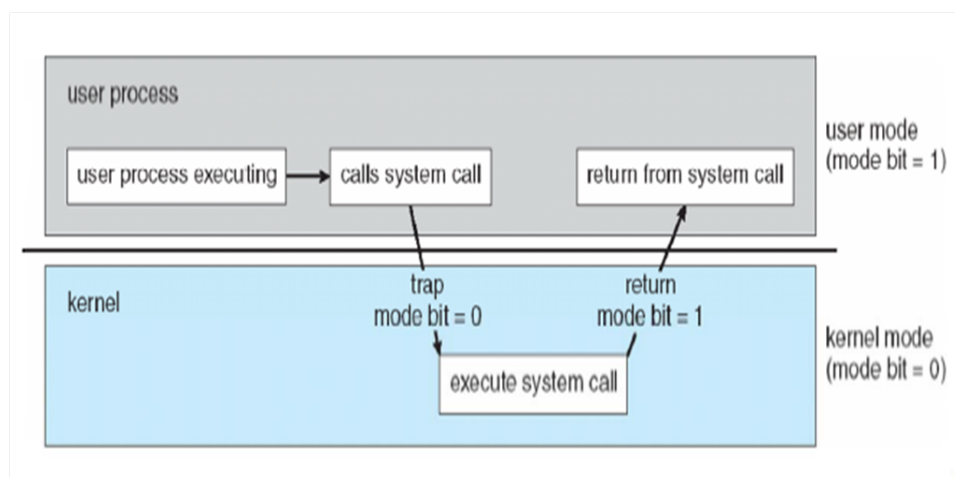


Рис. 1.18: Systemcall

- Дескриптор (например: *stdin*, ...) — интерфейс связи с ресурсом
- Файловый дескриптор — соответственно, связь с файлом
- **POSIX** — стандарт, минимальный интерфейс (**Unix**)
- **POSIX** compatible operating system
- Различные мнемоники для прав доступа (**S_IXOTH**, ...)
- Headers for 'function' in **\$ man 2 'function'** or in **POSIX**

1.17.3 Functions

- *creat()* — create new file or rewrite an existing one
'e' отсутствует, чтобы было быстрее писать

create.c

```
#include <stdio.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

int main() {
    const int rv = creat("/", 777);
    if (rv == -1) {
        printf("Cannot creat(2): %s\n", strerror(errno));
        printf("Cannot creat(2): %m\n");
    }
}
```

- *open()*
Можно открывать файл по умолчанию в двух режимах: **O_APPEND** или **O_TRUNC**
O_EXCL — позволяет сделать создание файла атомарным
- *close()* — закрывает файловый дескриптор
Важно "не терять" файловые дескрипторы и вовремя их закрывать
- *unlink()* — удаляет имя из файловой системы и, возможно, файл, к которому оно прикреплено
- *chdir()*, *getcwd()*
cwd процесса — текущая директория, в которой запущен процесс (resolve relative paths)
- *read()*, *write()*
Нужно писать цикл со счетчиком, для того чтобы записать точное количество информации
TODO iovec
- *lseek()*
- *chmod()*, *chown()* — смена прав файла

- *getdents()*

Лучше использовать С-шный *readdir()*, т.к. в его сигнатуре интерфейс вида **struct linux_dirent**

- *mkdir()*, *rmdir()*
- *mount()*, *umount()*
- *dup()*, *dup2()* — лекция про *IPC*
- scattered, gathered IO

TODO What is this?

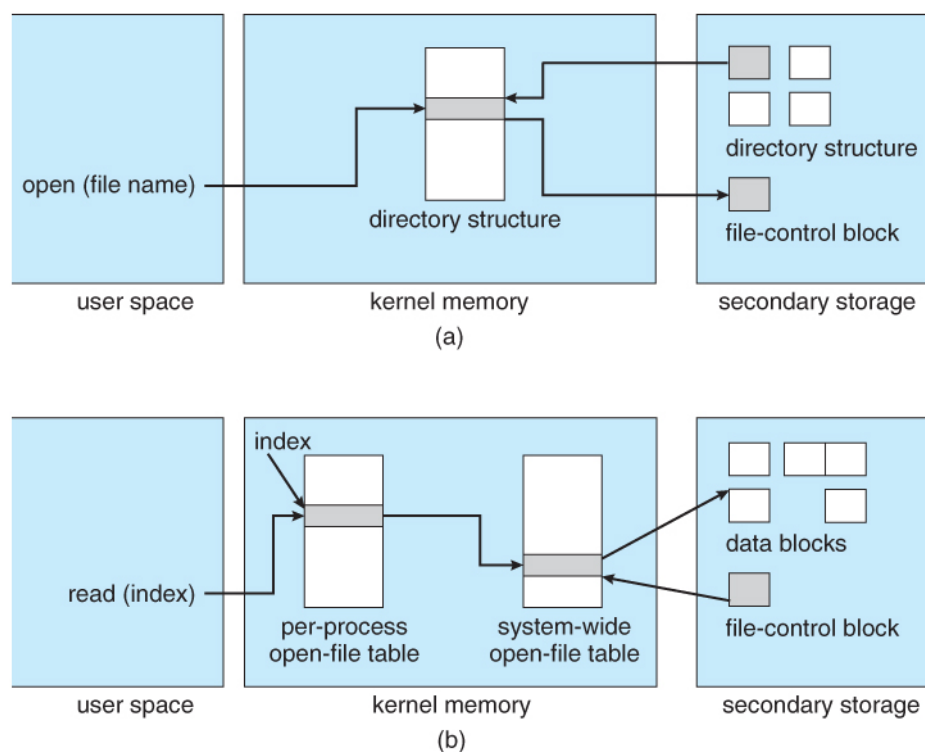


Рис. 1.19: Read() and Open()

1.18 Пару слов о типах

Лучше всего использовать следующие типы данных:

- **off_t**
- **size_t**
- **ssize_t**

TODO Why?

1.19 Common pitfalls

- Неатомарные операции (окно **race**)
- **TOCTOU** (пример **race condition**) — класс багов, связанных с изменением состояния объекта между проверкой и реальным использованием. (проверили что файл есть, захотели открыть, его кто-то удалил между этими действиями, проиграли).
- Утечка дескрипторов
- Файловая система гарантирует, что до тех пор пока ты держишь файловый дескриптор на файл с ним ничего не произойдет извне (функции оканчивающиеся на *"at"*, защита от **TOCTOU**)
- *openat*

1.20 Литература

- The Unix Programming Environment. Brian W. Kernighan, Rob Pike
- Advanced Programming in the Unix Environment. W. Richard Stevens

1.21 Домашнее задание №2

Необходимо написать подмножество утилиты `find`.
Программа должна:

- Первым аргументом принимать абсолютный путь, в котором будет производиться поиск файлов.
- По умолчанию выводить в стандартный поток вывода все найденные файлы по этому пути
- Поддерживать аргумент **-inum num**. Аргумент задает номер инода
- Поддерживать аргумент **-name name**. Аргумент задает имя файла
- Поддерживать аргумент **-size [—+]**size****. Аргумент задает фильтр файлов по размеру(меньше, равен, больше)
- Поддерживать аргумент **-nlinks num**. Аргумент задает количество `hardlink`'ов у файлов
- Поддерживать аргумент **-exec path**. Аргумент задает путь до исполняемого файла, которому в качестве единственного аргумента нужно передать найденный в иерархии файл
- Поддерживать комбинацию аргументов. Например хочется найти все файлы с размером больше 1GB и скормить их утилите `/usr/bin/sha1sum`.

- Выполнять поиск рекурсивно, в том числе во всех вложенных директориях.
- Сильные духом призываются к выполнению задания с использованием системного вызова *getdents(2)*. Остальные могут использовать *readdir* и *opendir* для чтения содержимого директории.