ПО сетевых устройств

Трещановский Павел Александрович, к.т.н.

06.03.20

Multics vs Unix

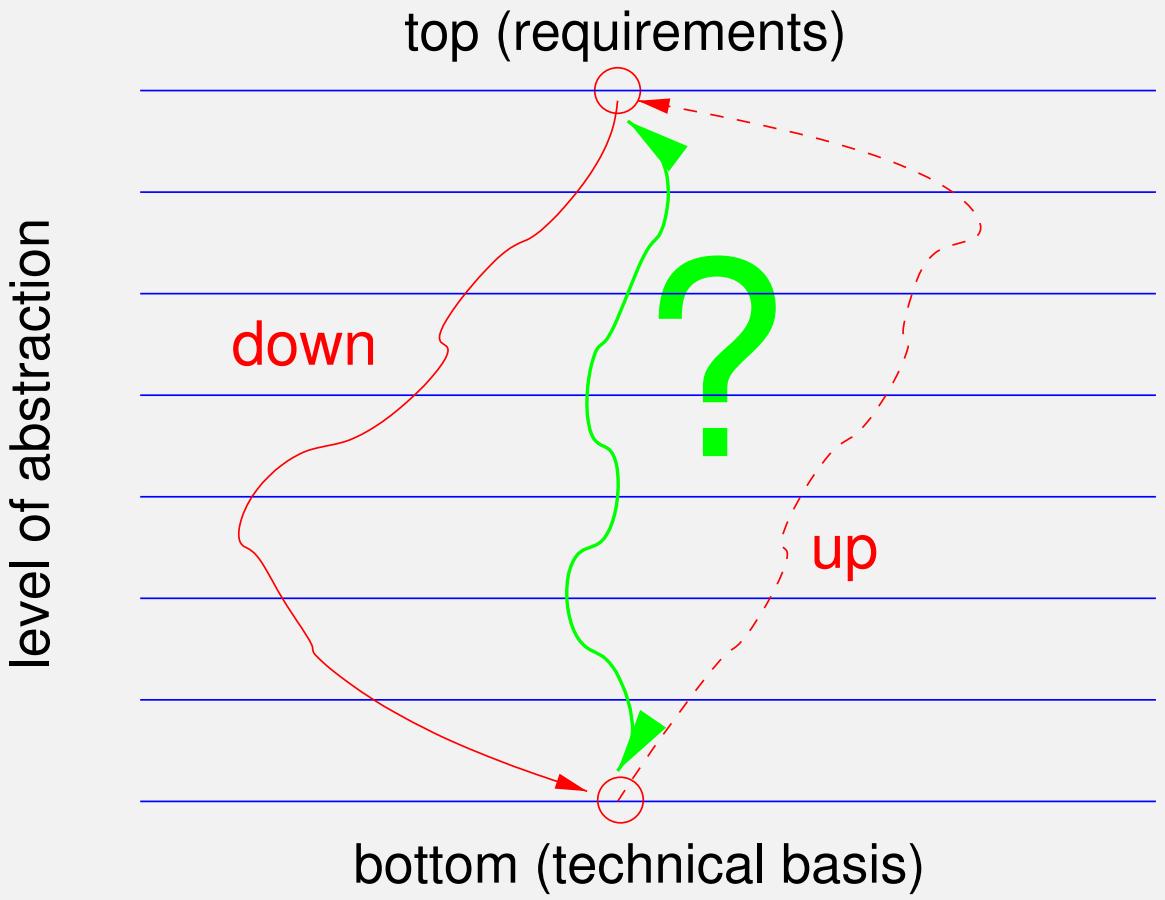
Multics

- Совместная разработка MIT, Bell Labs и General Electric.
- Высокоуровневый язык программирования PL/1.
- Инновационные идеи: файловая система, виртуальная страничная память, динамические библиотеки и др.
- Через пять лет разработки (64-69) - отсутствие рабочего продукта.

Unix

- 4-5 разработчиков первоначальной версии.
- Жесткие аппаратные ограничения.
- Минимальный функционал.
- Рабочий прототип был получен после нескольких месяцев разработки в 1969 году.

Top-Down vs Bottom-Up



Преимущества и недостатки

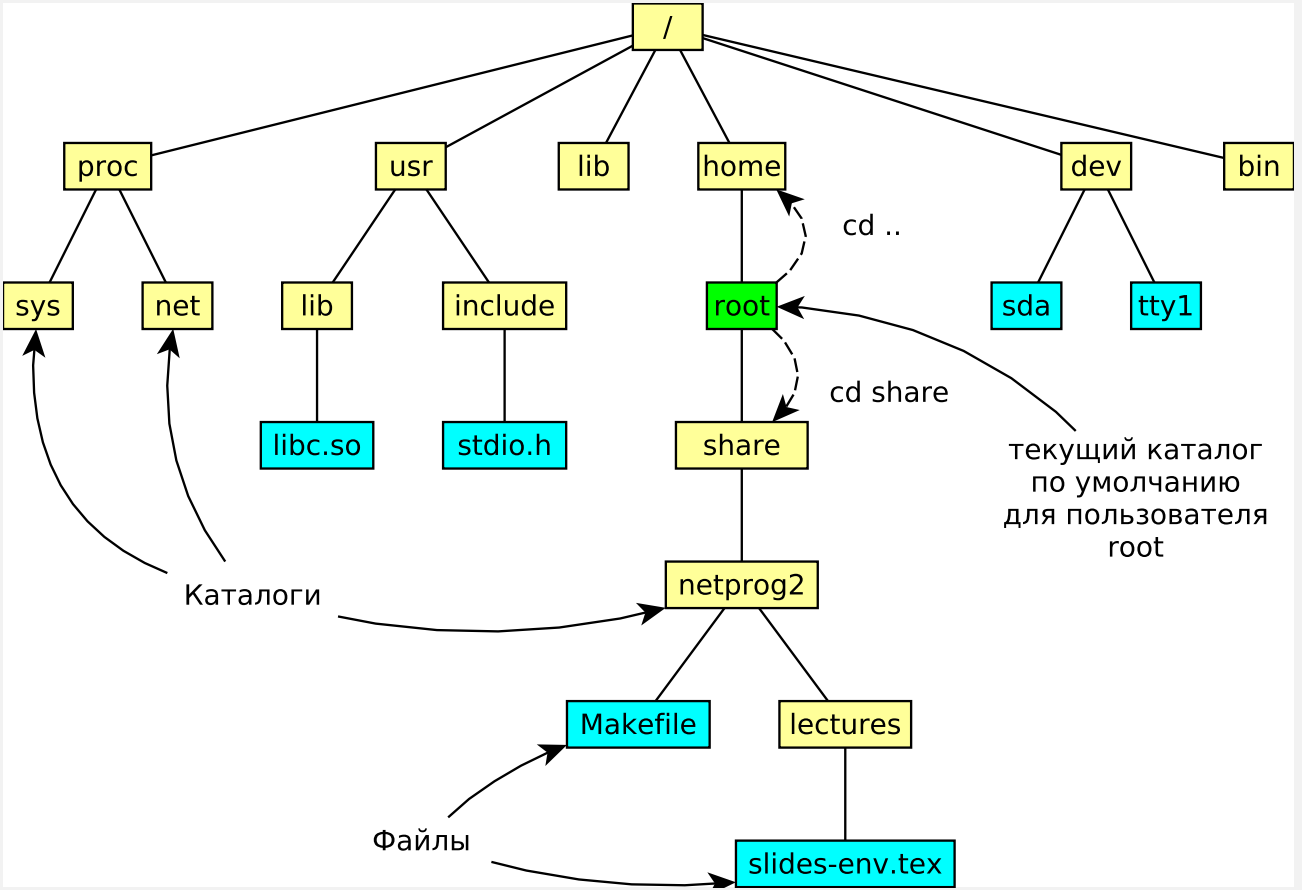
- Bottom-Up позволяет быстрее получить рабочий прототип.
- Bottom-Up лучше приспособлен к эволюции технических требований и возможностей. Части системы можно менять и/или адаптировать к новым условиям.
- Top-Down обычно более полно удовлетворяет исходным требованиям.
- Примеры Bottom-Up решений - файлы, файловые системы, большинство операционных систем.

Принципы разработки ОС Unix

McIlroy, предисловие к Unix Time-Sharing System (1978):

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new „features”.
- Expect the output of every program to become the input to another, as yet unknown, program. Don’t clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don’t insist on interactive input.
- Design and build software, even operating systems, to be tried early, ideally within weeks. Don’t hesitate to throw away the clumsy parts and rebuild them.

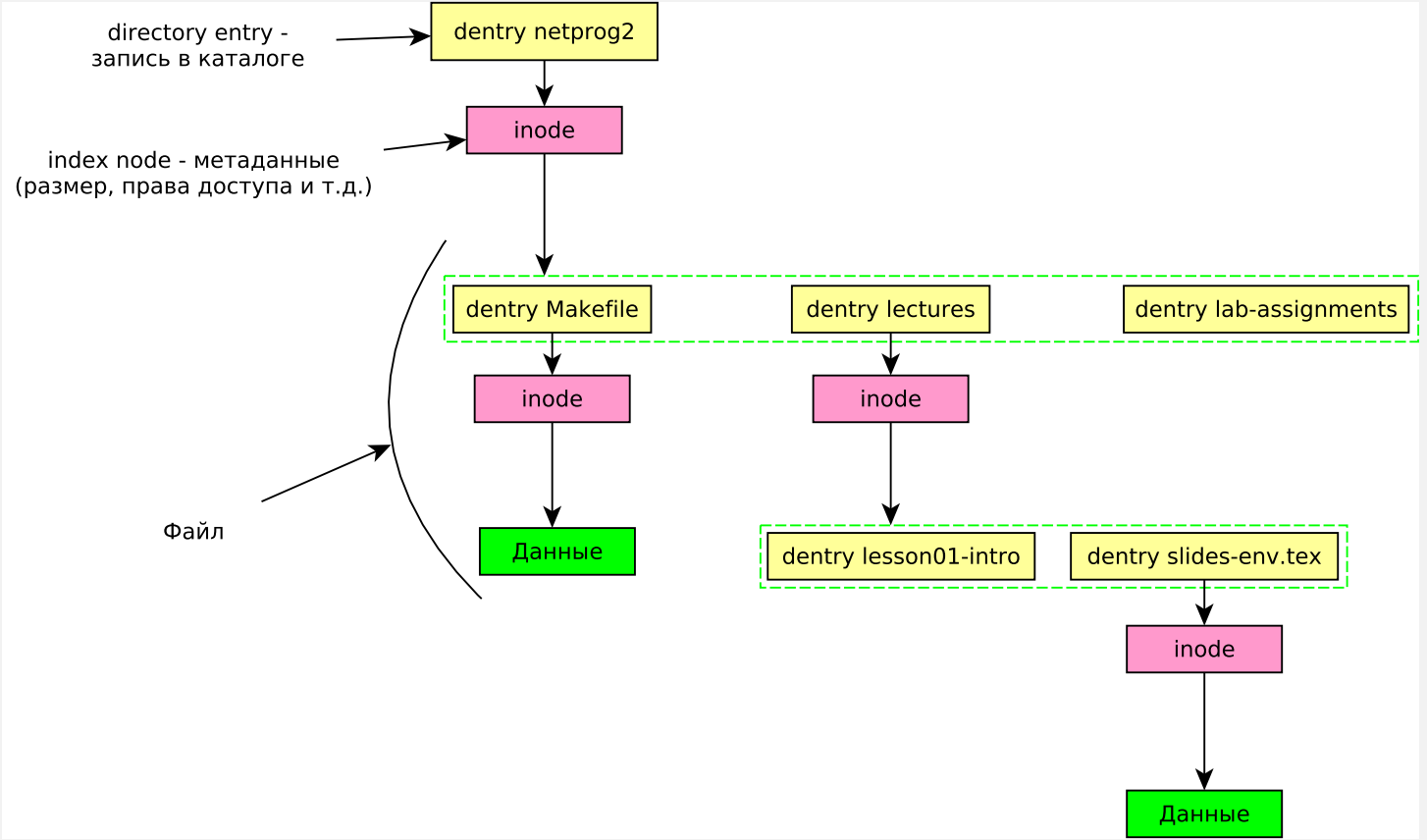
Иерархия каталогов и файлов



Иерархия каталогов, замечания

- У каждого процесса есть текущий каталог.
- Абсолютный путь - начинается с `/` и перечисляет имена каталогов от корня до цели. Например, `/home/root/share/netprog2`.
- Относительный путь - перечисляет имена каталогов от текущего каталога до цели. Например, `share/netprog2` или `./share/netprog2`.
- ~ (тильда) эквивалентна пути к домашнему каталогу пользователя. Для пользователя root - `/home/root`.

Внутреннее представление в ядре ОС



Метаданные

- Владелец файла.
- Тип файла (обычный, каталог, символьный, блочный).
- Права доступа к файлу - rwx, где r - доступ на чтение, w - на запись, x - на исполнение.
- Права доступа независимо задаются для владельца файла, для группы пользователей, к которой принадлежит владелец, и для всех остальных пользователей.
- Права часто записываются в виде 3 цифр. Например, 755 (111 101 101 в двоичной записи) разрешает любой доступ для владельца, но только чтение и исполнение для его группы и прочих пользователей.
- Время создания и последнего изменения файла.
- Размер файла.

Просмотр и изменение метаданных

- Просмотр метаданных:

```
$ ls -l
drwxr-xr-x 18 pavel pavel 4096 27 févr. 22:35 lab-assignments
```

d - каталог, 18 - количество файлов внутри.

- Задание прав доступа:

```
$ chmod 777 ./Makefile
```

- Задание владельца:

```
$ chown pavel:pavel ./Makefile
```

- Обновление времени доступа к файлу:

```
$ touch lab_00.c
```

Специальные файлы

- Доступ к большинству периферийных устройств осуществляется через специальные файлы - символьные и блочные файлы.
- Символьные файлы предоставляют доступ к устройствам, передающим и принимающим поток байтов. Например, последовательный интерфейс, модем GSM, терминал.
- Блочные файлы предоставляют доступ к устройствам, предоставляющим произвольный доступ к массиву байтов. Например, диски и разделы.
- Метаданные специальных файлов содержат 2 числа - старший и младший номера. Эти числа идентифицируют устройство. Имя файла не имеет значения для ядра ОС.
- Обычно файлы устройств находятся в каталоге /dev: /dev/tty1 - терминал, /dev/sda - жесткий диск и т.д.

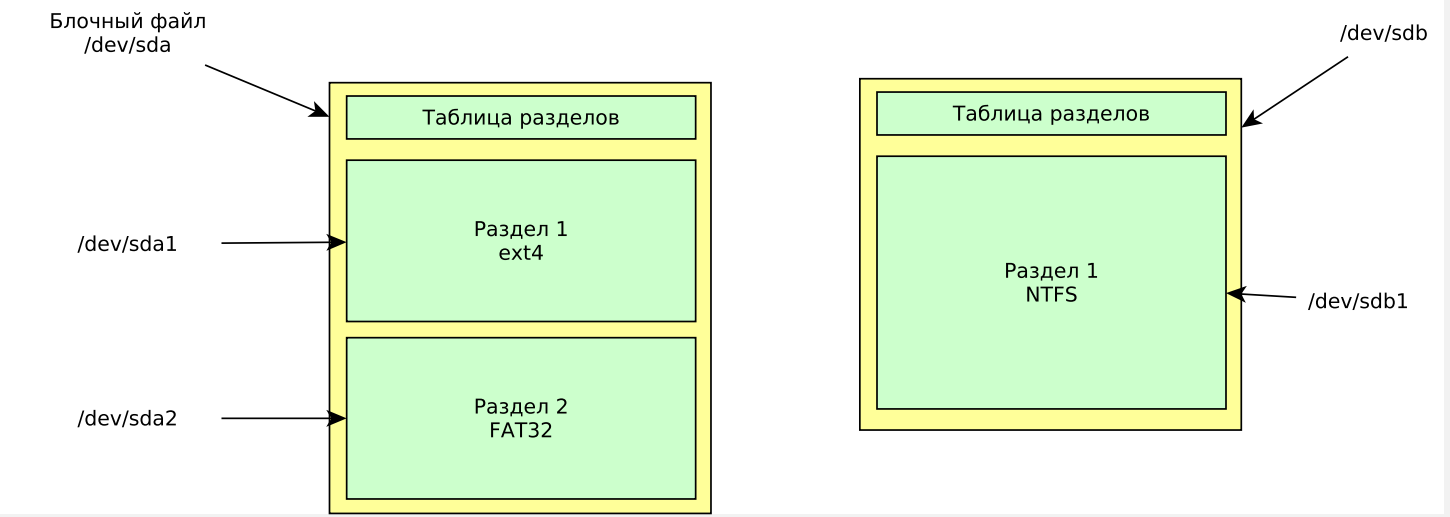
Специальные файлы, замечания

- Сами файлы не содержат никаких данных - все операции чтения и записи реализуются нижележащими устройствами.
- Файл - это не само устройство, а только интерфейс к нему. Наличие или отсутствие файла не означает наличие или отсутствие устройства.
- Обычно создаются и удаляются системой автоматически.
- Могут быть созданы вручную:

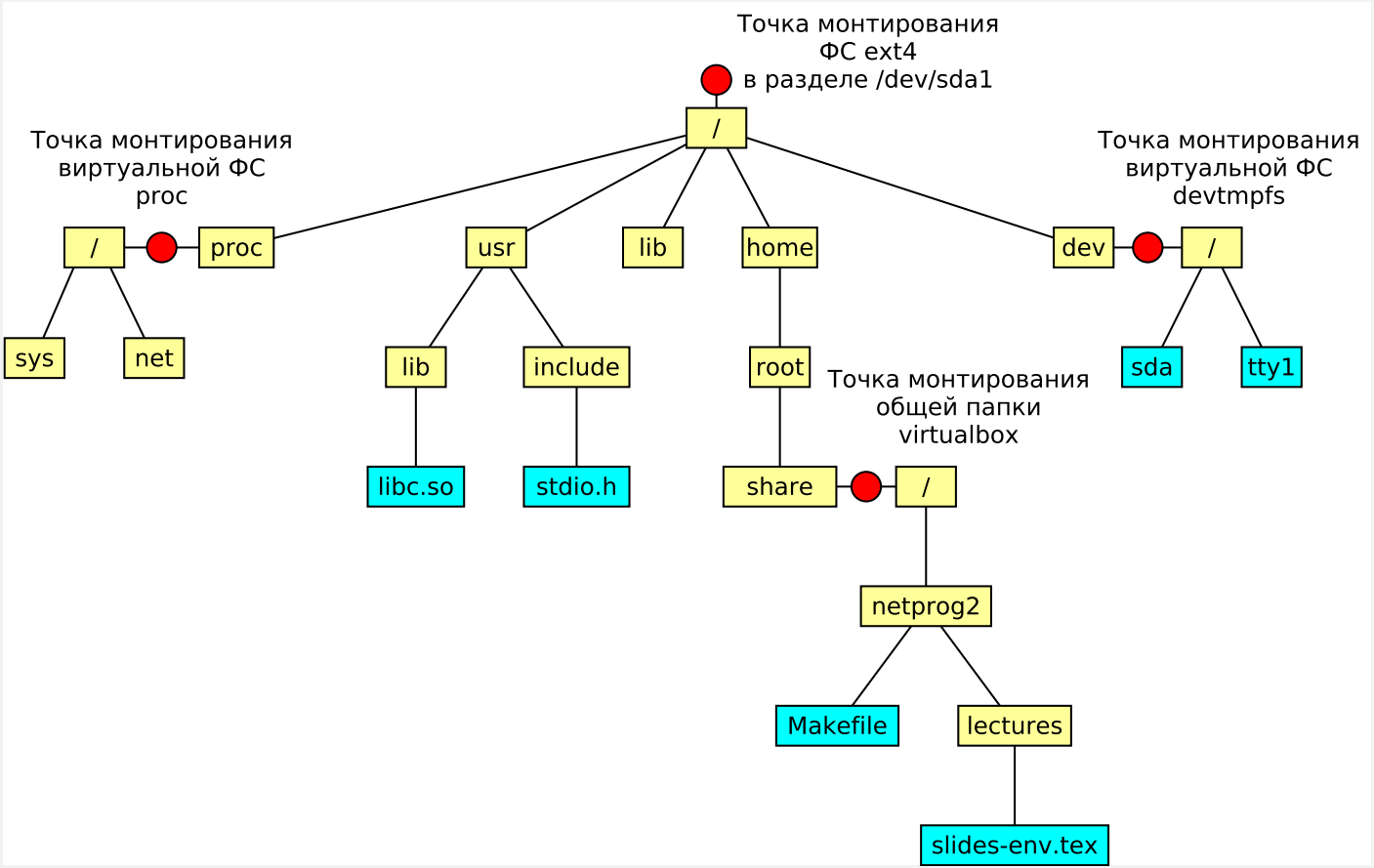
```
$ mknod another_tty1 c 4 1
```

  Создает символьный файл для терминала (старший номер 4, младший номер 1).

Диски и разделы



Точки монтирования



Командный интерфейс

- Монтирование:

```
$ mount -t ext4 /dev/sda1 /mnt
```

- Аргументы: тип (ext4), что (раздел /dev/sda1), куда (/mnt).
- Демонтирование:

```
$ umount /mnt
```

- Отображение точек монтирования:

```
$ mount
/dev/sda1 on / type ext4 (rw,noatime,data=ordered)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620,ptmxmode=000)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
```

Преимущества монтируемых файловых систем

- Косметика: отображение файловых систем на жестких дисках в произвольных точках дерева файлов по вкусу администратора.
- Более глубокое преимущество - отвязка дерева файлов от физического носителя данных, „виртуализация” файловой системы.
- Файловая система может предоставлять доступ внутренним объектам ядра Linux в целях конфигурирования, мониторинга и диагностики.
- Файловая система может предоставлять доступ к внешним ресурсам: FTP-сервер, проект в системе контроля версий git, база данных SQL.
- Для приложений виртуальная ФС выглядит как обычное (почти) дерево файлов.

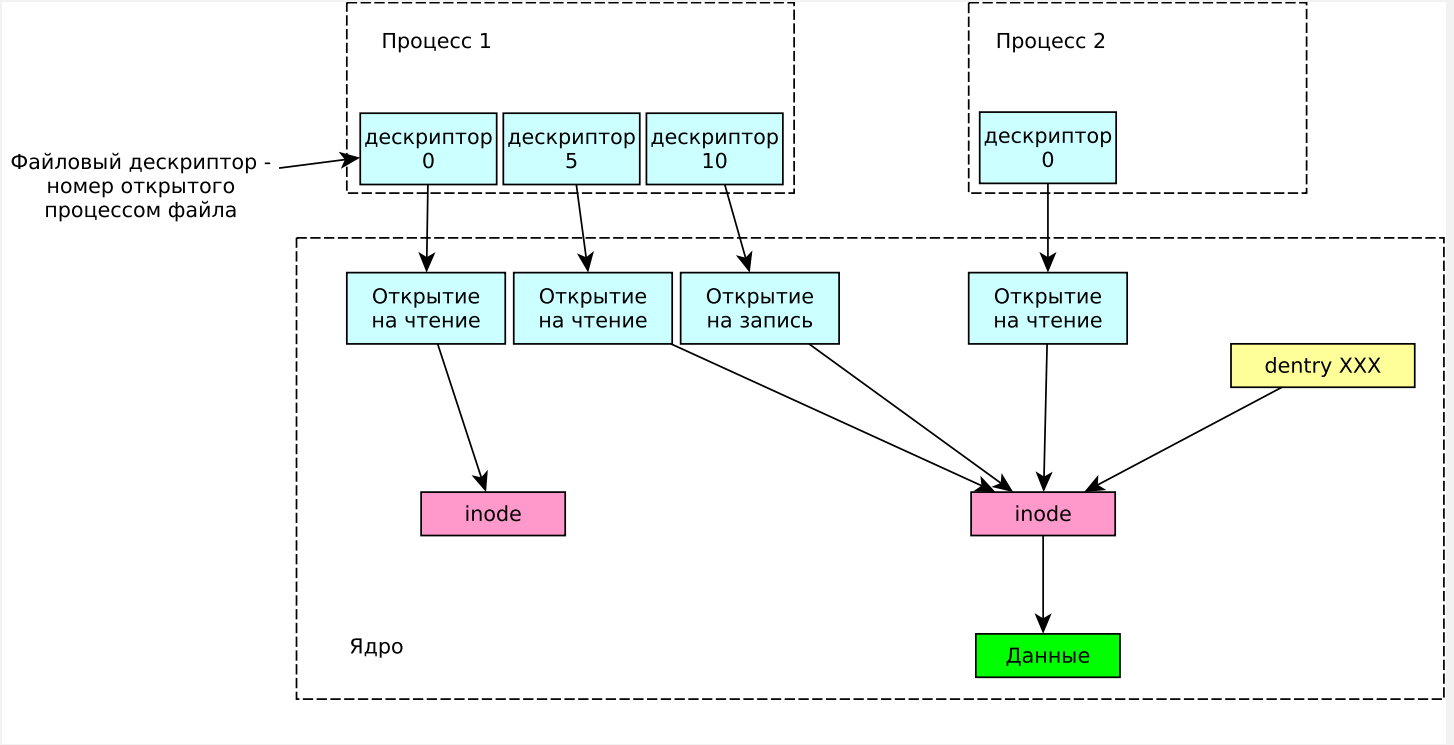
Виртуальные ФС proc и tmpfs

- proc предоставляет доступ к информации о процессах и общей информации о системе.
- Многие утилиты используют proc. Например, утилита ps:

```
$ ps -A
PID TTY          TIME CMD
  1 ?        00:00:03 init
  2 ?        00:00:00 kthreadd
  3 ?        00:00:00 ksoftirqd/0
```

- tmpfs представляет часть оперативной памяти в виде файловой системы.
- tmpfs используется для временных файлов, которые не должны сохранятся при перезапуске системы, а также в случаях, когда нужен очень быстрый обмен данными через файлы.

Открытый файл и файловый дескриптор



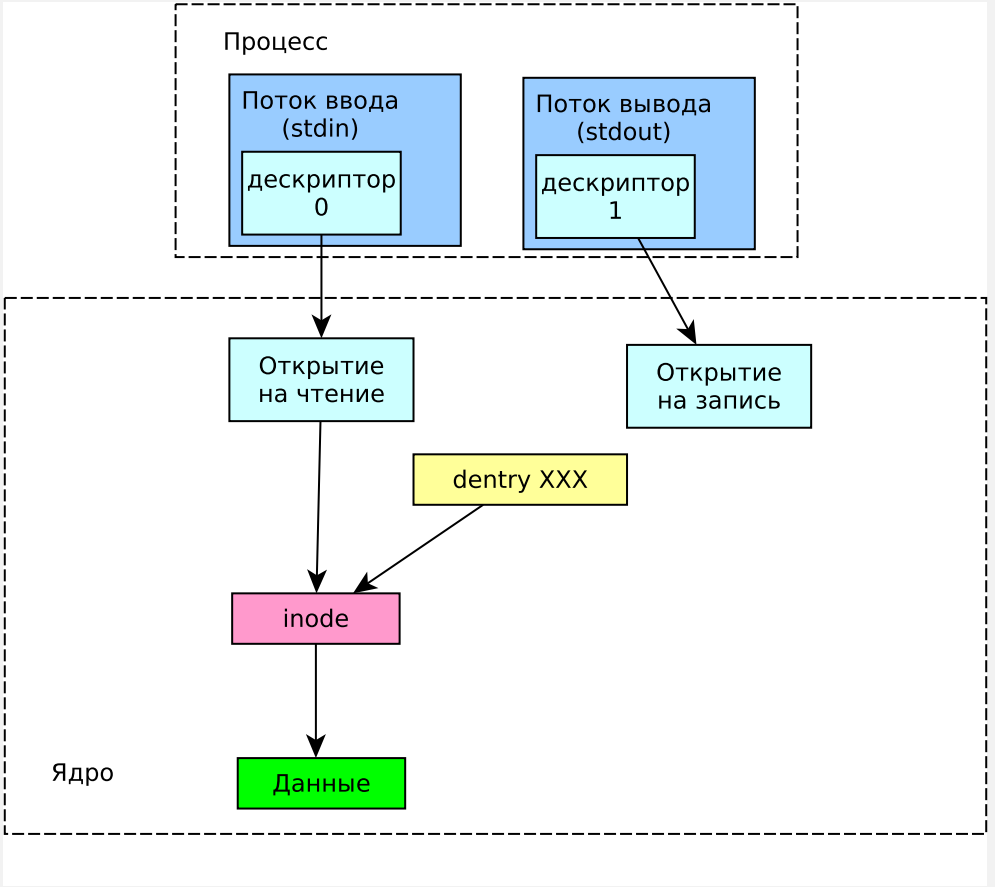
Низкоуровневый API

```
int fd;
char string[] = "sdfgsdfg";
int ret;

/* int open(const char *pathname, int flags); */
fd = open("/dir1/dir2/file1", O_WRONLY | O_TRUNC);
if (fd < 0) {
       printf("open failed: %d\n", strerror(errno));
       goto on_error;
}

/* ssize_t write(int fd, const void *buf, size_t count); */
ret = write(fd, write_buffer, strlen(string));
if (ret < 0) {
       printf("write failed: %d\n", strerror(errno));
       goto on_error;
}

close(fd);
```

Дескрипторы, замечания

- API: open(), close(), write(), read().
- Флаги `O_RDONLY`, `O_WRONLY`, `O_RDWR` открывают на чтение, на запись, на чтение и запись.
- Флаг `O_TRUNC` удаляет имеющееся содержимое файла.
- Флаги можно объединять операцией побитового или `|`.
- Обычно у процесса при запуске уже созданы дескрипторы 0, 1 и 2.

Потоки ввода-вывода



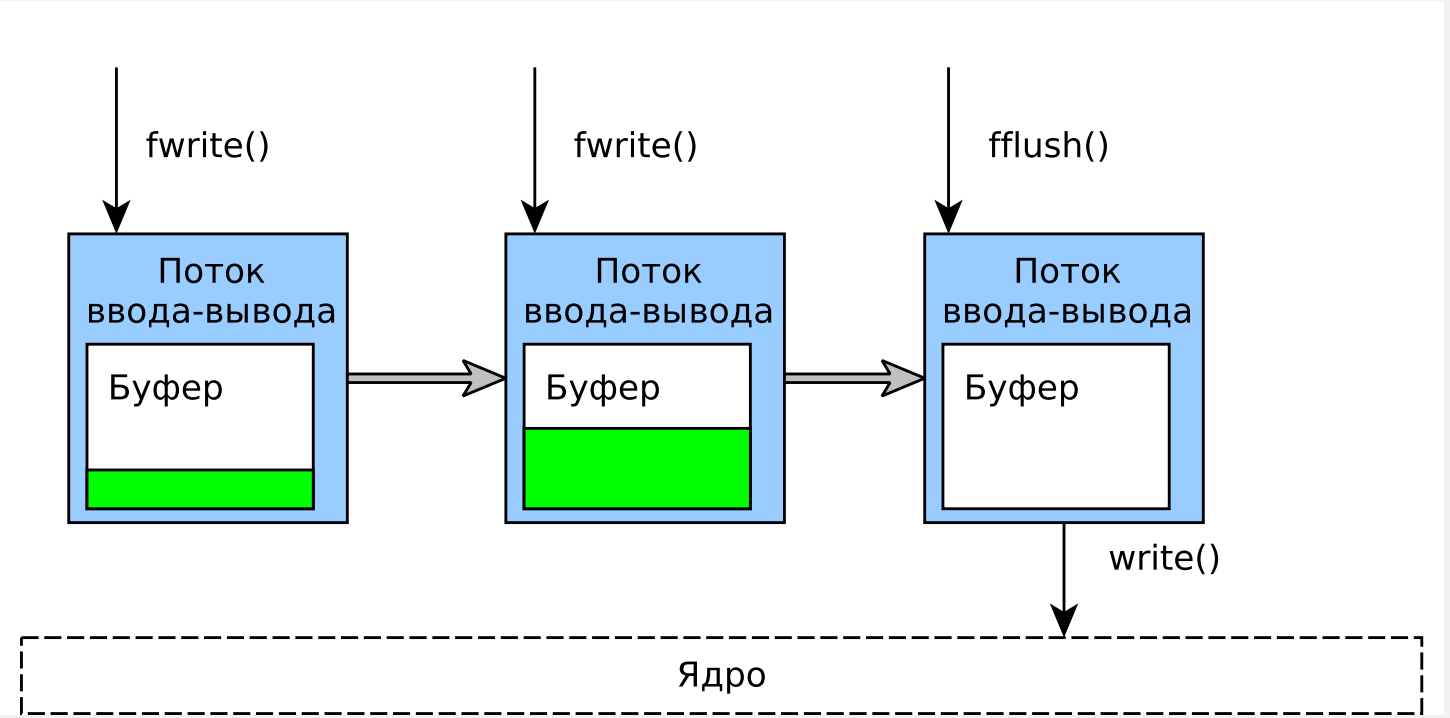
Более высокоуровневый API

```
FILE *fp;
char string1[] = "sdfgsdfg";
char string2[] = "32342";

/* FILE *fopen (const char *path, const char *mode); */
fp = fopen("/dir1/dir2/file1", "w");
if (!fp) {
       printf("fopen failed: %d\n", strerror(errno));
       goto on_error;
}

/* size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *stream); */
fwrite(string1, 1, strlen(string1), fp);
fwrite(string2, 1, strlen(string2), fp);

fclose(fp);
```

Исключительные ситуации

```
FILE *fp_in, *fp_out;
char buf[128];
size_t sz;

/*
 * Открытие файлов.
 *  .......
 */

/* size_t fread (void *ptr, size_t size, size_t nmemb, FILE *stream); */
while (1) {
       sz = fread(buf, 1, sizeof(buf), fp_in);
       fwrite(buf, 1, sz, fp_out);

       /* fread()/fwrite() _не_ возвращают -1 в случае ошибки. */
       if (ferror(fp_in) || ferror(fp_out))
              goto on_error;
       if (feof(fp_in))
              break;
}
```

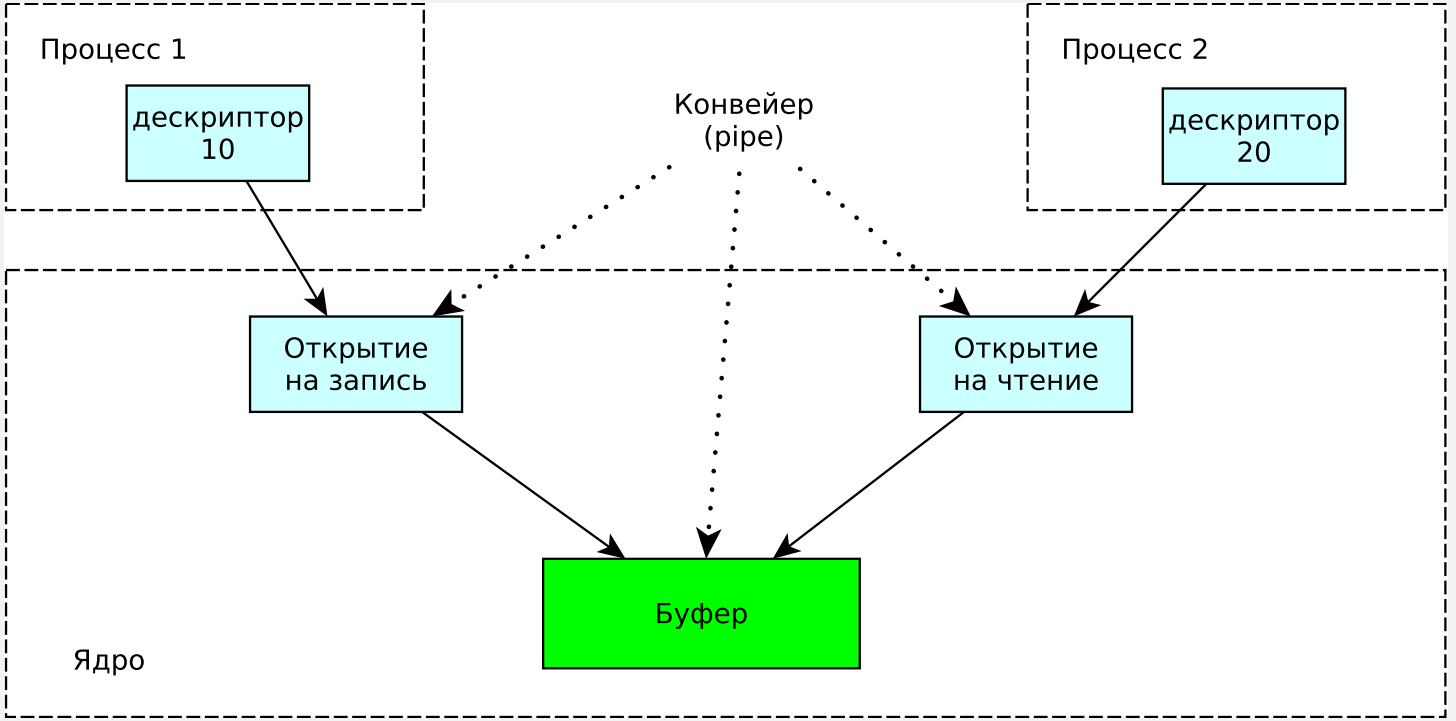
Буферизация



Потоки, замечания

- Буферизация данных на чтение и запись.
- Более удобный API.
- `fprintf()` - форматированный вывод в указанный поток.
- `fgets()` - чтение строки (последовательность, оканчивающаяся на ’\n’) из указанного потока.
- Каждый поток однозначно соответствует дескриптору. `fdopen()` создает поток из дескриптора.
- Стандартные потоки ввода (stdin), вывода (stdout) и ошибок создаются из дескрипторов 0, 1 и 2 при запуске процесса.

Конвейер (pipe)



Использование pipe из shell

Команда

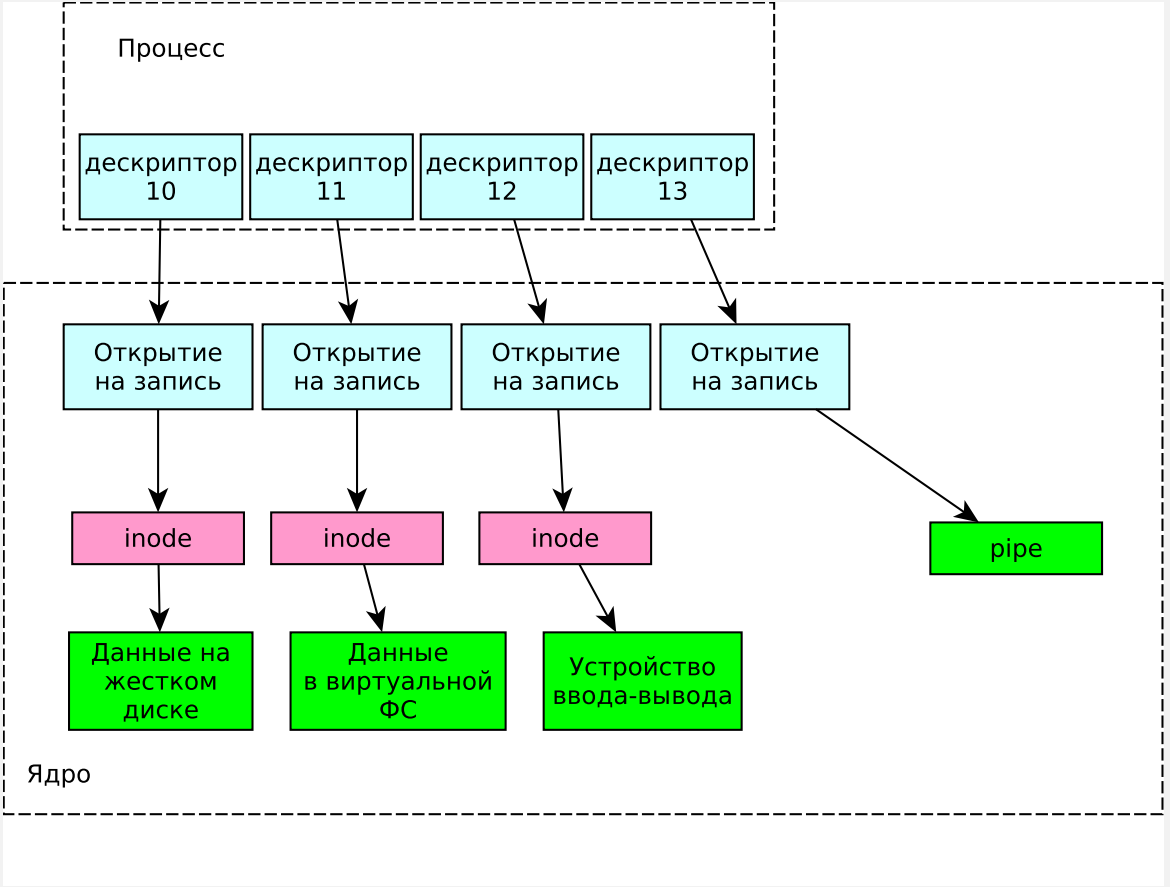
```
prog1 | prog2
```

подключает дескриптор 1 (вывод) программы prog1 к дескриптору 0 (ввод) программы prog2 через конвейер. prog1 и prog2 не знают о существовании конвейера.
Примеры:
`'cat students.txt | grep Маша'` выводит только строки с указанным именем в терминал.
`'cat students.txt | sed s/Маша/Даша/'` производит замену имени и выводит результат в терминал.
`'cat students.txt | wc -l'` подсчитывает количество строк в файле.

Конвейер может иметь произвольную длину:

```
prog1 | prog2 | prog3 | prog4 | ...
```

Управление потоками ввода-вывода



Перенаправление в shell

Команда

```
prog1 < file1 > file2
```

подключает дескриптор 0 (ввод) программы prog1 к файлу file1, а дескриптор 1 (вывод) - к файлу file2. prog1 не знает о перенаправлении.
`'grep Маша < students.txt > output.txt'` ищет строки с указанным именем в файле students.txt и записывает их в файл output.txt.

Применение принципов разработки ОС Unix

- Программы принимают на вход и передают на выход текстовые потоки.
- Система предоставляет набор инструментов (утилит) для решения элементарных задач.
- Сложные задачи решаются связыванием нескольких утилит через файлы и конвейеры.
- Система предоставляет утилиты-фильтры для обработки потоков между приложениями: поиск (grep), замена(sed) и т.д.
- shell является универсальным клеем. Отдельные команды объединяются в скрипты.

Шаблоны поиска (glob pattern)

Пример:

```
$ ls *.pd?
data-redirection.pdf  file-hierarchy.pdf
```

- shell автоматически подменяет шаблон на последовательность соответствующих имен.
- Специальному символу * соответствует любая последовательность обычных символов (в т.ч. последовательность длины 0).
- Специальному символу ? соответствует любой один обычный символ.
- Специальные символы могут использоваться в одном шаблоне произвольное число раз.