Файловые системы

АКОС, МФТИ

26 сентября, 2024

Всё есть файл...

...овый дескриптор

Встроенные

= `STDIN_FILENO`
= `STDOUT_FILENO`
= `STDERR_FILENO`

- **Потоки консоли:** ввод, вывод, ошибки;
- Почти всегда равны `0`, `1` и `2`
- Но лучше использовать константы;
- Их даже можно закрыть (но зачем?).

Пользовательские

= `open(...)`
= `socket(...)`
= `accept(...)`
= `epoll(...)`
= `socketpair(...)` *
= `signalfd(...)`, …

- Файлы, сеть, мультиплексеры,…;
- Всё, что работает как поток данных.

Почему не `fopen(...)` ?

`fopen(...)` возвращает `FILE*`, а не дескриптор

Папки – тоже файлы?

```
1  int main() {
2    int fd = open("/", O_RDONLY, 0);
3    struct dirent ent = {};
4    while(getdents64(fd, &ent, sizeof(ent)) > 0) {
5        printf("%s\n", ent.d_name);
6        lseek(fd, ent.d_off, SEEK_SET);
7    }
8  }
```

Да, это тоже файловые дескрипторы. А ещё это директории, а не папки.

А если вызвать `read(...)` на директорию?

```c
int main() {
  int fd = open("/", O_RDONLY, 0);
  char buffer[16];
  if(read(fd, buffer, sizeof(buffer)) >= 0) {
    printf("%15s", buffer);
  } else perror("read");
}
```

Вывод: `read: Is a directory` . **Причина:** `read(...) == -EISDIR`

Файловый дескриптор - это

- **Число**, сопоставленное какому-то ресурсу.
- Его можно понимать, как **“указатель” на виртуальный класс**.
- **Разработчик должен помнить** типы дескрипторов. Это упрощают обёртки:
  - `FILE`, `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fseek()`, … для файлов;
  - `DIR`, `opendir()`, `closedir()`, `readdir()`, `seekdir()`, … для директорий;
  - **И практически ничего** для работы с сокетами / сигналами / pipe-ами.
- Активные дескрипторы процесса можно увидеть в `/proc/<pid>/fd/`

```
open(char *path, int flags, mode_t mode)
```

Открыть (создать) файл (директорию).

`int flags` **:**

|= `O_RDONLY` : Открыть на **чтение**;

|= `O_WRONLY` : Открыть на **запись**;

|= `O_RDWR` : Открыть на **чтение и запись**;

|= `O_TRUNC` : **Очистить** файл при открытии;

|= `O_CREAT` : **Создать** файл, если его нет;

|= `O_EXCL` : Сломаться, если файл уже есть.

`mode_t mode` **: маска прав для создаваемого файла.**

Чтение и запись

- Для этого служат самые известные системные вызовы:

```
read(int fd, void* buf, size_t count)
```

```
write(int fd, void* buf, size_t count)
```

- Существуют `pread(...)` и `pwrite(...)`, которые **явно принимают позицию файла**.
- Если хочется одновременно записать **несколько буферов**, можно использовать:

```
readv(int fd, struct iovec *vector, int count)
```

```
writev(int fd, struct iovec *vector, int count)
```

```
lseek(int fd, off_t offset, int whence)
```

Настройка позиции файла

`int whence` :

= `SEEK_SET` : Установить позицию на `offset` ;

= `SEEK_CUR` : Сдвинуть позицию на `offset` ;

= `SEEK_END` : Установить позицию в **конец** и сдвинуть на `offset` .

Узнать позицию можно системным вызовом `tell(int fd)`

`fcntl(int fd, int cmd, long arg)`

Системный вызов управления дескрипторами.

`int cmd` **:**

= `F_SETFD` : Установить флаги дескриптора;

= `F_SETFL` : Установить флаги статуса;

= `F_SETLK` : Установить **блокировку** на файл;

= `F_DUPFD` : **Дублировать** дескриптор;

= `F_SETSIG` : Получить сигнал, когда будет доступно чтение / запись;

= `F_SETPIPE_SZ` : Настроить размер очереди.

`long arg` **: аргумент, используется в** `F_SET*` **– командах**

Какие флаги можно настраивать?

Флаги дескриптора (`F_SETFD`):

- `FD_CLOEXEC` : автоматически закрыть файл при вызове `exec()`
- И всё, их ровно одна штука 🤷‍♂️.

Полезные флаги статуса (`F_SETFL`):

- `O_APPEND` : дописывать в конец файла, как в лог;
- `O_NONBLOCK` : запретить блокирующий ввод/вывод;
- `O_NOATIME` : не изменять время последнего доступа к файлу;

Бесполезные флаги статуса:

- `O_ASYNC` : получать сигнал по доступности чтения/записи;
- `O_DIRECT` : прямая запись, обход кеширования ядра;

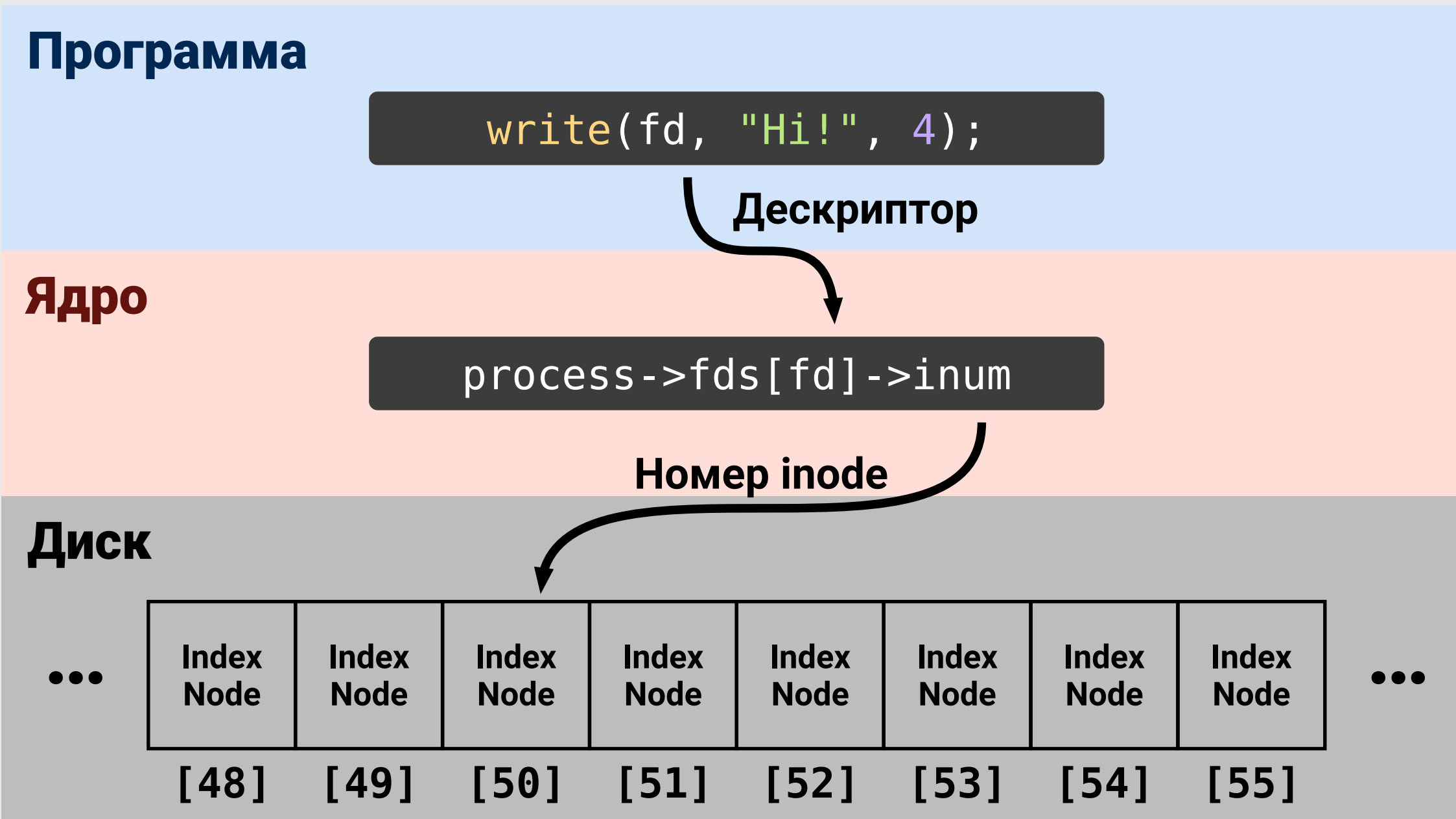
А что такое файл?

Файл – это не путь!

```
/home/you/test.txt
```

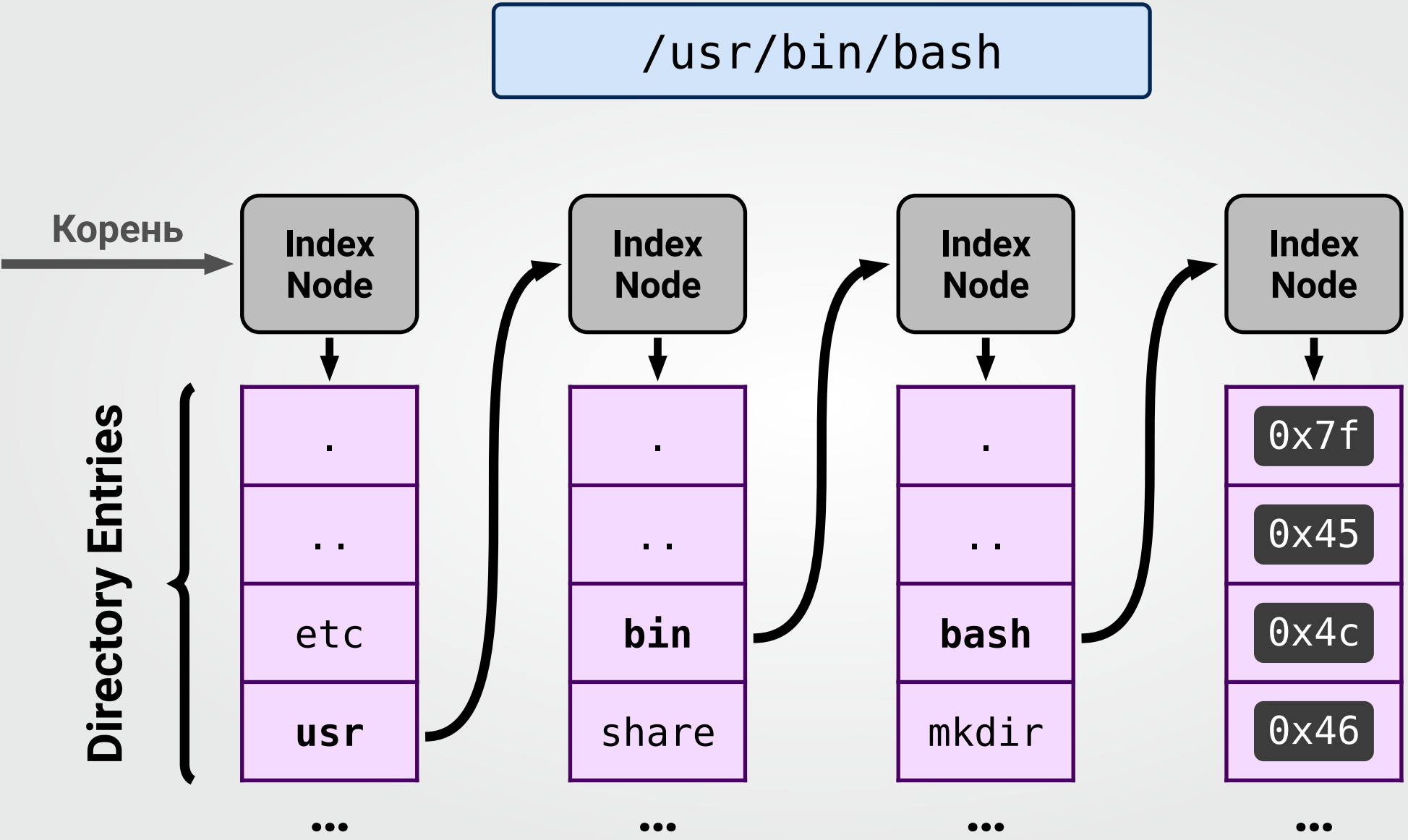
Может быть тем же файлом, что и:

```
/home/you/test2.txt
```


Программа
write(fd, "Hi!", 4);
Дескриптор
Ядро
process->fds[fd]->inum
Номер inode
Диск
...
Index Node
Index Node
Index Node
Index Node
Index Node
Index Node
Index Node
Index Node
...
[48]
[49]
[50]
[51]
[52]
[53]
[54]
[55]

Index Node

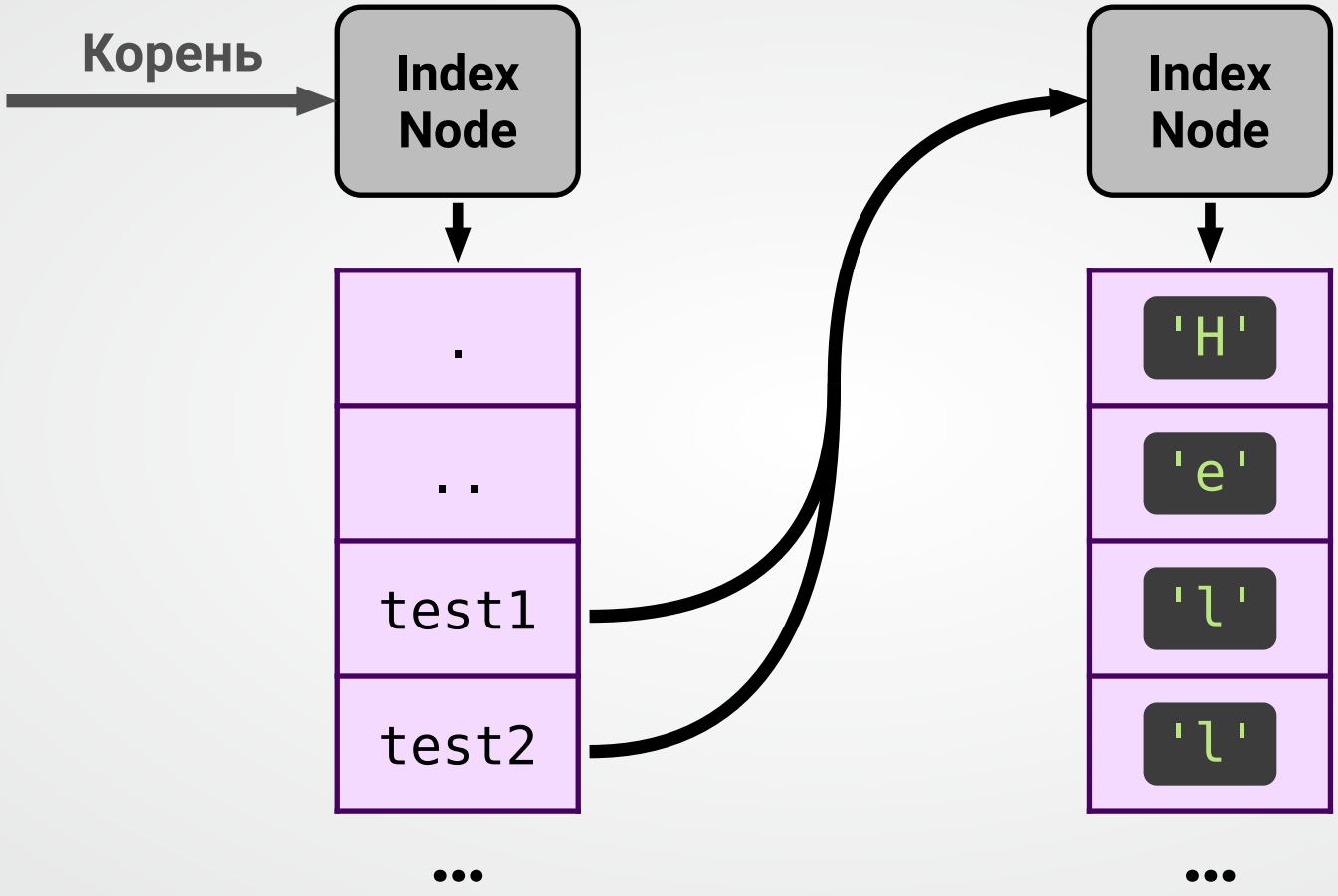
- **Структура, которую обычно называют “файлом”**;
- Хранится в длинном массиве на жестком диске;
- **Знает**, где на жестком диске хранится **содержимое её файла**;
- **Хранит число ссылок** на саму себя (как `std::shared_ptr`).


/usr/bin/bash
Корень
Index Node
Index Node
Index Node
Index Node
Directory Entries
.
..
etc
usr
.
..
bin
share
.
..
bash
mkdir
0x7f
0x45
0x4c
0x46
...
...
...
...

Directory Entry

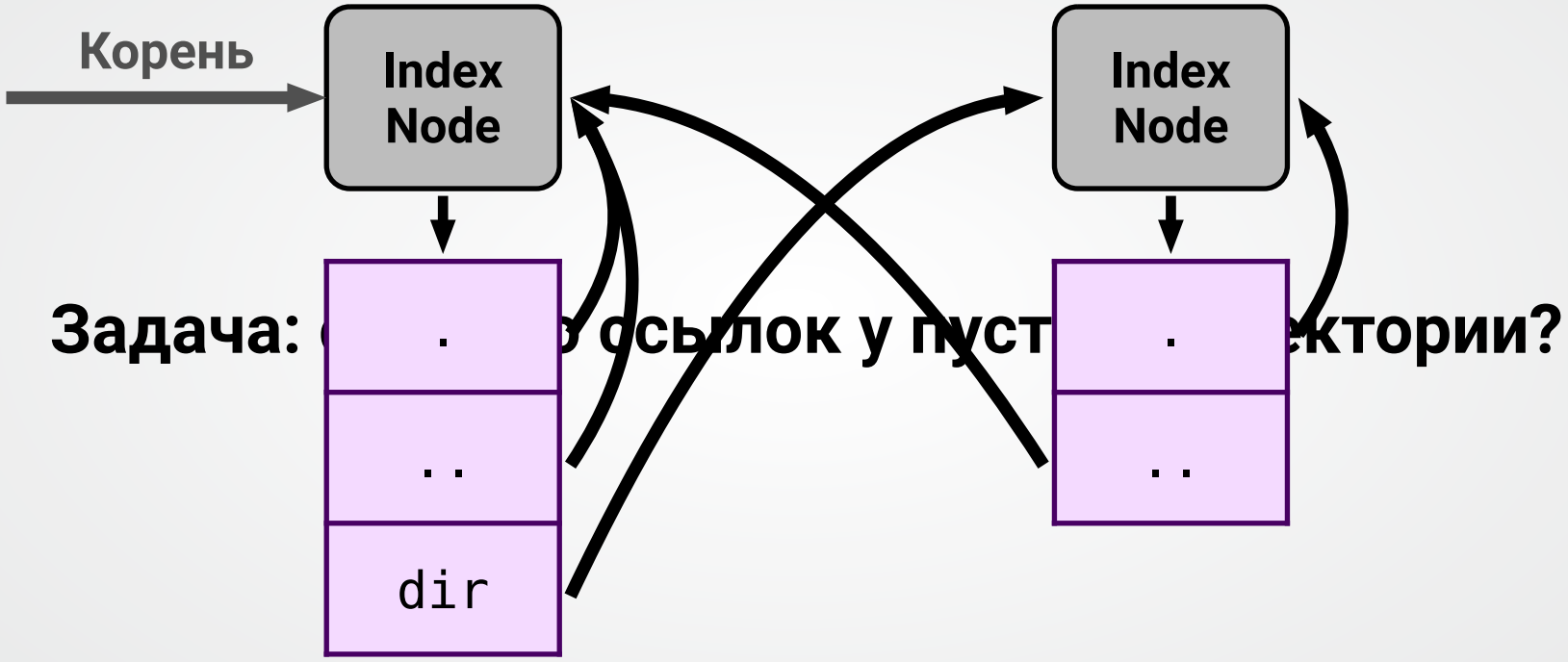
- Структура с **именем** и **адресом** Index Node;
- Может **ссылаться на свой же Index Node**(`./`);
- Живёт там, где обычно хранятся данные файла;
- Хранится в массиве других Directory Entry.

Что, если на Index Node несколько ссылок?



Это называется **жесткая ссылка.** Так можно только с файлами.

Задача: сколько ссылок у пустой директории?



Ответ: 2

VFS

Виртуальная файловая система

- К системе может быть подключено **много накопителей**: диски, USB-носители, …;
- Каждый из них может иметь свою файловую систему;

Как работать со всем сразу?

VFS - Обобщённый интерфейс для ФС.

Можно думать об этом, как о виртуальном классе:

```
class ExFat: public virtual VFS { /* ... */ }
class Fat32: public virtual VFS { /* ... */ }
class NTFS:  public virtual VFS { /* ... */ }
class FAT:   public virtual VFS { /* ... */ }
class HFS:   public virtual VFS { /* ... */ }
// ...
```

Теперь любую из этих файловых систем можно подключить к системе:

```
void System::mount(VFS* filesystem, const char* path);
```

`procfs` – не файловая файловая система.

`$ cat /proc/meminfo` : информация о памяти;

`$ cat /proc/cpuinfo` : информация о CPU;

`$ cat /proc/version` : версия ядра;

`$ cat /proc/schedstat` : информация от планировщика о каждом CPU;

`$ cat /proc/filesystems` : информация о файловых системах.

`$ cat /proc/<pid>/schedstat` : информация от планировщика о процессе;

`$ ls /proc/<pid>/fd` : открытые файловые дескрипторы;

Эта файловая система тоже реализует интерфейс VFS.

Другие примеры

Системные:

`sysfs` : содержит информацию об устройствах и драйверах;

`pipefs` : служит для создания и использования pipe-ов;

`ramfs` : использует оперативную память вместо диска;

`tmpfs` : как `ramfs` , но со сбросом на swap;

Общего назначения:

`ecryptfs` : хранит файлы в зашифрованном виде;

`unionfs` : объединяет несколько файловых систем вместе;

`overlayfs` : хранит разницу двух файловых систем

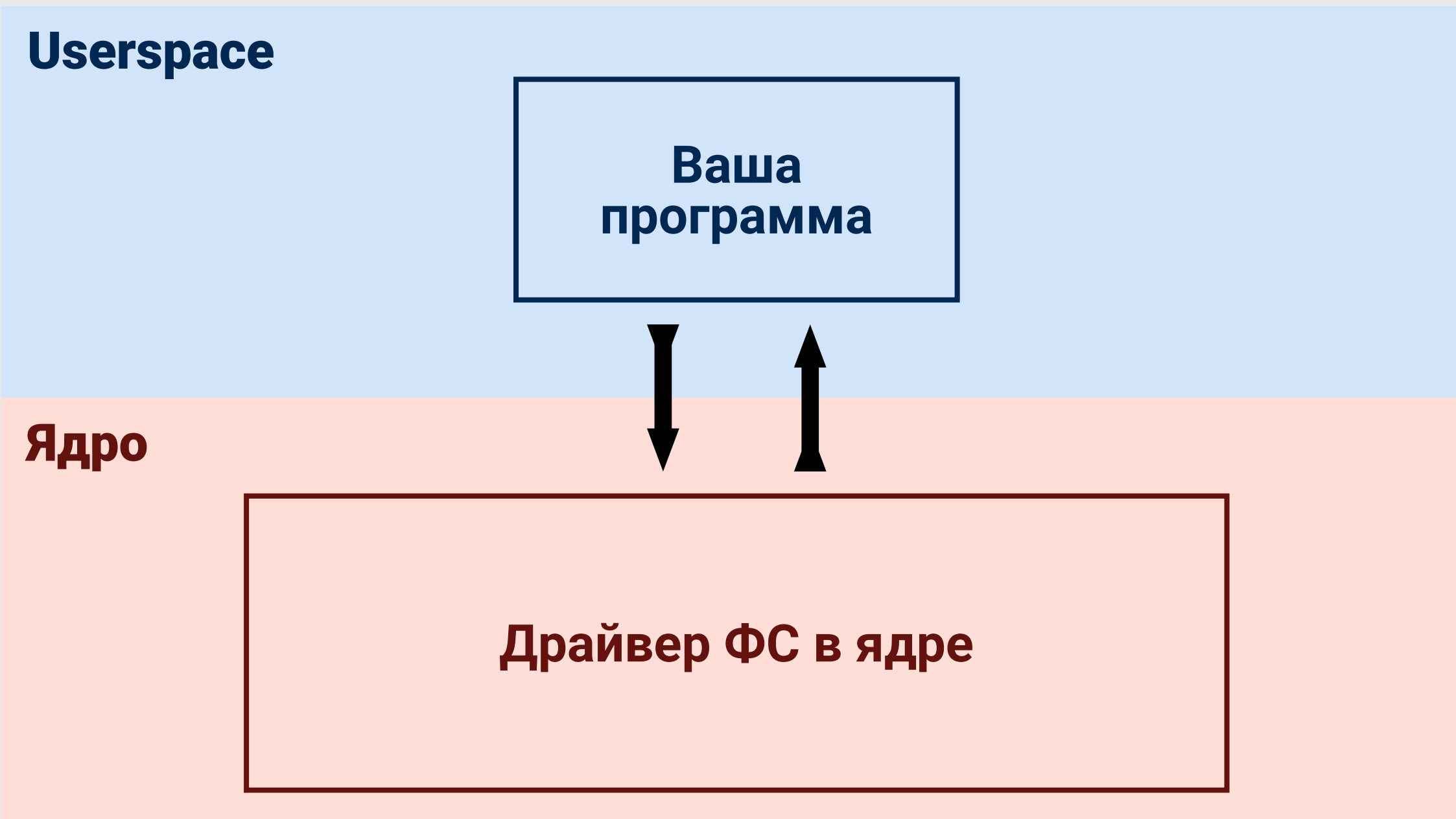
Как быть, если очень хочется свою ФС?

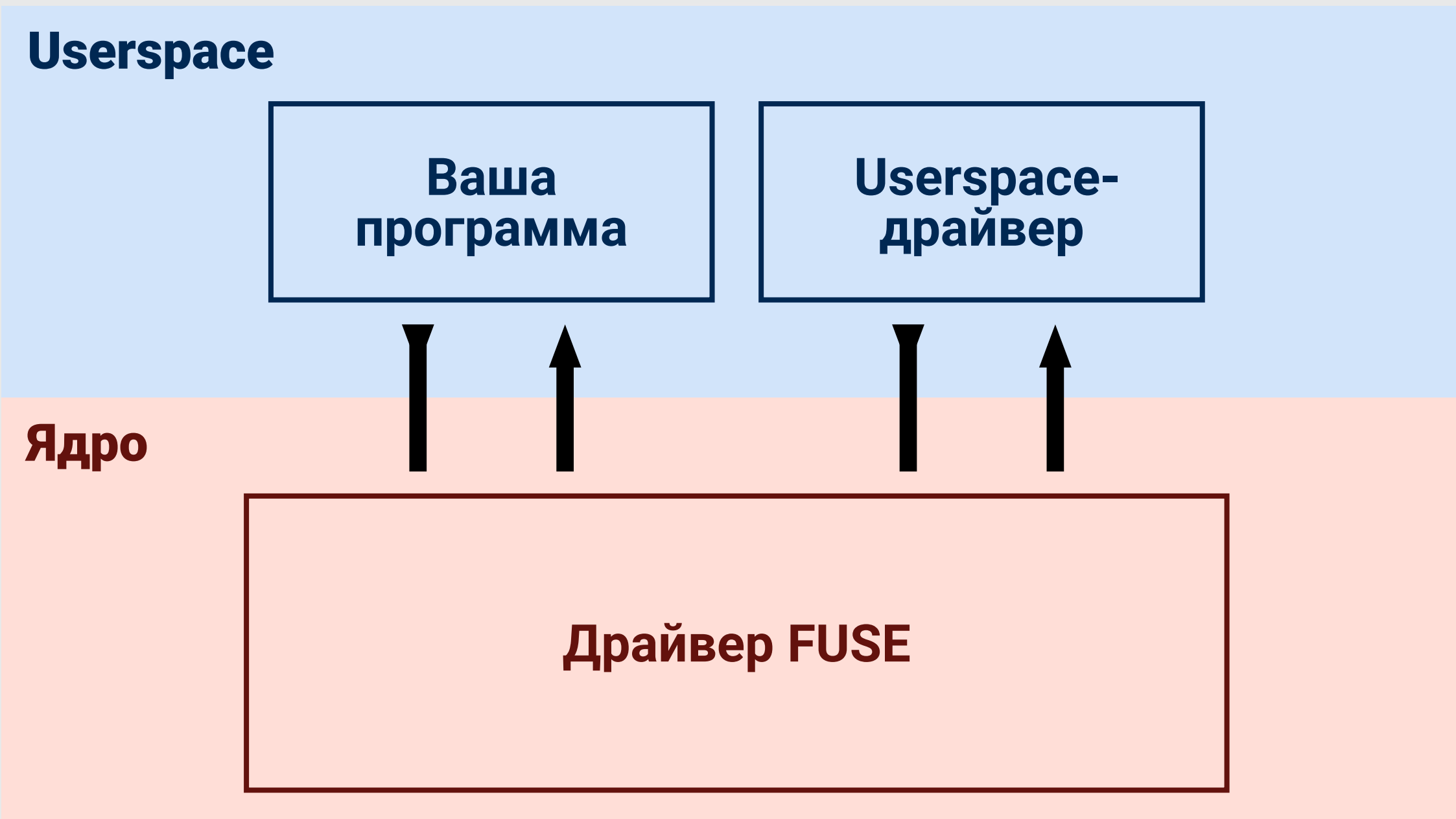
FUSE

Filesystem in userspace

Избавляет от необходимости разрабатывать драйвер для ядра;

Работает медленнее, чем встроенные в ядро файловые системы.


Userspace
Ваша программа
Ядро
Драйвер ФС в ядре

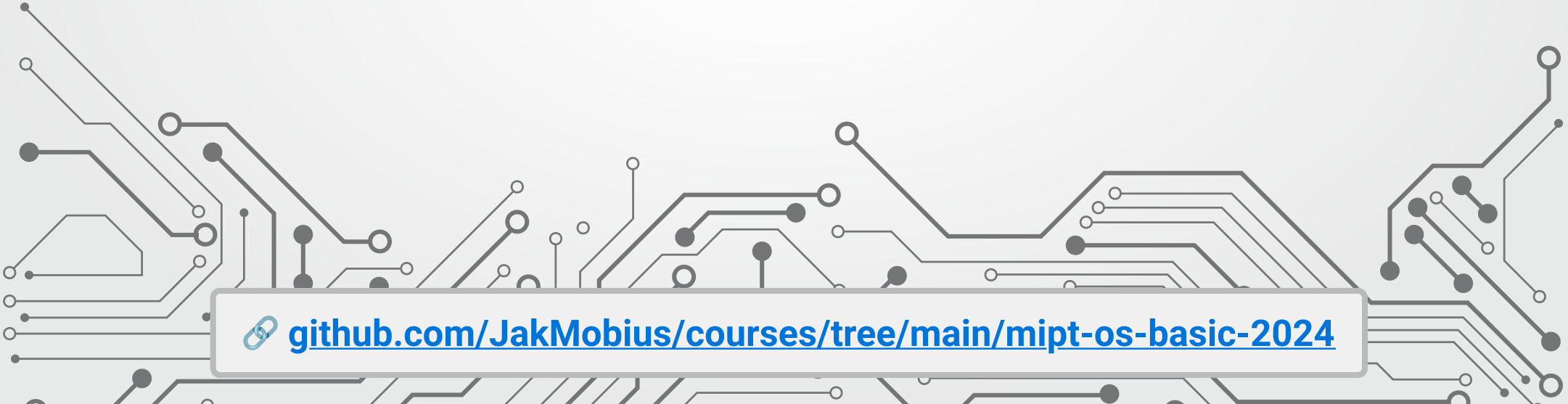

Userspace
Ваша программа
Userspace-драйвер
Ядро
Драйвер FUSE

Сеанс магии

И всё же, зачем?

- `$ sshfs` : проект сообщества, позволяет монтировать ФС через `$ ssh`
- Снижает поверхность атаки;
- Позволяет монтировать **почти любые образы дисков**, не имея прав администратора.

Спасибо за внимание!



🔗 github.com/JakMobius/courses/tree/main/mipt-os-basic-2024