

# Основы построения файловых систем



## Виртуальные FS в Linux: procfs

В Linux есть файловая система, каталоги в корне которой соответствуют исполняющимся процессам, а файлы внутри каждого каталога описывают состояние процесса.

```
artem@dev:~$ ls -lh /proc/self/  
total 0  
  
-r--r--r-- 1 artem artem 0 Oct  2 10:08 cmdline  
lrwxrwxrwx 1 artem artem 0 Oct  2 10:08 cwd -> /home/artem  
lrwxrwxrwx 1 artem artem 0 Oct  2 10:08 exe -> /bin/ls  
dr-x----- 2 artem artem 0 Oct  2 10:08 fd  
-r--r--r-- 1 artem artem 0 Oct  2 10:08 maps  
-r--r--r-- 1 artem artem 0 Oct  2 10:08 stat  
.....
```

Домашнее задание:

- `man 5 proc`,
- что находится в файле `/proc/PID/auxv` и как выглядит стек процесса сразу после `execve()`?
- напишите программу, которая спрячет первый аргумент командной строки из `/proc/PID/cmdline` (подсказка: `man prctl` + операция `PR_SET_MM`),
- напишите аналоги
  - `ps`
  - `lsof`

### Виртуальные FS в Linux: FUSE

FUSE (Filesystem in USer space) – это способ создания драйверов файловых систем, которые исполняются как пользовательские процессы, а не часть ядра.

Драйверы FUSE работают как сервера, которые обслуживают один pipe. Они читают оттуда команды вроде “найти элемент каталога по имени”, “открыть/закрыть файл”, “прочитать/записать данные в файл”. Клиентом такого сервера является ядро ОС.

Пример: sshfs.

## Виртуальные FS в Linux: FUSE

FUSE (Filesystem in USer space) – это способ создания драйверов файловых систем, которые исполняются как пользовательские процессы, а не часть ядра.

Драйверы FUSE работают как сервера, которые обслуживают один pipe. Они читают оттуда команды вроде “найти элемент каталога по имени”, “открыть/закрыть файл”, “прочитать/записать данные в файл”. Клиентом такого сервера является ядро ОС.

Пример: sshfs.

Преимущества FUSE:

- Позволяет легко экспериментировать,
- Реализации ФС могут иметь сложные зависимости, которые были бы нежелательны в ядре,
- Может использоваться непривилегированными пользователями.

Недостатки FUSE:

- Низкая производительность.

Домашнее задание:

- прочтите документацию по FUSE high level API,
- напишите драйвер для FUSE, который предоставляет ФС, состоящую из одного файла “hello”, из которого можно прочесть строку “hello, world!”; в каталоге, куда монтируется эта ФС, должны работать команды `ls` и `cat hello`.

## Memory-mapped files

```
int fd = open("file.txt", O_RDONLY);
char *str = mmap(NULL, length, PROT_READ, MAP_PRIVATE, fd, 0);

/* work with @str as if it were an array */
printf("%s\n", str);

munmap(str, length);
```

Как это работает?

### **Виртуальная память: зачем это надо?**

Процессы не имеют доступа к физической памяти.

Вместо этого, ОС предоставляют процессам линейное адресное пространство, которое может произвольно отображаться на физическую память.

## Виртуальная память: зачем это надо?

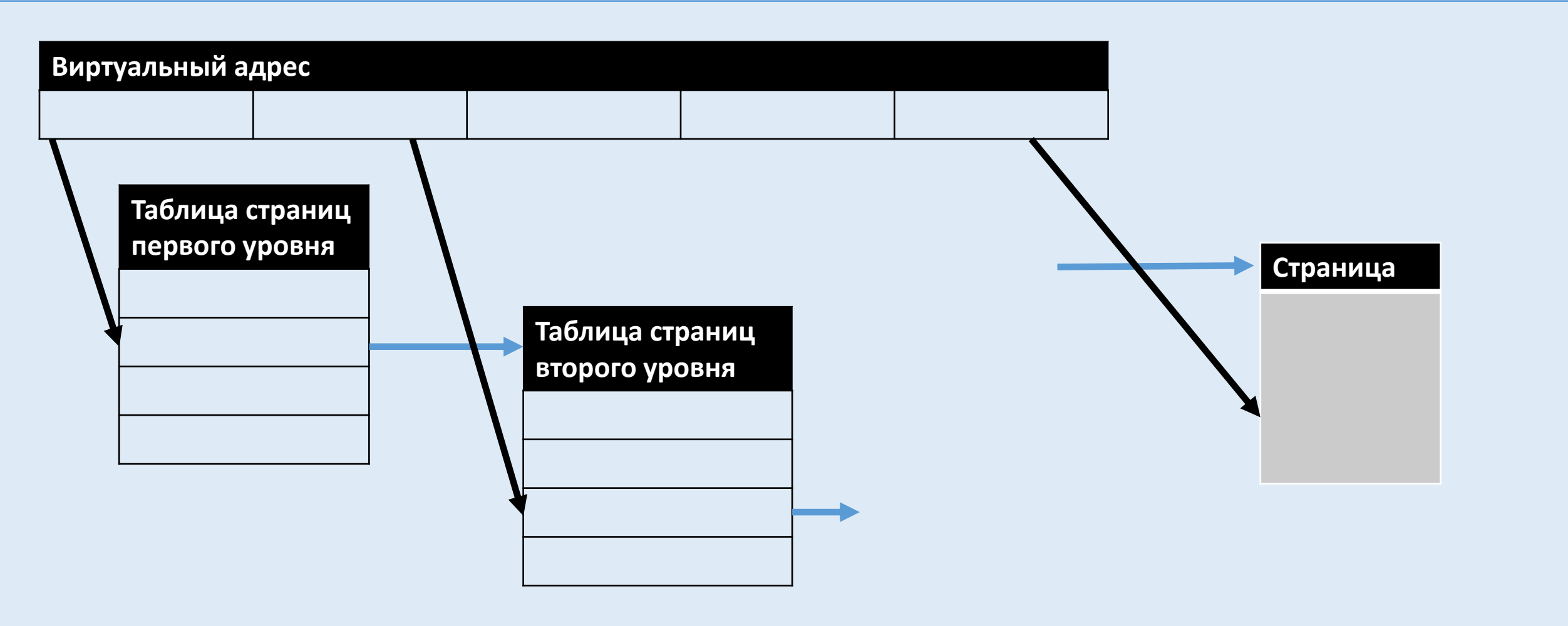
Процессы не имеют доступа к физической памяти.

Вместо этого, ОС предоставляют процессам линейное адресное пространство, которое может произвольно отображаться на физическую память.

Задачи, которые решает введение виртуального адресного пространства:

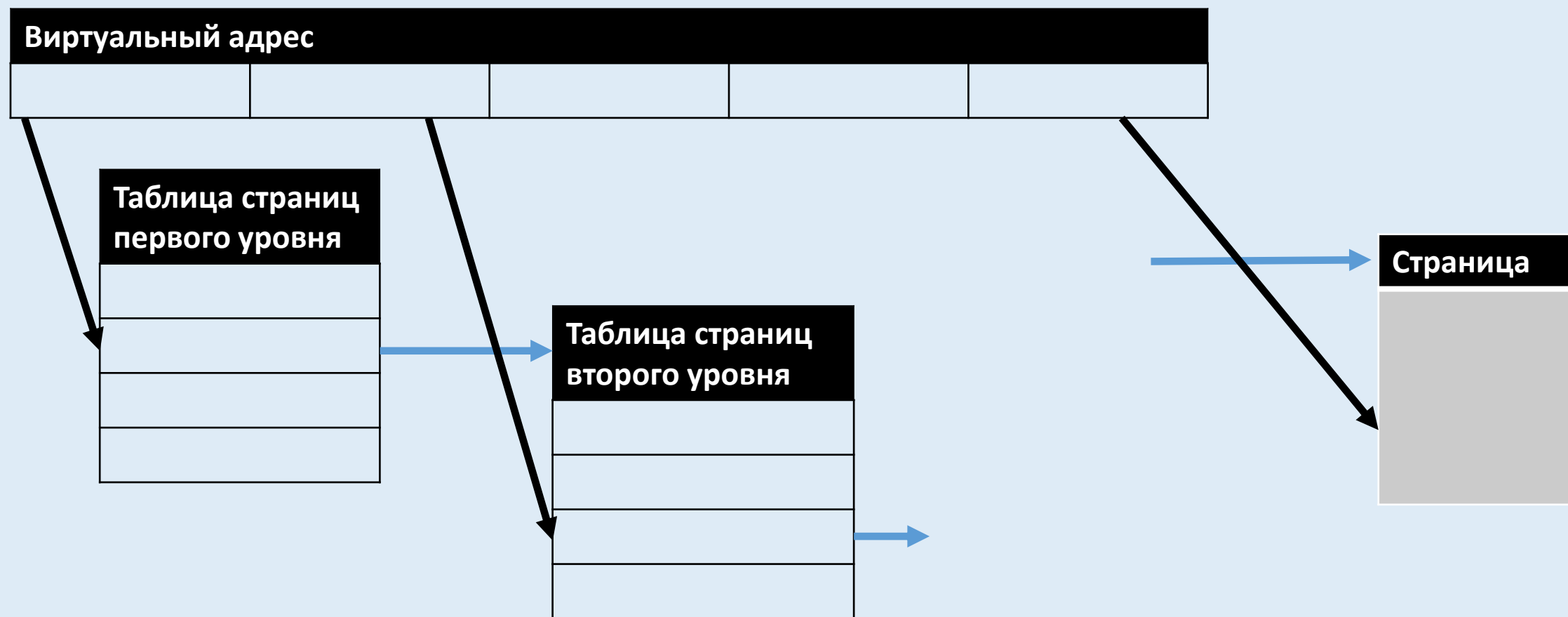
1. Возможность предоставить каждому процессу единообразное адресное пространство: процесс просто считает, что ему доступны все адреса в диапазоне  $[0, \text{MAX\_ADDR})$ ,
2. Изоляция процессов,
3. Возможность прозрачно разделять часть памяти между процессами (shared libraries, text segments, etc.),
4. Возможность «незаметно» для процесса заполнять/выгружать его части из памяти: memory-mapped files, swapping,

Виртуальная память с точки зрения CPU





## Виртуальная память с точки зрения CPU



- Таблицы разрешается заполнять частично, чтобы не тратить много памяти.
- Поиск по таблицам требует много обращений к памяти, поэтому результаты преобразований адресов кешируются в TLB (Translation Look-aside Buffer)

### Виртуальная память с точки зрения ОС

Для операционной системы память процесса представляется как набор VMA (Virtual Memory Area).

Каждая VMA указывает

- диапазон адресов,
- права доступа (и флаги вроде copy-on-write),
- правило, как подгружать страницы из данной VMA.

## Memory-mapped files: проблемы

Если файл виден как массив в памяти, то чтение и запись делаются очень просто.

Но как

1. увеличивать размер файла?
2. обрабатывать ошибки чтения из файла?
3. обрабатывать ошибки записи в файл?

## Memory-mapped files: проблемы

Если файл виден как массив в памяти, то чтение и запись делаются очень просто.

Но как

1. увеличивать размер файла?
2. обрабатывать ошибки чтения из файла?
3. обрабатывать ошибки записи в файл?

Ответ: никак.

До недавнего времени ошибки при отложенной записи (writeback) можно было легко потерять:

- <https://lwn.net/Articles/718734/>
- <http://stackoverflow.com/q/42434872/398670>

### Page cache и отложенная запись (writeback)

Аналогичные проблемы с записью есть и в POSIX API:

```
int fd = open("file.txt", O_RDWR);  
pwrite(fd, buf, size, 0);  
fsync(fd);  
close(fd);
```

## Page cache и отложенная запись (writeback)

Аналогичные проблемы с записью есть и в POSIX API:

```
int fd = open("file.txt", O_RDWR);  
pwrite(fd, buf, size, 0);  
fsync(fd);  
close(fd);
```

Вызов `pwrite()` не записывает данные в файл, а только помещает их в page cache.

Данные будут записаны на диск только после вызова `fsync()` или когда ОС решит сбросить page cache на диск.

- ошибки записи будут возвращены из `fsync()`
- `close()` тоже может завершаться с ошибкой.

### Page cache и отложенная запись (writeback)

`fsync()` и `fdatasync()`

- могут сказать, что записать данные не удалось,
- не указывают диапазон страниц, которые не удалось записать.

Как с этим бороться?

## Page cache и отложенная запись (writeback)

`fsync()` и `fdatasync()`

- могут сказать, что записать данные не удалось,
- не указывают диапазон страниц, которые не удалось записать.

Как с этим бороться?

Упорядочивать записи в файл:

1. записать новые данные,
2. `fsync()`,
3. записать заголовок, который ссылается на новые данные,
4. `fsync()`.



## Page cache и отложенная запись (writeback)

`fsync()` и `fdatasync()`

- могут сказать, что записать данные не удалось,
- не указывают диапазон страниц, которые не удалось записать.

Как с этим бороться?

Упорядочивать записи в файл:

1. записать новые данные,
2. `fsync()`,
3. записать заголовок, который ссылается на новые данные,
4. `fsync()`.

Как быть с перезаписями?

1. append-only files (только append и punch holes),
2. следить за использованием областей и перезаписывать только те, которые не используются.

POSIX API

open(const char \*path, int mode, int flags)

read(int fd, void \*buf, size\_t count)

write(int fd, const void \*buf, size\_t count)

close(int fd)

Windows API

```
HANDLE WINAPI CreateFile(
    _In_      LPCTSTR      lpFileName,
    _In_      DWORD        dwDesiredAccess,
    _In_      DWORD        dwShareMode,
    _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_      DWORD        dwCreationDisposition,
    _In_      DWORD        dwFlagsAndAttributes,
    _In_opt_  HANDLE       hTemplateFile
);
```

```
BOOL WINAPI ReadFile(
    _In_      HANDLE      hFile,
    _Out_     LPVOID       lpBuffer,
    _In_      DWORD        nNumberOfBytesToRead,
    _Out_opt_ LPDWORD      lpNumberOfBytesRead,
    _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

```
BOOL WINAPI WriteFile(
    _In_      HANDLE      hFile,
    _In_      LPCVOID      lpBuffer,
    _In_      DWORD        nNumberOfBytesToWrite,
    _Out_opt_ LPDWORD      lpNumberOfBytesWritten,
    _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

```
BOOL WINAPI CloseHandle(
    _In_ HANDLE hObject
);
```

POSIX API

open(const char \*path, int mode, int flags)

read(int fd, void \*buf, size\_t count)

write(int fd, const void \*buf, size\_t count)

close(int fd)

Windows API

```
HANDLE WINAPI CreateFile(  
_In_ LPCTSTR lpFileName,  
_In_ DWORD dwDesiredAccess,  
_In_ DWORD dwShareMode,  
_In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
_In_ DWORD dwCreationDisposition,  
_In_ DWORD dwFlagsAndAttributes,  
_In_opt_ HANDLE hTemplateFile  
);
```

```
BOOL WINAPI ReadFile(  
_In_ HANDLE hFile,  
_Out_ LPVOID lpBuffer,  
_In_ DWORD nNumberOfBytesToRead,  
_Out_opt_ LPDWORD lpNumberOfBytesRead,  
_Inout_opt_ LPOVERLAPPED lpOverlapped  
);
```

```
BOOL WINAPI WriteFile(  
_In_ HANDLE hFile,  
_In_ LPCVOID lpBuffer,  
_In_ DWORD nNumberOfBytesToWrite,  
_Out_opt_ LPDWORD lpNumberOfBytesWritten,  
_Inout_opt_ LPOVERLAPPED lpOverlapped  
);
```

```
BOOL WINAPI CloseHandle(  
_In_ HANDLE hObject  
);
```

### Синхронный и асинхронный ввод-вывод, `pipelining` и `multiplexing`

Рассмотрим пример: копирование файла с одного диска на другой.

Тривиальная реализация будет такой:

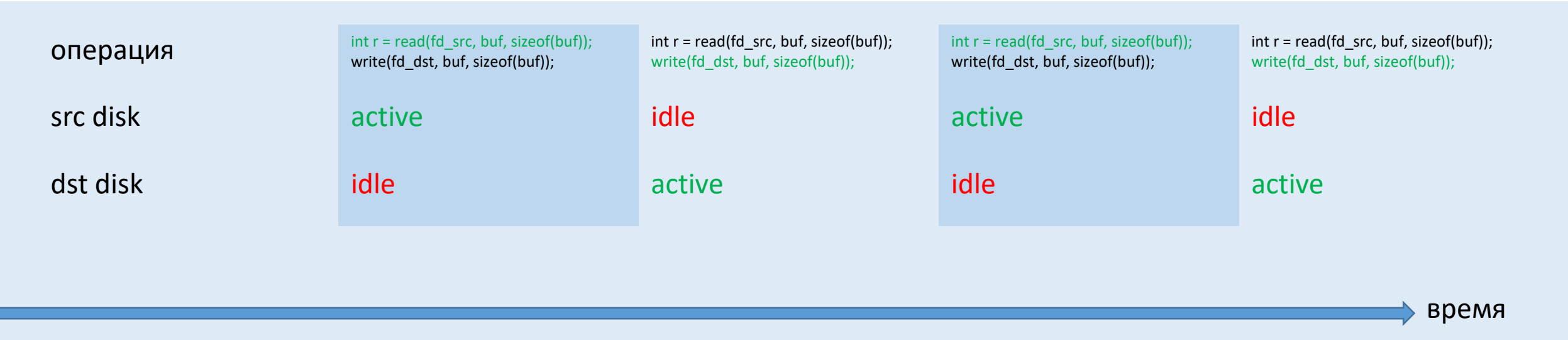
```
for (;;) {  
    int r = read(fd_src, buf, sizeof(buf));  
    write(fd_dst, buf, sizeof(buf));  
}
```

Эффективна ли она?

Синхронный и асинхронный ввод-вывод

Диск, если начал операцию, не прерывает её до тех пор, пока она не завершится.  
API для работы с файлами сохранили это же свойство – они не отдают управление, пока не завершатся.

```
for (;;) {  
    int r = read(fd_src, buf, sizeof(buf));  
    write(fd_dst, buf, sizeof(buf));  
}
```



## Синхронный и асинхронный ввод-вывод

Диск, если начал операцию, не прерывает её до тех пор, пока она не завершится.

API для работы с файлами сохранили это же свойство – они не отдают управление, пока не завершатся.

```
for (;;) {
    int r = read(fd_src, buf, sizeof(buf));
    write(fd_dst, buf, sizeof(buf));
}
```

операция

src disk

dst disk

```
int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));
```

active

idle

```
int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));
```

active

active

```
int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));
```

idle

active

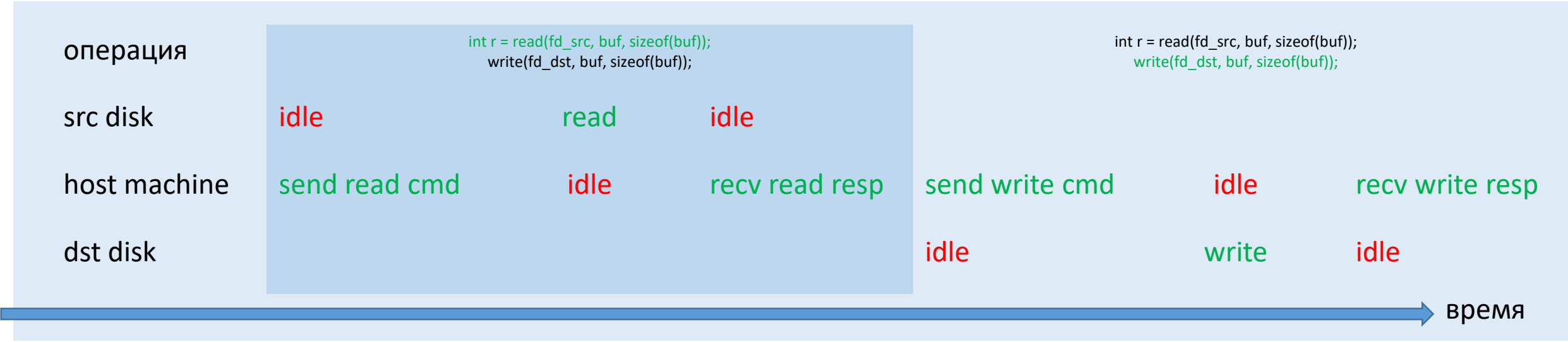
→ время

Решение: запросы на запись надо отправлять асинхронно и тут же переходить к чтению следующего блока.

Pipelining и multiplexing

Также надо учитывать время, которое требуется для того, чтобы отослать команду на чтение или запись. Оно было пренебрежимо мало для HDD, но оно велико для сетей и для SSD и NVMe.

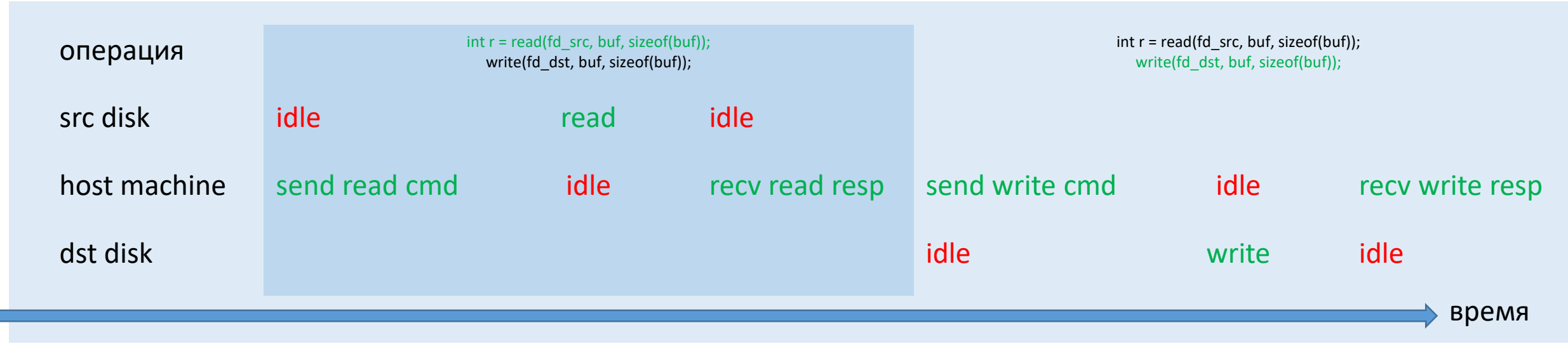
```
for (;;) {
  int r = read(fd_src, buf, sizeof(buf));
  write(fd_dst, buf, sizeof(buf));
}
```



Pipelining и multiplexing

Также надо учитывать время, которое требуется для того, чтобы отослать команду на чтение или запись. Оно было пренебрежимо мало для HDD, но оно велико для сетей и для SSD и NVMe.

```
for (;;) {
  int r = read(fd_src, buf, sizeof(buf));
  write(fd_dst, buf, sizeof(buf));
}
```



Решение: запросы на чтение надо отправлять в таком количестве, чтобы у диска всегда была непустая очередь команд. Первая команда всё равно увидит задержку на отправку запроса и получение ответа, но для последующих этой задержки не будет.



### Pipelining и head-of-line blocking

Предположим, что мы послали много запросов к диску (или к серверу). В каком порядке будут отсылаться ответы?

Есть два возможных варианта:

- в порядке получения запросов,
- в порядке завершения.

## Pipelining и head-of-line blocking

Предположим, что мы послали много запросов к диску (или к серверу). В каком порядке будут отсылаться ответы?

Есть два возможных варианта:

- в порядке получения запросов,
- в порядке завершения.

Первый вариант (pipelining) зачастую можно реализовать для протоколов, где изначально не позаботились о мультиплексировании.

Второй вариант требует поддержки в протоколе: у запросов должны быть уникальные номера.

Pipelining имеет существенный недостаток: если серверу были отправлены запросы  $R_1, R_2, \dots$ , то  $R_2$  и последующие должны ждать, пока закончится  $R_1$ . Если он окажется очень медленным, то все следующие за ним проведут много времени в очереди, даже если бы могли исполниться быстро. Такое явление называется head-of-line blocking.

## Pipelining и head-of-line blocking

Предположим, что мы послали много запросов к диску (или к серверу). В каком порядке будут отсылаться ответы?

Есть два возможных варианта:

- в порядке получения запросов,
- в порядке завершения.

Первый вариант (pipelining) зачастую можно реализовать для протоколов, где изначально не позаботились о мультиплексировании.

Второй вариант требует поддержки в протоколе: у запросов должны быть уникальные номера.

Pipelining имеет существенный недостаток: если серверу были отправлены запросы  $R_1, R_2, \dots$ , то  $R_2$  и последующие должны ждать, пока закончится  $R_1$ . Если он окажется очень медленным, то все следующие за ним проведут много времени в очереди, даже если бы могли исполниться быстро. Такое явление называется head-of-line blocking.

Дополнительное чтение:

- Google, “The QUIC Transport Protocol”, <https://research.google.com/pubs/archive/46403.pdf>
- Daniel Bernstein, “HTTP 2 explained”, <https://legacy.gitbook.com/book/bagder/http2-explained/details>