Основы построения файловых систем

Memory-mapped files

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?

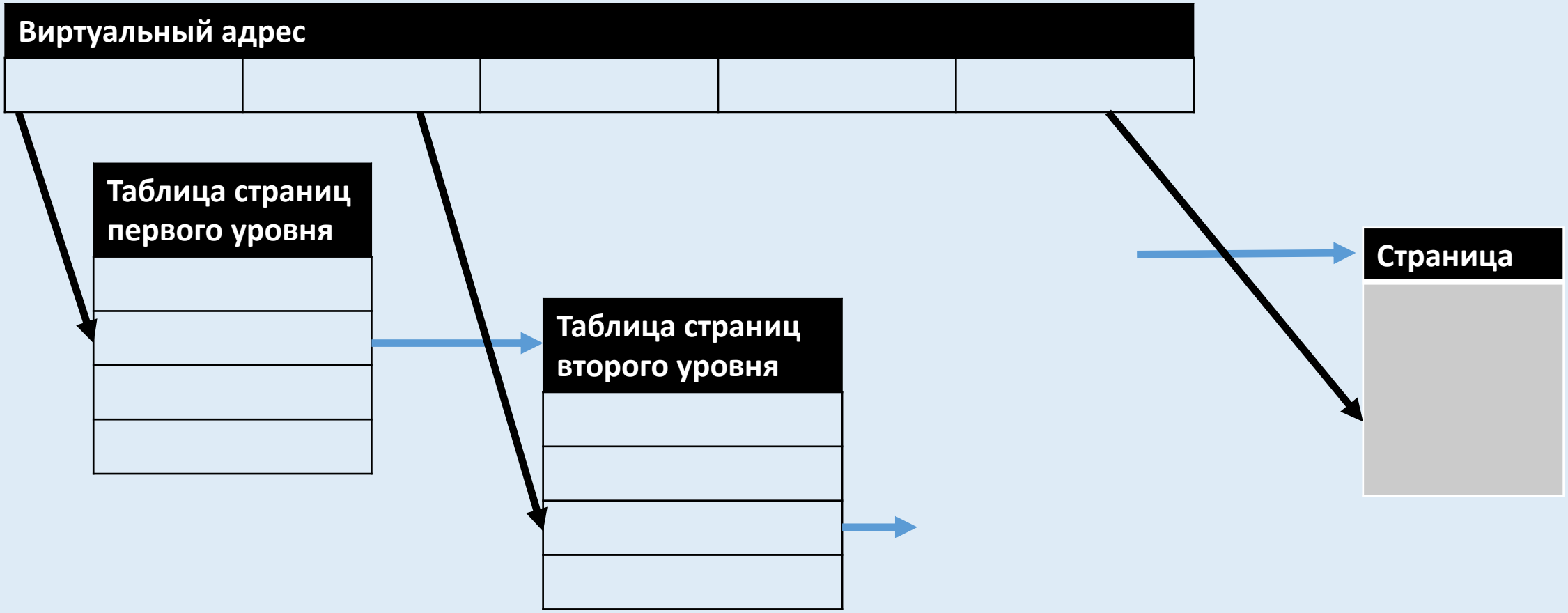
Процессы не имеют доступа к физической памяти.

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

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, 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



- 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

Вернёмся к POSIX API

Какой интерфейс хочется иметь:

```
f = open(“./pstorage-fes/src/fes.c");
read(f, buffer, size);
.....
write(f, buffer, size);
.....
close(f);

---

f = fopen(“./pstorage-fes/src/hello.txt”, “w”);
fprintf(f, “hello, world!\n”);
fclose(f);
```

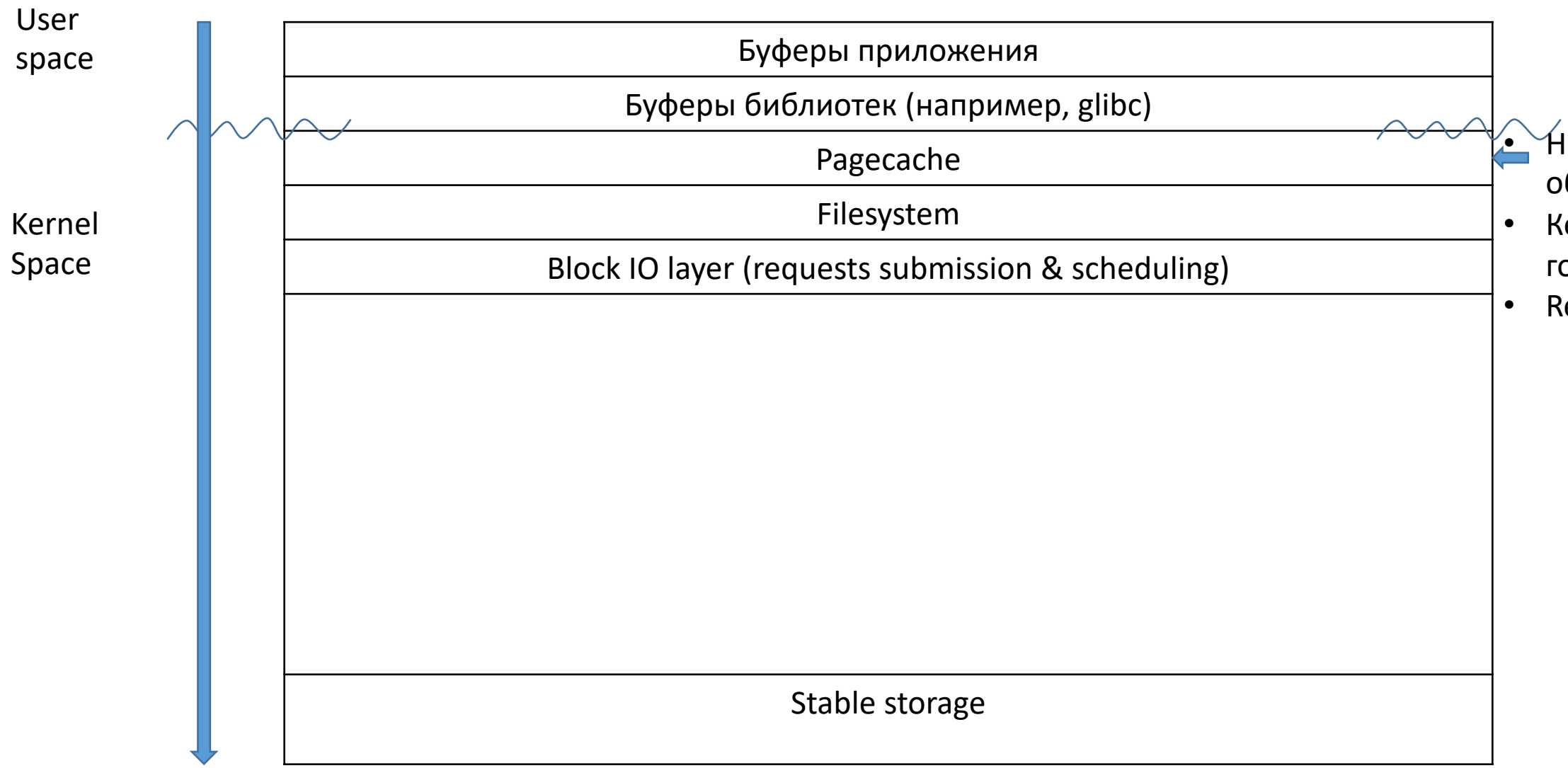
Какой интерфейс есть у жёсткого диска:

* прочесть сектор* номер N,
* записать сектор номер M.

** сектор – блок длиной 512 или 4096 байт*

В какой момент невыравненные чтения и записи превращаются в выравненные?

Путь данных от приложения до диска (обзорно)



Page cache и отложенная запись (writeback)

- Системные вызовы read()/write() копируют данные из буферов приложений в page cache – область памяти ядра, которая ведёт себя как memory mapped file.
- В отличие от инструкций процессора load/store, системные вызовы могут возвращать ошибки.
- С помощью системного вызова fsync() можно потребовать выписать все изменённые страницы page cache на диск.

Page cache и отложенная запись (writeback)

- Системные вызовы read()/write() копируют данные из буферов приложений в page cache – область памяти ядра, которая ведёт себя как memory mapped file.
- В отличие от инструкций процессора load/store, системные вызовы могут возвращать ошибки.
- С помощью системного вызова fsync() можно потребовать выписать все изменённые страницы page cache на диск.

Когда приложение узнает о возникновении ошибки записи?

Page cache и отложенная запись (writeback)

- Системные вызовы read()/write() копируют данные из буферов приложений в page cache – область памяти ядра, которая ведёт себя как memory mapped file.
- В отличие от инструкций процессора load/store, системные вызовы могут возвращать ошибки.
- С помощью системного вызова fsync() можно потребовать выписать все изменённые страницы page cache на диск.

Когда приложение узнает о возникновении ошибки записи?

- В момент fsync(),
- В момент write(), если свободной памяти мало и write() привёл к writeback,
- В момент close().

Page cache и отложенная запись (writeback)

- Системные вызовы read()/write() копируют данные из буферов приложений в page cache – область памяти ядра, которая ведёт себя как memory mapped file.
- В отличие от инструкций процессора load/store, системные вызовы могут возвращать ошибки.
- С помощью системного вызова fsync() можно потребовать выписать все изменённые страницы page cache на диск.

Когда приложение узнает о возникновении ошибки записи?

- В момент fsync(),
- В момент write(), если свободной памяти мало и write() привёл к writeback,
- В момент close().

Почему не делать IO сразу в вызове write()?

Page cache и отложенная запись (writeback)

- Системные вызовы read()/write() копируют данные из буферов приложений в page cache – область памяти ядра, которая ведёт себя как memory mapped file.
- В отличие от инструкций процессора load/store, системные вызовы могут возвращать ошибки.
- С помощью системного вызова fsync() можно потребовать выписать все изменённые страницы page cache на диск.

Когда приложение узнает о возникновении ошибки записи?

- В момент fsync(),
- В момент write(), если свободной памяти мало и write() привёл к writeback,
- В момент close().

Почему не делать IO сразу в вызове write()?

- Более длинные IO-запросы к блочному устройству,
- Delayed allocation,
- Etc.

Есть и проблемы: как ограничить время исполнения write() и 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?

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,
2. следить за использованием областей и перезаписывать только те, которые не используются.

Sparse files

Как быть с перезаписями?

1. append-only files <-- файл будет расти бесконечно? как удалять старые данные?
2. ...

Sparse files

Как быть с перезаписями?

1. append-only files <-- файл будет расти бесконечно? как удалять старые данные?
2. ...

POSIX API поддерживает операцию «заменить часть файла нулями». ФС вроде ext*, xfs, bsd ffs реализуют её эффективно и не хранят нули на диске.

Файлы с дырками называются sparse files.

Sparse files

Как быть с перезаписями?

1. append-only files,
2. ...

POSIX API поддерживает операцию «заменить часть файла нулями». ФС вроде ext*, xfs, bsd ffs реализуют её эффективно и не хранят нули на диске.

Файлы с дырками называются sparse files.

См. также:

1. fallocate(2),
2. logical & physical size в struct stat.

POSIX API

Windows API

POSIX API	Windows API
open(const char *path, int mode, int flags)	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);
read(int fd, void *buf, size_t count)	BOOL WINAPI ReadFile(_In_ HANDLE hFile, _Out_ LPVOID lpBuffer, _In_ DWORD nNumberOfBytesToRead, _Out_opt_ LPDWORD lpNumberOfBytesRead, _Inout_opt_ LPOVERLAPPED lpOverlapped);
write(int fd, const void *buf, size_t count)	BOOL WINAPI WriteFile(_In_ HANDLE hFile, _In_ LPCVOID lpBuffer, _In_ DWORD nNumberOfBytesToWrite, _Out_opt_ LPDWORD lpNumberOfBytesWritten, _Inout_opt_ LPOVERLAPPED lpOverlapped);
close(int fd)	BOOL WINAPI CloseHandle(_In_ HANDLE hObject);

POSIX API	Windows API
open(const char *path, int mode, int flags)	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);
read(int fd, void *buf, size_t count)	BOOL WINAPI ReadFile(_In_ HANDLE hFile, _Out_ LPVOID lpBuffer, _In_ DWORD nNumberOfBytesToRead, _Out_opt_ LPDWORD lpNumberOfBytesRead, _Inout_opt_ LPOVERLAPPED lpOverlapped);
write(int fd, const void *buf, size_t count)	BOOL WINAPI WriteFile(_In_ HANDLE hFile, _In_ LPCVOID lpBuffer, _In_ DWORD nNumberOfBytesToWrite, _Out_opt_ LPDWORD lpNumberOfBytesWritten, _Inout_opt_ LPOVERLAPPED lpOverlapped);
close(int fd)	BOOL WINAPI CloseHandle(_In_ HANDLE hObject);

Синхронный и асинхронный ввод-вывод, pipelining и multiplexing

Рассмотрим наивное копирование файла с одного диска на другой:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r1 = write(fd_out, buf, r);
  ...
}
```

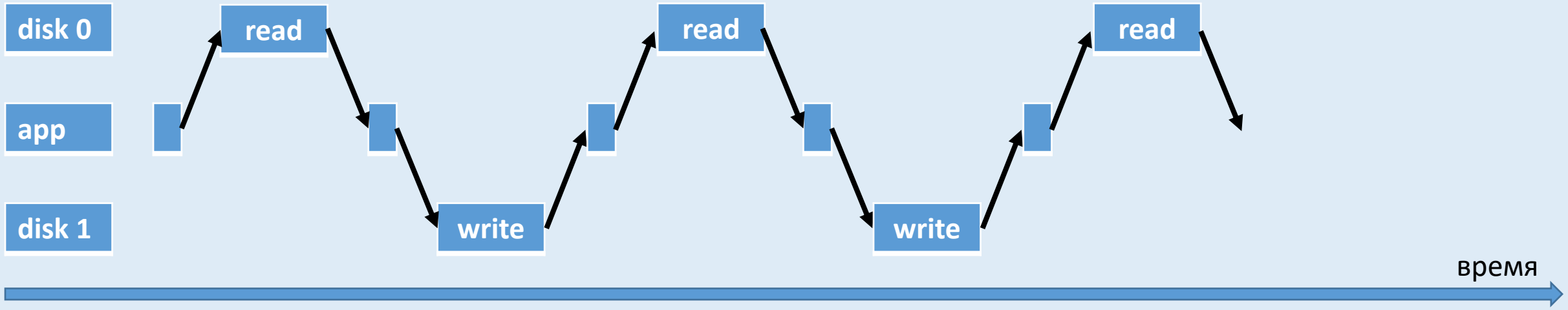
Будет ли оно работать эффективно?

Синхронный и асинхронный ввод-вывод, pipelining и multiplexing

Рассмотрим наивное копирование файла с одного диска на другой:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r1 = write(fd_out, buf, r);
  ...
}
```

Как расположены во времени обращения к дискам?

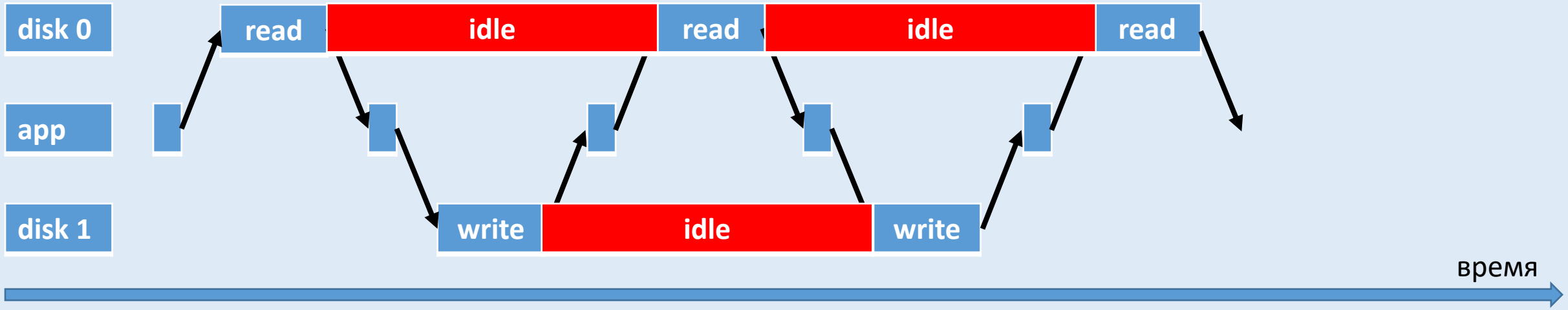


Синхронный и асинхронный ввод-вывод, pipelining и multiplexing

Рассмотрим наивное копирование файла с одного диска на другой:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r1 = write(fd_out, buf, r);
  ...
}
```

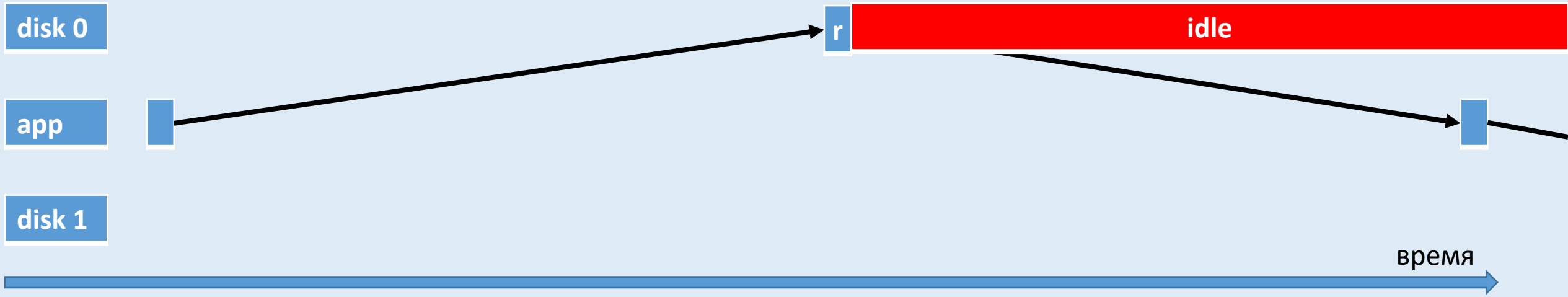
Как расположены во времени обращения к дискам?



Синхронный и асинхронный ввод-вывод, pipelining и multiplexing

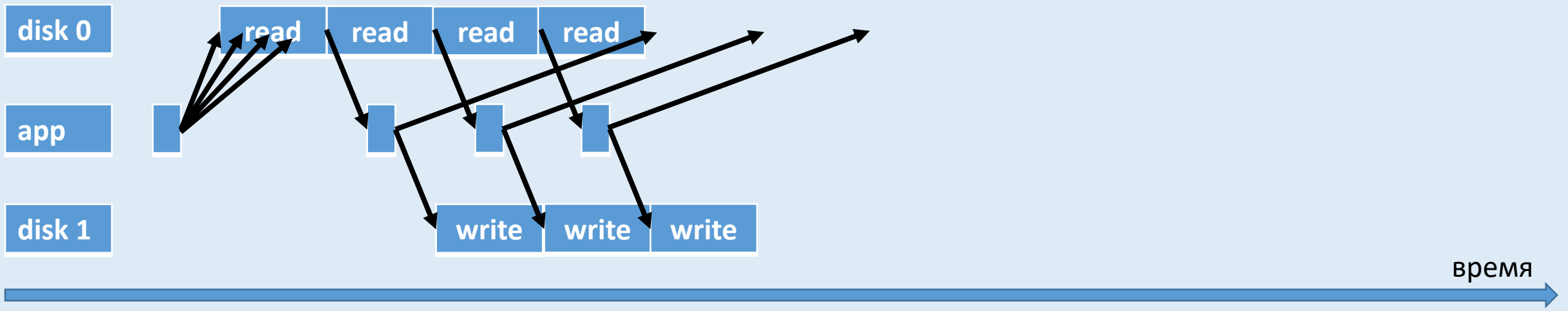
Обычно проблема хуже. Если ФС, между которыми делается копирование, сетевые, или расположены на быстрых NVMe-устройствах, то картина будет выглядеть так:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r1 = write(fd_out, buf, r);
  ...
}
```



Синхронный и асинхронный ввод-вывод, pipelining и multiplexing

Улучшение: 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.

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, …, 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.

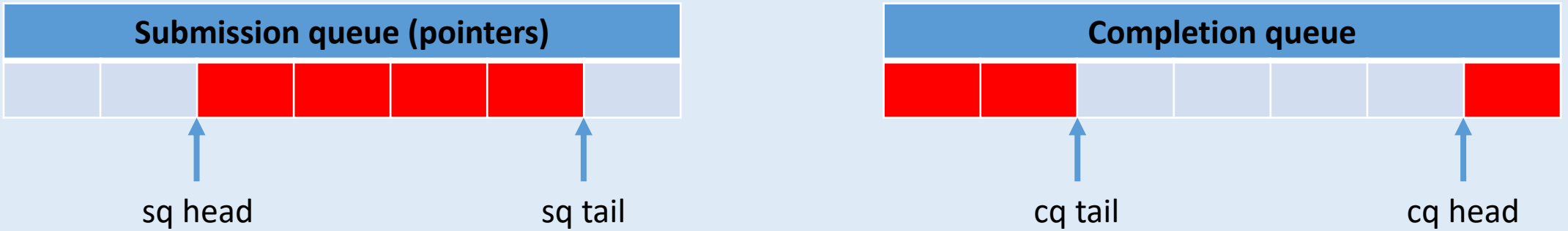
Второй вариант требует поддержки в протоколе: у запросов должны быть уникальные номера.

Pipelining имеет существенный недостаток: если серверу были отправлены запросы R_1, R_2, …, 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

Асинхронный ввод-вывод в Linux (io_uring)

Приложение может поддерживать разделяемую с ядром область памяти с двумя кольцевыми буферами:

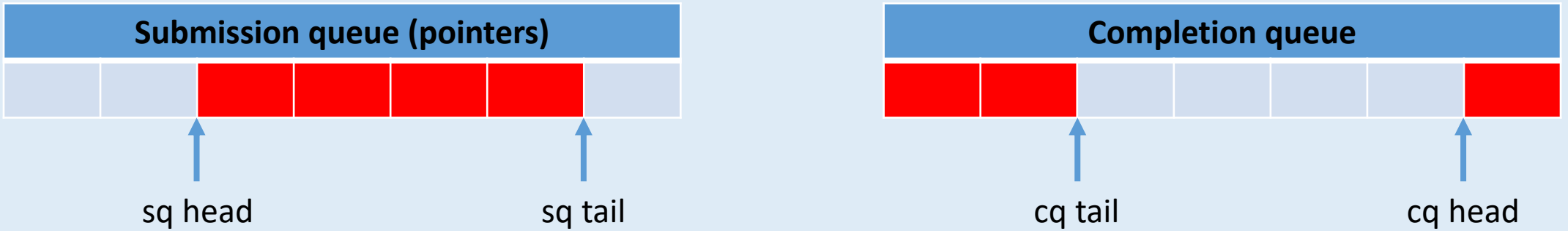


Приложение помещает в submission queue описание IO-операций, которые надо исполнить, продвигает sq tail и просит ядро исполнить IO. Ядро по мере обработки запросов продвигает sq head.

Когда IO завершается, ядро сохраняет добавляет запись в completion queue и продвигает cq tail.

Асинхронный ввод-вывод в Linux (io_uring)

Приложение может поддерживать разделяемую с ядром область памяти с двумя кольцевыми буферами:



Приложение помещает в submission queue описание IO-операций, которые надо исполнить, продвигает sq tail и просит ядро исполнить IO. Ядро по мере обработки запросов продвигает sq head.

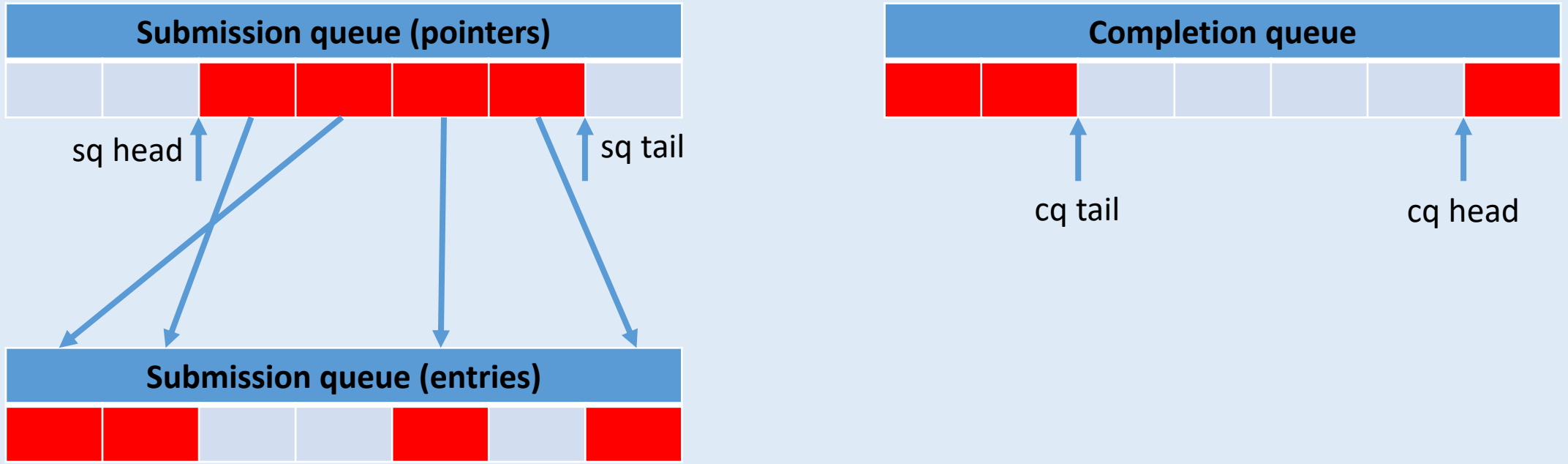
Когда IO завершается, ядро сохраняет добавляет запись в completion queue и продвигает cq tail.

Вопрос: в pread(), pwrite() и прочие можно было бы добавить аналог параметра lpOverlapped из Windows API. Чем лучше схема с кольцевыми буферами?

Вопрос: что будет, если IO-операция, стоящая в голове submission queue, будет исполняться дольше всех остальных?

Асинхронный ввод-вывод в Linux (io_uring)

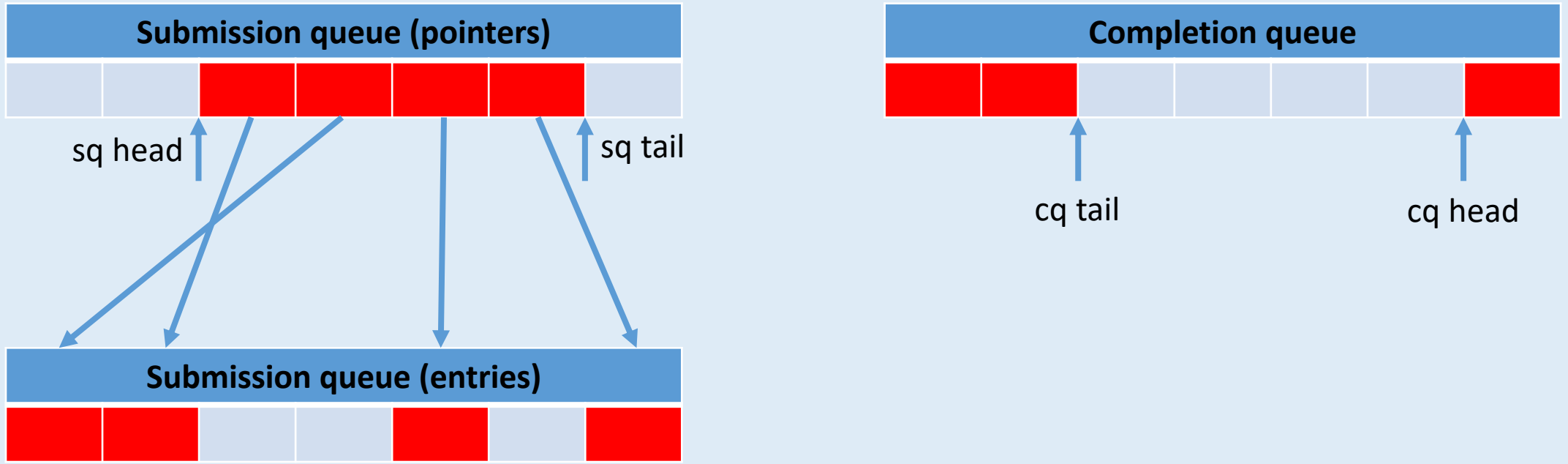
Приложение может поддерживать разделяемую с ядром область памяти с двумя кольцевыми буферами:



Если бы submission queue состояла именно из запросов на ввод-вывод, ядро могло бы продвигать sq head только когда завершится IO-запрос из головы списка. Это приводило бы к head-of-line blocking.

Асинхронный ввод-вывод в Linux (io_uring)

Приложение может поддерживать разделяемую с ядром область памяти с двумя кольцевыми буферами:



Домашнее задание:

1. https://lwn.net/Articles/776703/,
2. Изучите API liburing https://github.com/axboe/liburing,
3. Напишите cp, который работает следующим образом:
    1. испустить N последовательных запросов на чтение, N = 4 или N = 8, длина запроса – 256K или 512K,
    2. когда исполнится чтение №0, испустить запрос на запись и ещё один запрос на чтение,
    3. когда исполнится чтение №1, ….
    4. и т.д.