The basics of file systems

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);
```

How does this work?

Virtual memory: why do we need it?

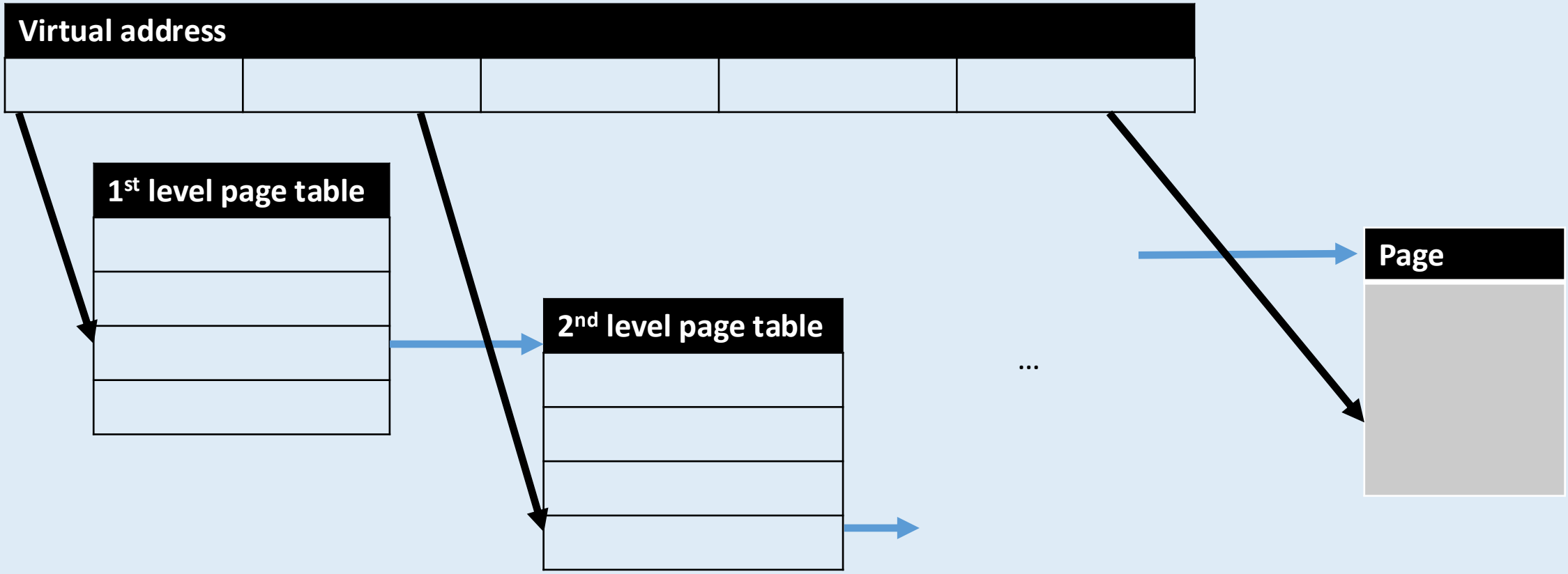
Processes have no direct access to the physical RAM in a computer.

Instead, the OS supplies them with a virtual address space. Parts of the virtual address space are mapped to the physical memory in a way that is most convenient for the OS.

What are the benefits of virtual address spaces:

1. The address space of a process is very simple, and every process has the same address space. Essentially, a process believes it can use all addresses in the range `[0, MAX_ADDR)`.
2. Processes are isolated one from another.
3. Processes can share memory without even knowing that. For example, it suffices to load shared libraries into the memory only once, and then share them between processes.
4. The memory of a process can be partially swapped out in a way that is transparent to the process.

Virtual memory from the point of view of a CPU



- A virtual address is split into several segments. Each segment is used as an offset into a page table or a page.
- Tables may be filled only partially to avoid wasting memory.
- Address lookups need multiple access to the RAM, so their results are cached by the TLB (Translation Look-aside Buffer).
- See also https://lwn.net/Articles/888914/ about pointer tagging.

TLBs and the latency of munmap()

- Address lookups need multiple access to the RAM, so their results are cached by the TLB (Translation Look-aside Buffer).

TLB entries become stale when `munmap()` or `mprotect()` change the page table of a process. In a multithreaded process they need to send an IPI to inform other cores that their TLB entries became invalid.

TLBs and the latency of munmap()

- Address lookups need multiple access to the RAM, so their results are cached by the TLB (Translation Look-aside Buffer).

TLB entries become stale when `munmap()` or `mprotect()` change the page table of a process. In a multithreaded process they need to send an IPI to inform other cores that their TLB entries became invalid.

Quiz: why does `mmap()` not need an IPI?

TLBs and the latency of munmap()

- Address lookups need multiple access to the RAM, so their results are cached by the TLB (Translation Look-aside Buffer).

TLB entries become stale when `munmap()` or `mprotect()` change the page table of a process. In a multithreaded process they need to send an IPI to inform other cores that their TLB entries became invalid.

Quiz: why does `mmap()` not need an IPI?

Experiment:

1. write a program that `mmap()`s a region of memory and immediately `munmap()`s it in a loop,
2. measure the duration of calls to `mmap()` and `munmap()`,
3. modify the program to start several threads that run a `mmap()`/`munmap()` loop,
4. modify both versions to access the `mmap()`ed region of memory.

To measure the syscall latency use

```
% perf record -e 'syscalls:sys_*_mmap,syscalls:sys_*_munmap' ./test_program
% perf script --deltatime --ns --tid=<tid of one of threads>
```

See also:

- https://lwn.net/Articles/893906/
- https://lwn.net/Articles/932298/
- https://lwn.net/Articles/974392/
- https://lwn.net/Articles/845507/

Virtual memory from the point of view of an OS

Linux represents the memory of a process as a list of VMAs (Virtual Memory Area).

Each VMA tracks

- the range of addresses that it describes,
- the access rights of the memory region (and flags like copy-on-write),
- the rule which describes how to page-in pages in this memory range (from a file, from the swap, from userfaultfd, etc.).

Memory overcommit

Linux represents the memory of a process as a list of VMAs (Virtual Memory Area).

Each VMA tracks
- the range of addresses that it describes,
- the access rights of the memory region (and flags like copy-on-write),
- the rule which describes how to page-in pages in this memory range (from a file, from the swap, from userfaultfd, etc.).

VMAs need not allocate backing physical pages immediately. They can be faulted in as the application accesses its memory.

In particular, the size of VMAs of a process may be greater than the amount of physical RAM reserved for the process.
This is called **overcommit**. This is the default behaviour of Linux.

See also:
- sysctl vm.overcommit_ratio,
- /proc/sys/vm/overcommit_ratio.

Memory overcommit

Linux represents the memory of a process as a list of VMAs (Virtual Memory Area).

Each VMA tracks
- the range of addresses that it describes,
- the access rights of the memory region (and flags like copy-on-write),
- the rule which describes how to page-in pages in this memory range (from a file, from the swap, from userfaultfd, etc.).

VMAs need not allocate backing physical pages immediately. They can be faulted in as the application accesses its memory.

In particular, the size of VMAs of a process may be greater than the amount of physical RAM reserved for the process.
This is called **overcommit**. This is the default behaviour of Linux.

See also:
- sysctl vm.overcommit_ratio,
- /proc/sys/vm/overcommit_ratio.

Even though Linux overcommits memory, a call to `malloc()` or `mmap()` may return `NULL`. The number of VMAs per process is limited, and it is often easy to reach that limit.

See also:
- sysctl vm.max_map_count,
- /proc/sys/vm/max_map_count.

Usability problems of memory-mapped files

When a file is visible as an array in memory, it is very easy to read it and to write it.

But how does one

1. grow or shrink the file?
2. handle read errors?
3. handle write errors?

With memory-mapped files there is no way to do that.

Usability problems of memory-mapped files

When a file is visible as an array in memory, it is very easy to read it and to write it.

But how does one

1. grow or shrink the file?
2. handle read errors?
3. handle write errors?

With memory-mapped files there is no way to do that.

Until recently, it was even possible to lose writeback errors:

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

Let us get back to the POSIX API

The basics of file systems

The desired interface to a file system:

```
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);
```

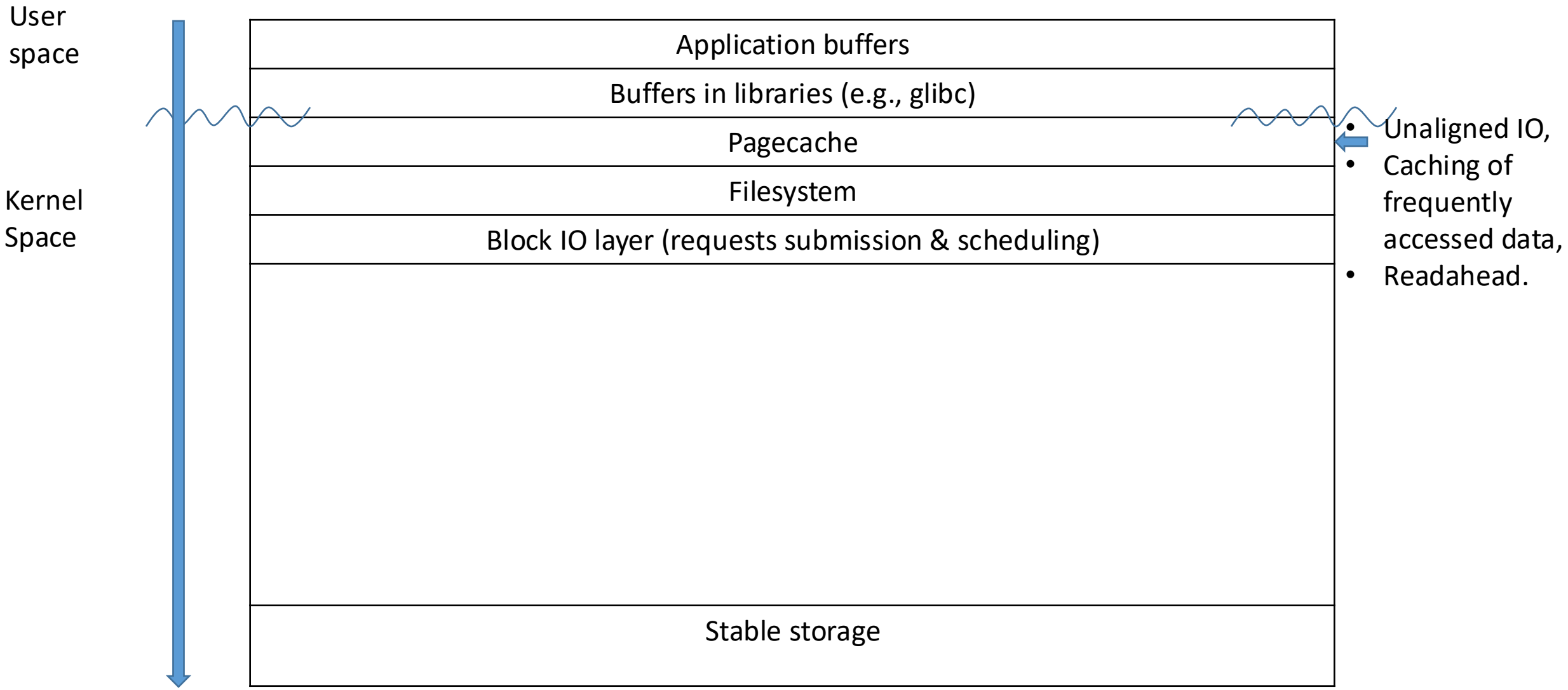
The interface of a storage device:

* read a sector* nr. N,
* write a sector* nr. M.

** a sector is a contiguous piece of a storage device that is 512 bytes or 4096 bytes long; the start of a sector is a multiple of the sector size*

When are unaligned reads and write replaced with aligned reads and writes?

The path of writes from applications to a disk (very approximate)



The basics of file systems

What does this program print?

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    fprintf(stdout, "hello");
    fork();
    return 0;
}
```

hellohello

And this one?

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    fprintf(stderr, "hello");
    fork();
    return 0;
}
```

hello

How does this happen?

The basics of file systems

What does this program print?

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
   fprintf(stdout, "hello");
   fork();
   return 0;
}
```

hellohello

How does this happen?

Glibc buffers writes to stdout. A call to `fork()` produces a copy of this buffer in the new process.

Upon exit, each of the 2 processes flush the buffer to file descriptor 1.

And this one?

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
   fprintf(stderr, "hello");
   fork();
   return 0;
}
```

hello

Stderr is not buffered. A call to `fprintf()` writes the message to file descriptor 2 immediately. A process forks when it has nothing more to write.

The basics of file systems

What does this program print?

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    fprintf(stdout, "hello");
    fork();
    return 0;
}
```

hellohello

And this one?

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    fprintf(stderr, "hello");
    fork();
    return 0;
}
```

hello

How does this happen?

Quiz: what happens to mutexes when a multithreaded process calls `fork()`?

Page cache and writeback

- System calls like `read()` and `write()` copy data between application buffers and the page cache in the kernel. The page cache works like a memory-mapped file. File pages are faulted into the page cache when accessed.
- Unlike CPU instruction that load and store values, system calls have a return value and can signal IO errors.
- By calling `fsync()`, an application requests the page cache to write all dirty (modified) pages to the disk.

Page cache and writeback

- System calls like `read()` and `write()` copy data between application buffers and the page cache in the kernel. The page cache works like a memory-mapped file. File pages are faulted into the page cache when accessed.
- Unlike CPU instruction that load and store values, system calls have a return value and can signal IO errors.
- By calling `fsync()`, an application requests the page cache to write all dirty (modified) pages to the disk.

When can an application detect a write error?

Page cache and writeback

- System calls like `read()` and `write()` copy data between application buffers and the page cache in the kernel. The page cache works like a memory-mapped file. File pages are faulted into the page cache when accessed.
- Unlike CPU instruction that load and store values, system calls have a return value and can signal IO errors.
- By calling `fsync()`, an application requests the page cache to write all dirty (modified) pages to the disk.

When can an application detect a write error?

- When calling `fsync()`,
- When calling `write()`, if the system is low on RAM and `write()` triggers writeback,
- (Possibly) when calling* `close()`.

** This is a very unfortunate API design. The only sure way is to call `fsync()` before `close()`.*

Page cache and writeback

- System calls like `read()` and `write()` copy data between application buffers and the page cache in the kernel. The page cache works like a memory-mapped file. File pages are faulted into the page cache when accessed.
- Unlike CPU instruction that load and store values, system calls have a return value and can signal IO errors.
- By calling `fsync()`, an application requests the page cache to write all dirty (modified) pages to the disk.

When can an application detect a write error?

- When calling `fsync()`,
- When calling `write()`, if the system is low on RAM and `write()` triggers writeback,
- (Possibly) when calling* `close()`.

Why do OSes not do IO immediately in `write()` itself?

** This is a very unfortunate API design. The only sure way is to call `fsync()` before `close()`.*

Page cache and writeback

- System calls like `read()` and `write()` copy data between application buffers and the page cache in the kernel. The page cache works like a memory-mapped file. File pages are faulted into the page cache when accessed.
- Unlike CPU instruction that load and store values, system calls have a return value and can signal IO errors.
- By calling `fsync()`, an application requests the page cache to write all dirty (modified) pages to the disk.

When can an application detect a write error?

- When calling `fsync()`,
- When calling `write()`, if the system is low on RAM and `write()` triggers writeback,
- (Possibly) when calling* `close()`.

Why do OSes not do IO immediately in `write()` itself?

- Multiple writes or re-writes to a file may be coalesced into a single write to a disk,
- It becomes possible to issue fewer IO requests to a disk, and make them longer,
- File systems can do “delayed allocation” to decrease the fragmentation of files.

Delaying the IO also has downsides. For example, it may introduce thrashing and make it impossible to control the time it takes to call `write()` and `fsync()`.

** This is a very unfortunate API design. The only sure way is to call `fsync()` before `close()`.*

Page cache and writeback

`fsync()` and `fdatasync()`

- report whether a writeback succeeded or failed,
- do **not** specify ranges that were written successfully and ranges that failed*.

How do we design our file formats so that an error “some (unspecified) writes failed” can be handled?

Quiz: if a program overwrites a region in a file, what can we assume about the region if `fsync()` fails?

Page cache and writeback

`fsync()` and `fdatasync()`

- report whether a writeback succeeded or failed,
- do **not** specify ranges that were written successfully and ranges that failed.

How do we design our file formats so that an error “some (unspecified) writes failed” can be handled?

We can split our data into two parts. The data files itself and metadata files that keep (small) pointers to data. With this arrangement we can order the writes this way:

1. write a new data file,
2. `fsync()` the data file,
3. write a header that points to the new data:
    a. append a new header to the metadata and `fsync()`,
    b. write a temporary metadata file, `fsync()`, and `rename()` it atop the previous metadata file.

POSIX API

Windows API

POSIX API	Windows API

`size_t 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`
	`);`

`size_t 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`
	`);`

POSIX API	Windows API

`size_t 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`
	`);`

`size_t 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`
	`);`

Synchronous and asynchronous IO, pipelining and multiplexing

Consider a naïve implementation of a routine that copies a file from one disk to another:

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

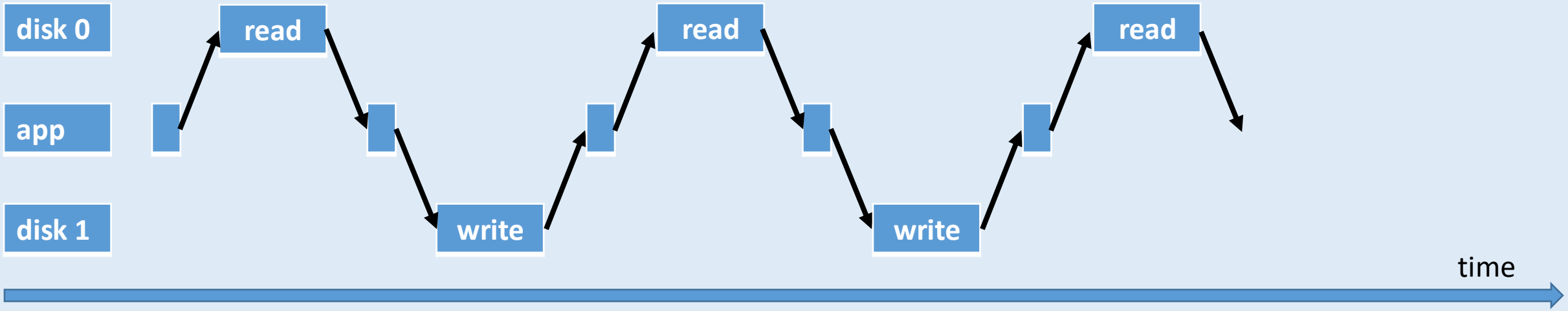
Is the performance of this routine any good?

Synchronous and asynchronous IO, pipelining and multiplexing

Consider a naïve implementation of a routine that copies a file from one disk to another:

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

Let us draw time intervals when each disk is accessed:

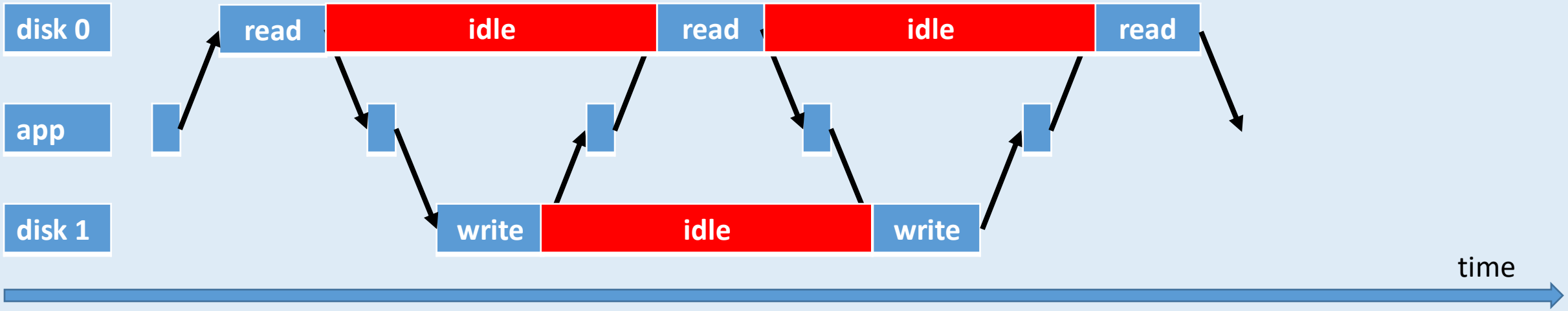


Synchronous and asynchronous IO, pipelining and multiplexing

Consider a naïve implementation of a routine that copies a file from one disk to another:

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

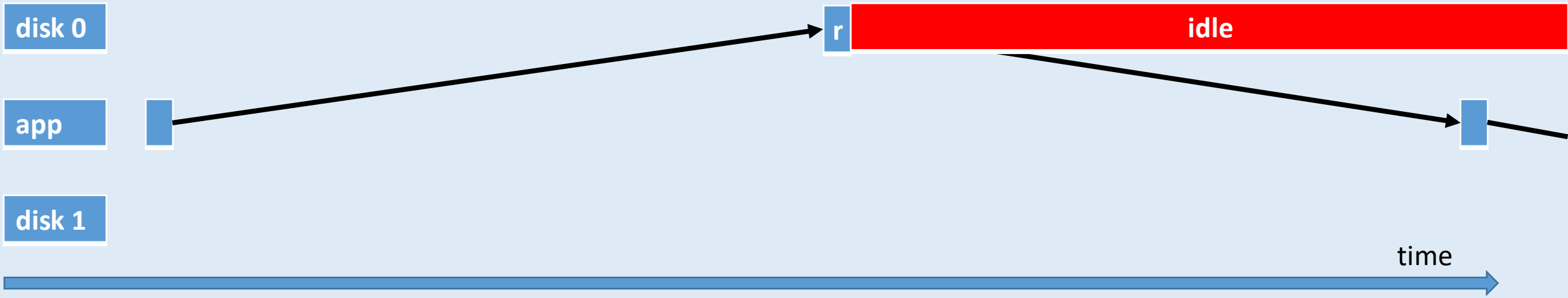
Let us draw time intervals when each disk is accessed:



Synchronous and asynchronous IO, pipelining and multiplexing

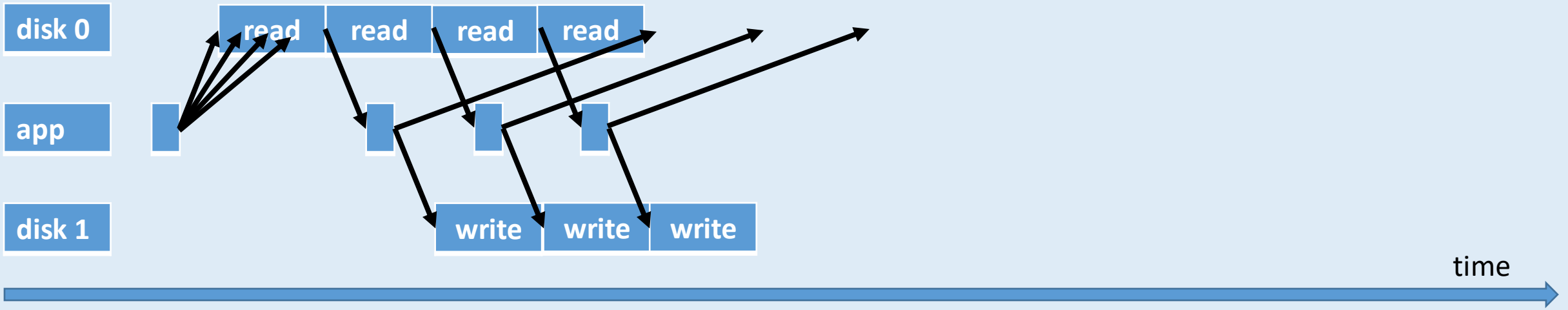
Typically, the problem is even worse. If the FS is networked, or the FS is located on a fast NVMe device, then the timeline is going to look this way:

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



Synchronous and asynchronous IO, pipelining and multiplexing

An improvement: issue multiple read requests so that the source disk always have some work to do. The first command still suffers the latency penalty, but subsequent requests have their issue latency masked by preceding requests.



Pipelining and head-of-line blocking

Suppose we have sent multiple requests to a server. What are the options for the order of replies?

1. the order of replies matches the order of requests,
2. replies may be reordered and sent as soon as a request completes.

The first options is called pipelining. It can be added to almost any protocol*.

The second option is possible if requests have unique numbers.

** Yet, HTTP 1 has no pipeling. Why?*

Pipelining and head-of-line blocking

Suppose we have sent multiple requests to a server. What are the options for the order of replies?

1. the order of replies matches the order of requests,
2. replies may be reordered and sent as soon as a request completes.

The first options is called pipelining. It can be added to almost any protocol.

The second option is possible if requests have unique numbers.

Pipelining has an important inefficiency. Suppose that we've issued requests R_1, R_2, …. The request R_2 can send the reply only after R_1 even if R_2 completes much sooner than R_1. Thus, a slow request blocks all subsequent requests. This scenario is called head-of-line blocking.

Pipelining and head-of-line blocking

Suppose we have sent multiple requests to a server. What are the options for the order of replies?

1. the order of replies matches the order of requests,
2. replies may be reordered and sent as soon as a request completes.

The first options is called pipelining. It can be added to almost any protocol.

The second option is possible if requests have unique numbers.

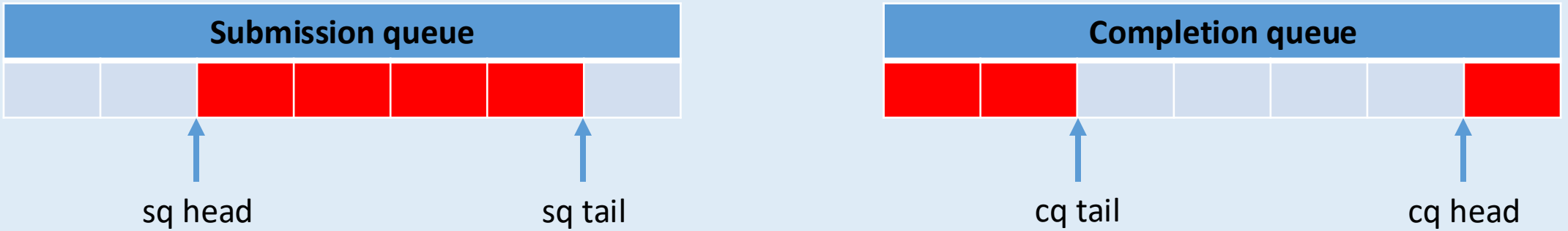
Pipelining has an important inefficiency. Suppose that we've issued requests R_1, R_2, …. The request R_2 can send the reply only after R_1 even if R_2 completes much sooner than R_1. Thus, a slow request blocks all subsequent requests. This scenario is called head-of-line blocking.

To read at home:

- Google, "The QUIC Transport Protocol", https://research.google.com/pubs/archive/46403.pdf

The Linux approach to asynchronous IO (io_uring)

An application and the kernel share a memory are that contains two circular buffers:

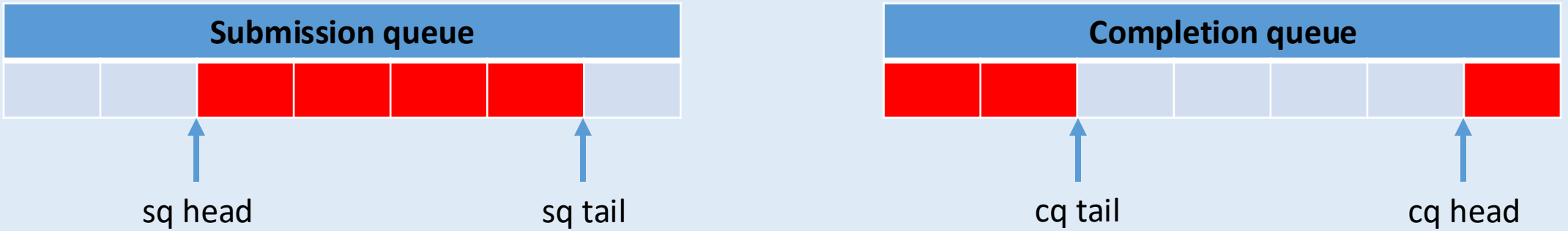


The application places IO requests to the tail of the submission queue, advances the sq tail and requests the kernel to execute the IO. The kernel advances the sq head as it consumes requests.

When an IO request completes, the kernel writes the result to the tail of the completion queue and advances the cq tail.

The Linux approach to asynchronous IO (io_uring)

An application and the kernel share a memory are that contains two circular buffers:



The application places IO requests to the tail of the submission queue, advances the sq tail and requests the kernel to execute the IO. The kernel advances the sq head as it consumes requests.

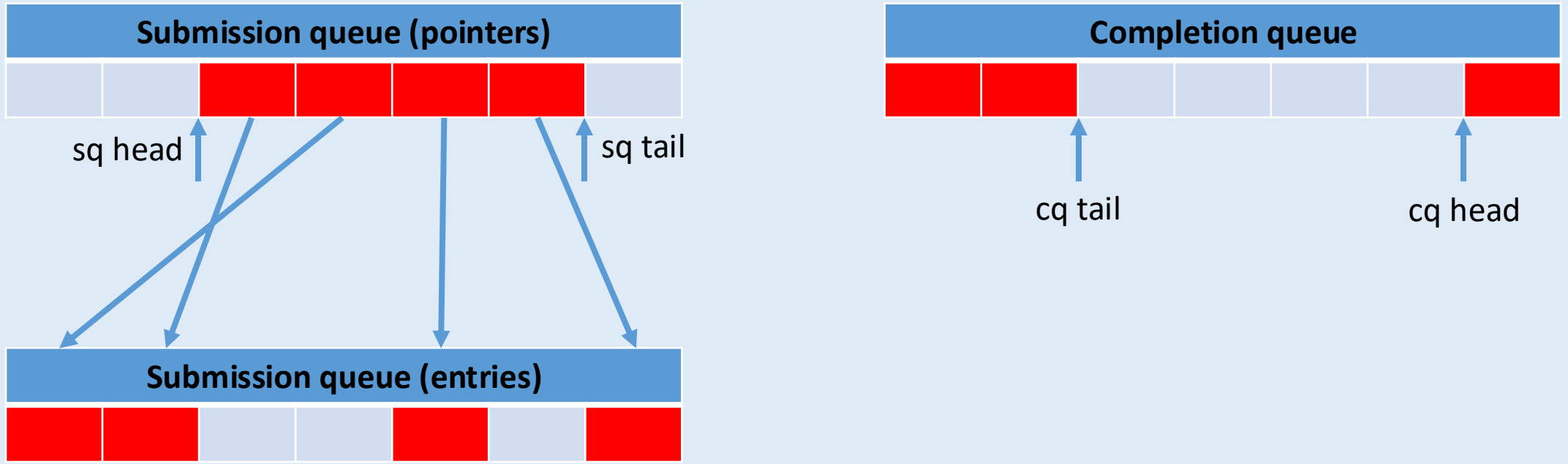
When an IO request completes, the kernel writes the result to the tail of the completion queue and advances the cq tail.

Quiz: one could make `pread()` and `pwrite()` asynchronous by adding an argument `lpOverlapped` like in the Windows API. Why is the mechanism with circular buffers preferable?

Quiz: what happens if an IO request at the head submission queue takes more time than all following IO requests?

The Linux approach to asynchronous IO (io_uring)

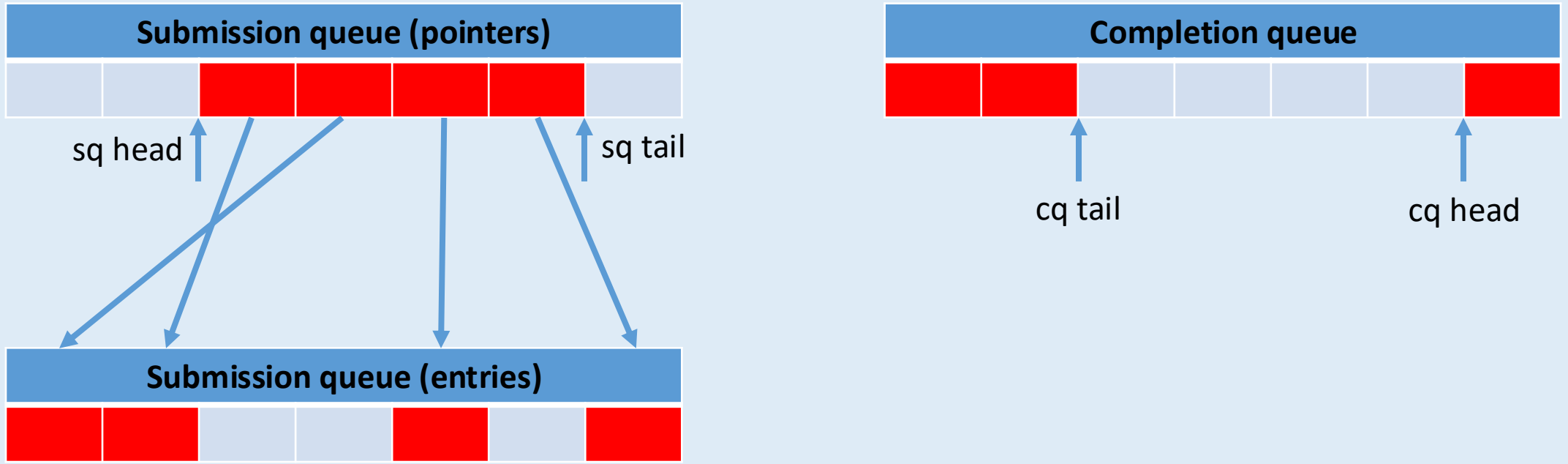
An application and the kernel share a memory are that contains two circular buffers:



To avoid the head-of-line blocking, the submission queue contains only pointers to structs with request arguments. This way, the sq head may be advances as soon as the kernel begins executing an IO request.

The Linux approach to asynchronous IO (io_uring)

An application and the kernel share a memory are that contains two circular buffers:



To do at home:

1. Read https://lwn.net/Articles/776703/,
2. Learn the API of liburing https://github.com/axboe/liburing,
3. Write an implementation of cp that works this way:
   a. start by issuing N read requests (N = 4 or N = 8), each request being 256K or 512K long,
   b. when the read #0 completes, issue a write request,
   c. when the read #1 completes, ….
   d. as write requests complete and buffers become available for reuse, issue more reads.