INF113: Long-Term Storage

Kirill Simonov

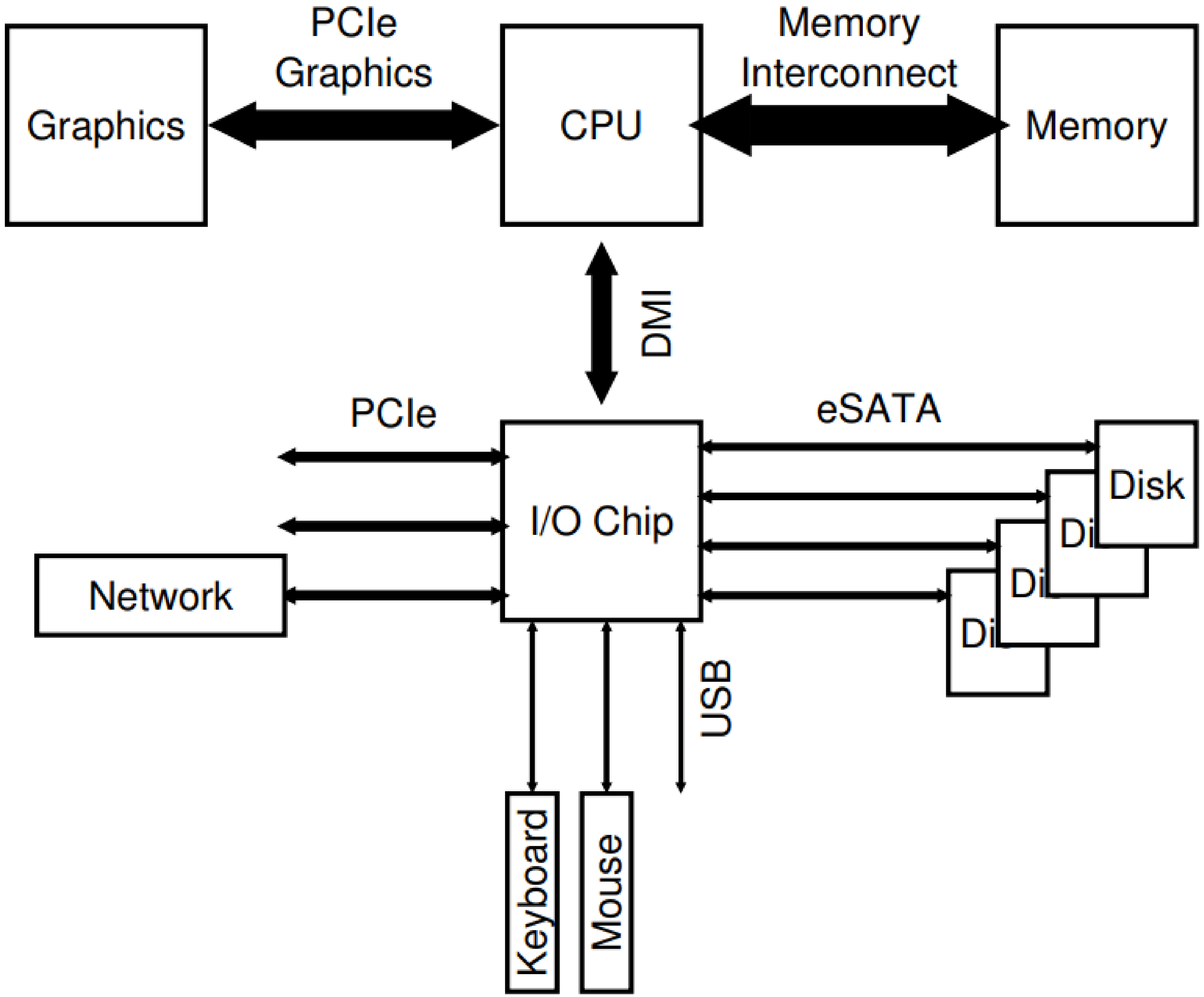
29.10.2025

Persistence

- We covered many aspects of the OS:
  - How to let the processes “time-share” the CPU
  - How to share memory access
  - How to run multiple parallel threads over the same address space
- All of this vanishes once the power goes out
- **Persistence:** How to manage long-term storage?
- Challenges:
  - Slow devices
  - Reliability
  - Abstraction

System hierarchy

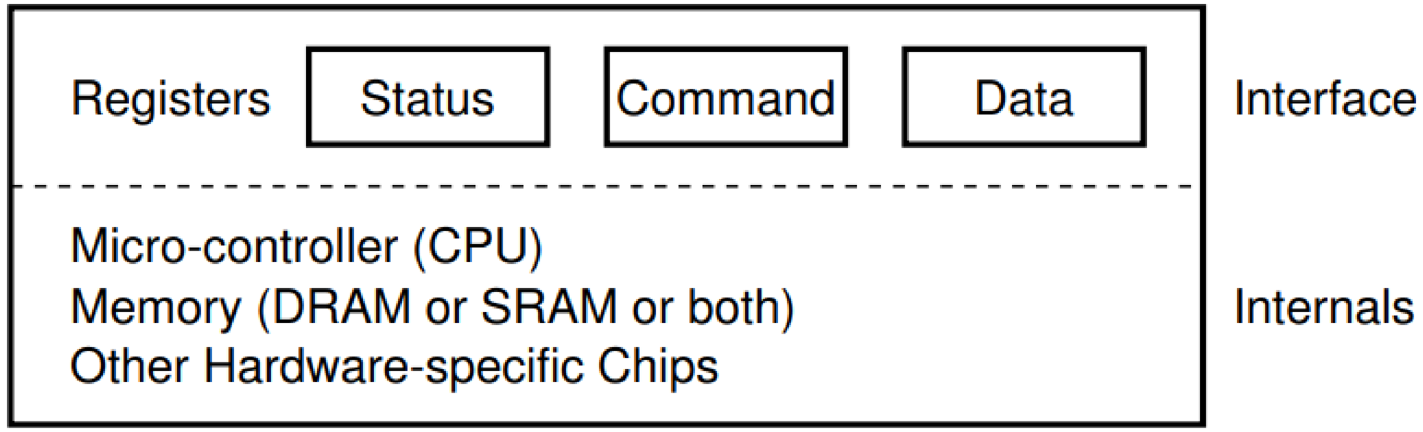
- Slower rates—farther from CPU
- The slower the device, the more "custom" it can be
- Modern systems would have specialized chips and faster point-to-point interconnects
  – e.g., to the GPU



Intel Z270 (2017)

A canonical device

- How to work with wildly different devices?
- Every device will have registers for control

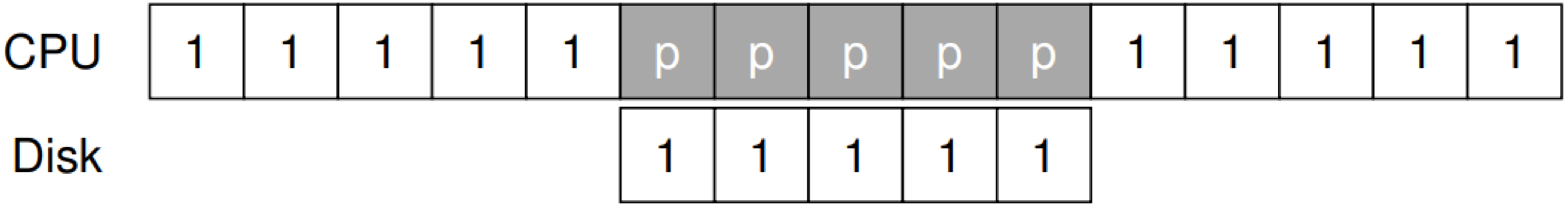


- The canonical interaction protocol:

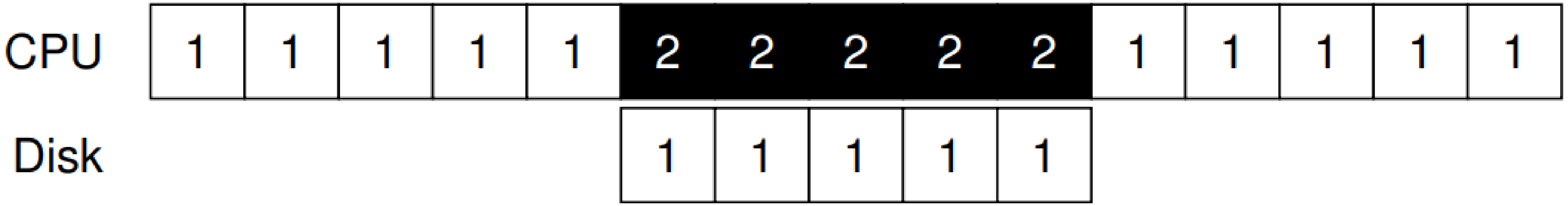
```
while (STATUS == BUSY)
; // wait until device is not busy
write data to DATA register
write command to COMMAND register
(starts the device and executes the command)
while (STATUS == BUSY)
; // wait until device is done with your request
```

Optimization 1: Interrupts

- Polling wastes CPU cycles

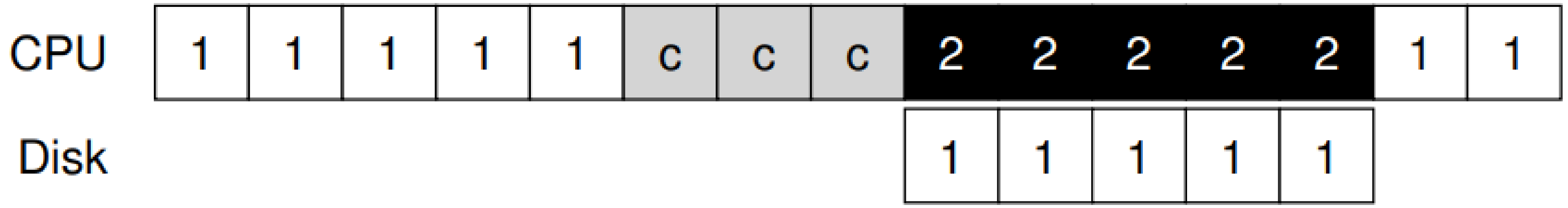


- Interrupts instead of active polling:
  1. Request for a status of a device
  2. Process is put to sleep
  3. Interrupt raised when the device is done
  4. Operation is completed

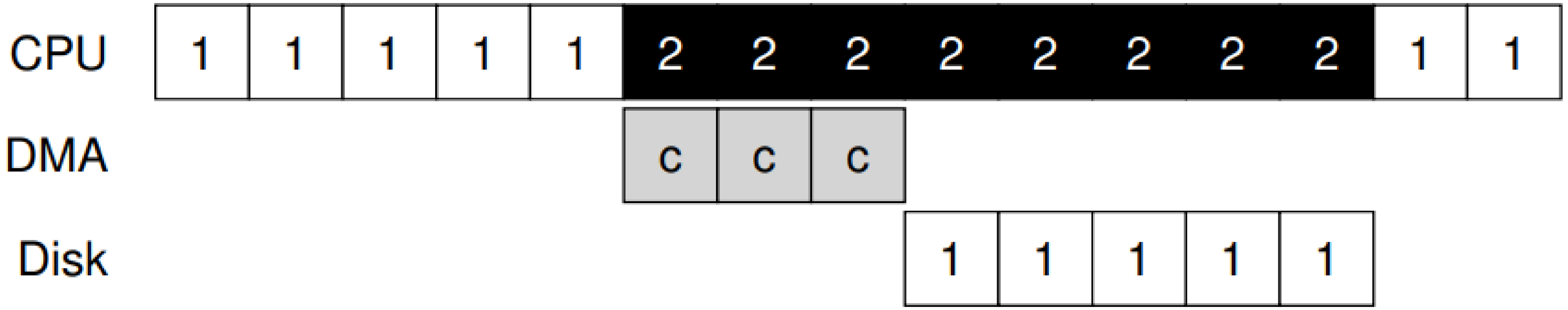


Optimization 2: DMA

- Write to disk:
  1. Copy data to disk interface
  2. Disk performs the write

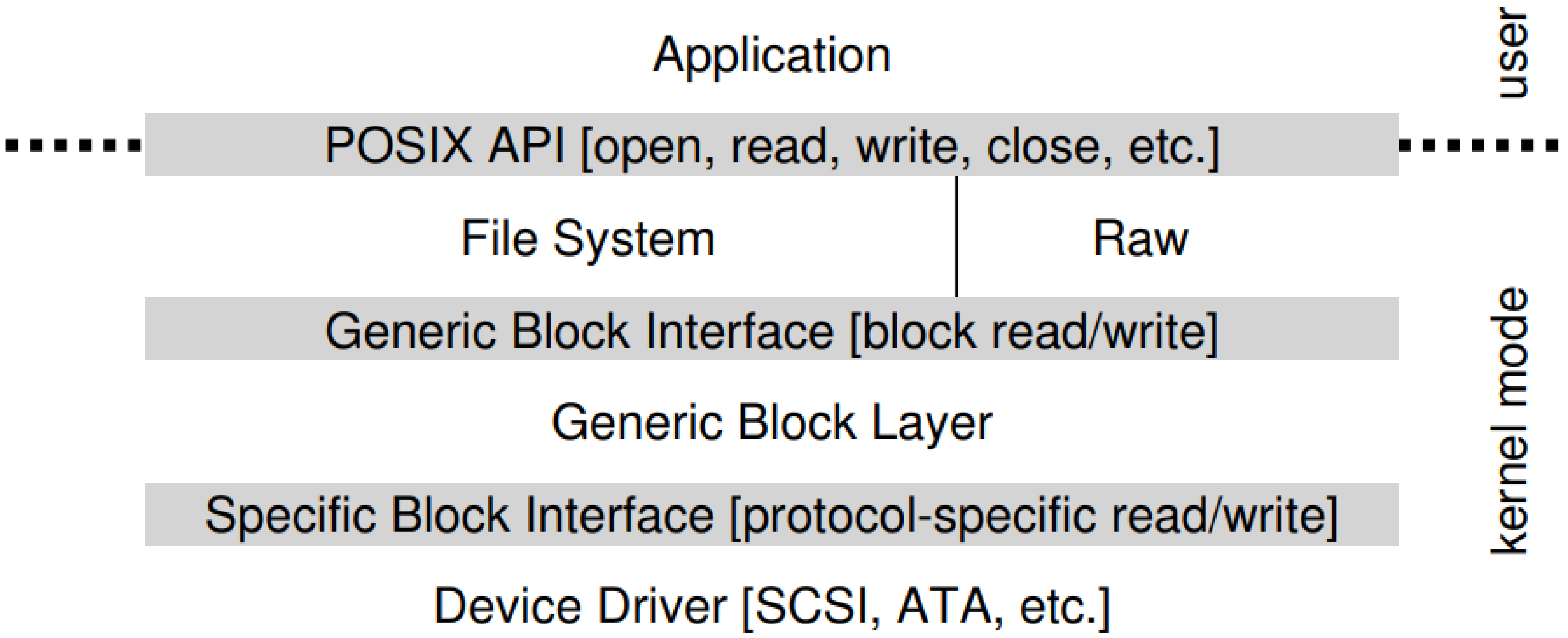


- Direct Memory Access (DMA)
  1. DMA engine receives instructions
  2. Performs copy from memory in parallel
  3. Raises interrupt when it's done



Device drivers

- Certain priveleged instructions are used to interact with the device registers
  - e.g., `in` and `out` on x86
  - need to specify **port**, i.e., which device, and the register
- **Driver** provides an abstract interface to the device to the rest of the OS
  - Hides the specific register locations, signal values, interaction protocol



write "`hello`" to `foo.txt`

↓

write to location 0
within block #1113

↓

wait until register `0x1f7` has code 6
put block address into `0x1f3`
...

Hard disk drive (HDD)

- Has a spinning magnetic **platter** and a **head** that shifts to the desired sector

- Cons:
  - Slower than SSDs
  - Bulky and noisy
  - Sensitive to mechanical action

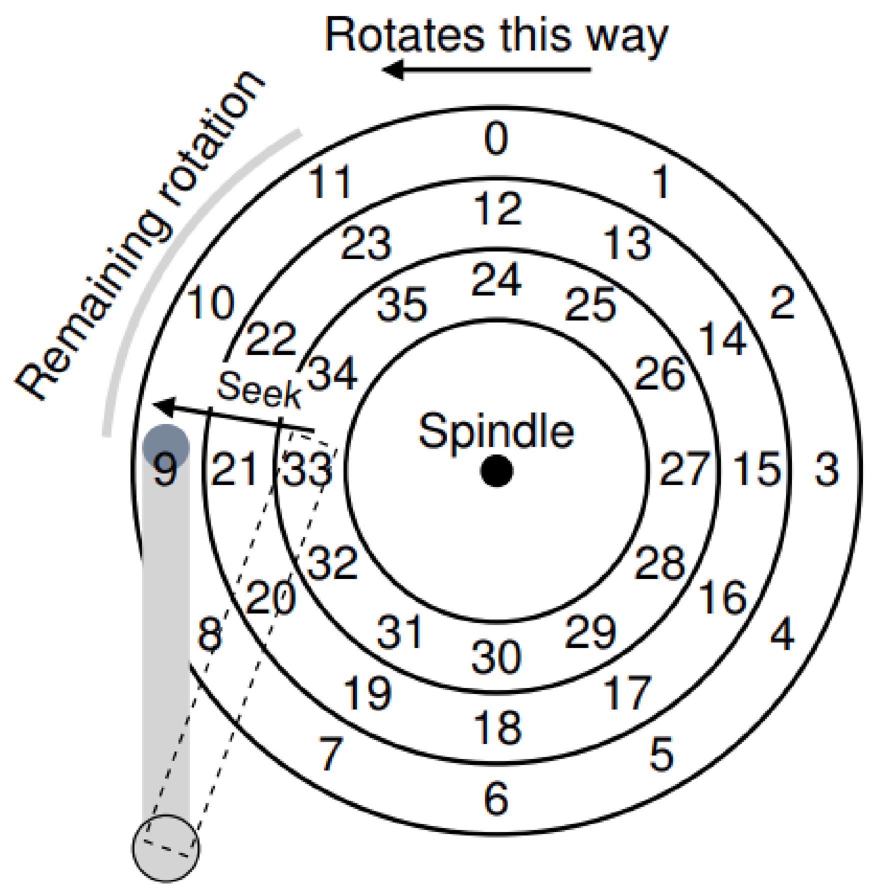
- Pros:
  - Cheaper per 1TB
  - Reliable long-term
  - Faster with continuous writing

- Still most of the datacenter storage

cc wiki

Understanding HDDs

- The platter consists of 512-byte **sectors**/**blocks**, indexed from 0 to $n-1$
- The head reads/writes one sector at a time, atomically
- The head needs to mechanically reach the sector
- **Rotational delay**: waiting for a sector on the same track
  - For 10 000 RPM, 6ms for the whole rotation
- **Seek**: moving the head to the right track
  - Settling time 0.5-2ms
- Once the head is in place, transfer is fast—e.g., 30μs
- Much faster to access sectors sequentially
  - Random access: 5–8 ms, next sector: 30 μs



Solid-state drive (SSD)

cc wiki

- Stores individual bit values in transistors

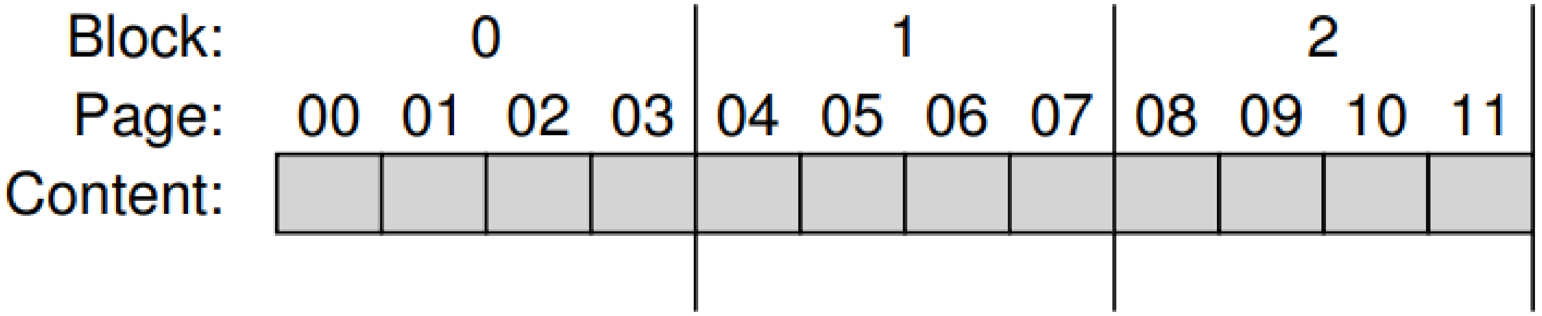
- Pros:
  - Fast
  - No mechanical parts

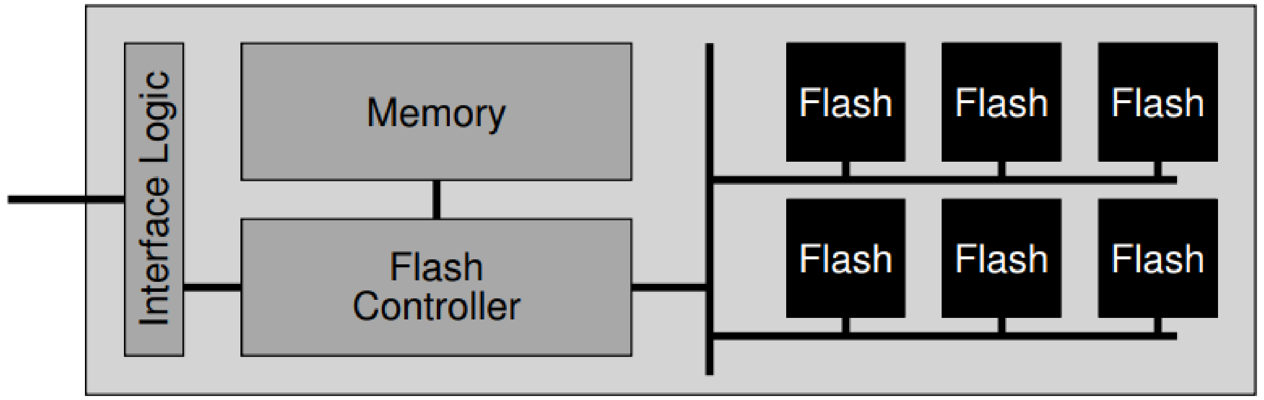
- Cons:
  - More expensive
  - Wears out on rewrites
  - Writing is complicated

	Random		Sequential	
Device	**Reads (MB/s)**	**Writes (MB/s)**	**Reads (MB/s)**	**Writes (MB/s)**
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Understanding SSDs

- Bits are grouped in 4KB **pages**, pages in 256KB **blocks** and blocks into **banks**

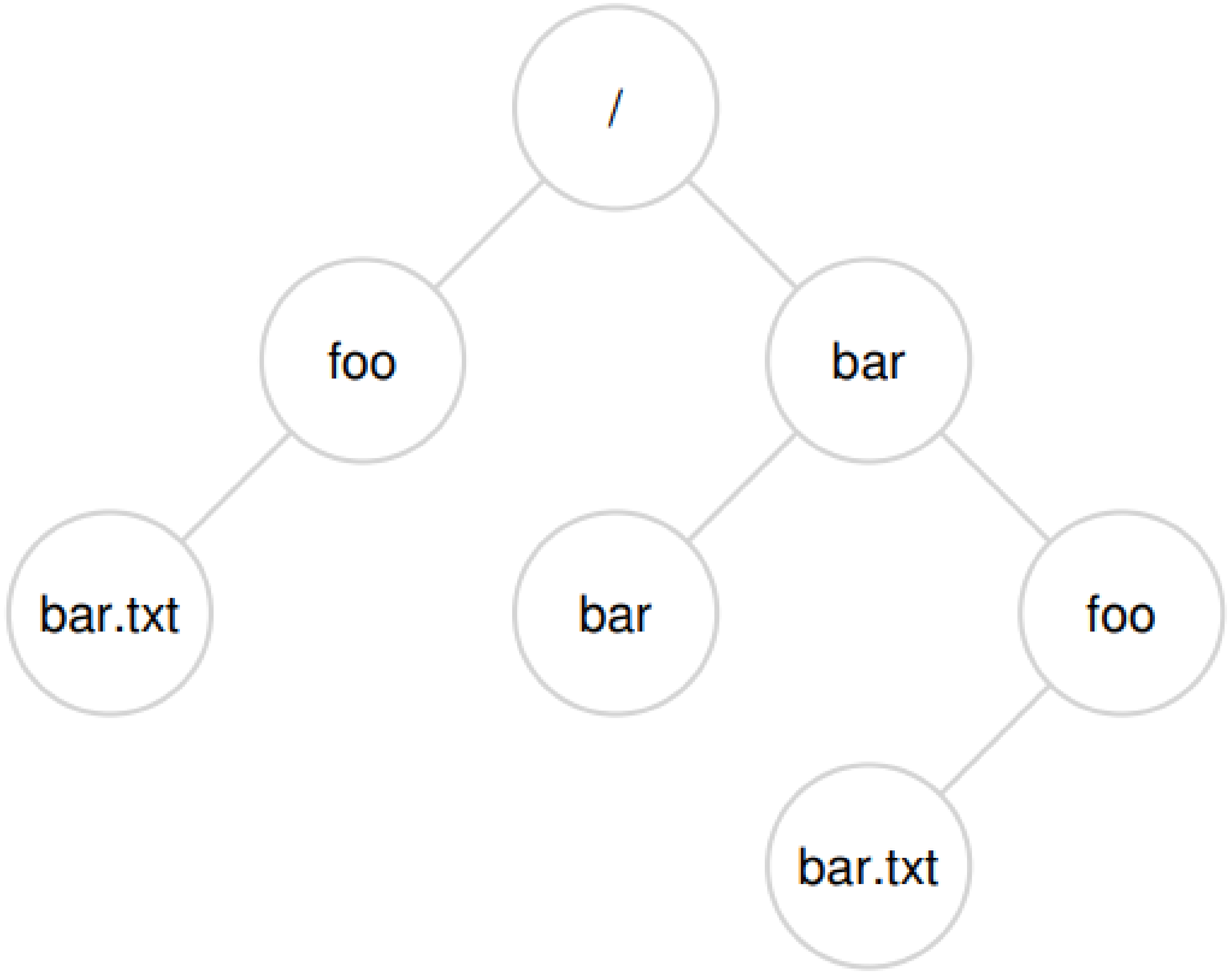




- **Read:** Get contents of a page by its index, $\sim 10\mu$s
- Cannot write to a page at will, instead:
  - **Erase a block:** sets the entire block to 1, ~ 3ms
  - **Program a page:** sets an erased page to the desired contents, $\sim 100\mu$s
- **Wear out:** each block can only do a finite number of erase cycles
- **Solution: logging**—append changes, instead of rewriting directly
- **Solution: wear leveling**—spread writes across all pages

Files and directories

- **File** is an array of bytes
  - OS keeps a low-level identifier, **inode number**
- **Directory** stores a list of pairs: (user-readable name, inode number)
  - Is really a special kind of file
  - Also has an inode number
- Files and directories form a tree-like hierarchy
  - Starts from the root directory /
  - Each file can be referenced by the **absolute pathname** `/bar/foo/bar.txt`
- In UNIX systems, everything is accessible within a single tree
  - Also devices, pipes, processes



File system interface

- Creating files: possible by open syscall

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- Return value: **file descriptor**, a local index of an opened file
  - OS stores the list of files opened by a process

```
struct proc {
  ...
  struct file *ofile[NOFILE]; // Open files
  ...
};
```

0: stdin
1: stdout
2: stderr
3: first file
...

- Reading files

```
$ echo hello > foo
$ cat foo
hello
```

```
$ strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6) = 6
hello
read(3, "", 4096) = 0
close(3) = 0
...
```

Non-sequential reading and writing

- A process need not to go through the whole file byte-by-byte
- `lseek` syscall shifts the "next character" by the desired offset

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

- For each file opened by the process, OS tracks the current offset
  - updates implicitly by `read`/`write`
  - updates explicitly by `lseek`

```
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

Renaming files

```
$ mv foo bar
```

- `rename` syscall "atomically" renames a file

```
int rename(char *old, char *new);
```

- Can be used to safely edit files:

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
    S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

- `fsync` syscall forces the write to disk as soon as possible, instead of buffering

File metadata

- `stat` syscall gives information about the file

```
$ stat foo
  File: foo
  Size: 6               Blocks: 8          IO Block: 4096   regular file
Device: 259,4 Inode: 23598029    Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/ seemann)   Gid: ( 1000/ seemann)
Access: 2025-10-27 19:58:53.527389303 +0100
Modify: 2025-10-27 19:58:50.937389495 +0100
Change: 2025-10-27 19:58:50.937389495 +0100
 Birth: 2025-10-27 19:58:50.937389495 +0100
```

- OS stores this information about files in a structure called **inode**

Directories

- Directory files are stored in a special format so should not be read directly
- Create empty directory: `mkdir`

```
$ strace mkdir foo
...
mkdir("foo", 0777) = 0
...
```

- Cycle over directory entries: `opendir`, `readdir`, `closedir`

```
$ ls
```

```
struct dirent {
  char d_name[256]; // filename
  ino_t d_ino; // inode number
  off_t d_off; // offset to the next dirent
  unsigned short d_reclen; // length of this record
  unsigned char d_type; // type of file
};
```

Links

- Removing a file simply calls `unlink()`

```
$ strace rm foo
...
unlink("foo") = 0
...
```

- `link()` makes a **hard link**—a copy of the directory entry

```
$ ln file file2
$ ls -i file file2
67158084 file
67158084 file2
```

- OS keeps track of the number of hard links to the actual file (i.e., inode)
  - When created, the file is linked to its human-readable name
  - When linked, counter increases
  - When unlinked, counter decreases (also in `rm foo`)
  - When it reaches zero, the whole entry at inode is removed

Soft links

- **Soft link** is a special type of file that contains the pathname of another file
  - Can point to a directory
  - Can point to a different device

```
$ ln -s foo foo2
```

- We can see that soft link is a special file by running `ls -l` and `stat`

```
$ ls -l
-rw-rw-r-- 1 ... foo
lrwxrwxrwx 1 ... foo2 -> foo
```

```
$ stat foo2
  File: foo2 -> foo
  Size: 3           Blocks: 0          IO Block: 4096   symbolic link
Device: 259,4 Inode: 23593299    Links: 1
```

Owners and permission bits

- Each file has an owner (user) and a group (collection of users)
- A file has also permission settings for each of the three access types: by own user, by own group, or by anyone else

```
$ ls -l foo
-rw-rw-r-- 1 user group 0 Oct 28 19:13 foo
```

- Permission could be to read the file (`r`), to write to the file (`w`), or execute it (`x`)
- Permission set has a corresponding bitmask, e.g., `rwx` is 111 or 7

```
$ chmod 640 foo
```

6—to owner	4—to group	0—to others
110 = read and write	100 = read only	000 = nothing

Making and mounting a file system

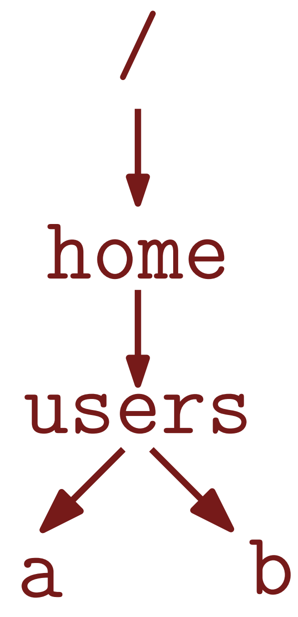
- `mkfs` creates an empty file system of given fs type on a given device

  don't run on your devices that already have a file system!

- To be part of the directory tree, the file system has to be **mounted**
- Mount attaches the directory tree of the file system under the given **mount point**

```
$ mount -t ext3 /dev/sda1 /home/users
```

```
$ cd /home/users/a
```



- List all mounted fs by calling `mount` without arguments:

```
$ mount
/dev/sda1 on / type ext3 (rw)
...
```

- With mount, all different file systems and devices are in the same directory tree