Week 10

Persistent Storage: Intro to File Systems

Oana Balmau
March 7, 2023

Class Admin

Assignment 2 due.

Assignment 3 released.
You have 4 weeks to solve it.

Week 10 **File Systems**	mar 6 No lab. Work on Assignment 2 ~~**Scheduling Assignment Due**~~	mar 7 **Intro to File Systems (1/2)** Recorded lecture. Do not come to class. Optional reading: OSTEP Chapters 36, 37, 39	mar 8 **Memory Management Assignment Released**	mar 9 **Intro to File Systems (2/2)** Memory Management Assignment Overview — with Jiaxuan	mar 10
Week 11 **File Systems**	mar 13 **Scheduling Assignment Due** ~~**Graded Exercises Due**~~ C Review: Complex structs	mar 14 **Basic File System Implementation (1/2)** Optional reading: OSTEP Chapters 40, 41, 45	mar 15	mar 16 **Basic File System Implementation (2/2)** • ~~Grades released for Scheduling Assignment~~	mar 17
Week 12 **File Systems**	mar 20 **Graded Exercises Due** C Review: Pointers & Memory Allocation II	mar 21 **Advanced File System Implementation (1/2)**	mar 22	mar 23 **Advanced File System Implementation (2/2)** • Grades released for Scheduling Assignment	mar 24
Week 13 **File Systems**	mar 27 C Review: Advanced debugging	mar 28 **Handling Crashes & Performance (1/2)** Optional reading: OSTEP Chapters 38, 43	mar 29	mar 30 **Handling Crashes & Performance (2/2)** • ~~Grades released for Exercises Sheet~~ • Practice Exercises Sheet: File Systems	mar 31
Week 14 **Advanced Topics**	apr 3 No lab. Work on Assignment 3 ~~**Memory Management Assignment Due**~~	apr 4 **Advanced topics: Virtualization**	apr 5	apr 6 **Advanced topics: Operating Systems Research** (Invited Speaker: TBD) Grades released for Exercises Sheet	apr 7
Week 15 **Wrap-up**	apr 10 No Lab. Prepare for end-of-semester. **Memory Management Assignment Due**	apr 11 **End-of-semester Q&A– not recorded**	apr 12	apr 13 **End-of-semester Q&A — not recorded.** Last class!	apr 14 Grades released for Memory Management Assignment

Key Concepts

- I/O devices
  - OS role for integrating I/O devices in systems
  - Polling, Interrupts, Drivers
- Notion of “permanent” storage
- File system interface
- Disk Management for HDDs
  - Disk Allocation, Disk Scheduling, Optimizations

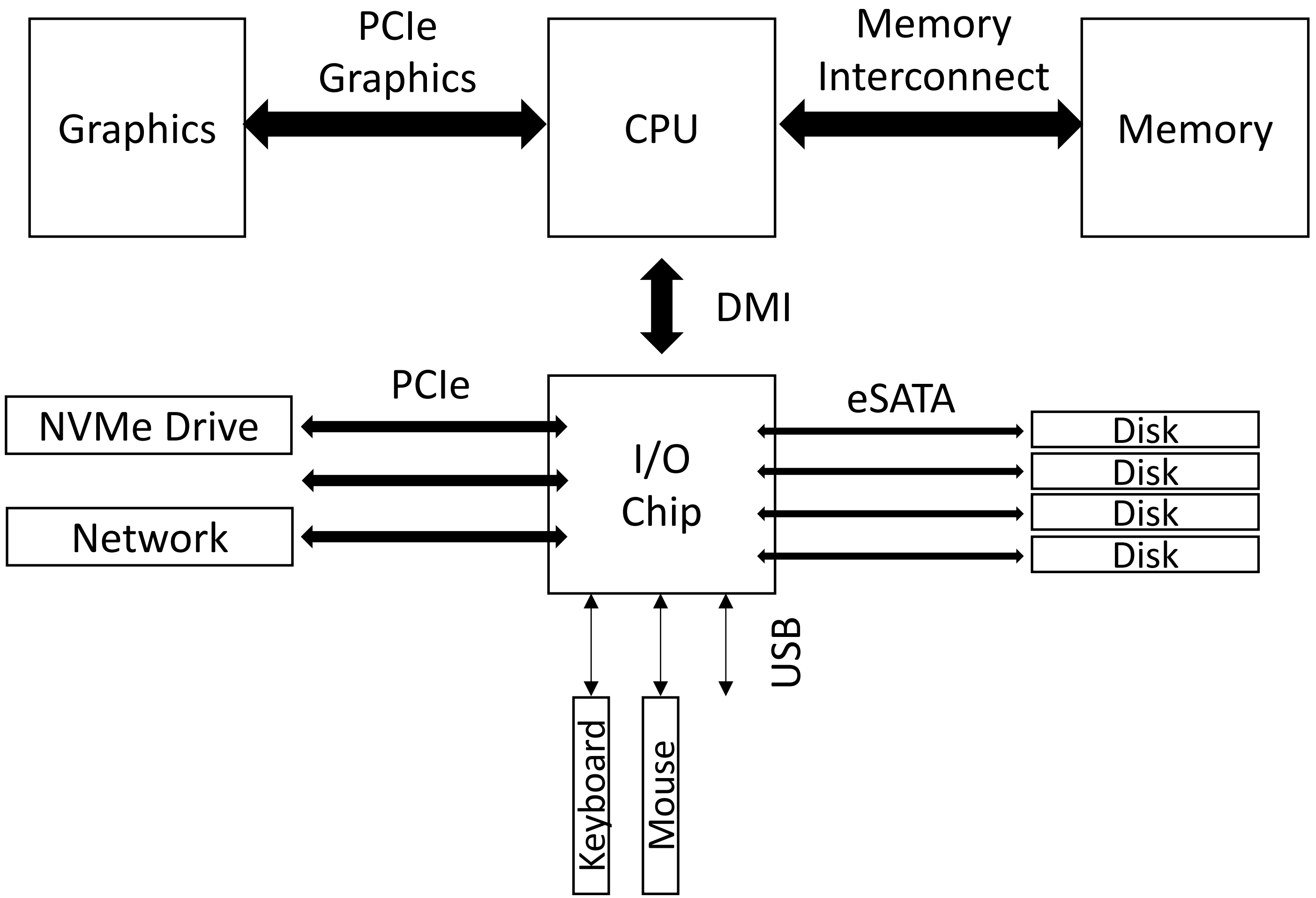
How should I/O be integrated into systems?

- I/O = Input/Output
- For computer systems to be interesting, both input and output are required.
- Many, many I/O devices

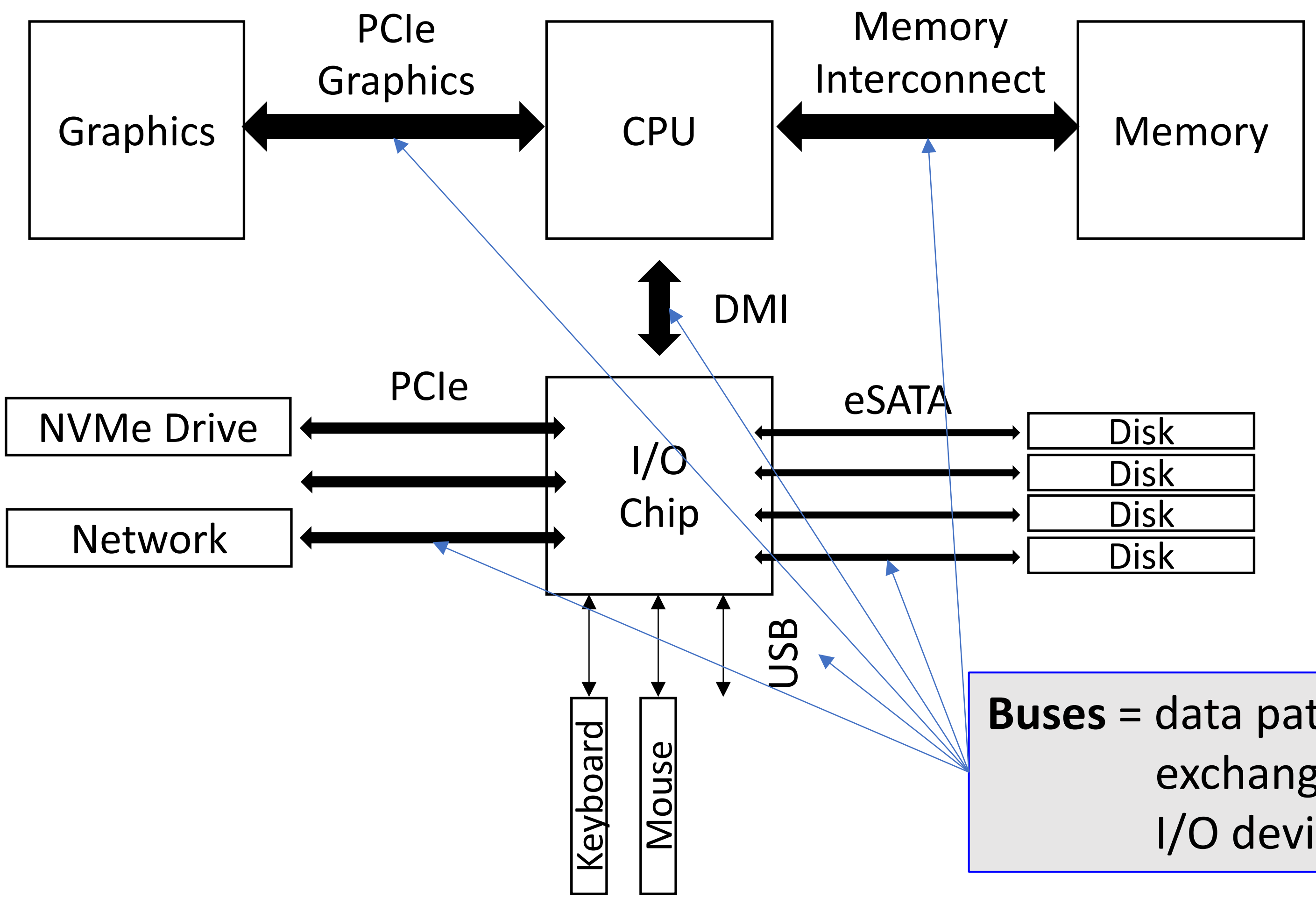




I/O System Architecture

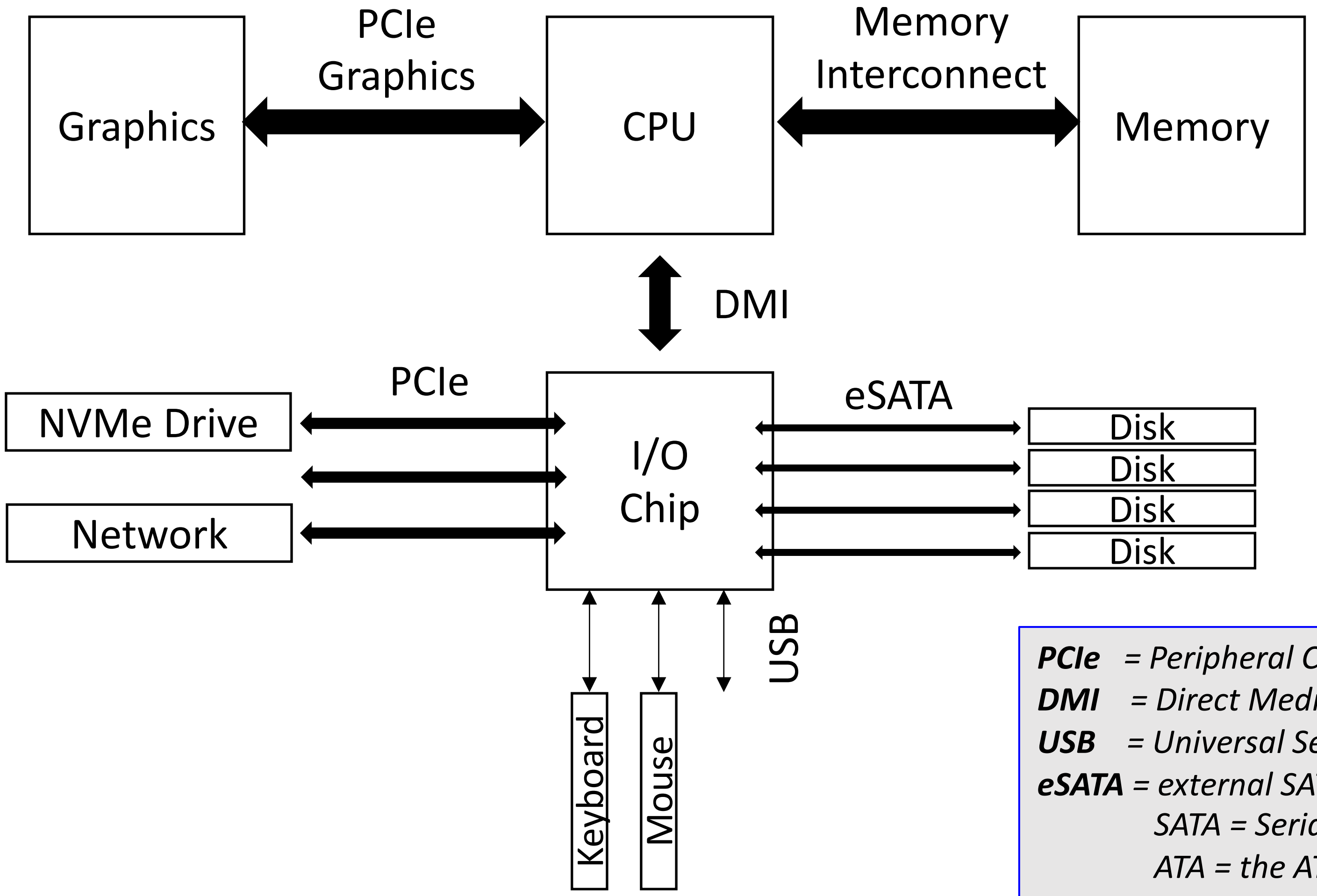


I/O System Architecture



Buses = data paths that enable information exchange between CPU, RAM and I/O devices

I/O System Architecture



PCIe *= Peripheral Component Interconnect Express*
DMI *= Direct Media Interface*
USB *= Universal Serial Bus*
eSATA *= external SATA*
SATA = Serial ATA
ATA = the AT Attachment, in reference to providing connection to the IBM PC AT

How does OS communicate with I/O devices?

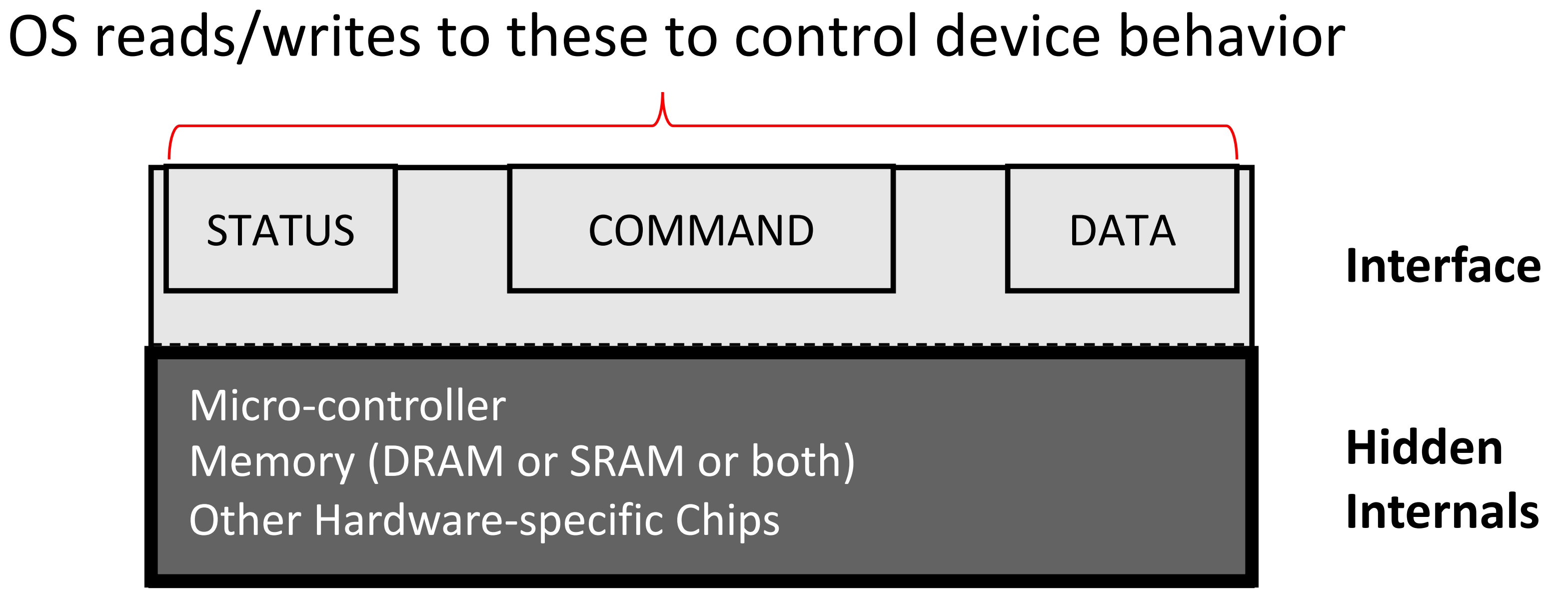
Canonical Device Interface

Interface made of **3 registers:**

- **Status** – current status of device
- **Command** – OS tells device what command to perform
- **Data** – send/receive data to/from device

By reading and writing these **3 registers,**
the OS can **control device behavior.**

Canonical Device Interface



How does OS use device interface?

- Polling

- Interrupts

Polling

- OS waits until device is ready
- OS **repeatedly checks** the STATUS register in a loop

Advantage: Simple, works

Disadvantage: Wasted CPU cycles

Polling

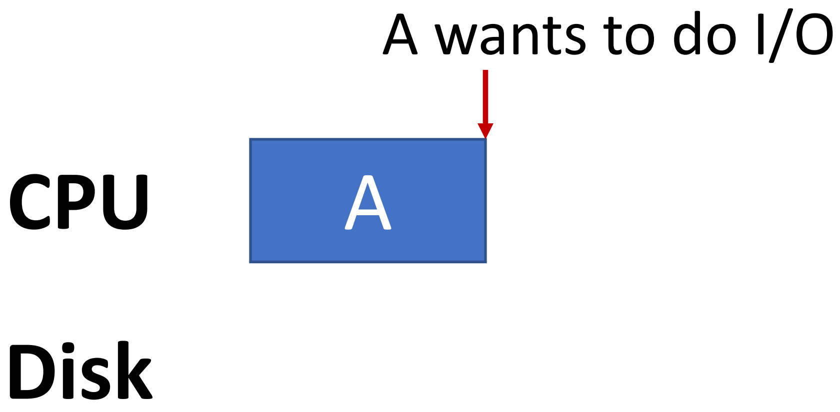
```
write data to DATA register
write command to COMMAND register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //spin-wait until device is done with your request
```

Polling

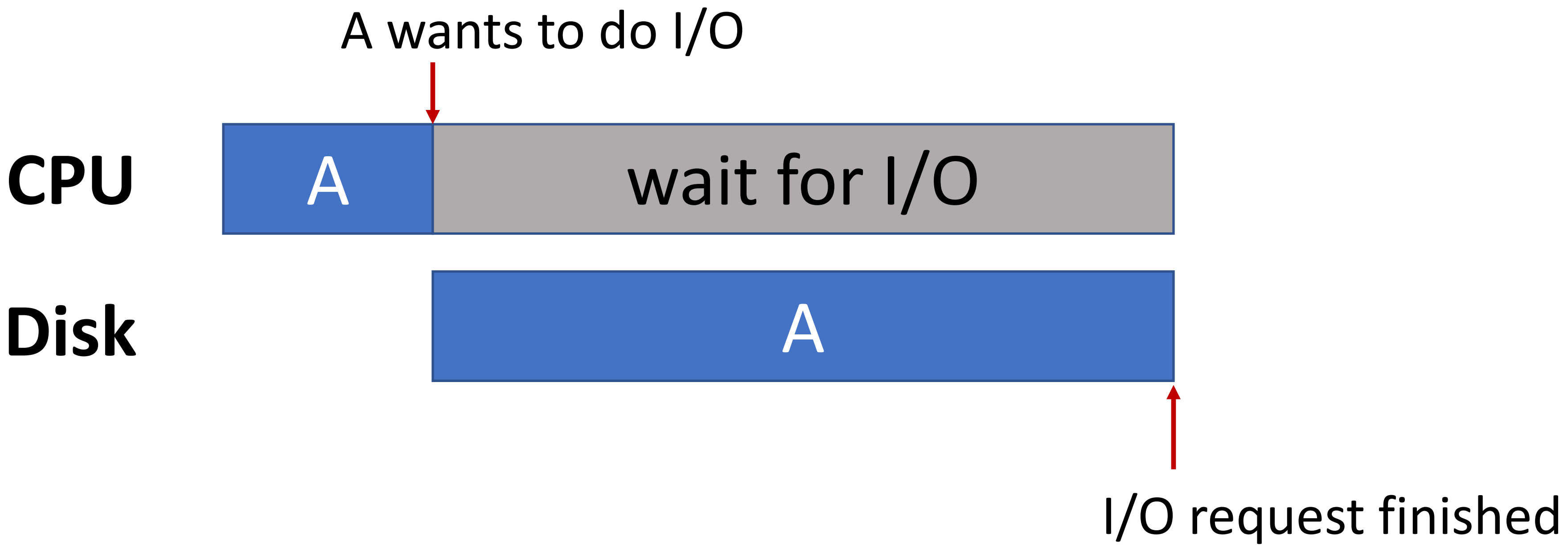
```
write data to DATA register
write command to COMMAND register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //spin-wait until device is done with your request
```

Wasted CPU cycles

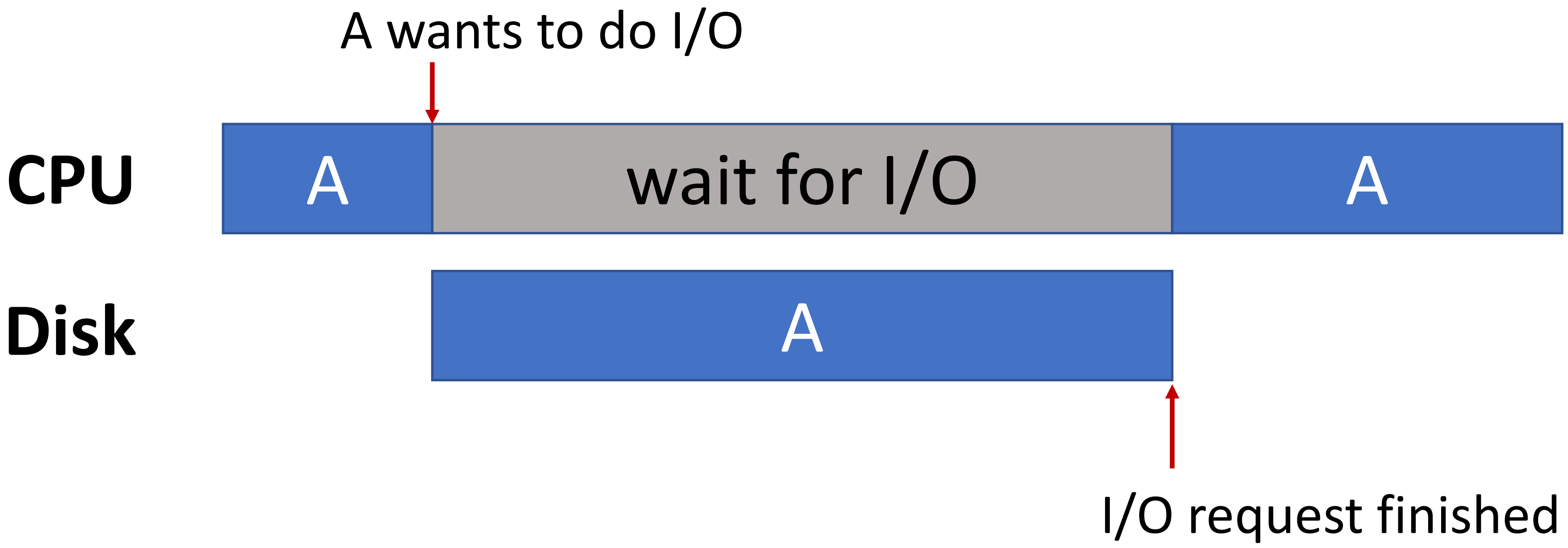
Polling Example



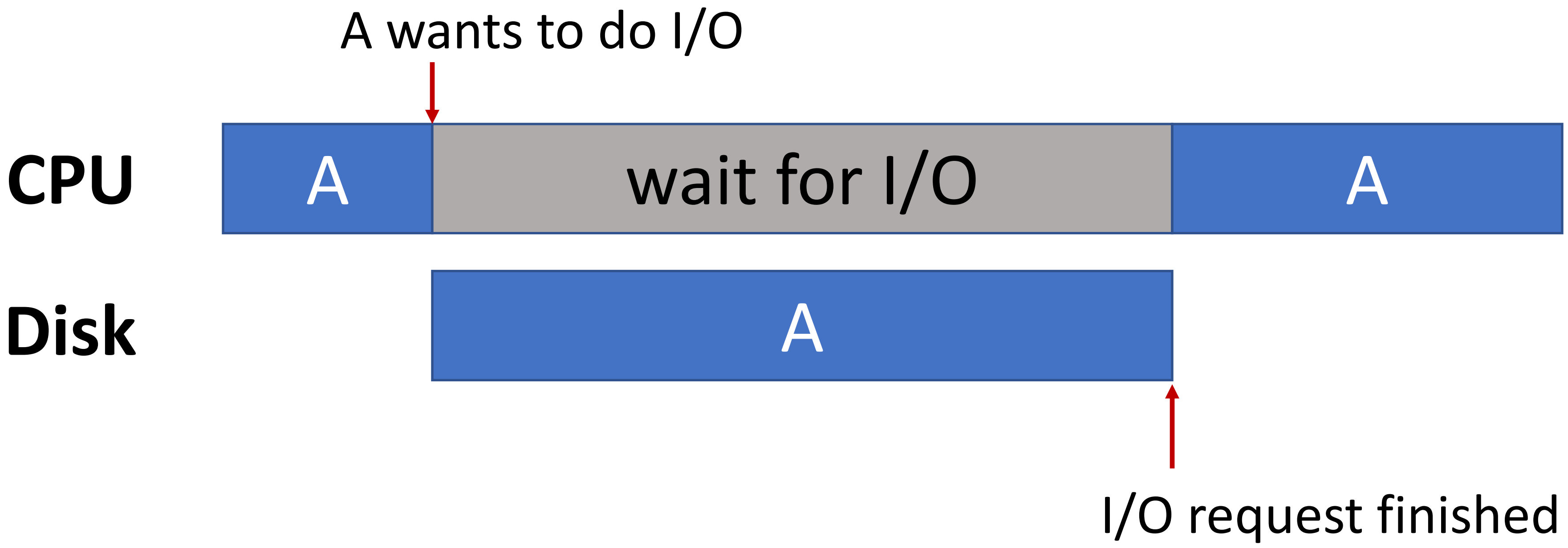
Polling Example



Polling Example



Polling Example



→ How can we avoid waiting for I/O?

Interrupts

- Put process requesting I/O to **sleep**
- **Context switch** to a different process
- When I/O finishes, wake sleeping process with an **interrupt**
- CPU jumps to **Interrupt Handler** in the OS

Remember: Same interrupt mechanism used for demand paging last week.

Interrupts

- Put process requesting I/O to **sleep**
- **Context switch** to a different process
- When I/O finishes, wake sleeping process with an **interrupt**
- CPU jumps to **Interrupt Handler** in the OS

Remember: Same interrupt mechanism used for demand paging last week.

Remember: Interrupts vs Syscalls.

- **Interrupt:** generated by hardware to initiate a context switch
- **Syscall:** generated by process, to request functionality from kernel mode. Also, initiates a context switch.

Interrupts

```
write data to DATA register
write command to COMMAND register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    go to sleep; wait for interrupt
```

Interrupts Example

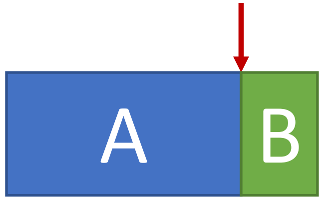
A wants to do I/O;
Send I/O request **and sleep**

CPU

Disk

Interrupts Example

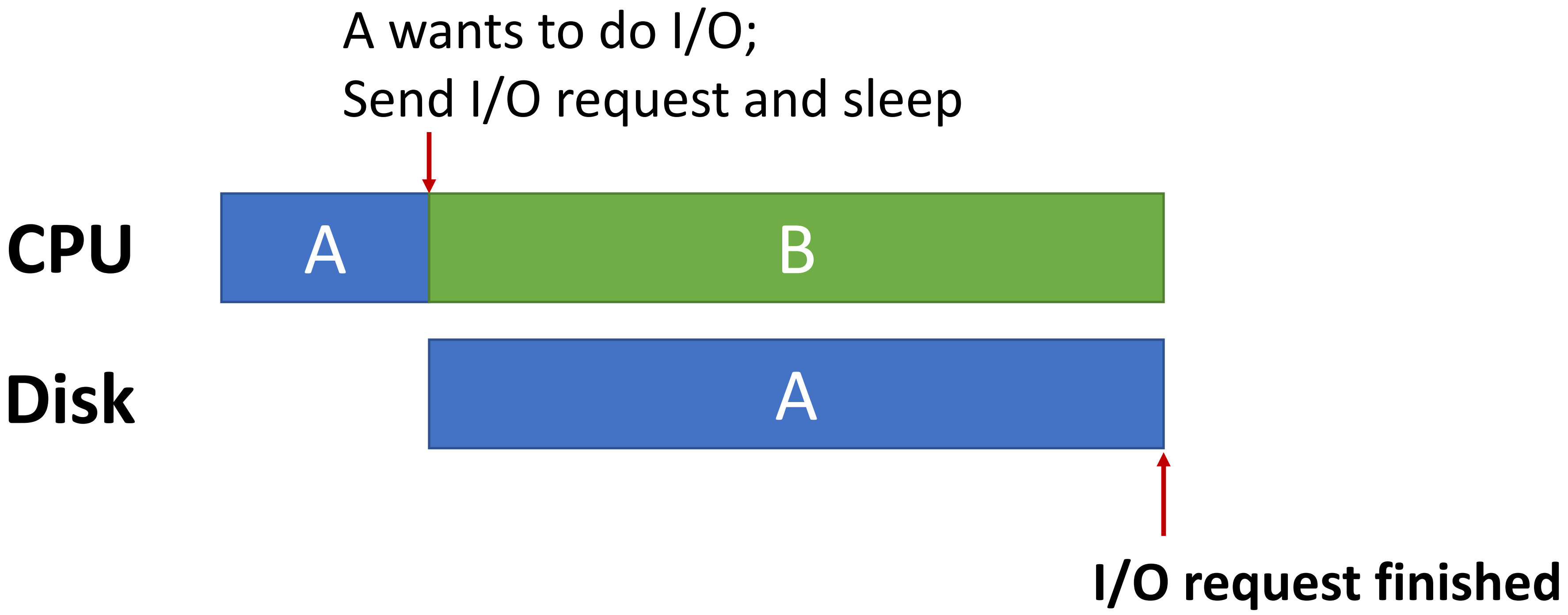
A wants to do I/O;
Send I/O request and sleep

CPU 

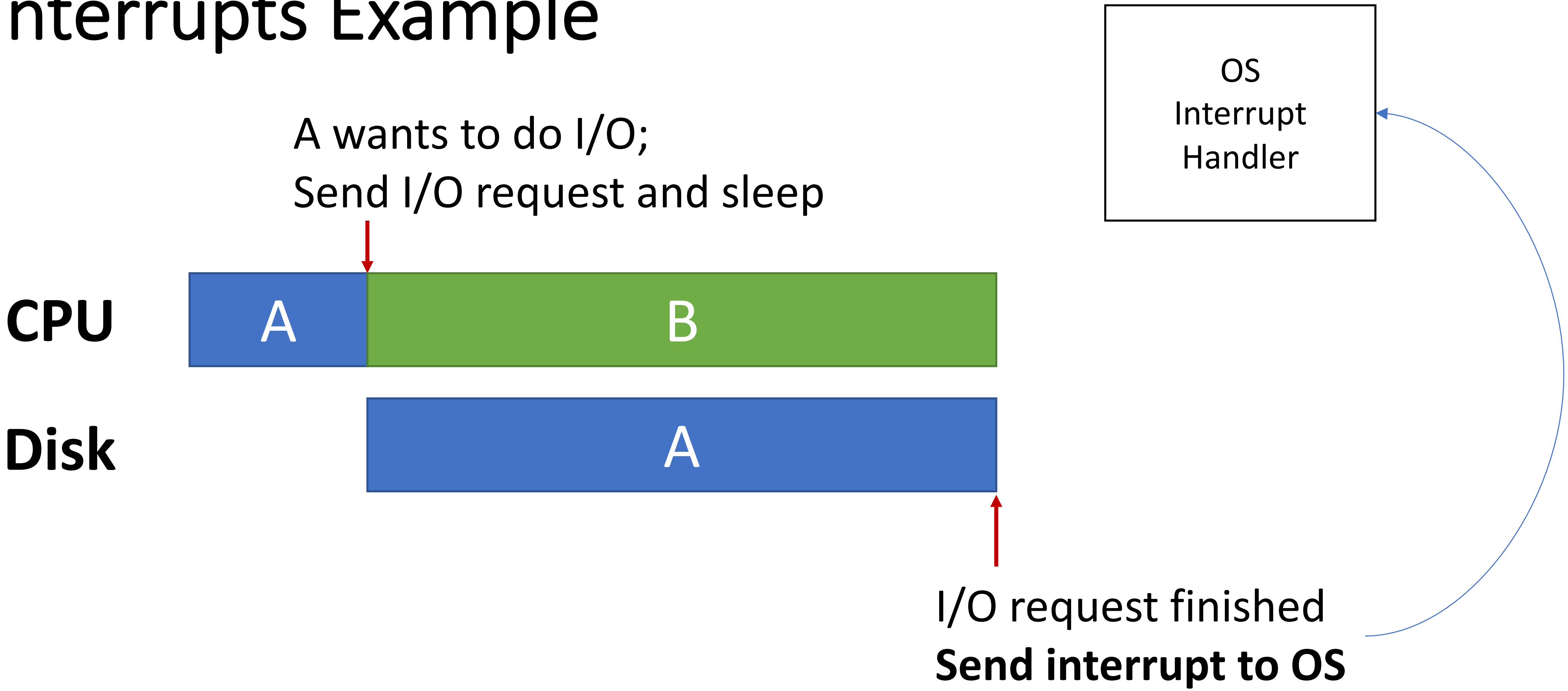
Context switch to B

Disk

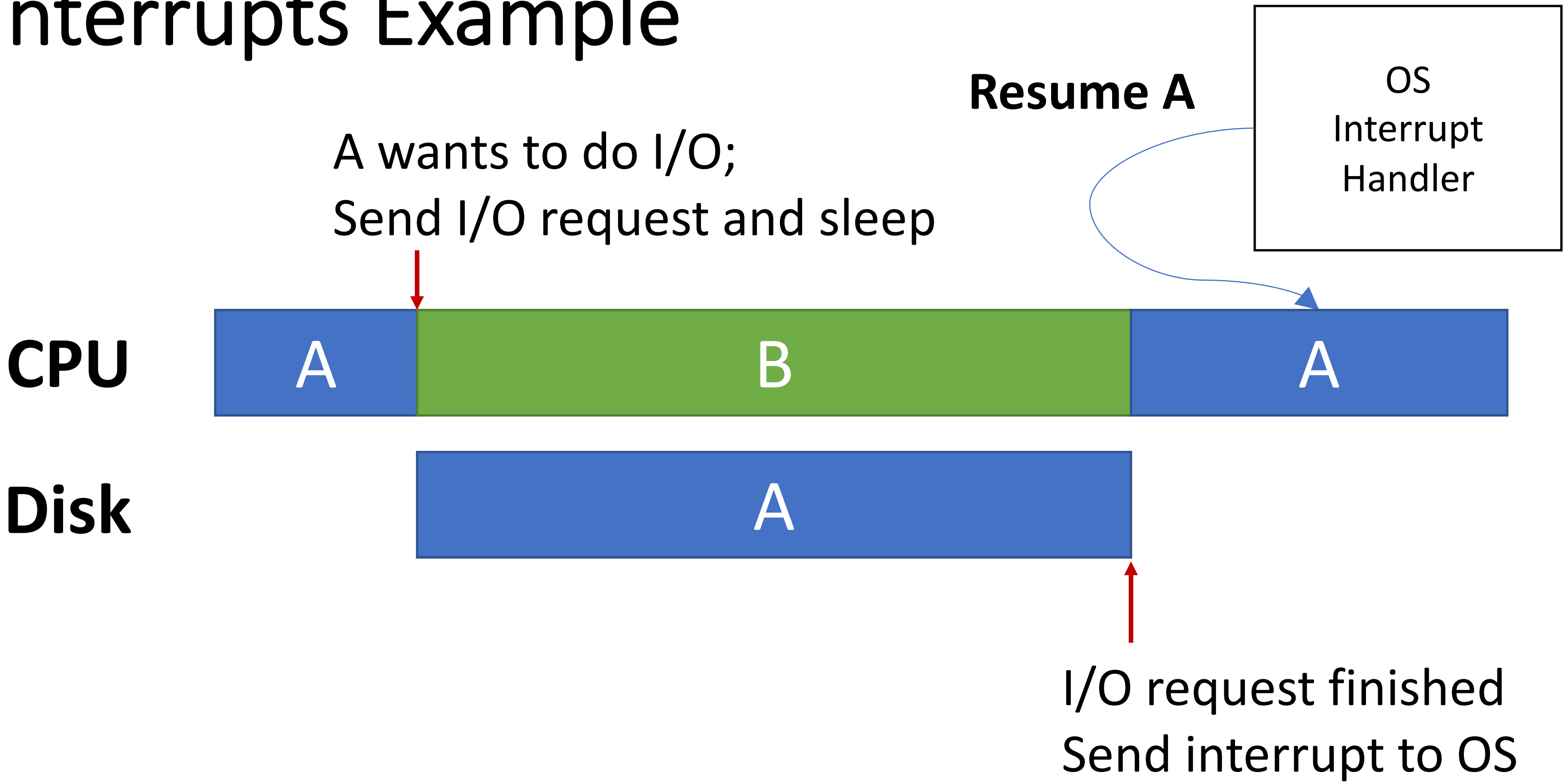
Interrupts Example



Interrupts Example



Interrupts Example



Interrupts

Advantage: No waste of CPU cycles

Disadvantages:

- Expensive to context switch

→**polling can be better for fast devices**

Problem: How to handle different devices in OS?

- Many, many devices
- Each has its own protocol
- **Variety is a challenge**

Solution: Device Drivers

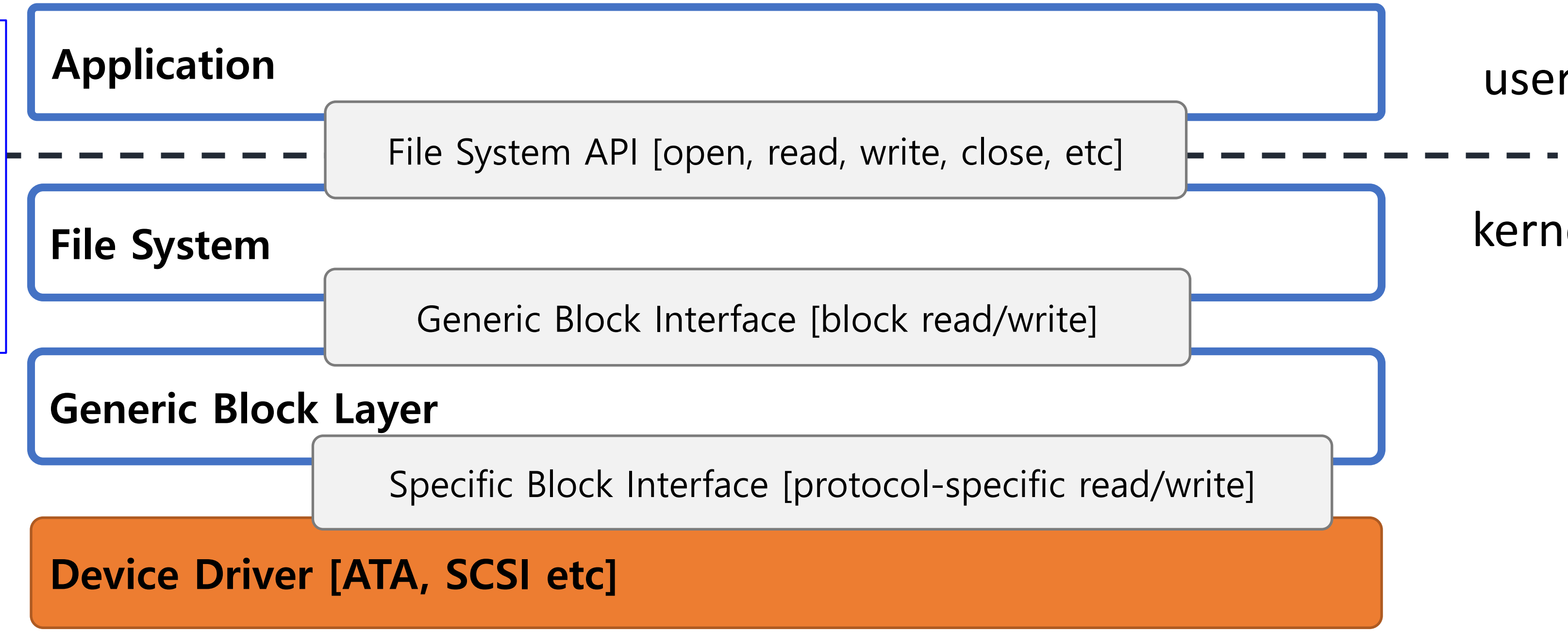
- Driver is a piece of software in the OS
- Must know in detail how a device works
- Need a device driver for each device

→Drivers are **~70%** of Linux source code

So when we say the OS has millions of lines of code, we're really saying that "the OS has millions of lines of device driver code."

Example: File System Stack

Applications &
File System
are oblivious to the
inner workings of the
storage device(s)



Summary – I/O Devices

- I/O System Architecture
- Canonical Device Interface
- Device Access
  - Polling, interrupts,
  - Direct memory access (DMA)
- Device driver abstraction

“Permanent” Storage

“Permanent” Storage

How permanent is permanent?

- Across program invocations
- Across login
- Across machine failures/restarts
- Across disk failures
- Across multiple disk (data center) failures

For this course

Permanent Storage Media

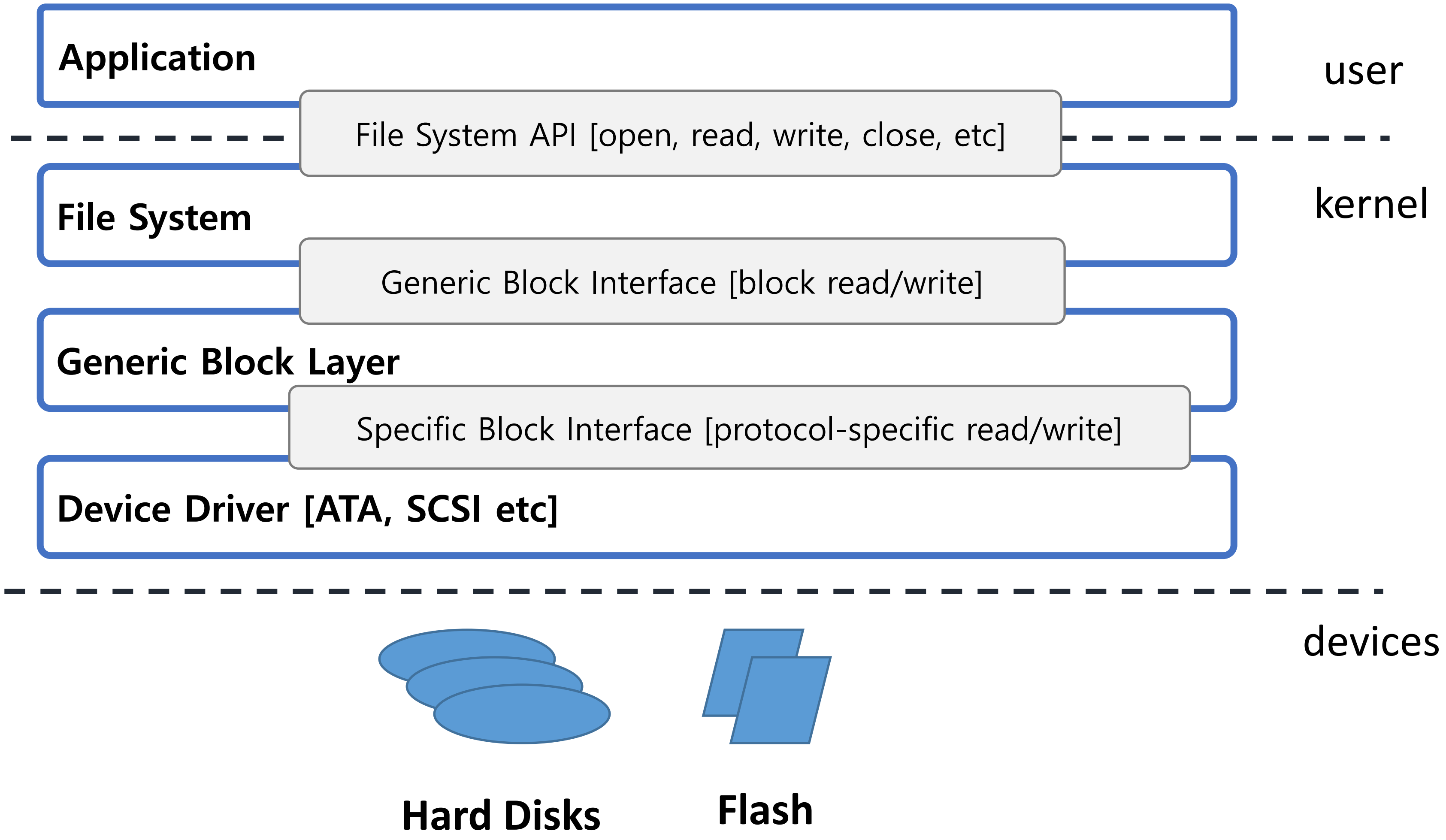
- Main memory – not suitable

- Battery-backed memory
- Nonvolatile memory
  - Flash SSDs
  - 3DXpoint SSDs (released in 2018)
- Hard Disks (HDDs)
- Tapes

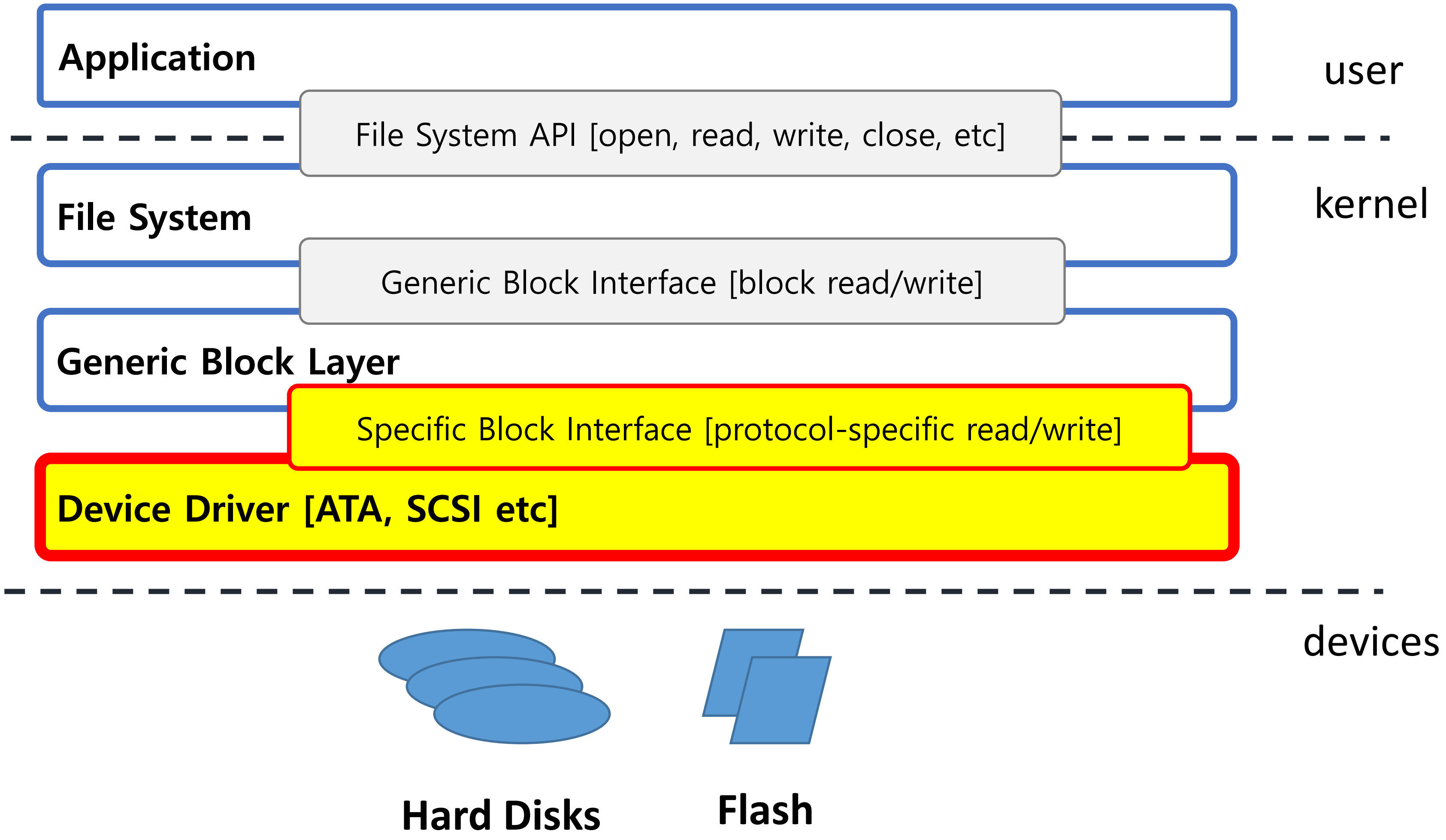
For this course



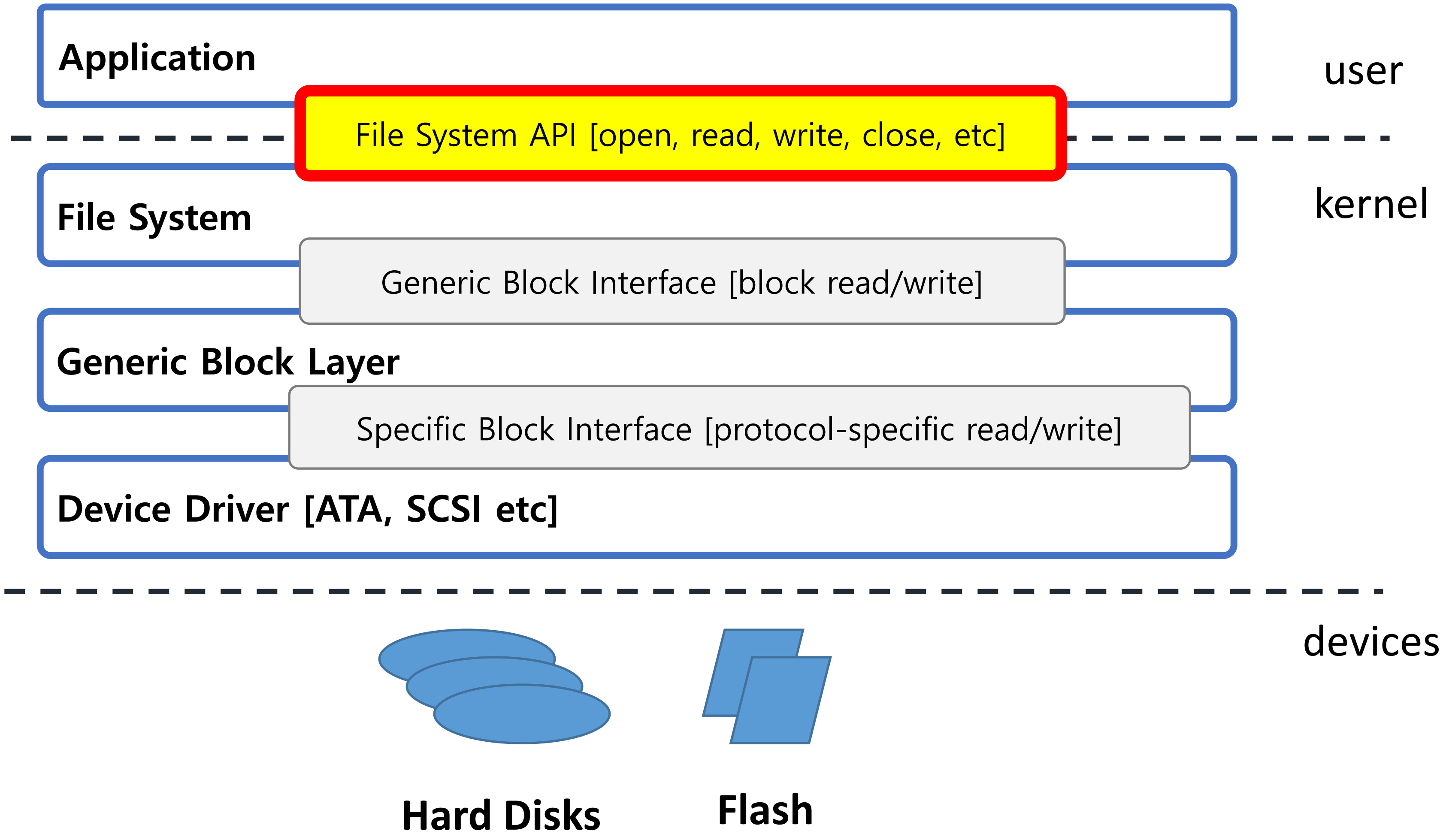
Overall Picture



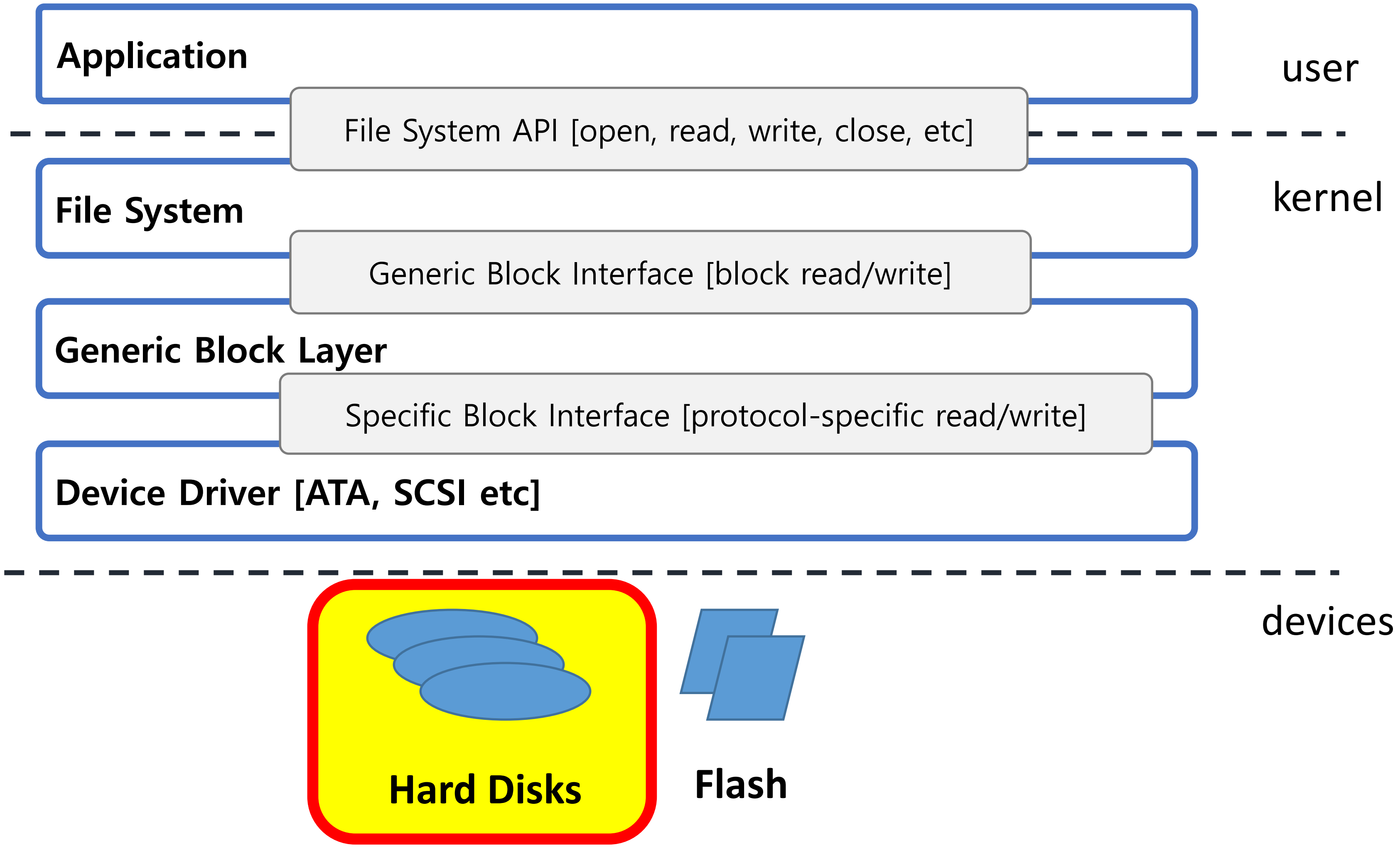
We just talked about this part



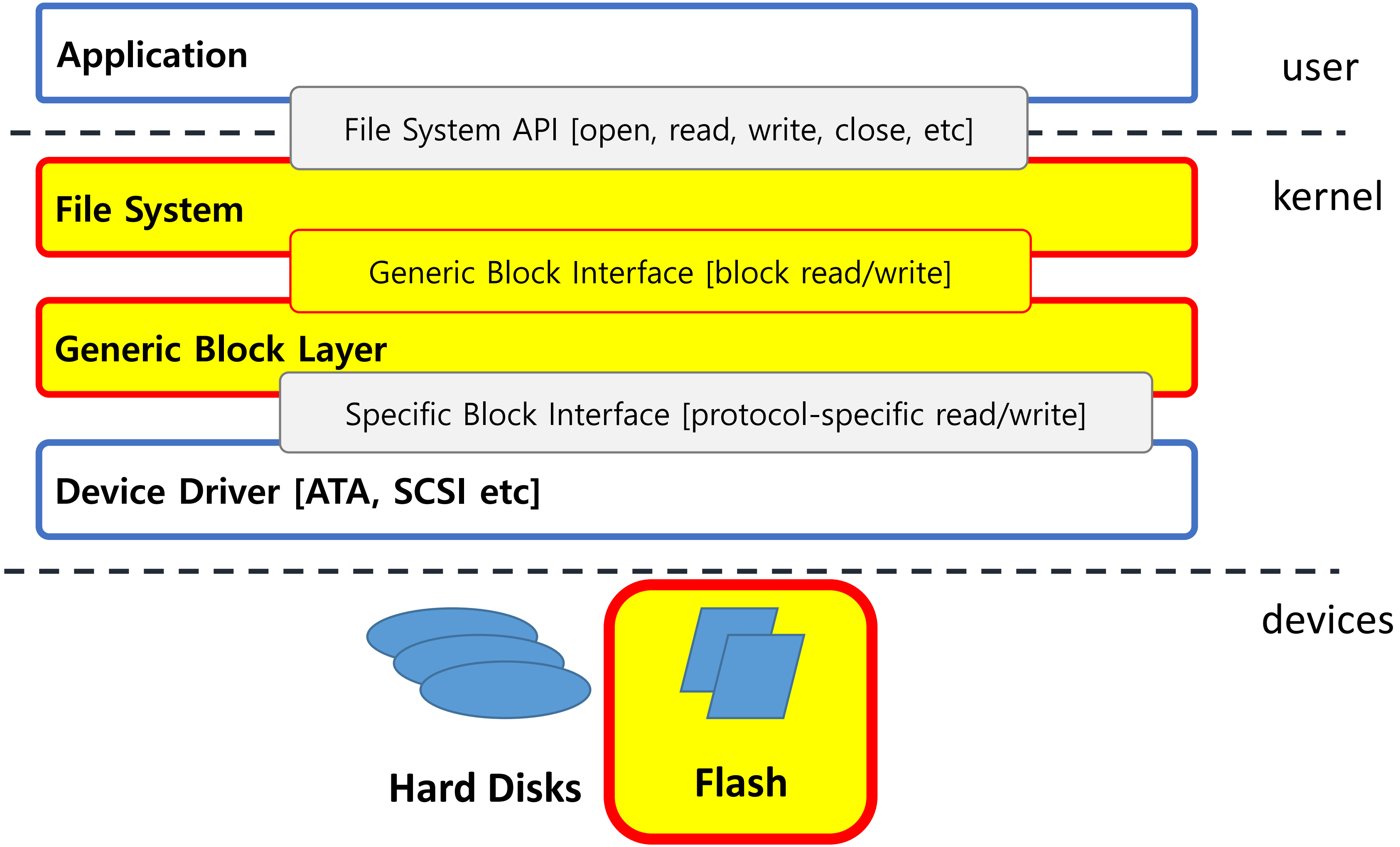
Now we will talk about this part



And then we will talk about this part



Next Weeks' Lectures



File System API

File System API

What is a file?

- Un-interpreted collection of objects
  - Bytes, records …
  - → We will look at **bytes (as in Linux)**
- Un-interpreted ~=
  - File system does not know what data means
  - Only application knows

Typed or Untyped?

Typed = File System knows what the object means

Advantages:
- Invoke certain programs by default
- Prevent errors
- More efficient storage

Disadvantages:
- Can be inflexible (typecast)
- Can become a lot of code (many types)

→ We will look at **untyped files**

Aside: File Name Extensions = Types ?

`foo.txt`

- Pure convention (Linux)
  - User knows, system does not do anything with it
- Known to the system (Windows)
  - User knows, systems knows (and enforces)

File System Primitives

- Access
- Concurrency
- Naming
- Protection

File System Primitives

- **Access**
- Concurrency
- Naming
- Protection

Main Access Primitives

- Create()
- Delete()

- Read()
- Write()

Create() and Delete()

```
uid = Create( [optional arguments] )
```

- ***uid*** **unique identifier**, not human-readable string
- Creates an empty file

```
Delete( uid )
```

- Deletes file with identifier *uid*
- Usually also deletes all of its contents

Read()

`Read( uid, buffer, from, to )`

- Reads from file with identifier *`uid`*
- From byte *`from`* to byte *`to`*
  - Can cause EOF (End-of-file) condition
- Into a memory buffer *`buffer`*
  - previously allocated
  - **must be of sufficient size**

Write()

```
Write( uid, buffer, from, to )
```

- Write to file with identifier *`uid`*
- Into byte *`from`* to byte *`to`*
- From a memory buffer *`buffer`*

Sequential vs Random Access

Read() and Write() in previous slide:

- ***Random-access* primitives**
- No connection between two successive accesses

Sequential access is very common:

- Read from where you stopped reading
- Write to where you stopped writing
- In particular, whole file access is common

→ For this reason, need sequential access methods

Sequential Read()

- File system keeps **file pointer *fp*** (initially 0)
- `Read( uid, buffer, bytes )`
  - Read from file with unique identifier *`uid`*
  - **Starting from byte *`fp`***
  - *Bytes* `bytes`
  - Into memory buffer *`buffer`*
  - ***fp += bytes***

Sequential can be built on Random

- Maintain *fp*-equivalent in user code

```
…
myfp = 0
Read( uid, buffer, myfp+bytes-1 )
myfp += bytes
Read( uid, buffer, myfp+bytes-1 )
…
```

Can Random be built on Sequential?

Not without an additional primitives

- `Seek( uid, to )`

Using Seek() to implement Random **Read(uid, from, to, buffer):**

```
…
Seek( uid, from )
Read( uid, buffer, to-from+1 )
…
```

Sequential vs. Random

- Sequential access is very common
- **All systems provide sequential access**

- Some systems provide
  - **Only sequential access**
  - Plus Seek()

File System Primitives

- Access
- **Concurrency**
- Naming
- Protection

Concurrent (Sequential) Access

- Two processes access the same file
- What about *fp*?

Concurrent (Sequential) Access

- Two processes access the same file
- What about *fp*?

→ The notion of an **“Open” File**

- Open()
- Close()

Open()

```
tid = Open(uid, [optional args])
```

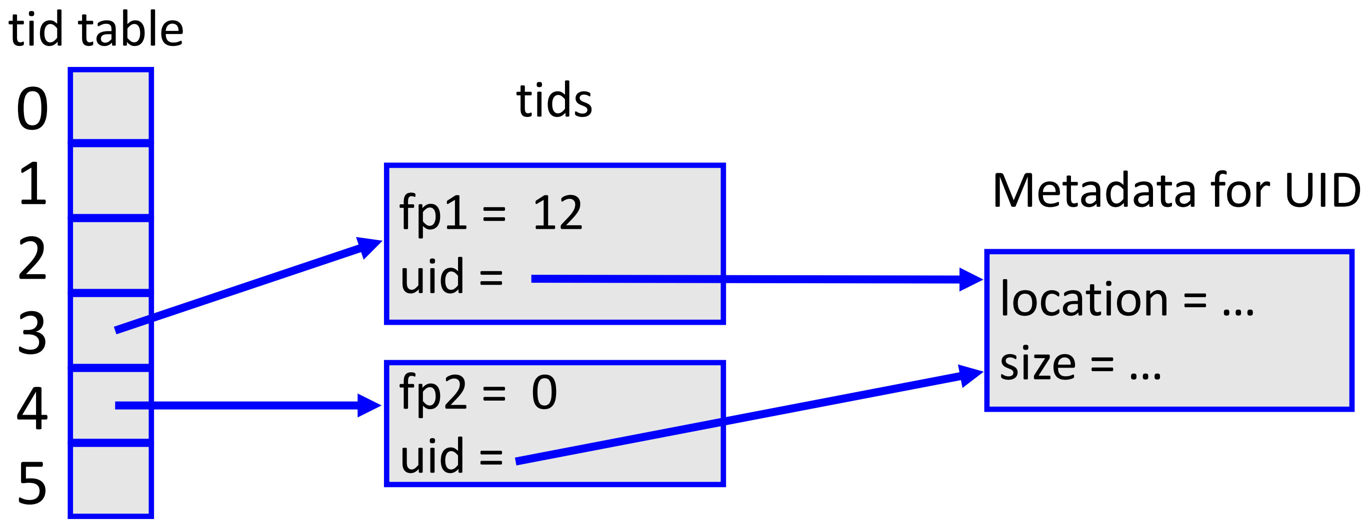
- Creates an instance of file with *uid*
- **Accessible by this process (or thread) only**
- With the temporary process-unique id *tid*
- ***fp* is associated with *tid*, not with *uid***

Close()

```
Close(tid)
```

- Destroys the instance

Putting Open() together with Read()

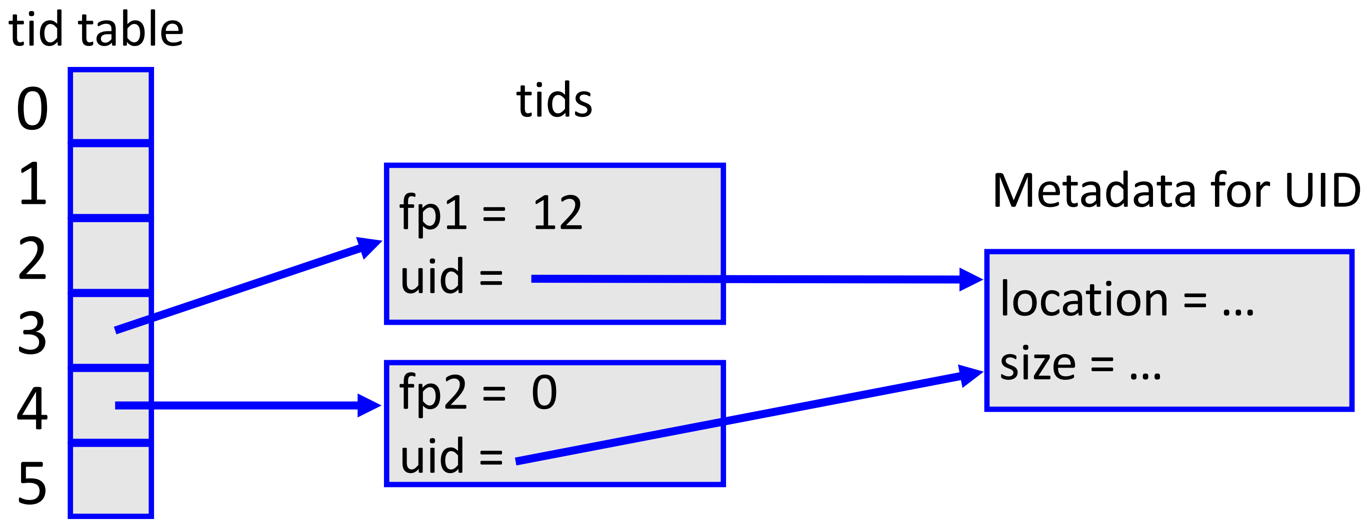



```
tid1 = Open(UID);          // returns 3
Read(tid1, buf, 12);
tid2 = Open(UID);          // returns 4
Close(tid1);
Close(tid2);
```

Putting Open() together with Read()

Why start at 3?

Each running process has 3 files open:
- 0: standard input
- 1: standard output
- 2: standard error




```
tid1 = Open(UID);          // returns 3
Read(tid1, buf, 12);
tid2 = Open(UID);          // returns 4
Close(tid1);
Close(tid2);
```

Putting Open() together with Write()

Different possible semantics:

- Separate file instances altogether
  - Writes by one process not visible to others
- Separate file instances until Close()
  - Writes visible after Close().
  - How to manage merging of instances after close?
- One single instance of the file
  - Writes visible immediately to others

- ***In all cases, fp* is private!**

File System Primitives

- Access
- Concurrency
- **Naming**
- Protection

Naming Primitives

- Naming = mapping

  human-readable string ➔ uid

- Directory = collection of such mappings

Directory Structure

Different possibilities:

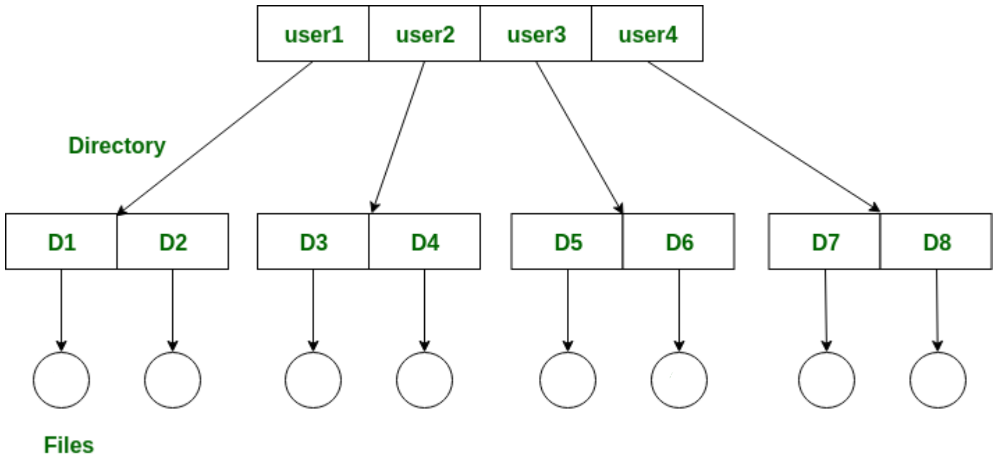
- **Flat** structure
- **Two-level**: [user] filename
- **Hierarchical**: /a/b/c …
  - Root directory
  - Working directory

Directory Primitives

```
CreateDirectory(string)                  // create new directory

DeleteDirectory(string)                  // remove a directory

SetWorkingDirectory(string)              // set directory where you are currently working

string = ListWorkingDirectory()          // retrieve working directory

List(directory)                          // display contents of directory
```

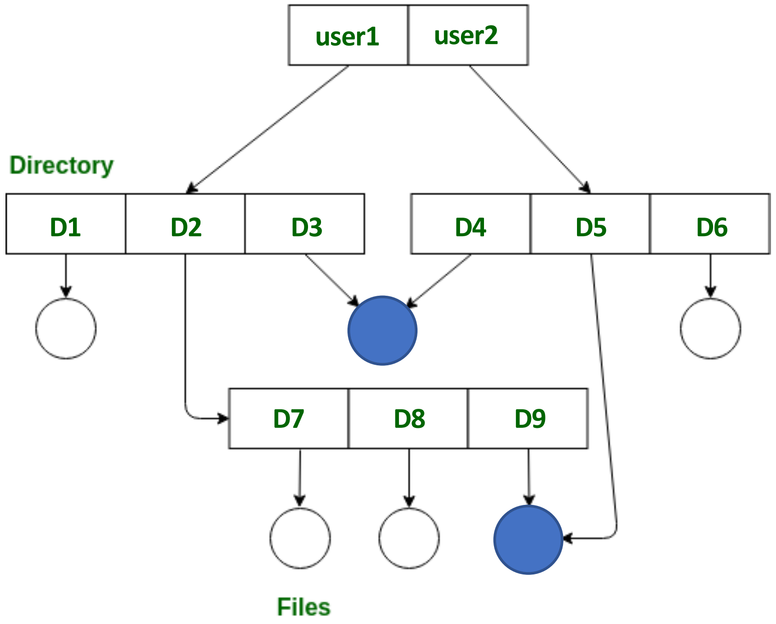
Hierarchical Directory Structures

Tree



(Acyclic) Graph

Allows sharing of *uids* under different names



Hard Links vs Soft Links

Assume mapping `(string1, uid)` already exists

Hard Link

`HardLink(string2, uid)`

After HardLink, two **mappings are equivalent**

Soft Link

`SoftLink(string2, string1)`

After SoftLink, two **mappings are different**

Hard/Soft Link Difference

Assume mapping `(string1, uid)` already exists

Hard Link

```
HardLink(string2, uid)
Remove(string1, uid)
```

Soft Link

```
SoftLink(string2, string1)
Remove(string1, uid)
```

Hard/Soft Link Difference

Assume mapping `(string1, uid)` already exists

Hard Link

```
HardLink(string2, uid)
Remove(string1, uid)
```

Mapping (string2, uid) remains

Soft Link

```
SoftLink(string2, string1)
Remove(string1, uid)
```

Mapping (string2, string1) dangling reference

Hard Links and Soft Links

After the following sequence of file system primitives are executed

Create(name1)
HardLink(name2, name1)
SoftLink(name3, name2)
SoftLink(name4, name2)
HardLink(name5, name3)
Delete(name3)
Open(name5)
Delete(name2)
Open(name4)

Describe the result of each of the two Open()s, and explain your answer.

Hard Links and Soft Links

Create(name1)

HardLink(name2, name1)

SoftLink(name3, name2)

SoftLink(name4, name2)

HardLink(name5, name3)

Delete(name3)

Open(name5)

Delete(name2)

Open(name4)

Hard Links and Soft Links

Create(name1)

HardLink(name2, name1)

SoftLink(name3, name2)

SoftLink(name4, name2)

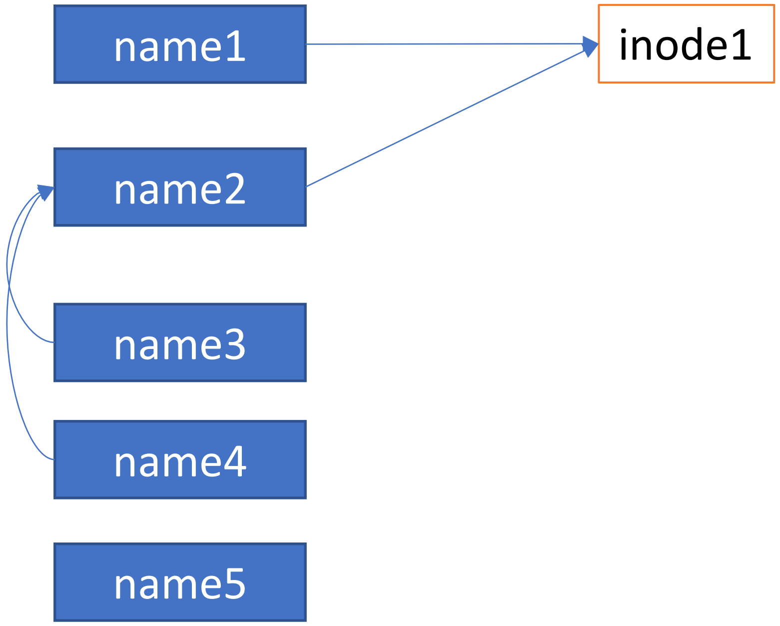
HardLink(name5, name3)

Delete(name3)

Open(name5)

Delete(name2)

Open(name4)



Hard Links and Soft Links

Create(name1)

HardLink(name2, name1)

SoftLink(name3, name2)

SoftLink(name4, name2)

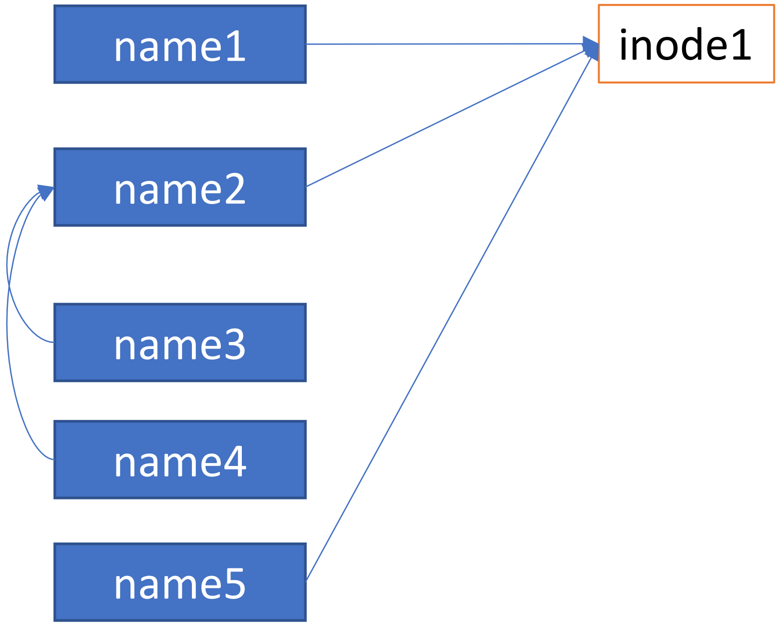
HardLink(name5, name3)

Delete(name3)

Open(name5)

Delete(name2)

Open(name4)



Hard Links and Soft Links

Create(name1)

HardLink(name2, name1)

SoftLink(name3, name2)

SoftLink(name4, name2)

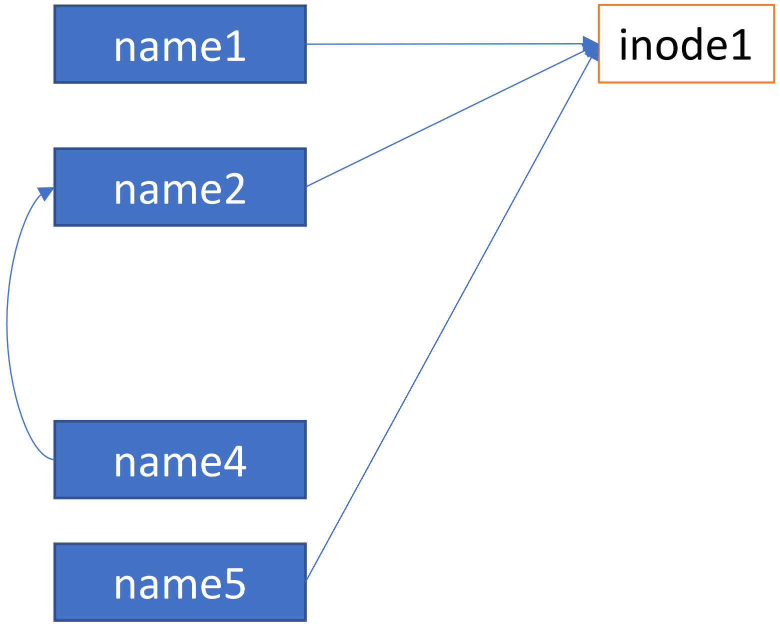
HardLink(name5, name3)

Delete(name3)

Open(name5)

Delete(name2)

Open(name4)



Hard Links and Soft Links

Create(name1)

HardLink(name2, name1)

SoftLink(name3, name2)

SoftLink(name4, name2)

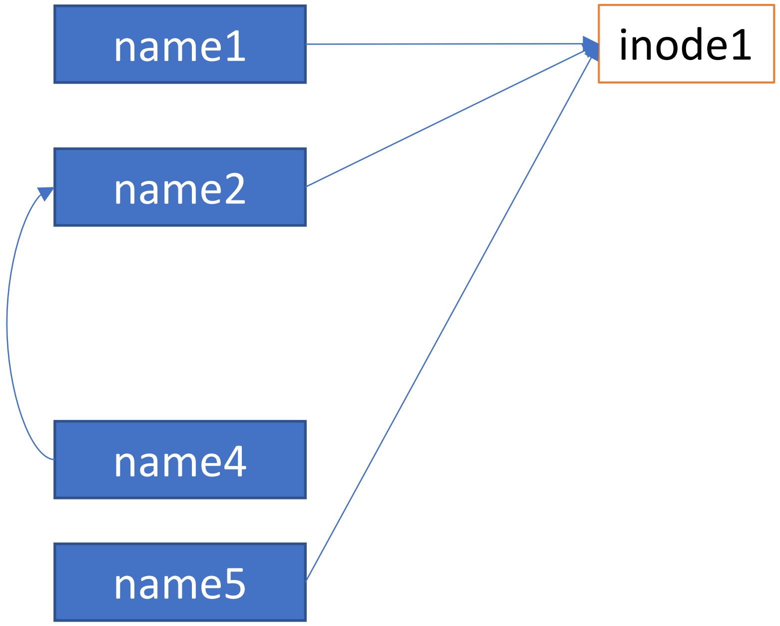
HardLink(name5, name3)

Delete(name3)

Open(name5) ← opens file

Delete(name2)

Open(name4)



Hard Links and Soft Links

Create(name1)

HardLink(name2, name1)

SoftLink(name3, name2)

SoftLink(name4, name2)

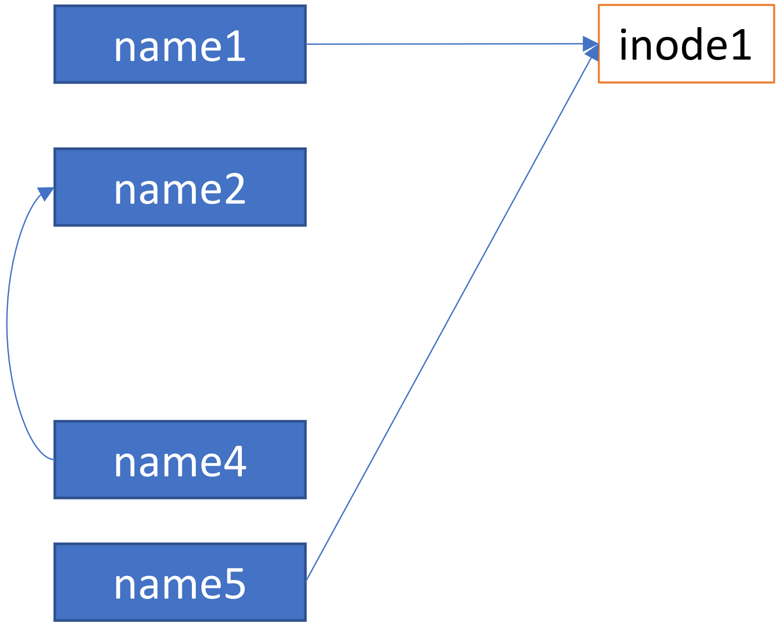
HardLink(name5, name3)

Delete(name3)

Open(name5) ← opens file

Delete(name2)

Open(name4)



Hard Links and Soft Links

Create(name1)

HardLink(name2, name1)

SoftLink(name3, name2)

SoftLink(name4, name2)

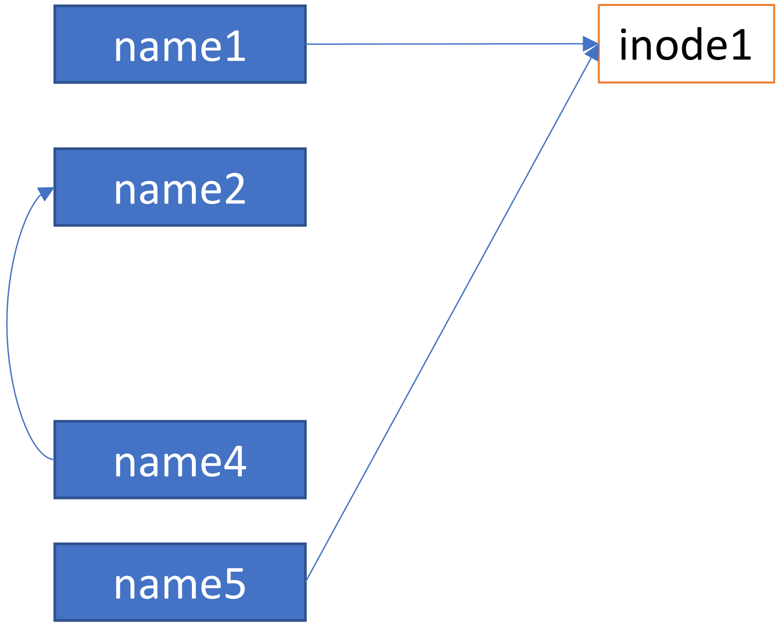
HardLink(name5, name3)

Delete(name3)

Open(name5) ← opens file

Delete(name2)

Open(name4) ← dangling reference



Linux Primitives

Collapses in a single interface:

- Access
- Concurrency
- Naming

```
Creat(string)
  • uid = Create()
  • Insert(string, uid)
fd = Open(string, [optional args])
  • uid = Lookup(string)
  • fd = (tid =) Open(uid, [optional args])
…
```

***uid* is never visible at the user level**

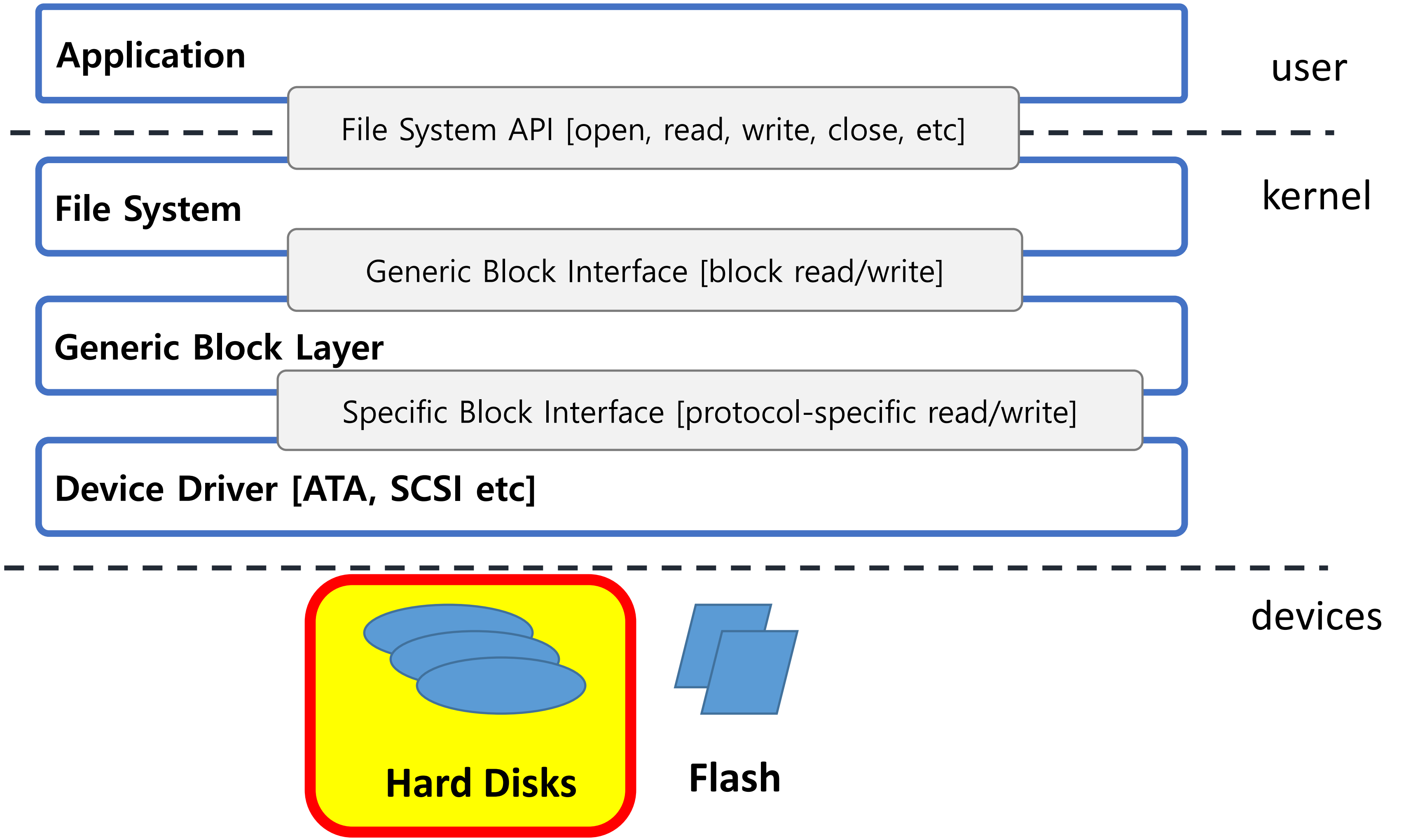
File System Primitives

- Access
- Concurrency
- Naming
- **Protection** ← Later in the course

Summary – File System Interface

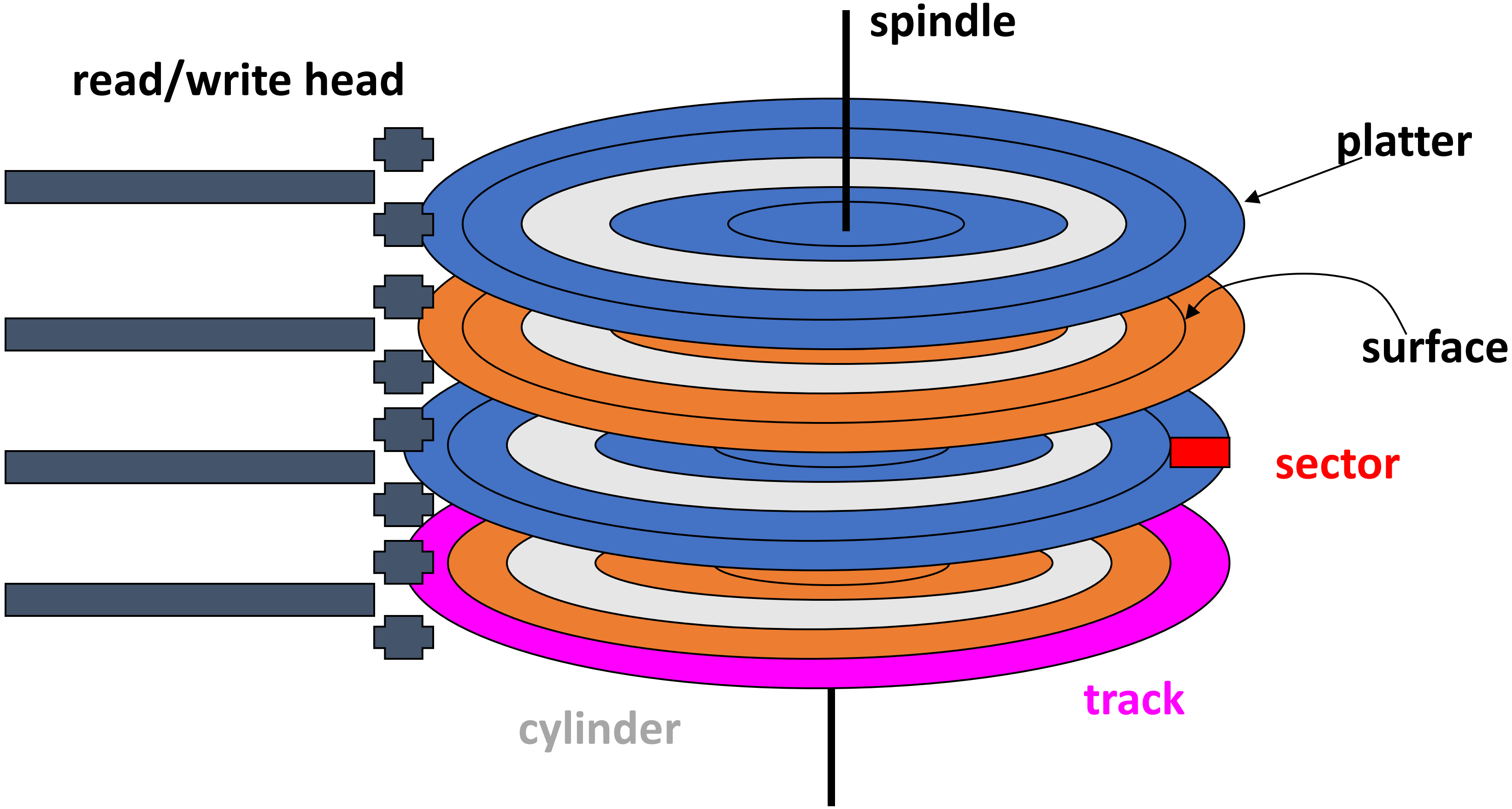
- Permanent storage
- Notion of File
- File system primitives
  - Access, concurrency, naming (, protection)

Disks

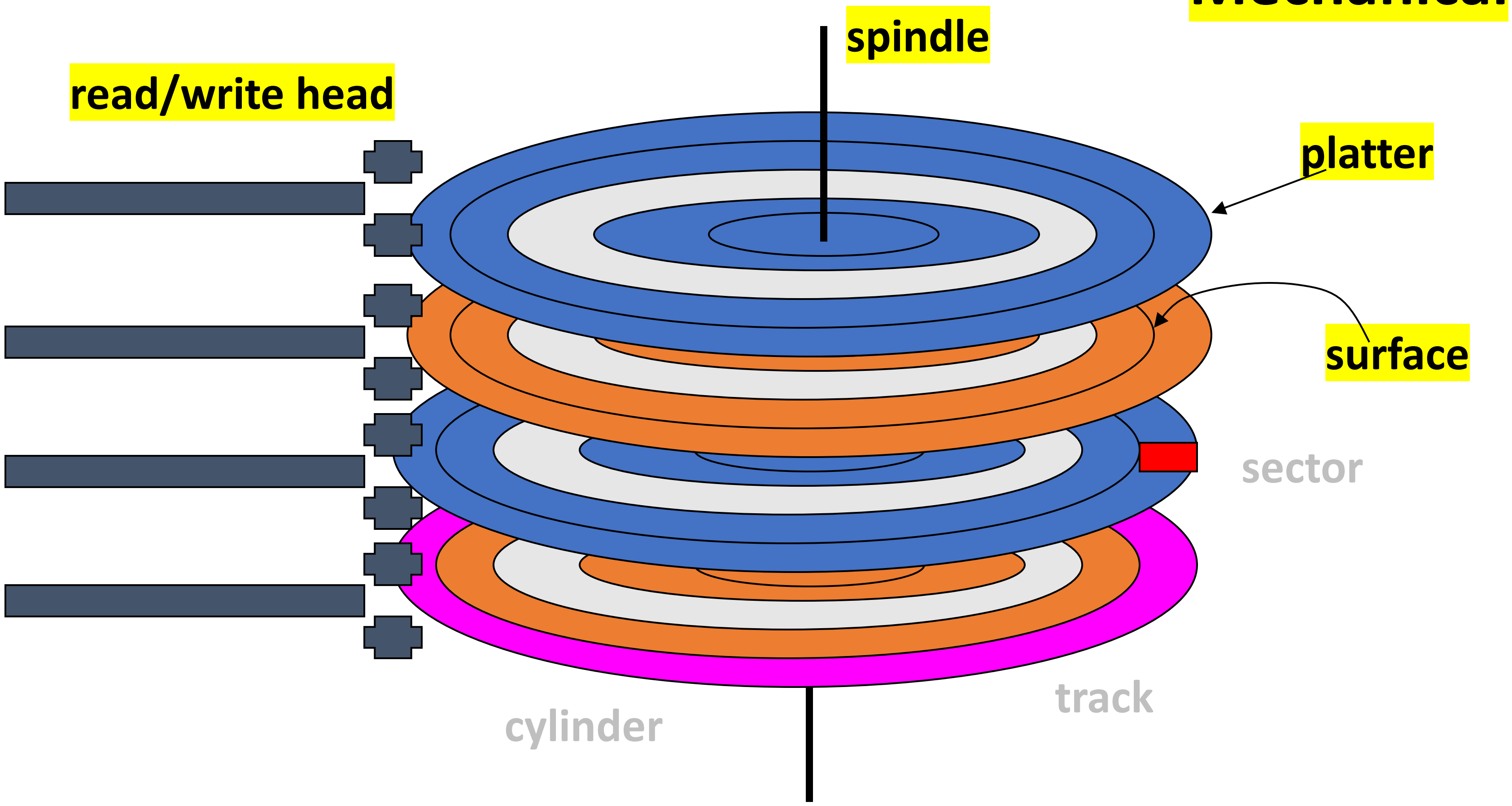

Application
user
File System API [open, read, write, close, etc]
File System
kernel
Generic Block Interface [block read/write]
Generic Block Layer
Specific Block Interface [protocol-specific read/write]
Device Driver [ATA, SCSI etc]
devices
Hard Disks
Flash

Disk

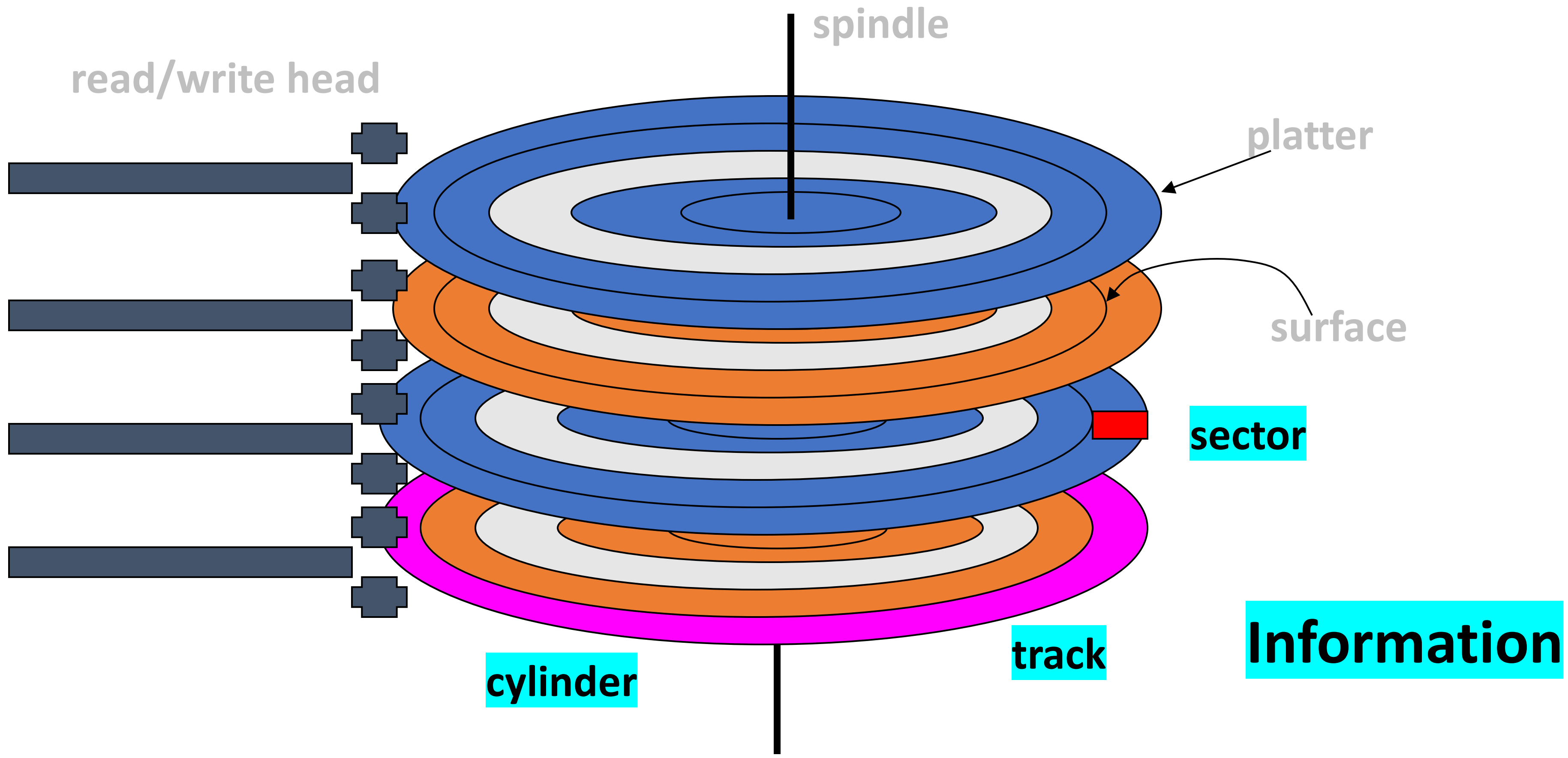
Disk Terminology



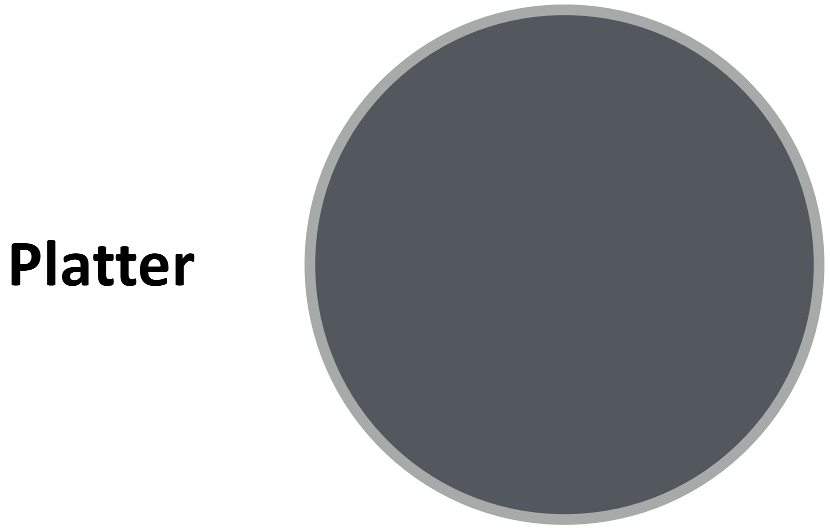
Disk Terminology



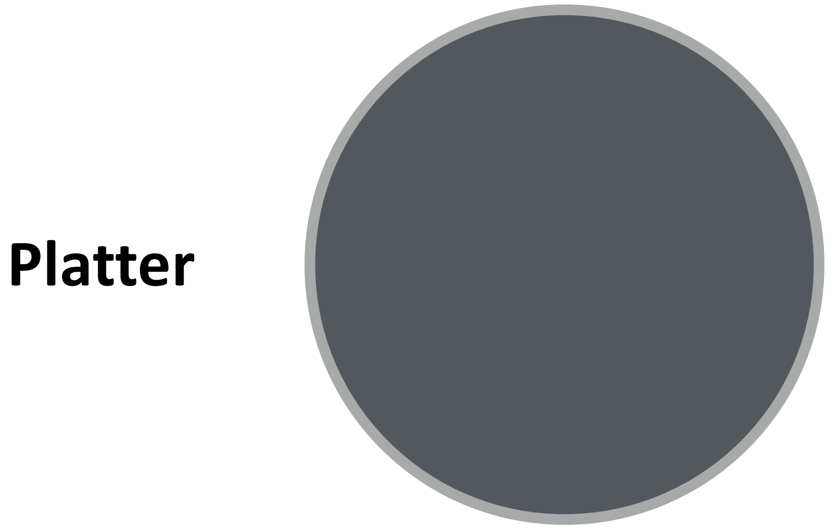
Disk Terminology



Disk Internals

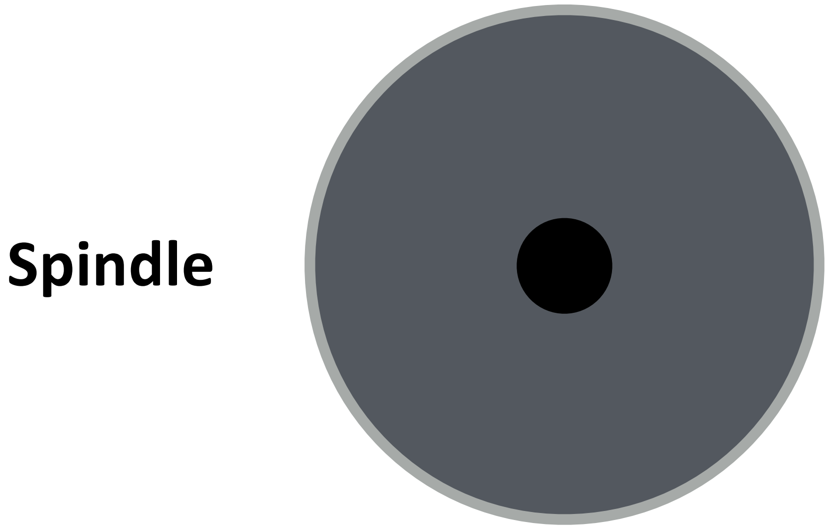


Disk Internals

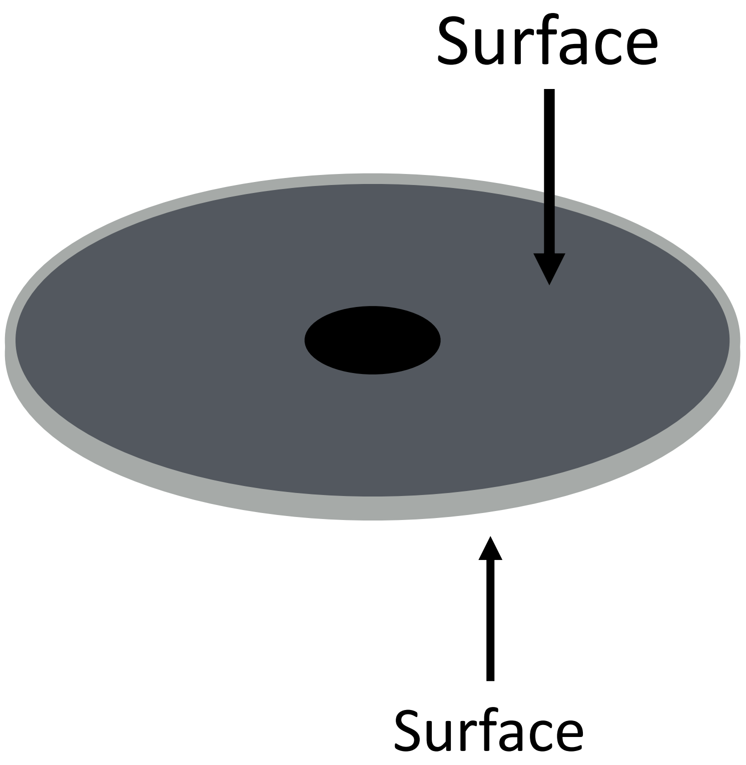


Platter is covered with a magnetic film.

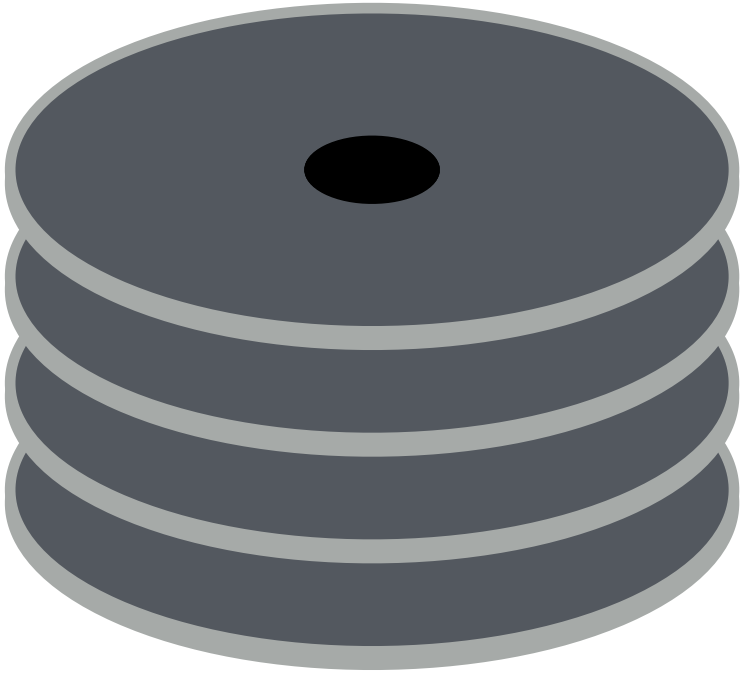
Disk Internals



Disk Internals

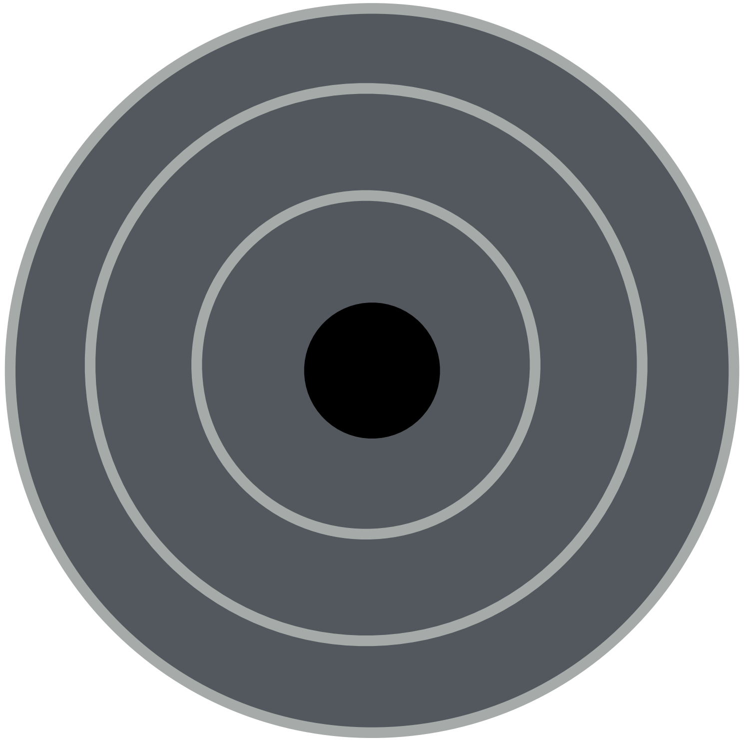


Disk Internals



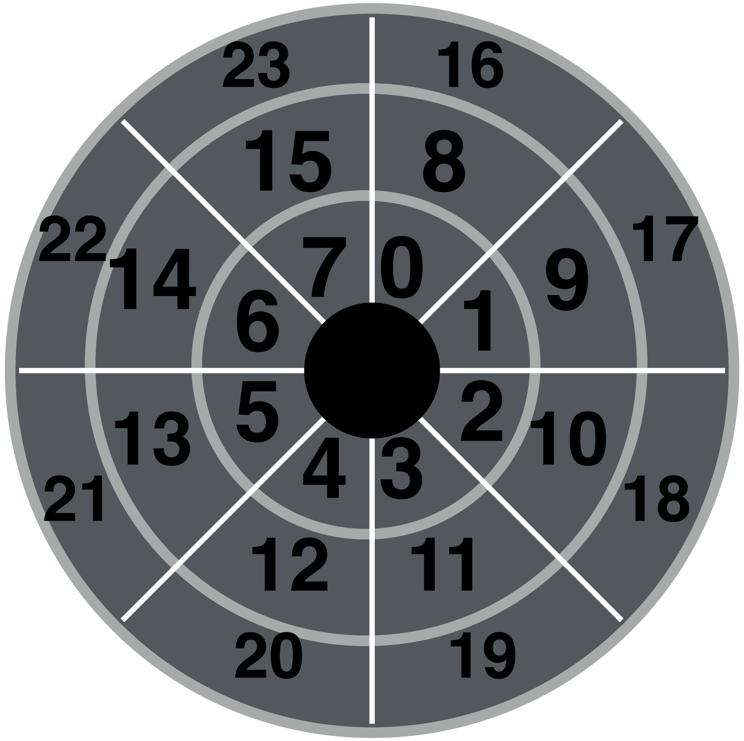
Many platters may be bound to the spindle.

Disk Internals



Each surface is logically divided into rings called **tracks.**
A stack of tracks (across platters) is called a **cylinder.**

Disk Internals

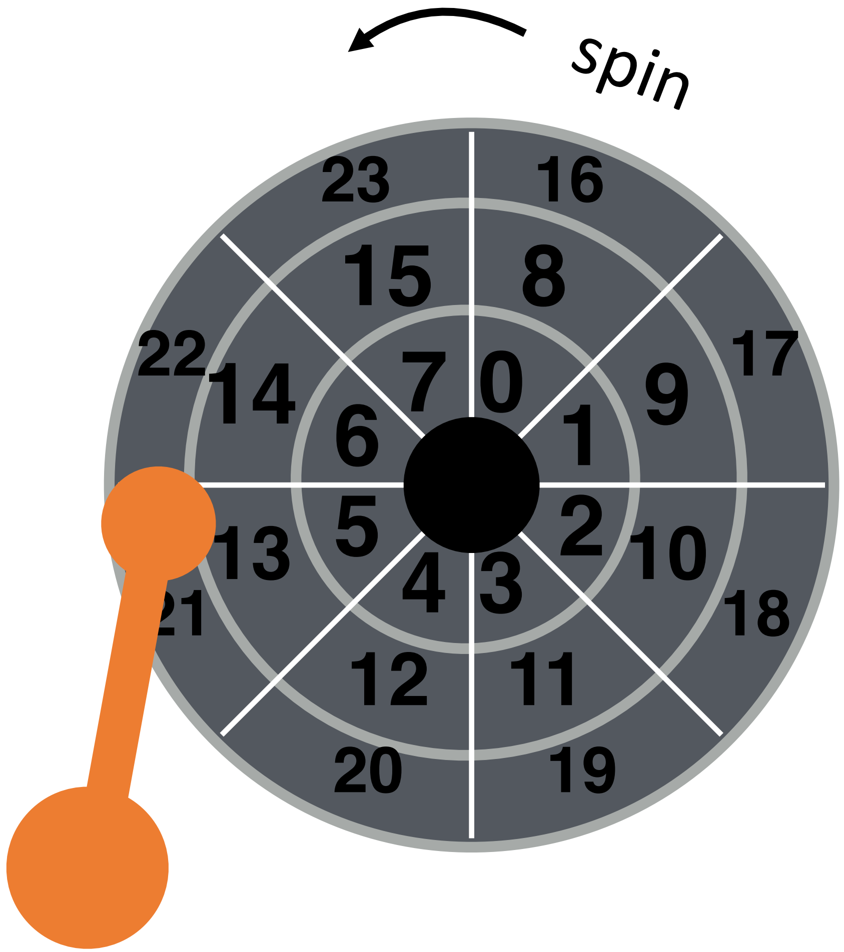


The tracks are logically divided into numbered **sectors**.

Disk Internals

Heads on a moving **arm** can read from each surface.

Disk Internals



Spindle/platters rapidly spin

Disk Characteristics

- **Size:** typically from 1.8” to 3.5”
- **Capacity:** from tens of Gb to a few Tb

Disk Interface

- **Accessible by sector only**
  - ReadSector (logical_sector_number, buffer)
  - WriteSector(logical_sector_number, buffer)

Disk Interface

- **Accessible by sector only**
  - ReadSector (**logical_sector_number**, buffer)
  - WriteSector(logical_sector_number, buffer)

- Logical_sector_number =
  - **Platter**
  - **Cylinder or track**
  - **Sector**

A Look Ahead at File System Implementation

The **main task of the file system** is to translate

From user interface

- `Read(uid, buffer, bytes)`

To disk interface

- `ReadSector(logical_sector_number, buffer)`

Two Small Simplifications

1. User Read() allows arbitrary number of bytes

→ Simplify to only allowing Read() of a block

Read(uid, block_number)

A block is fixed-size

Two Small Simplifications

1. User Read() allows arbitrary number of bytes

    → Simplify to only allowing Read() of a block

    Read(uid, block_number)

    A block is fixed-size

2. Typically *block_size = 2^n * sector_size*

    Example: Block size = 4,096 bytes, Sector size = 512 bytes

    → For simplicity of presentation in class

    block_size = sector_size

Back to Disk Interface

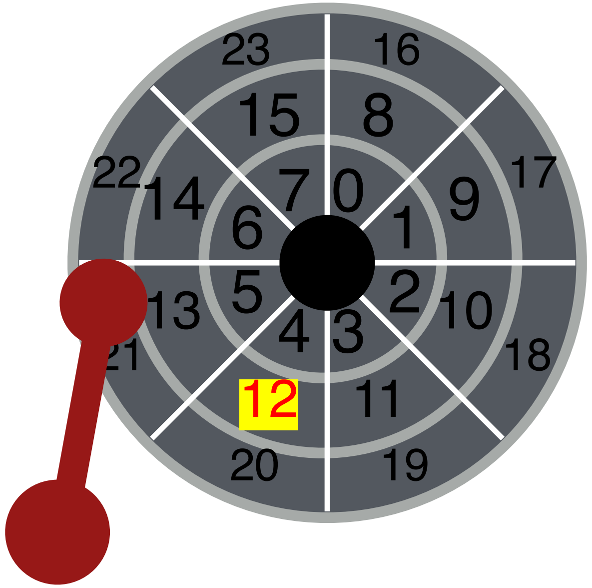
- **Accessible by sector only**
  - ReadSector (logical_sector_number, buffer)
  - WriteSector(logical_sector_number, buffer)

- Logical_sector_number =
  - Platter
  - Cylinder or track
  - Sector

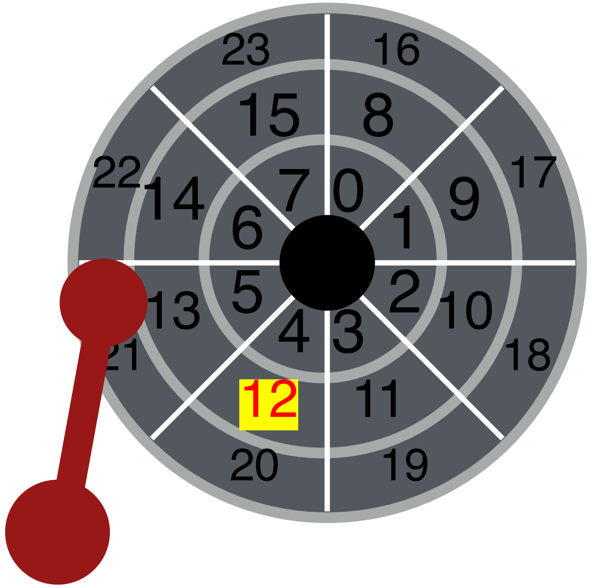
Disk Access

- Head selection – select platter
- Seek – move arm over cylinder
- Rotational latency – move head over sector
- Transfer time – read from sector

Let’s Read 12!



Select platter.



Head selection

- Electronic switch
- ~ nanoseconds

Seek to right track.

Seek Time

- Approx. linear in the number of cylinders
- 3 to 12 milliseconds

Seek to right track.

Seek Time

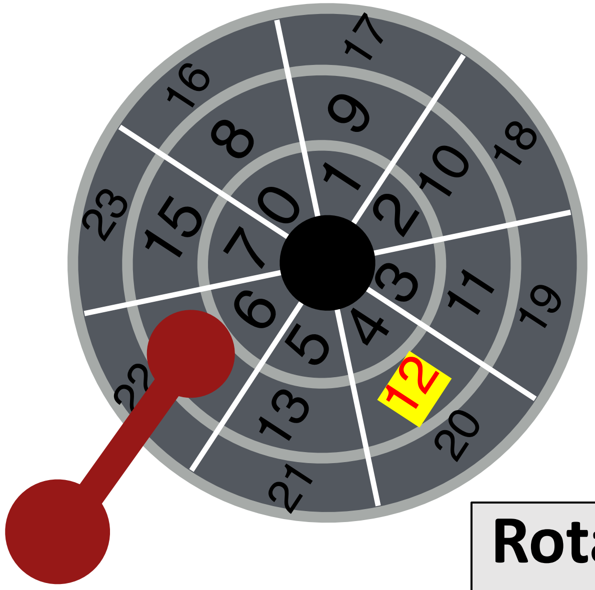
- Approx. linear in the number of cylinders
- 3 to 12 milliseconds

Seek to right track.

Seek Time

- Approx. linear in the number of cylinders
- 3 to 12 milliseconds

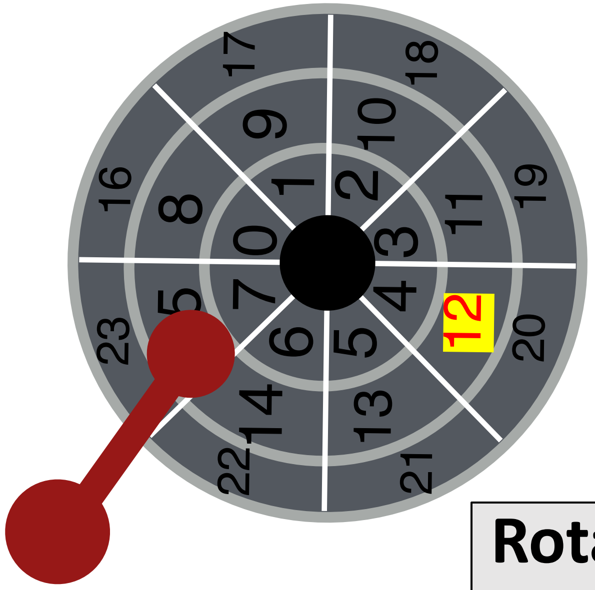
Wait for rotation.



Rotational Latency

- Linear in the number of sectors
- Rotational speed: 4,500 -15,000 RPM
- One revolution = 1 / (RPM/60)sec
- Avg. rotational latency = ½ revolution
- From 2 to 7.1 milliseconds

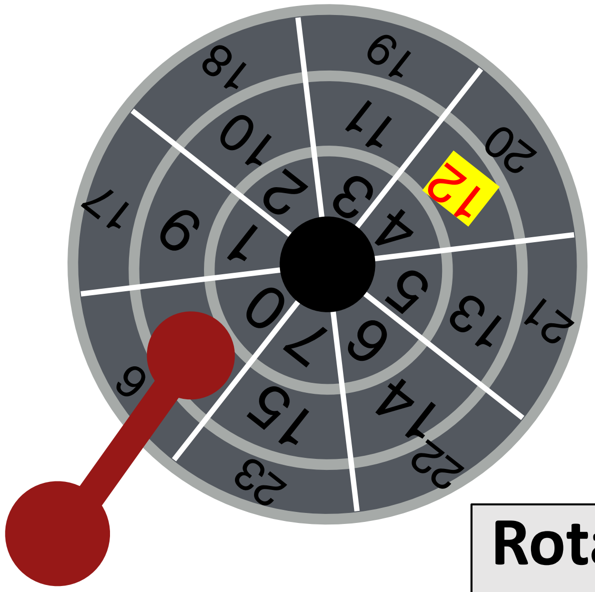
Wait for rotation.



Rotational Latency

- Linear in the number of sectors
- Rotational speed: 4,500 -15,000 RPM
- One revolution = 1 / (RPM/60)sec
- Avg. rotational latency = ½ revolution
- From 2 to 7.1 milliseconds

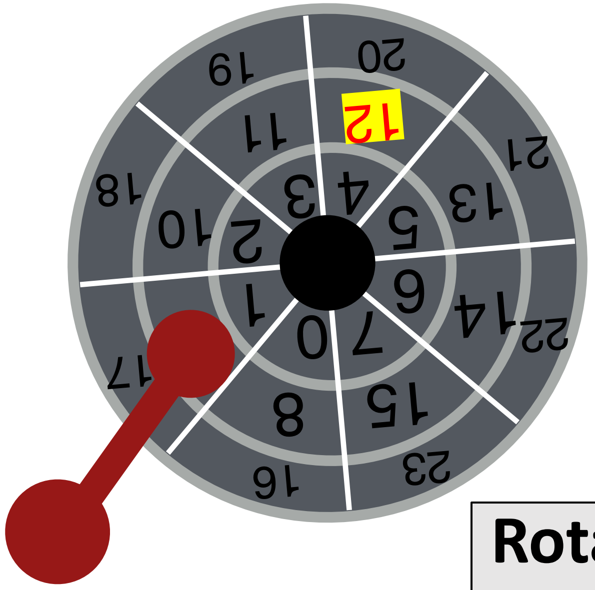
Wait for rotation.



Rotational Latency

- Linear in the number of sectors
- Rotational speed: 4,500 -15,000 RPM
- One revolution = 1 / (RPM/60)sec
- Avg. rotational latency = ½ revolution
- From 2 to 7.1 milliseconds

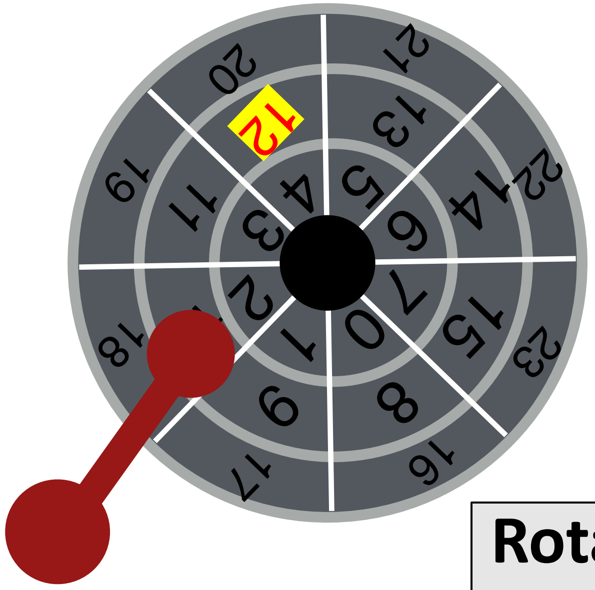
Wait for rotation.



Rotational Latency

- Linear in the number of sectors
- Rotational speed: 4,500 -15,000 RPM
- One revolution = 1 / (RPM/60)sec
- Avg. rotational latency = ½ revolution
- From 2 to 7.1 milliseconds

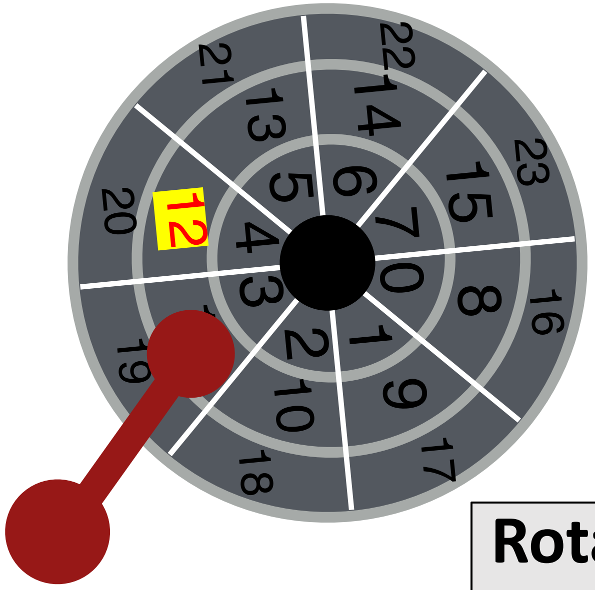
Wait for rotation.



Rotational Latency

- Linear in the number of sectors
- Rotational speed: 4,500 -15,000 RPM
- One revolution = 1 / (RPM/60)sec
- Avg. rotational latency = ½ revolution
- From 2 to 7.1 milliseconds

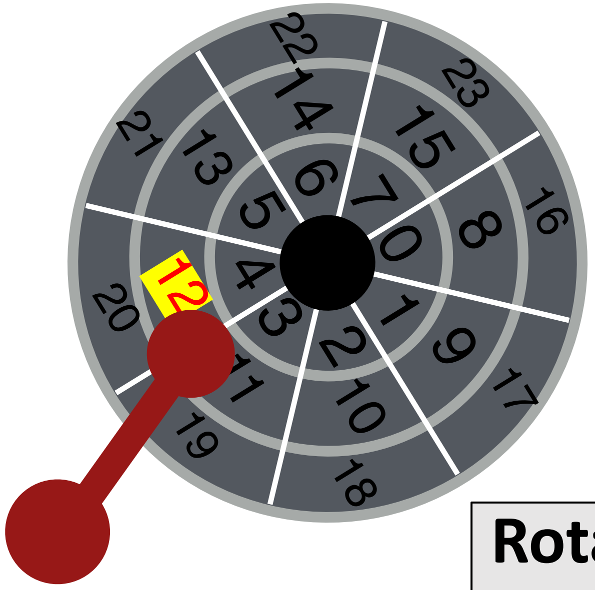
Wait for rotation.



Rotational Latency

- Linear in the number of sectors
- Rotational speed: 4,500 -15,000 RPM
- One revolution = 1 / (RPM/60)sec
- Avg. rotational latency = ½ revolution
- From 2 to 7.1 milliseconds

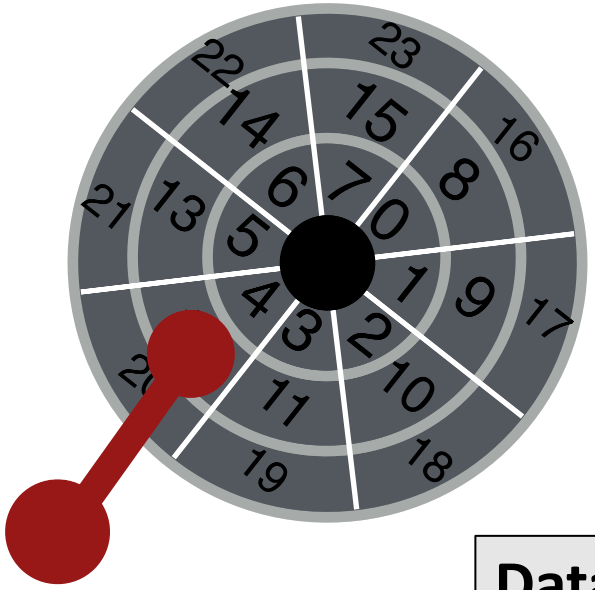
Transfer data.



Rotational Latency

- Linear in the number of sectors
- Rotational speed: 4,500 -15,000 RPM
- One revolution = 1 / (RPM/60)sec
- Avg. rotational latency = ½ revolution
- From 2 to 7.1 milliseconds

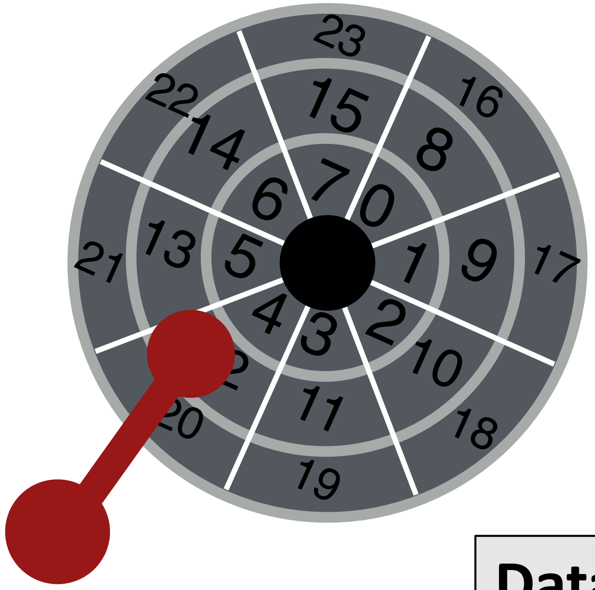
Transfer data.



Data Transfer

- Effective transfer rate ~ 1 Gbyte per second
- Sector = 512 bytes
- Transfer time ~ 0.5 microseconds

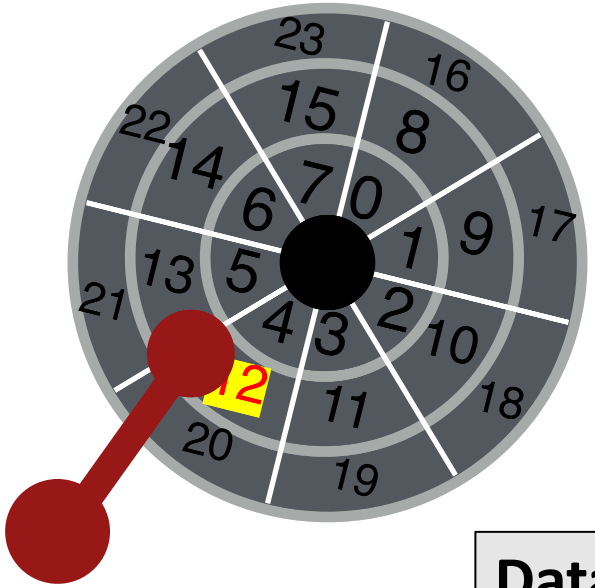
Transfer data.



Data Transfer

- Effective transfer rate ~ 1 Gbyte per second
- Sector = 512 bytes
- Transfer time ~ 0.5 microseconds

Done!



Data Transfer

- Effective transfer rate ~ 1 Gbyte per second
- Sector = 512 bytes
- Transfer time ~ 0.5 microseconds

Further Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 36, 37, 39.

https://pages.cs.wisc.edu/~remzi/OSTEP/

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University).