# Week 10

# Persistent Storage: Intro to File Systems

Max Kopinsky
11 March, 2025

# 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

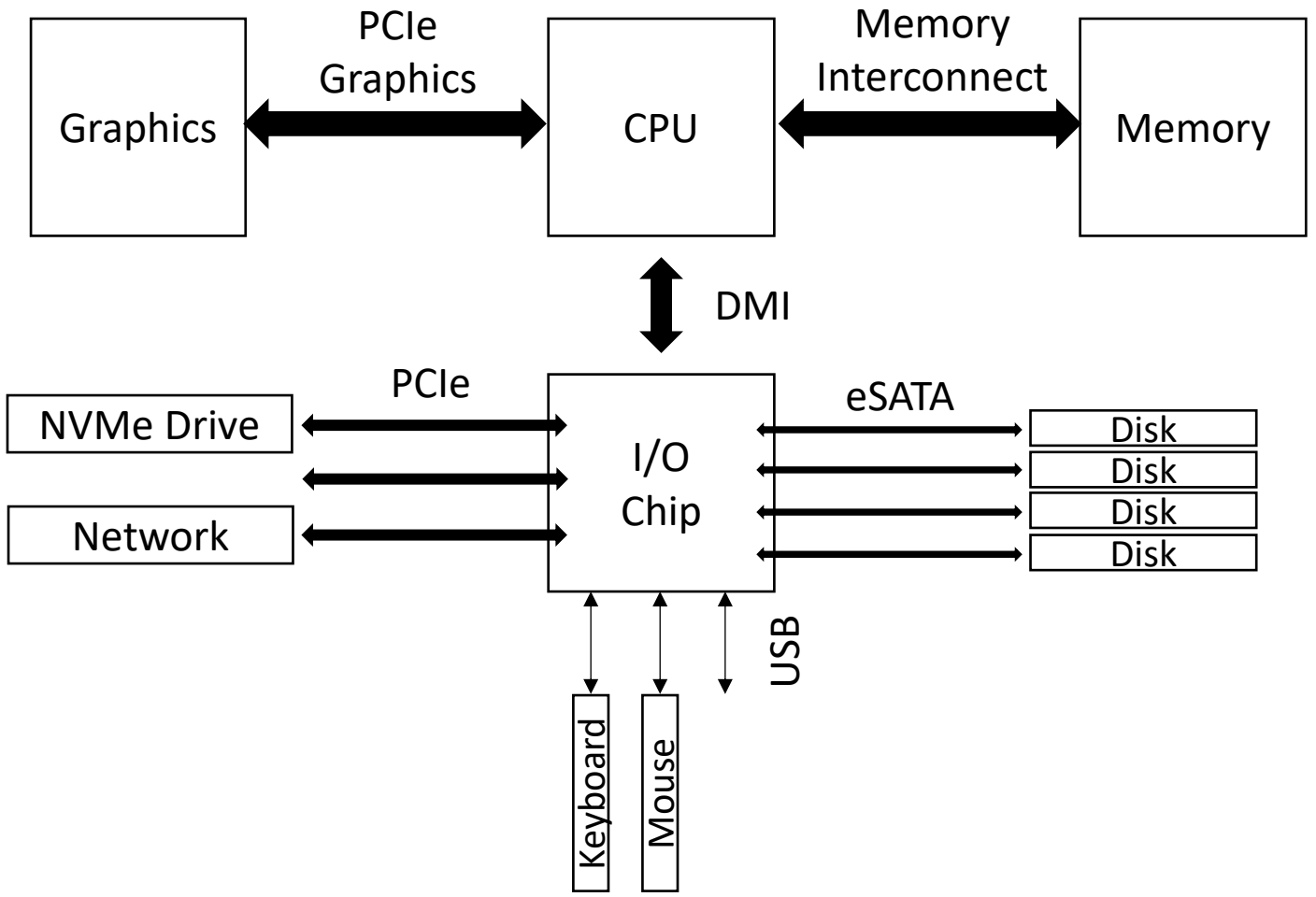
# How should I/O be integrated into systems?

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

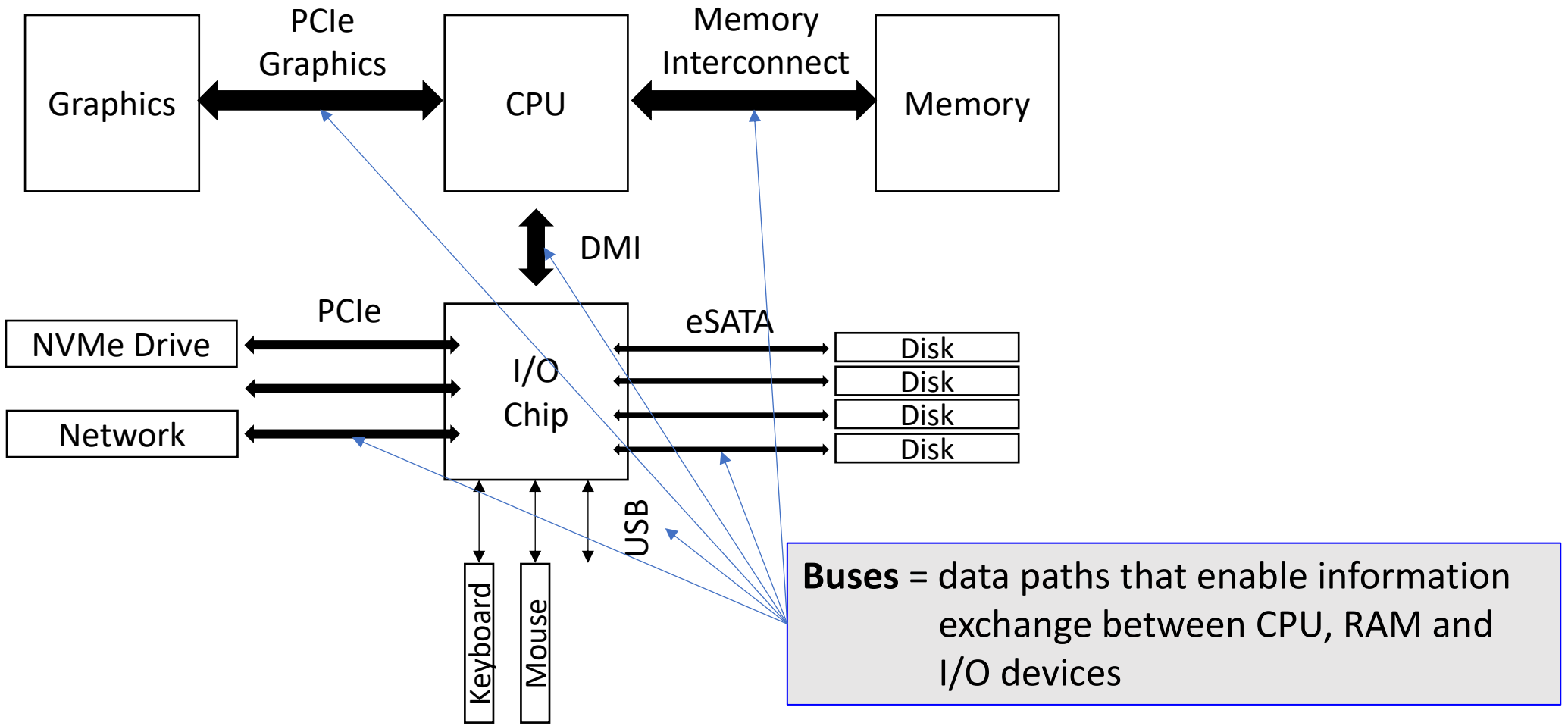




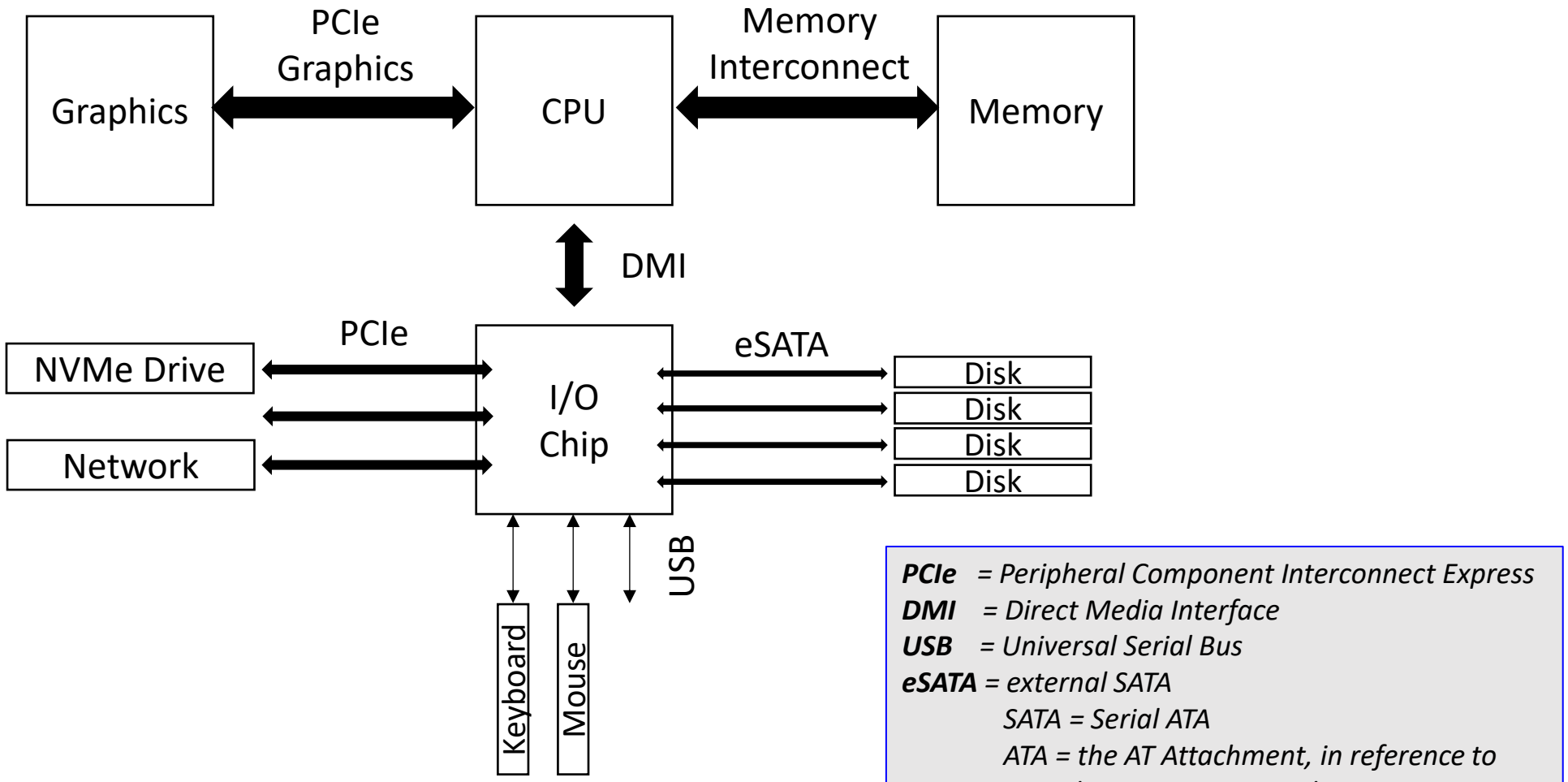
# I/O System Architecture (typical modern)



# I/O System Architecture (typical modern)



# I/O System Architecture (typical modern)




***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

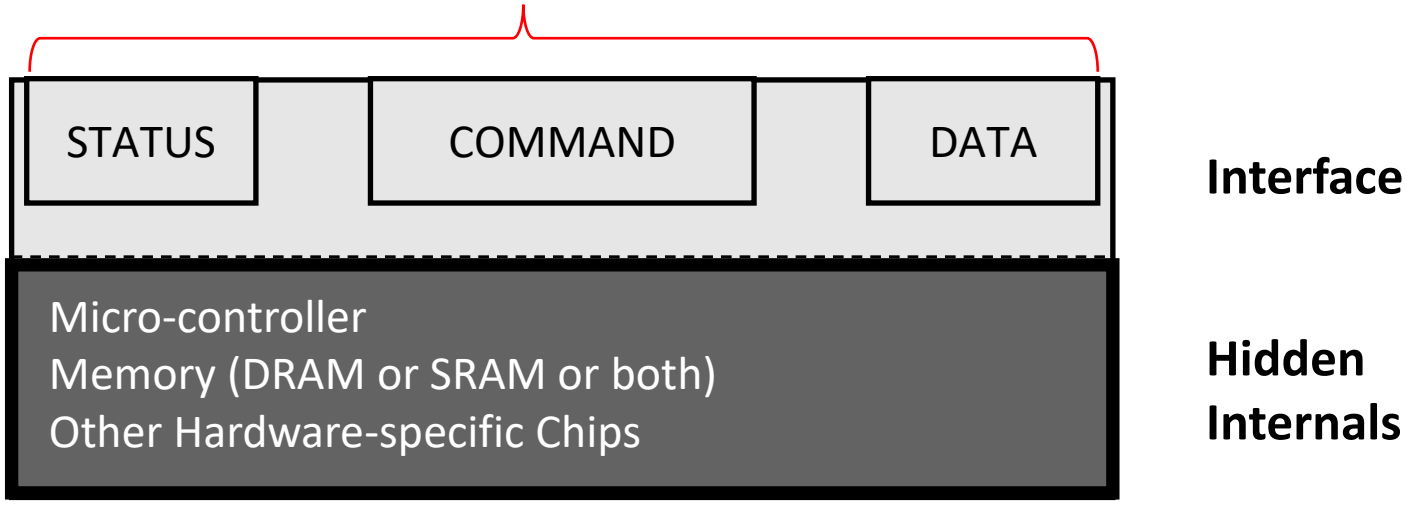
Interface made of **3 registers:**

- **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

OS reads/writes to these to control device behavior



# 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

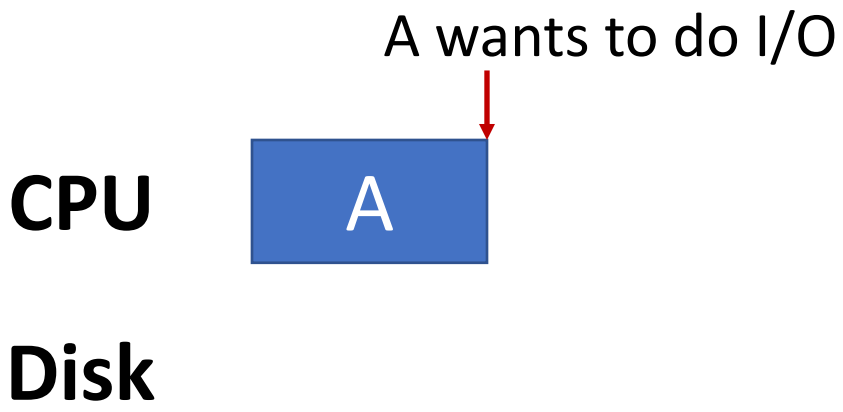
```
write data to DATA register
write command to COMMAND register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //spin-wait until device is done with your request
```

# Polling

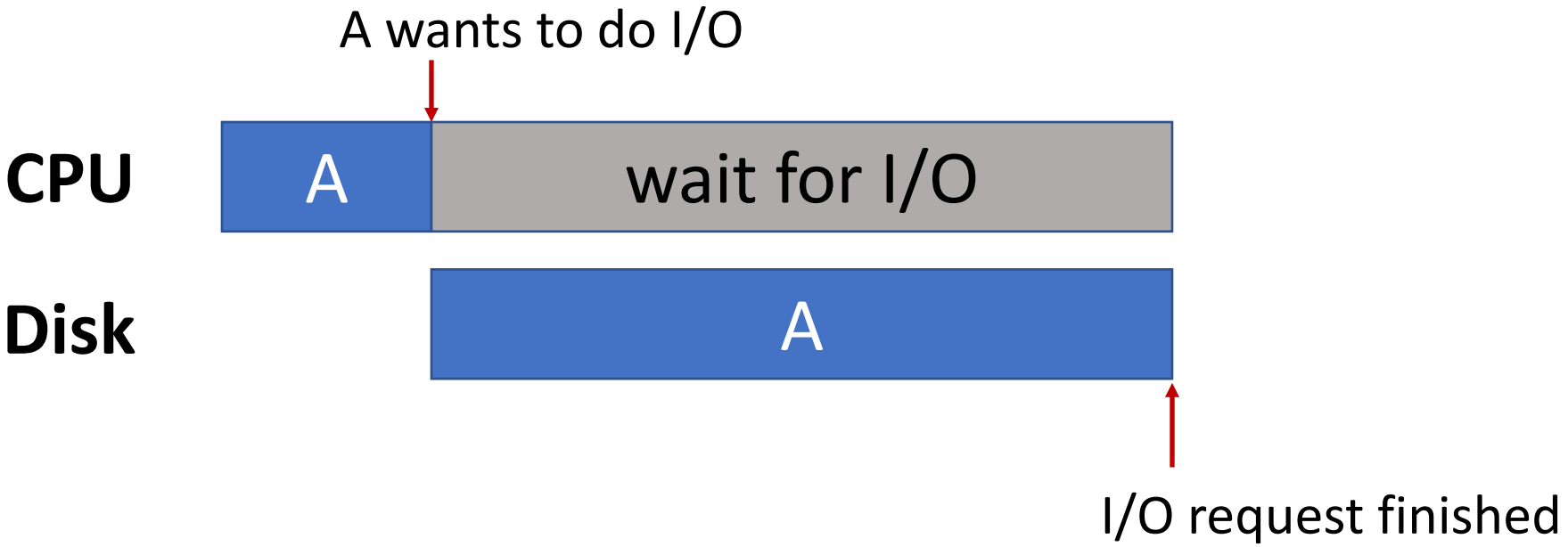
```
write data to DATA register
write command to COMMAND register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //spin-wait until device is done with your request
```

Wasted CPU cycles

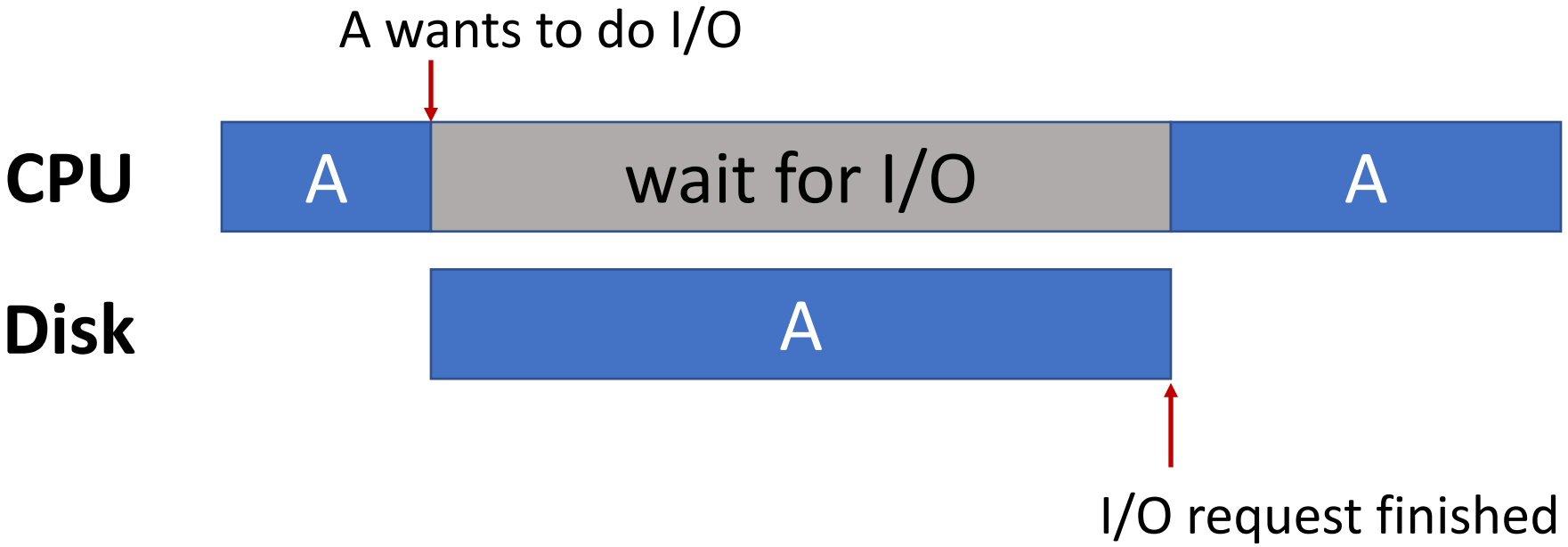
# Polling Example



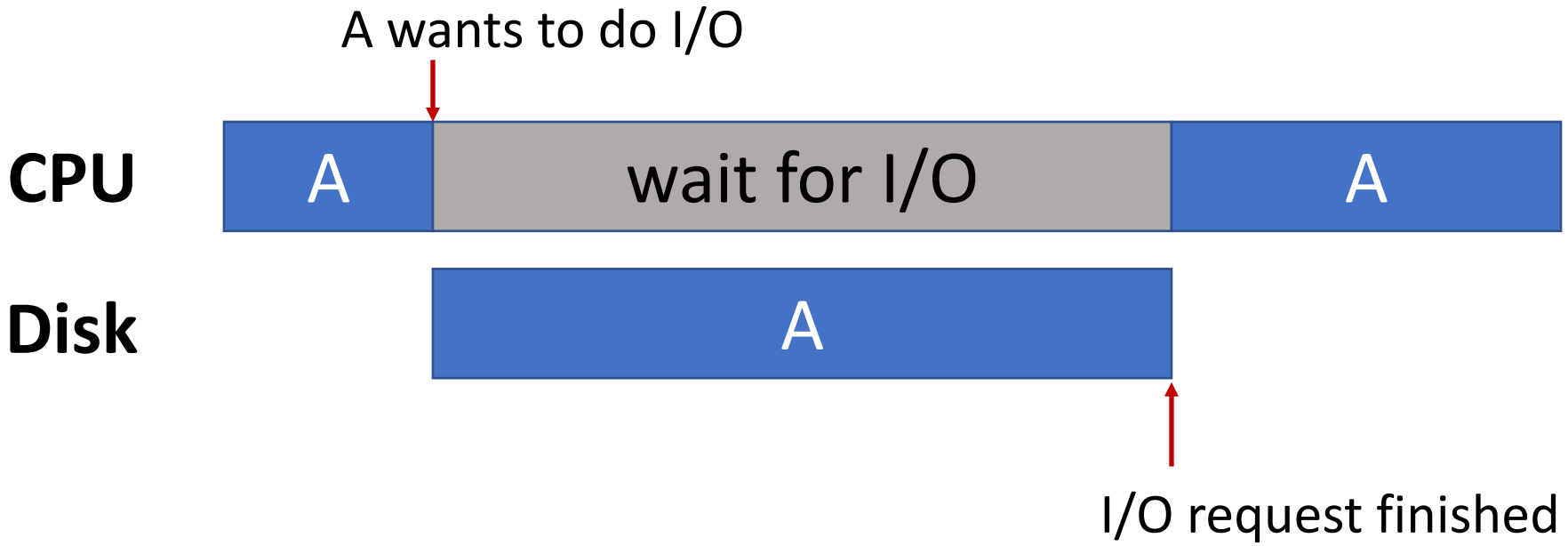
# Polling Example



# 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, alert OS 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, alert OS 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. OS handler will often cause context switch.
- **Syscall:** generated by process, to request functionality from kernel mode. Can also result in context switches, but most "fast" system calls won't.

# 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

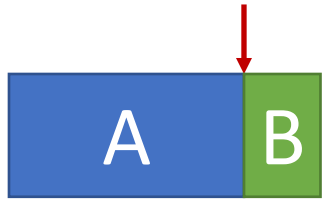
A wants to do I/O;
Send I/O request **and sleep**


**CPU**

**Disk**

# Interrupts Example

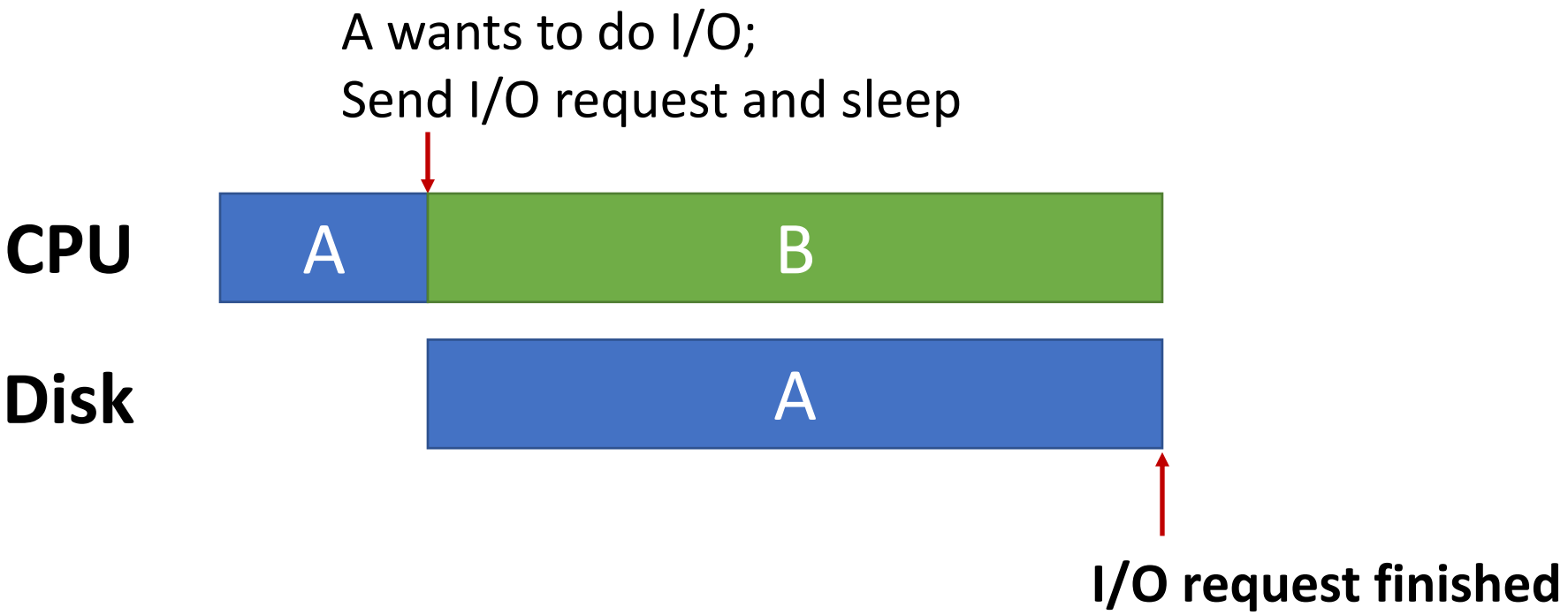
A wants to do I/O;
Send I/O request and sleep



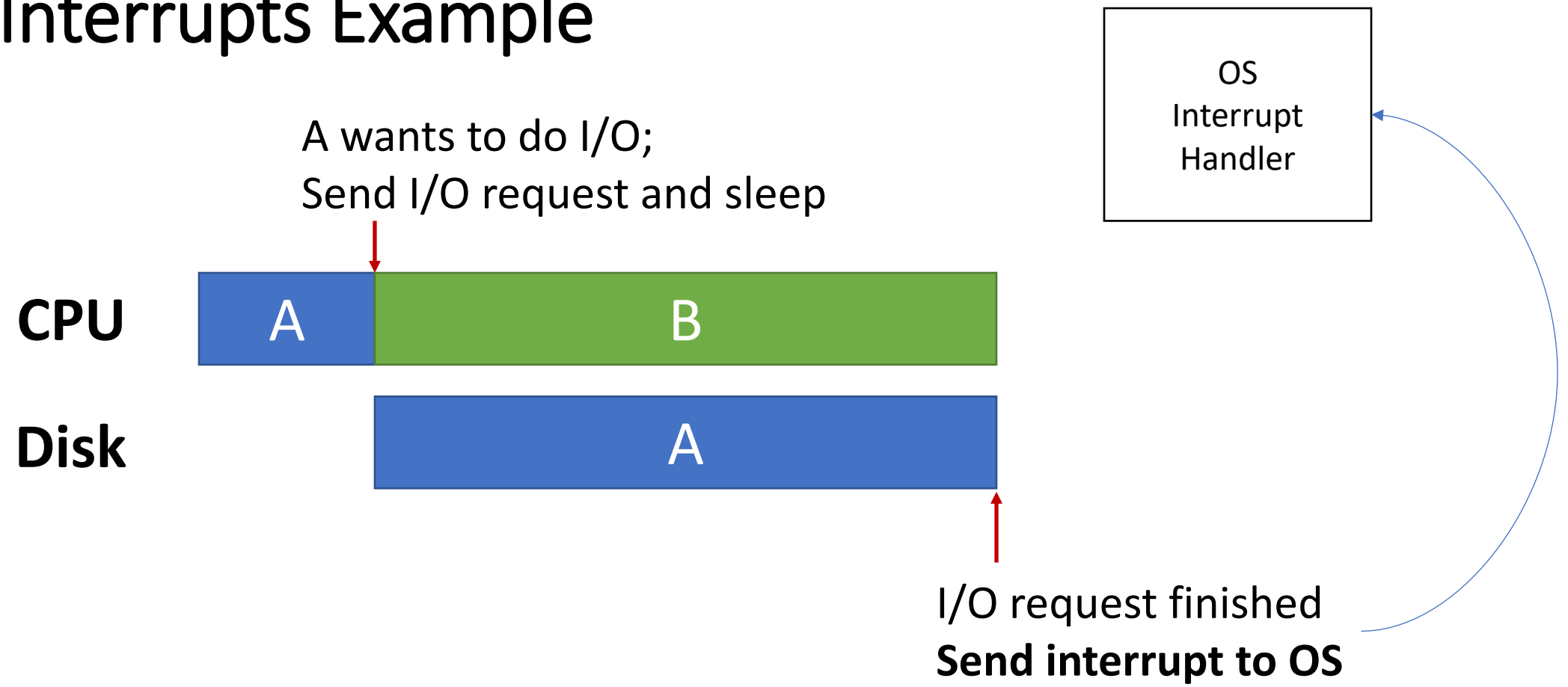

**CPU** **Context switch to B**

**Disk**

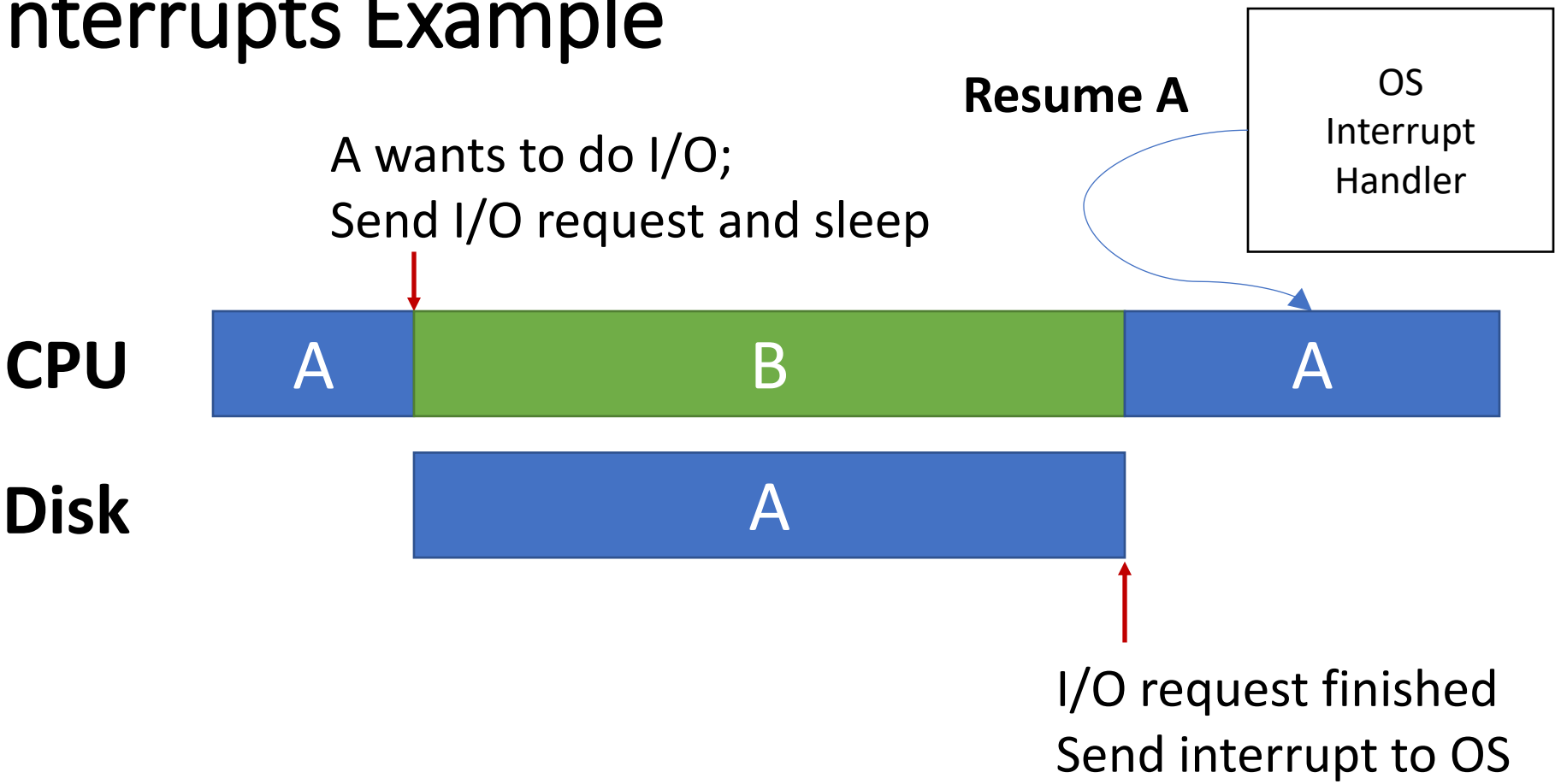
# Interrupts Example



# 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

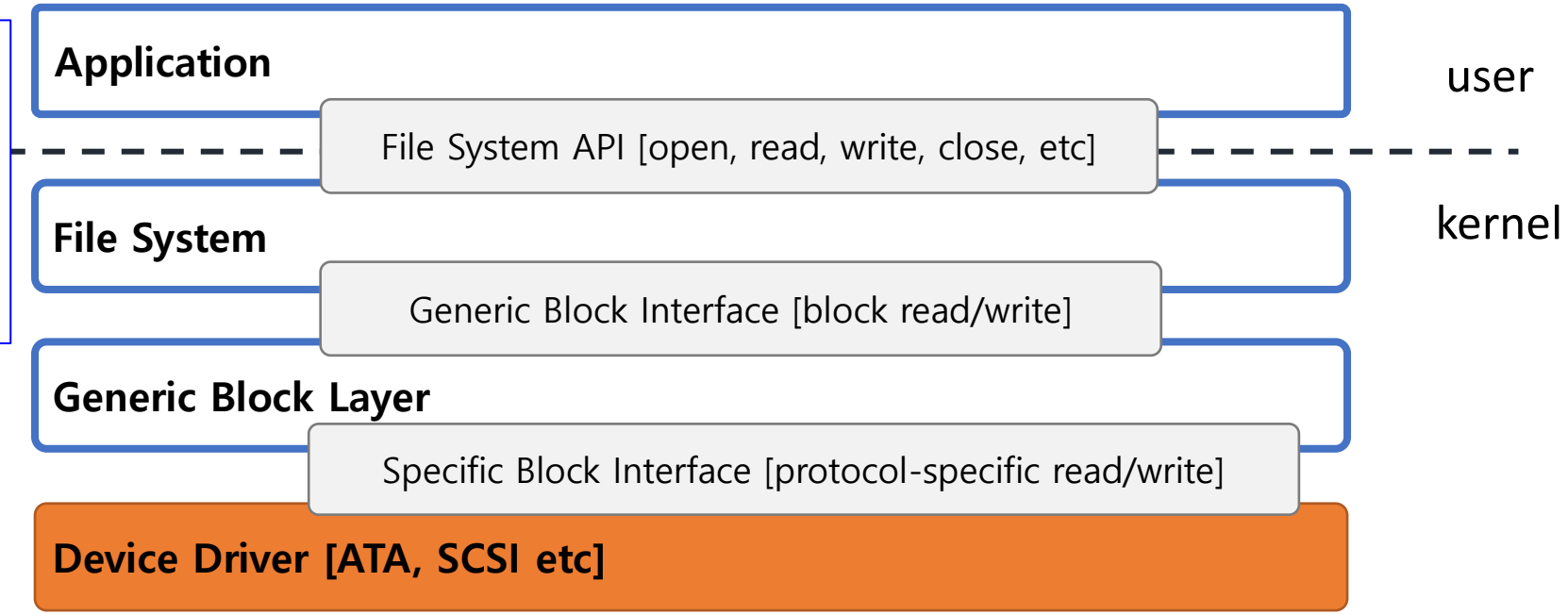
→Drivers are **~70%** of Linux source code

*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."*

*(but Linux also uses device drivers for lots of non-hardware tasks)*

# 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**

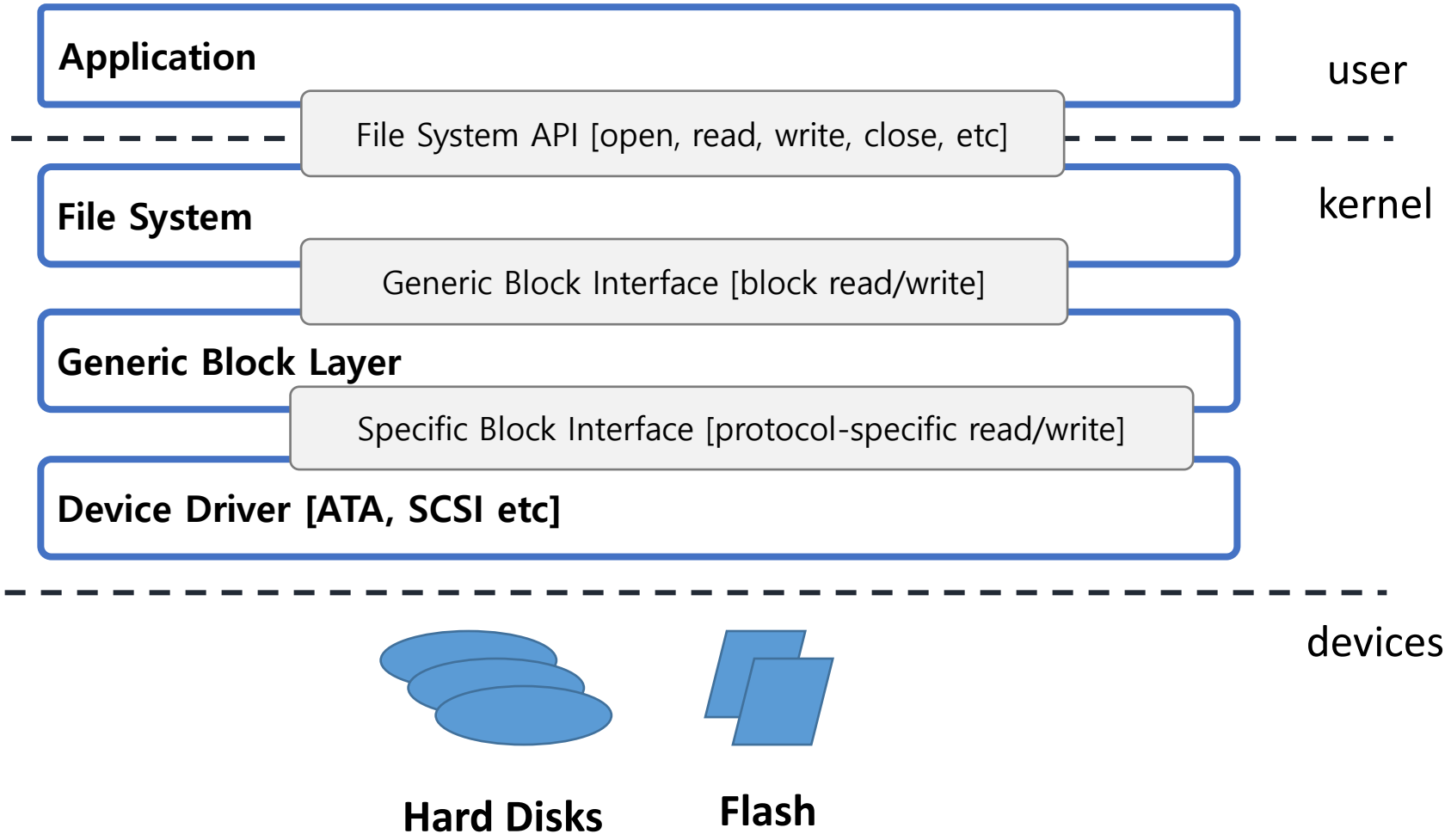
# Permanent Storage Media

- Main memory – not suitable

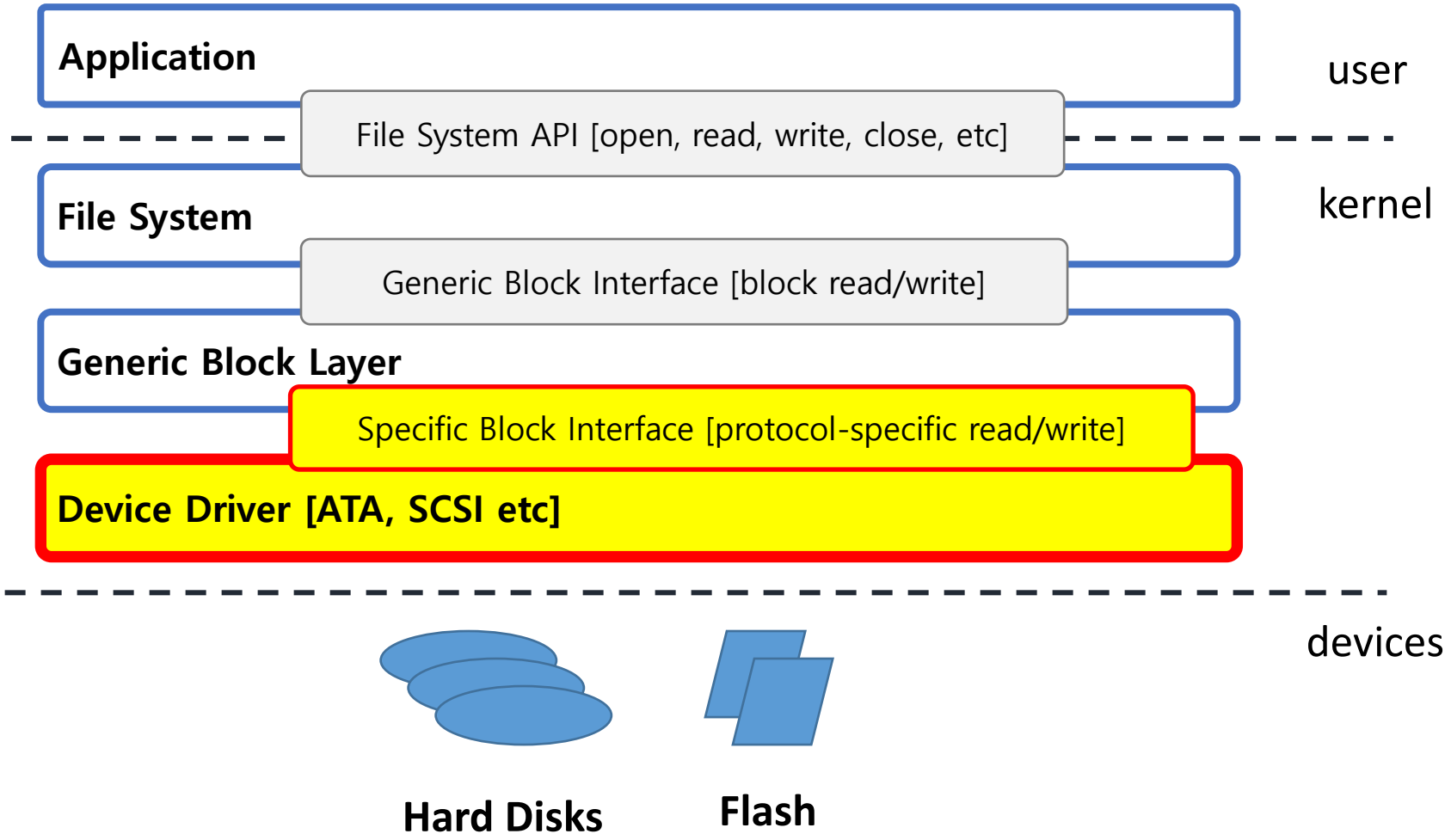

- Battery-backed memory
- Nonvolatile memory
  - Flash SSDs
  - 3DXpoint SSDs (2018-2022)
- Hard Disks (HDDs)
- Tapes
  - Surprisingly, tapes are used **a lot** even today.

**For this course**

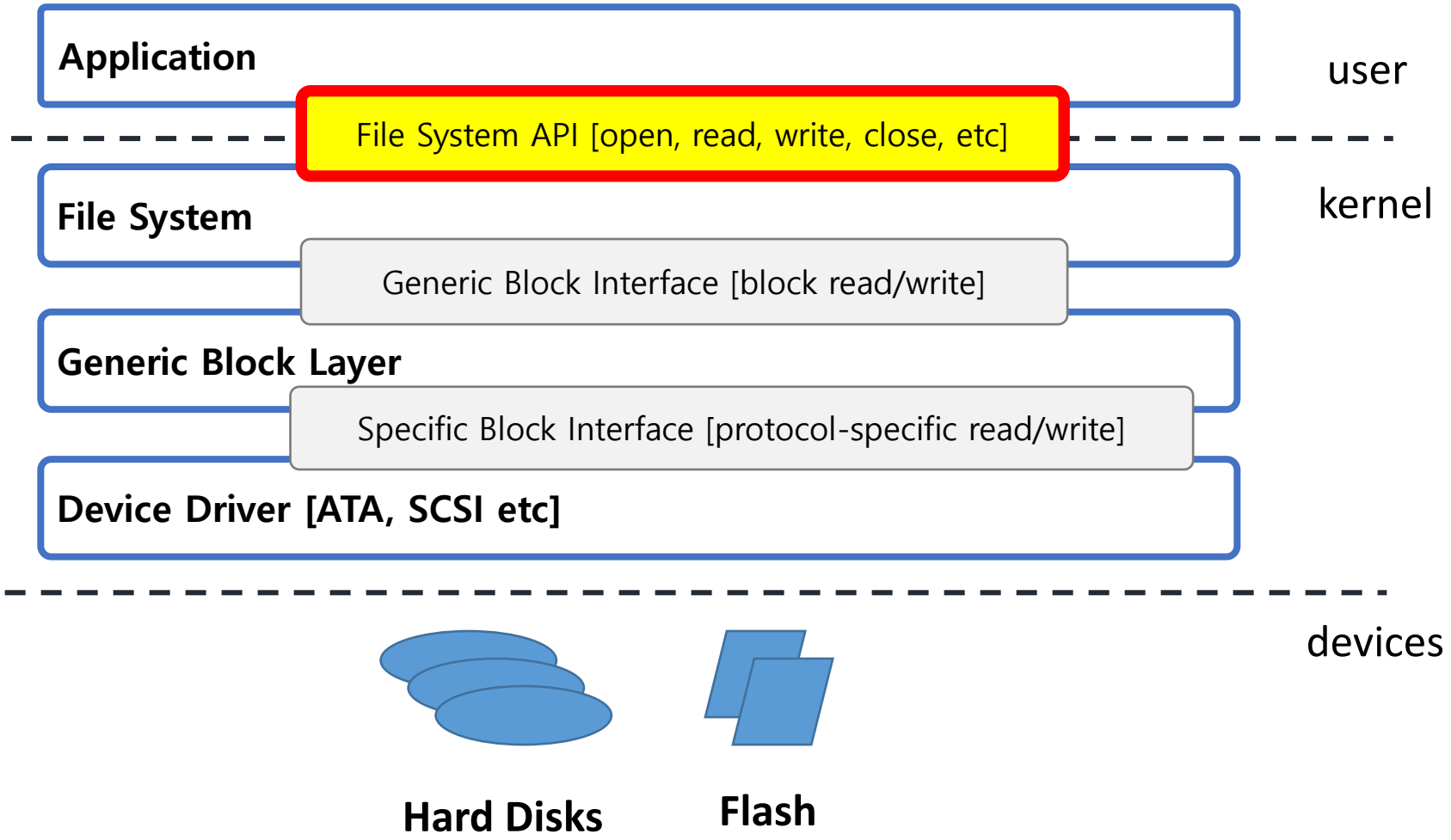
# Overall Picture



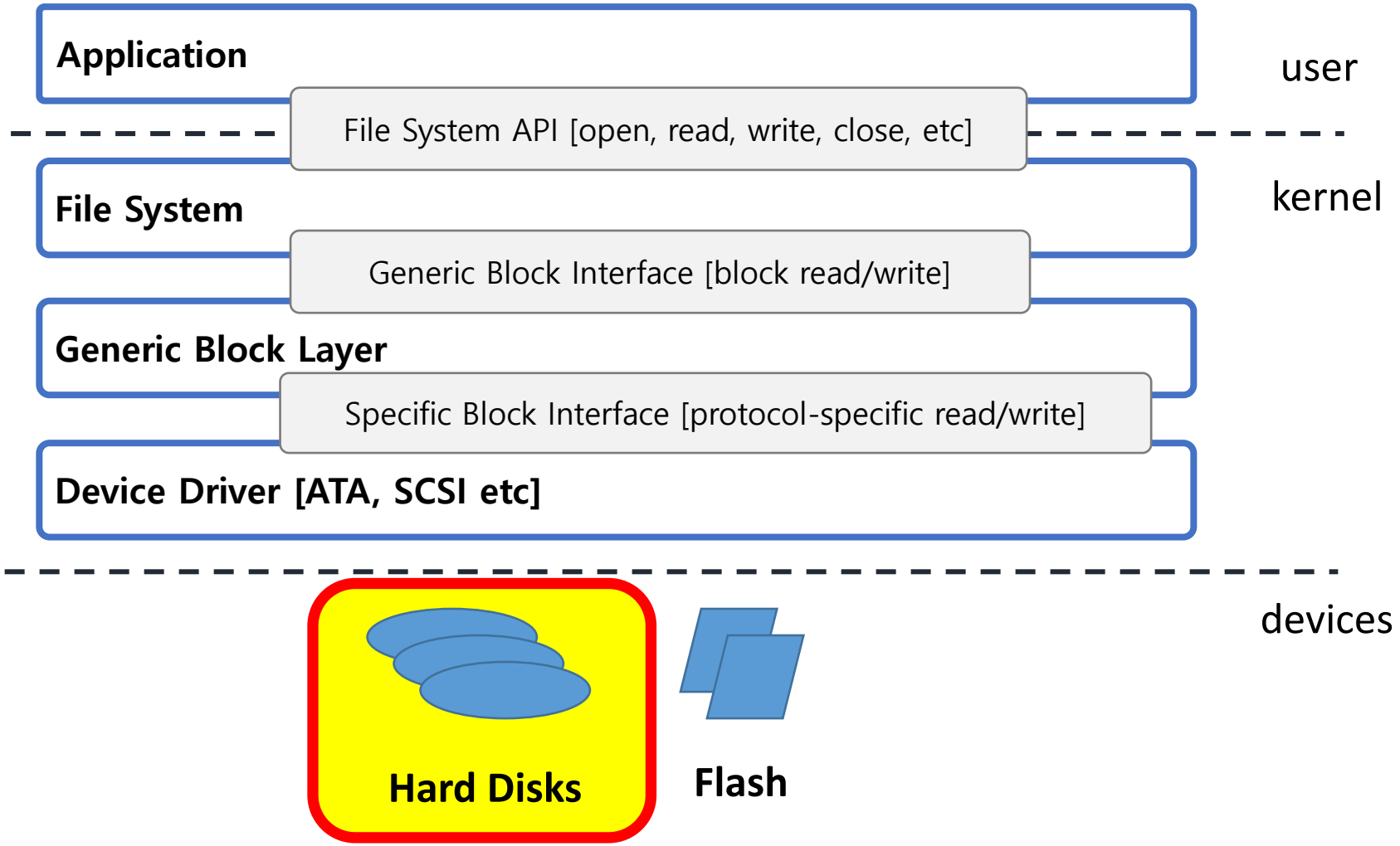
# We just talked about this part



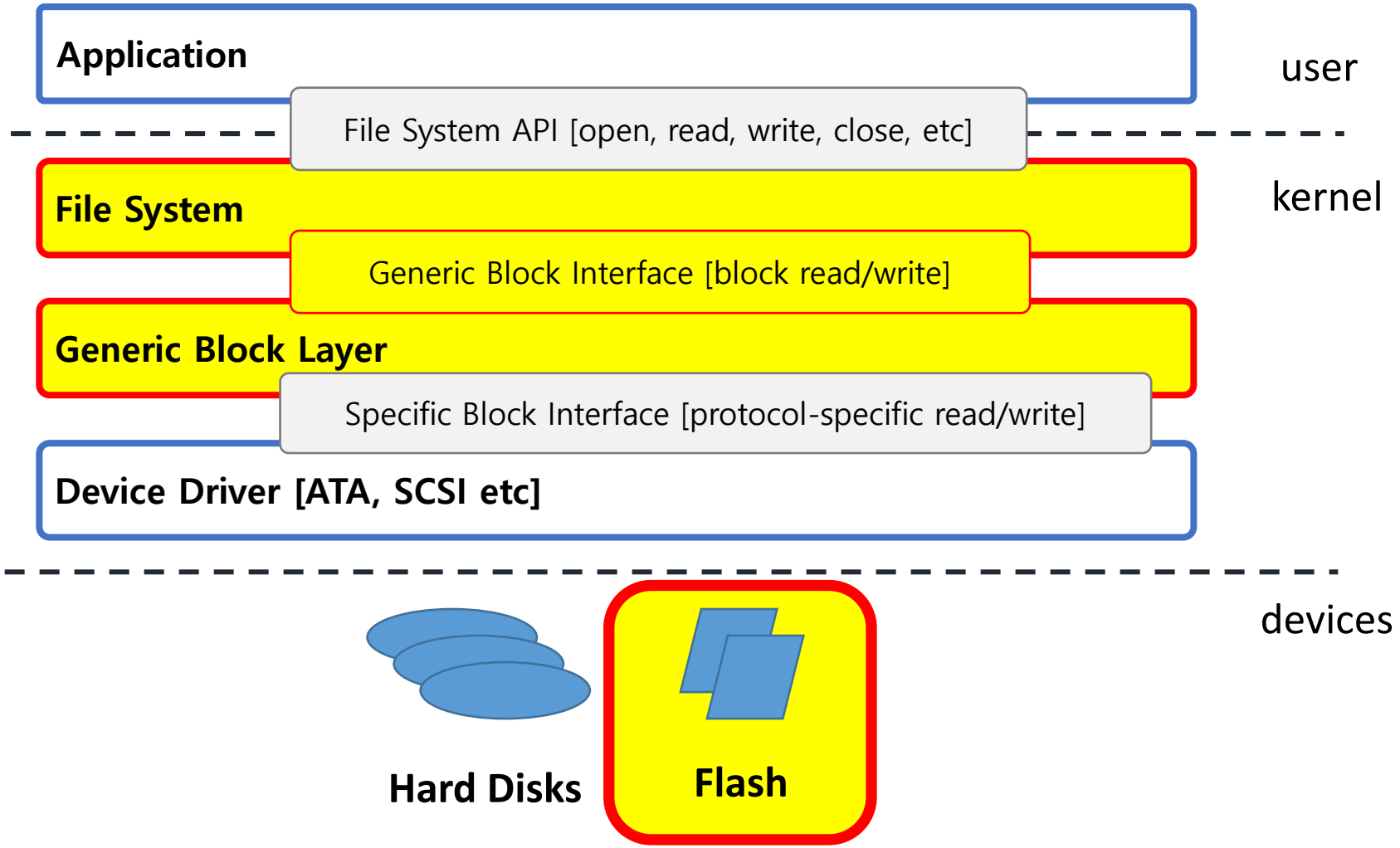
# Now we will talk about this part



# 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
  - System cannot know that a certain file is textual, binary, executable, etc

- Known to the system (Windows)
  - User knows, systems knows (and enforces)
  - Can make some filenames illegal in a way confusing to users
  - Can cause dangerous vulnerabilities (see windows .com extension)

# File System Primitives

- Access
- Concurrency
- Naming
- Protection

# File System Primitives

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

# Main Access Primitives (minimal, conceptual)

- 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`*
- Starting from byte offset *`from`* and ending at offset *`to`*
  - Also ends at EOF (End-of-file) condition
- Into a memory buffer *`buffer`*
  - previously allocated by caller
  - **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
- Sequential access is also **much** faster (will see why)

→ For this reason, want sequential access methods

# Sequential Read()

- File system keeps **file pointer *fp*** (initially 0) internally
- `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])
```

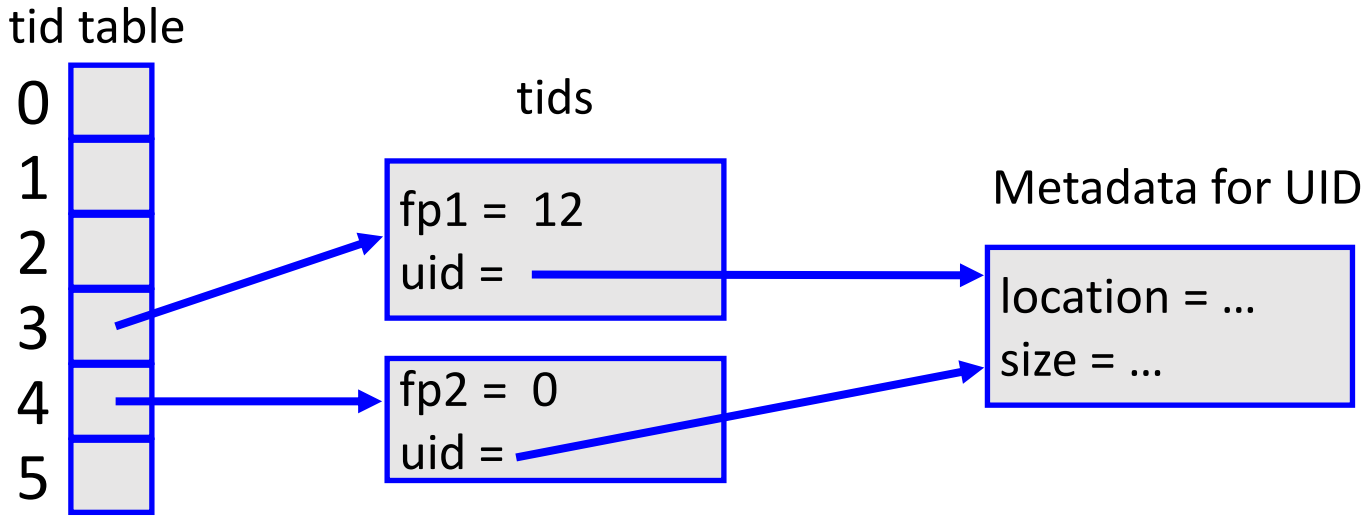
- Creates an instance of file with *uid*
- **Accessible by this process (or thread) only**
- With the temporary process-unique id *tid*
- ***fp* is associated with *tid*, not with *uid***
- tid is usually called fd or “file descriptor” but to avoid confusion with fp we will stick to tid for now.

# Close()

```
Close(tid)
```

- Destroys the instance

# Putting Open() together with Read()

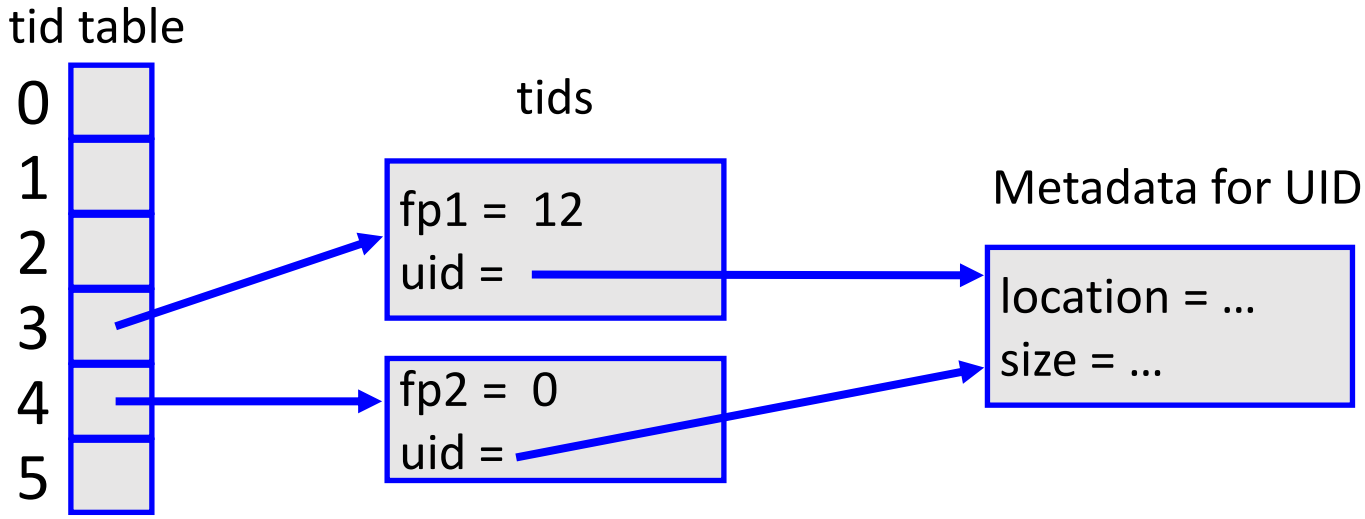



```
tid1 = Open(UID);          // returns 3
Read(tid1, buf, 12);
tid2 = Open(UID);          // returns 4
Close(tid1);
Close(tid2);
```

# 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:

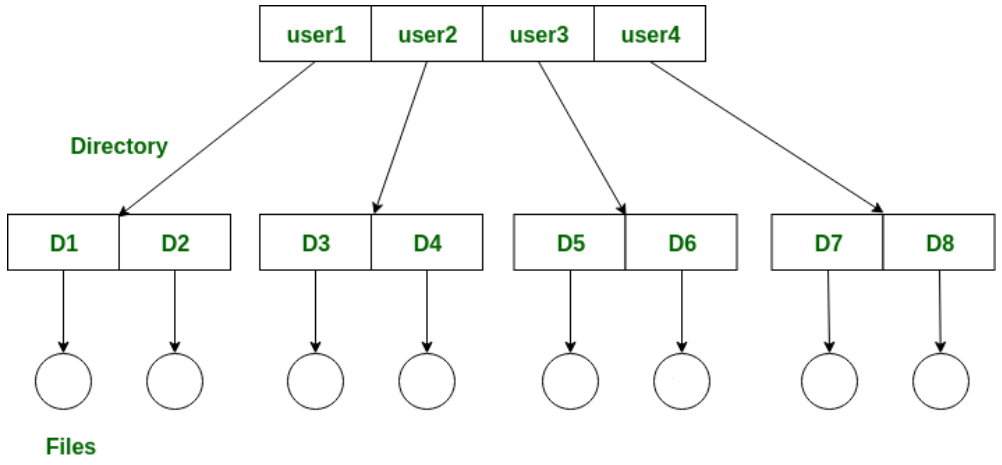
- **Flat** structure
- **Two-level**: [user] filename
- **Hierarchical**: /a/b/c …
  - Root directory
  - Working directory
  - **Modern standard**

# 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
```

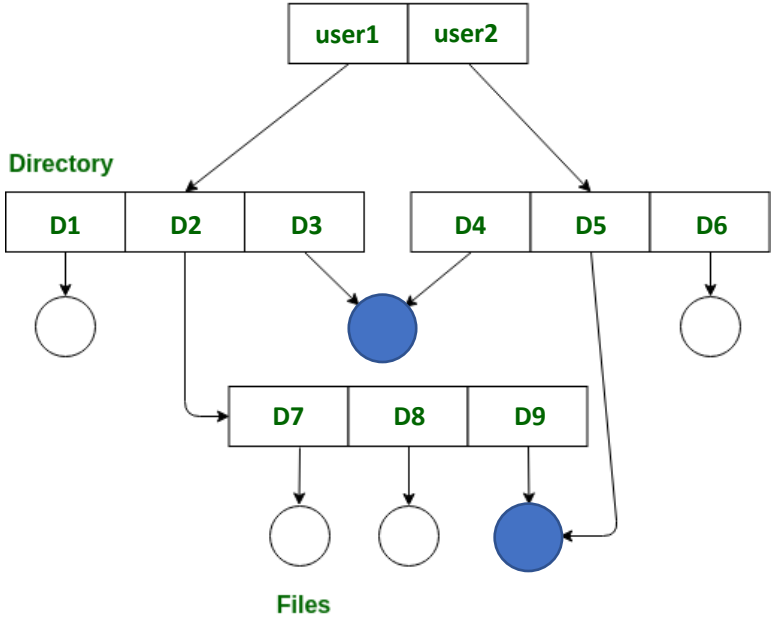
# Hierarchical Directory Structures

## Tree




## (Acyclic) Graph

Allows sharing of *uid*s 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 )**

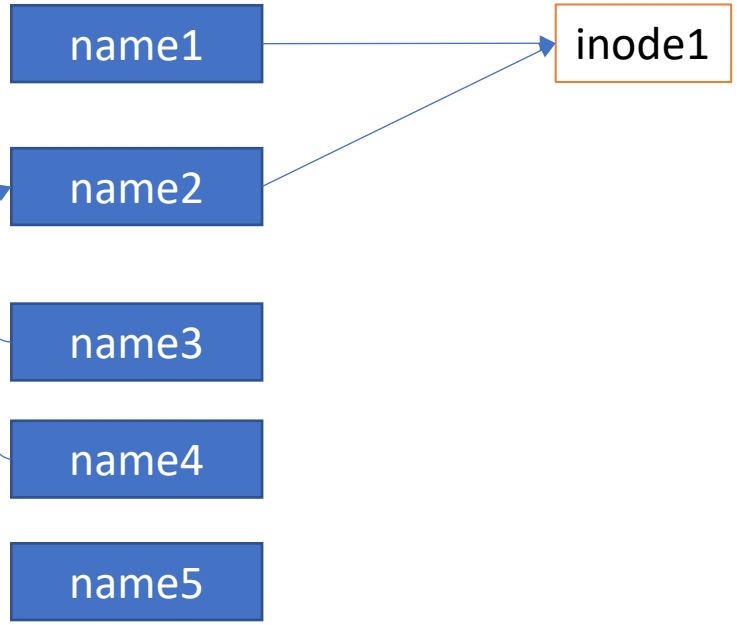
HardLink( name5, name3 )

Delete( name3 )

Open( name5 )

Delete( name2 )

Open( name4 )



# Hard Links and Soft Links

**Create( name1 )**

**HardLink( name2, name1 )**

**SoftLink( name3, name2 )**

**SoftLink( name4, name2 )**

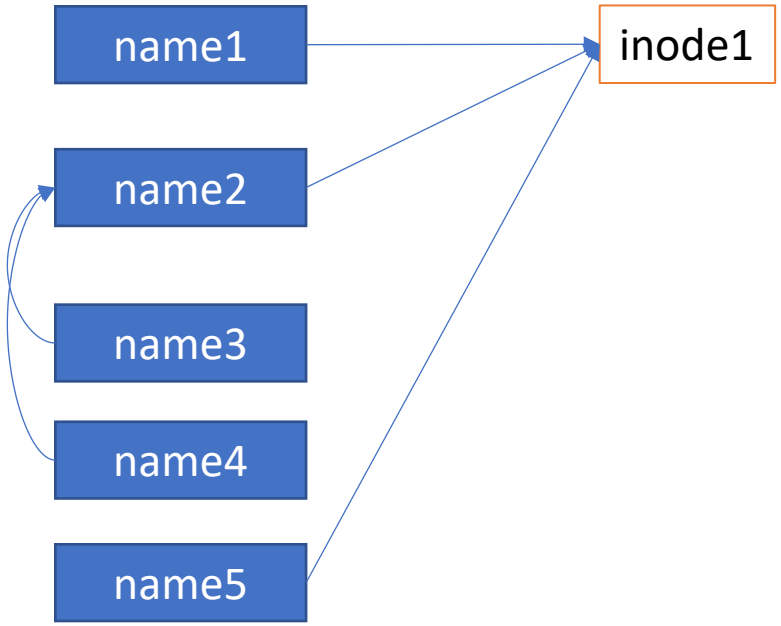
**HardLink( name5, name3 )**

Delete( name3 )

Open( name5 )

Delete( name2 )

Open( name4 )



# Hard Links and Soft Links

Create( name1 )

HardLink( name2, name1 )

SoftLink( name3, name2 )

SoftLink( name4, name2 )

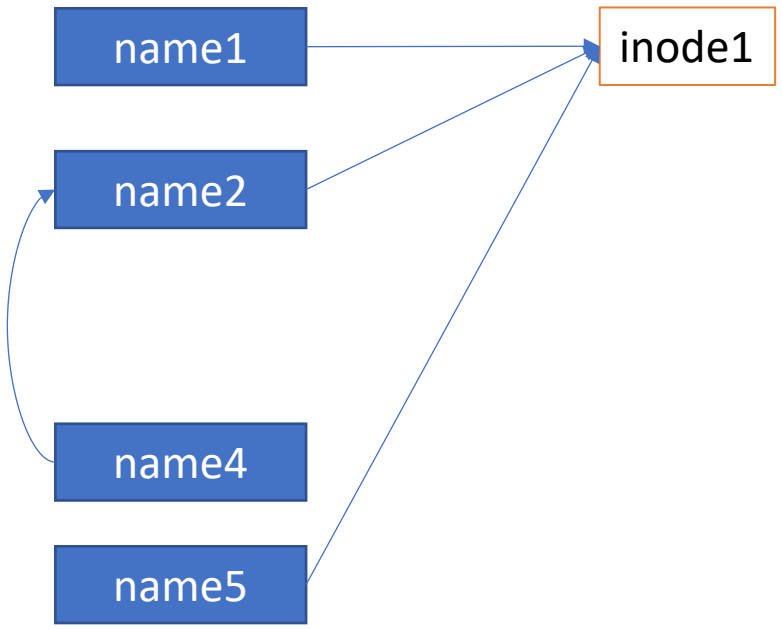
HardLink( name5, name3 )

**Delete( name3 )**

Open( name5 )

Delete( name2 )

Open( name4 )



# Hard Links and Soft Links

Create( name1 )

HardLink( name2, name1 )

SoftLink( name3, name2 )

SoftLink( name4, name2 )

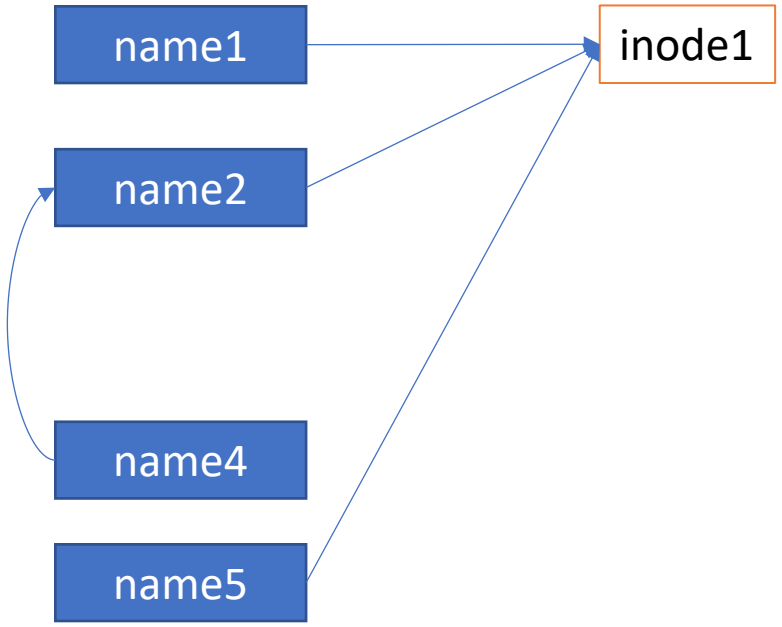
HardLink( name5, name3 )

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 )

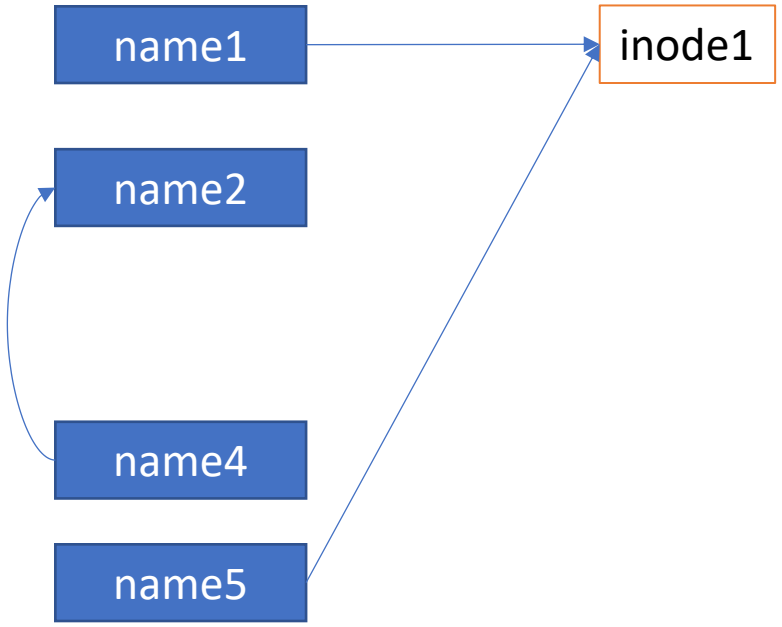
HardLink( name5, name3 )

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 )

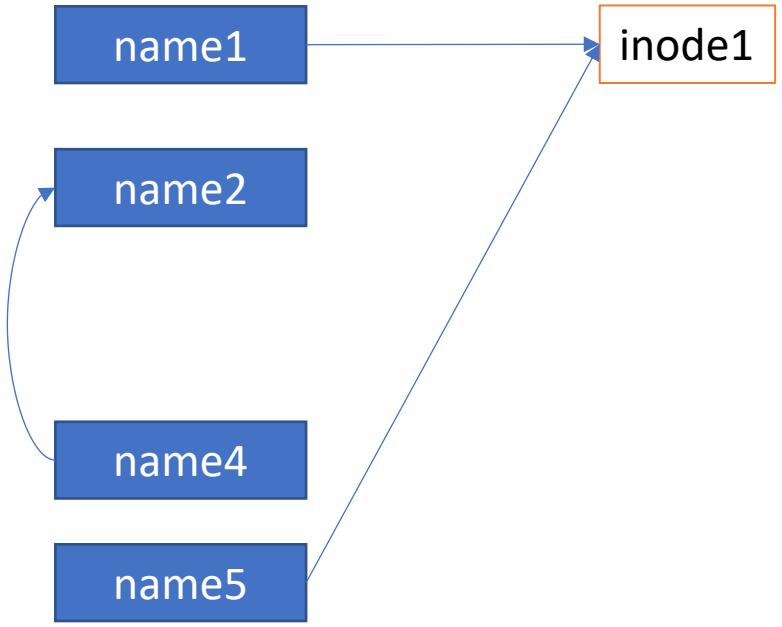
HardLink( name5, name3 )

Delete( name3 )

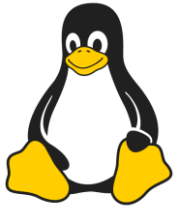
Open( name5 ) ← opens file

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

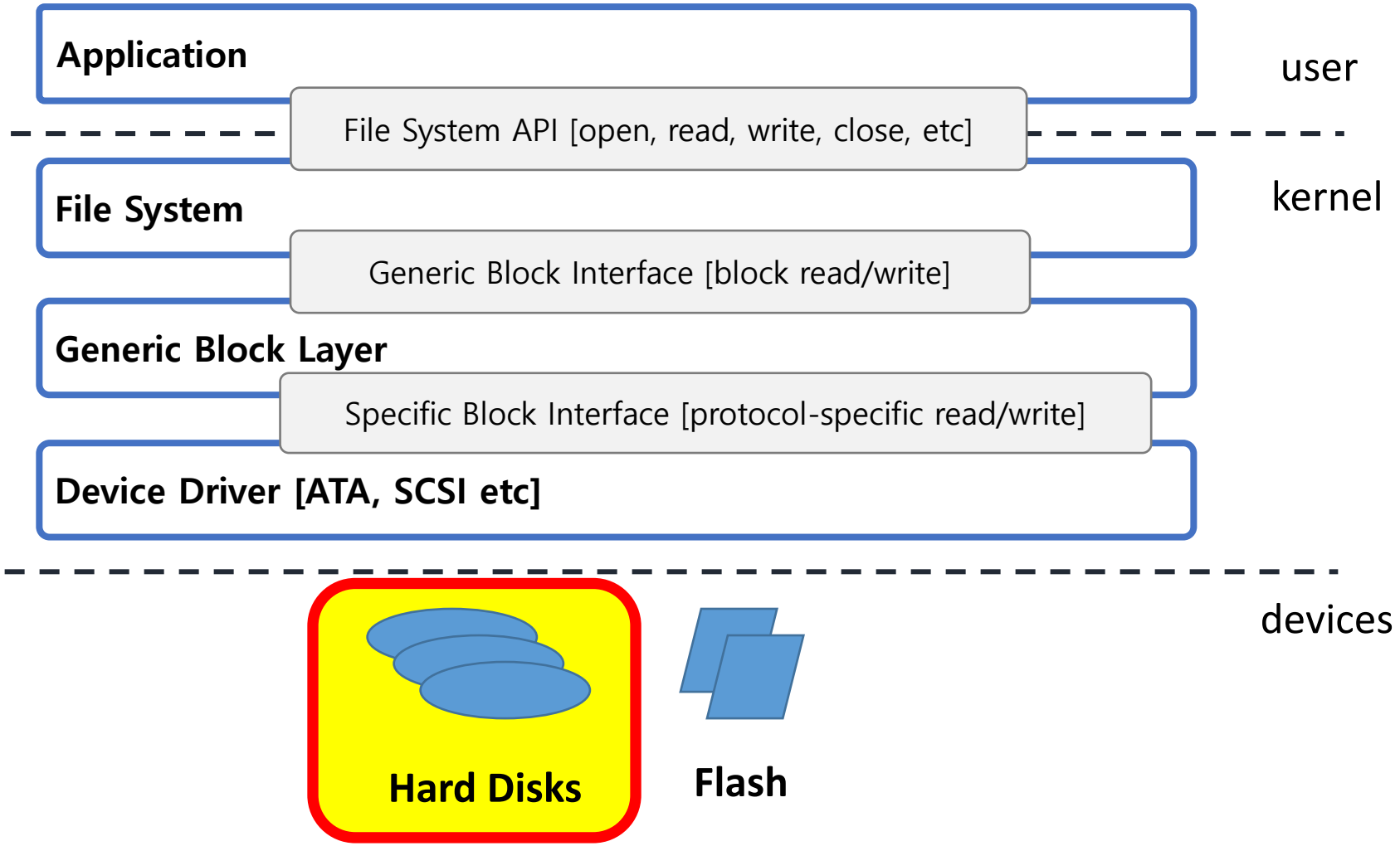
- Access
- Concurrency
- Naming
- **Protection** ← Later in the course

# Summary – File System Interface

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

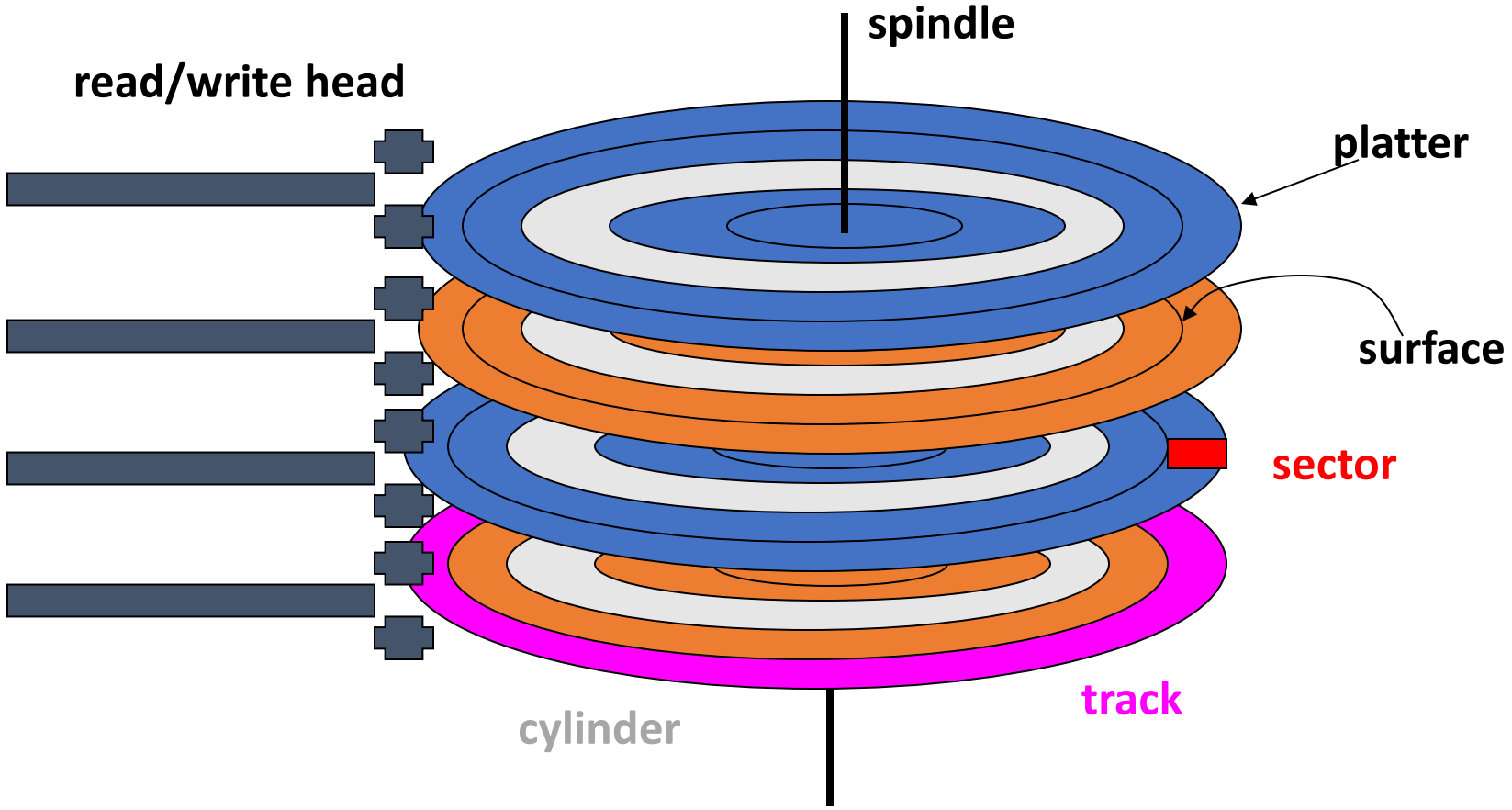
# Persistent Storage: Disks

# Disks

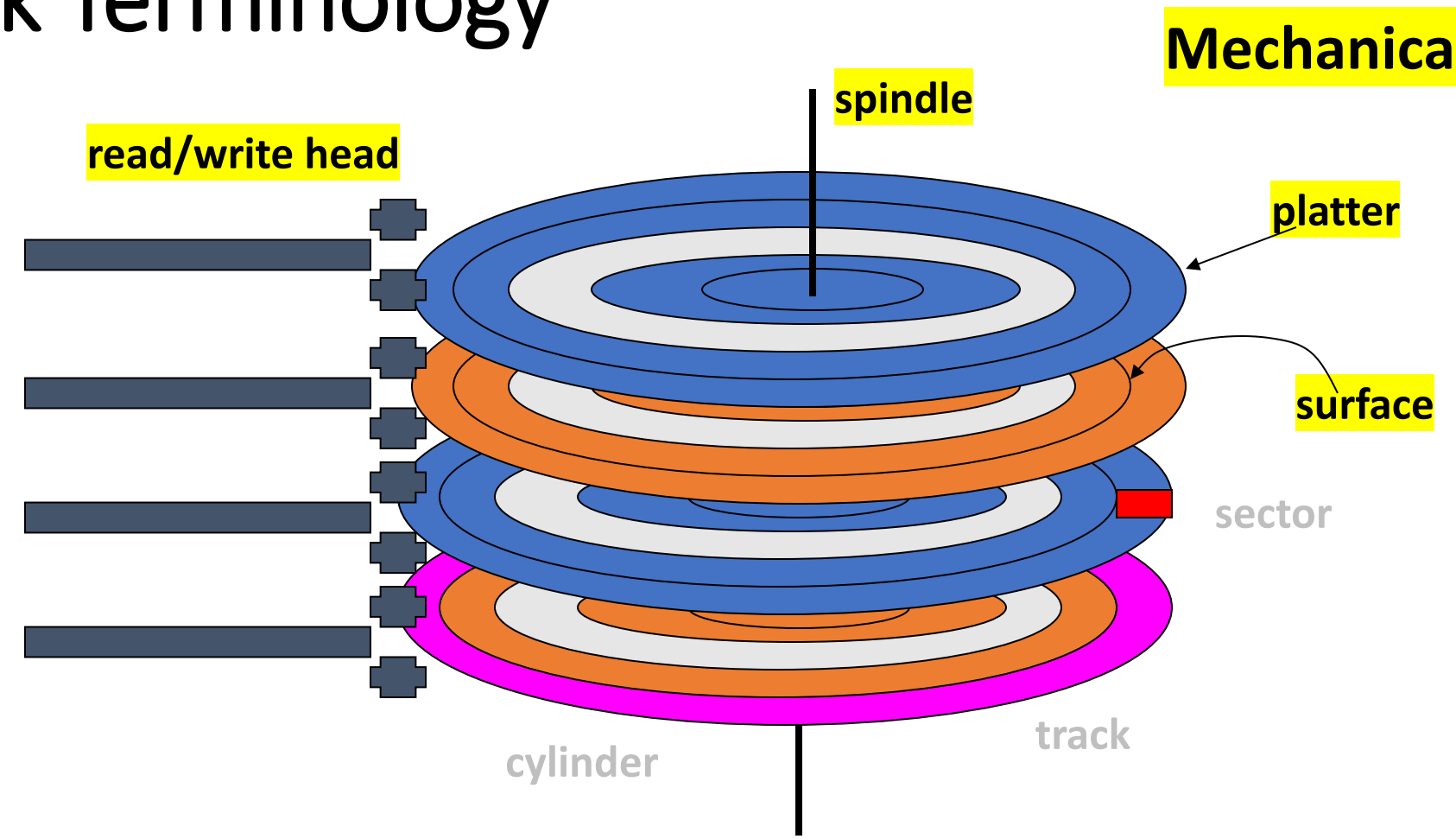

Application
File System API [open, read, write, close, etc]
File System
Generic Block Interface [block read/write]
Generic Block Layer
Specific Block Interface [protocol-specific read/write]
Device Driver [ATA, SCSI etc]
user
kernel
devices
Hard Disks
Flash

# Disk

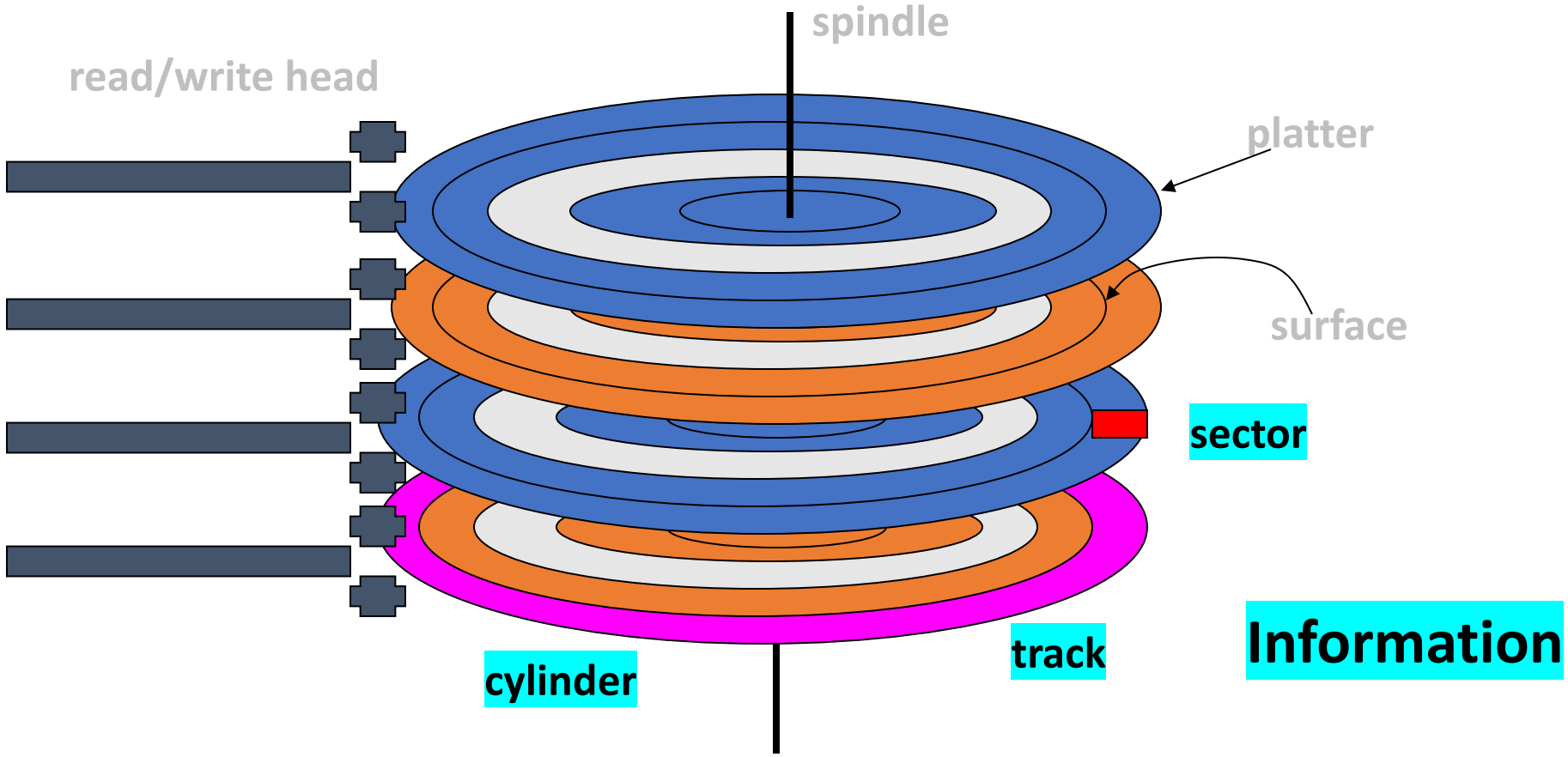
# Disk Terminology



# Disk Terminology

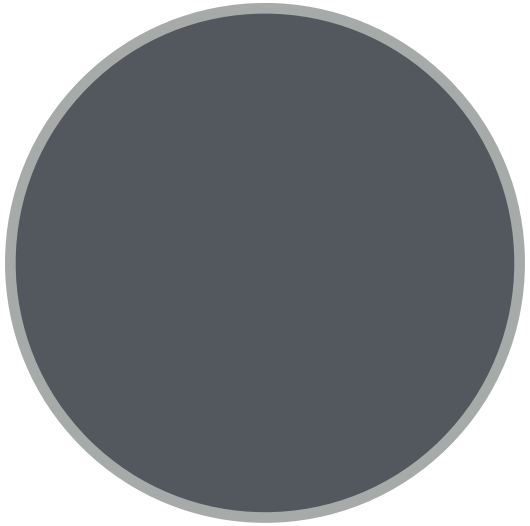


# Disk Terminology



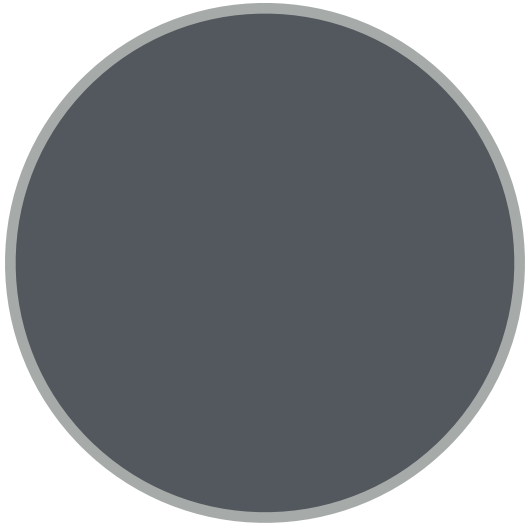
# Disk Internals

**Platter**



# Disk Internals

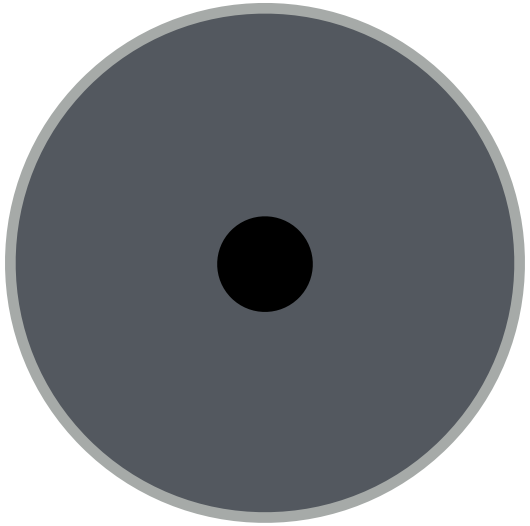
**Platter**



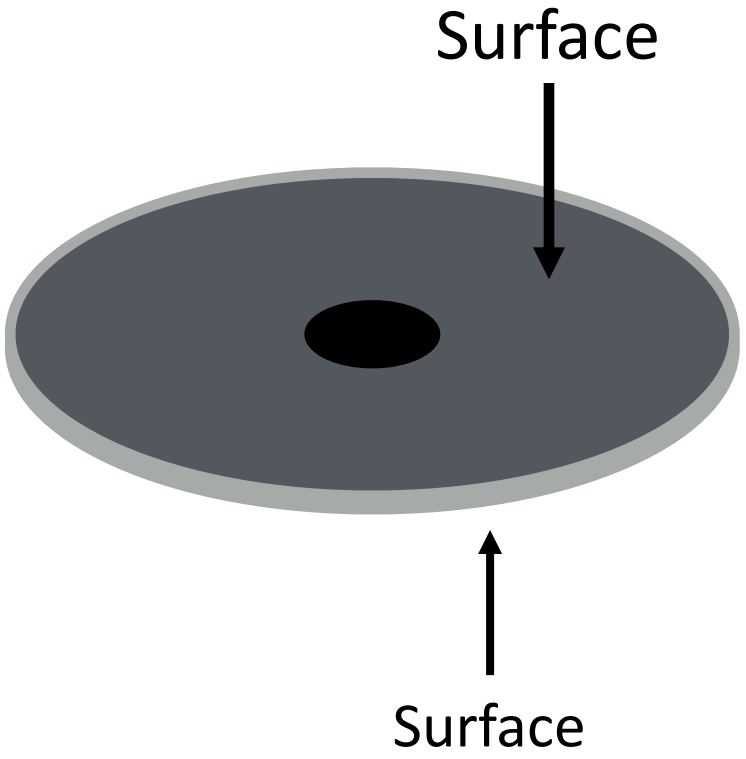
Platter is covered with a magnetic film.

# Disk Internals

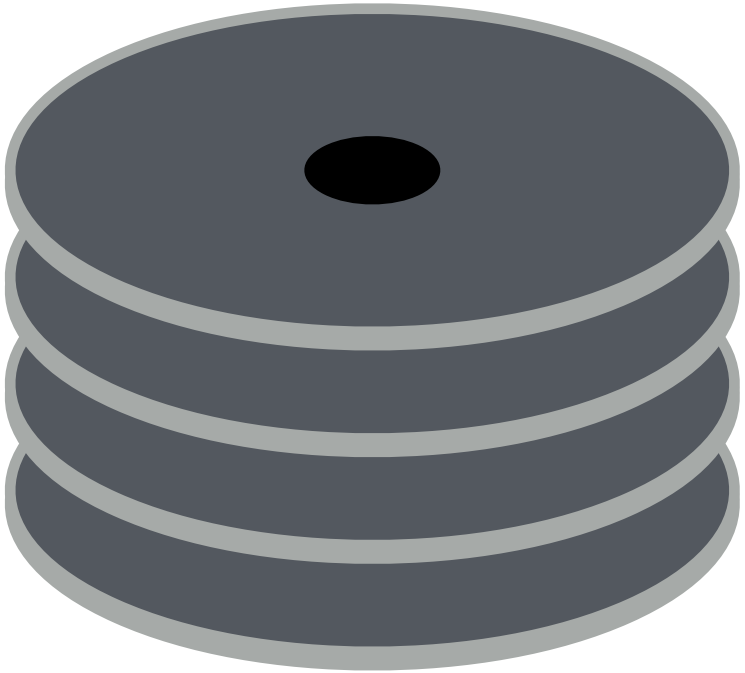
**Spindle**



# Disk Internals

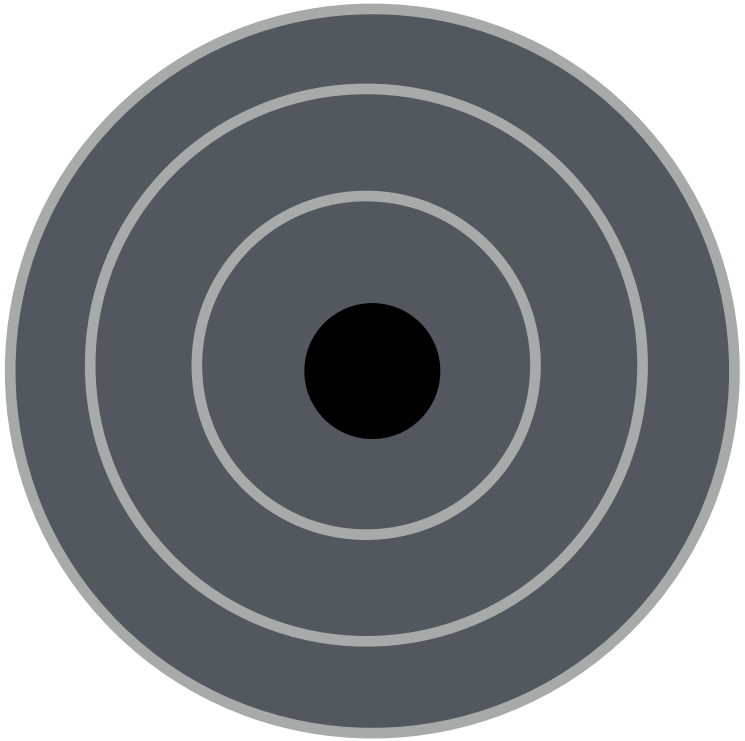


# Disk Internals



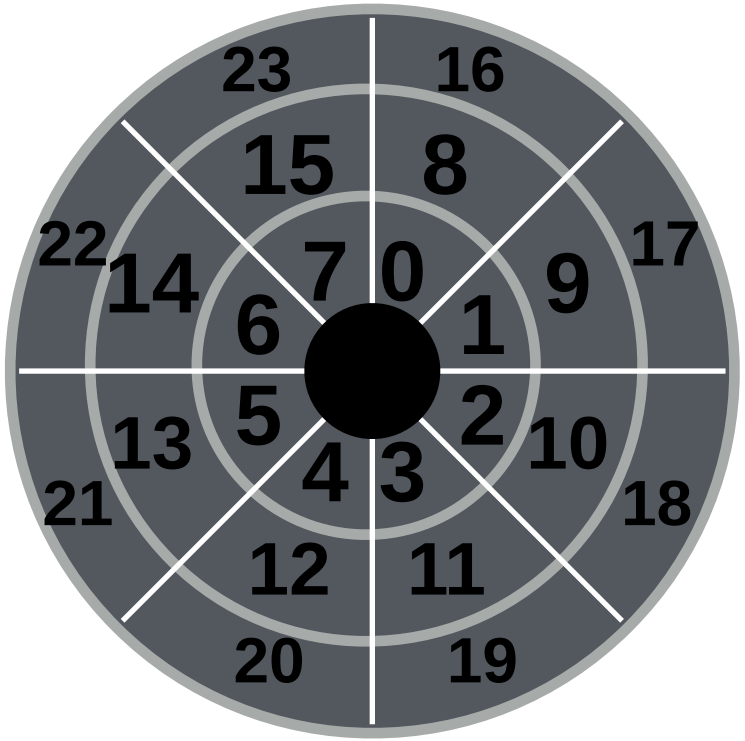
Many platters may be bound to the spindle.

# Disk Internals



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

# Disk Internals

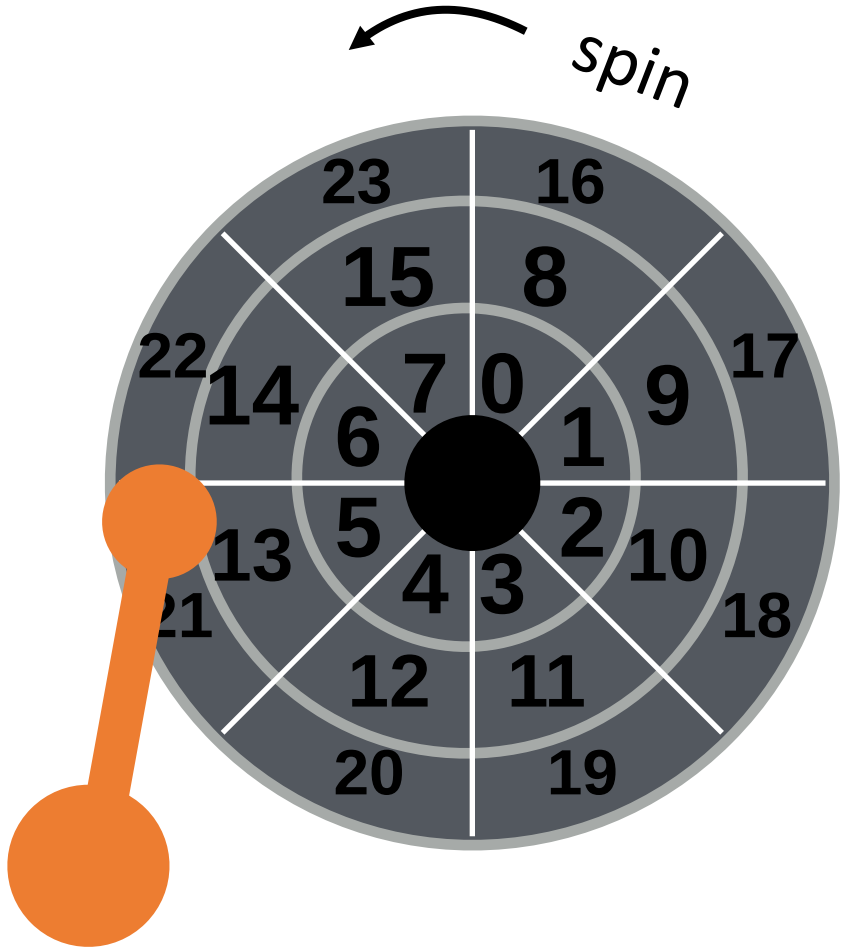



The tracks are logically divided into numbered **sectors**.

# 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*

*Often: Block size = 4,096 bytes, Sector size = 512 bytes*

→ For simplicity of presentation in class

**block_size = sector_size**

# Back to Disk Interface

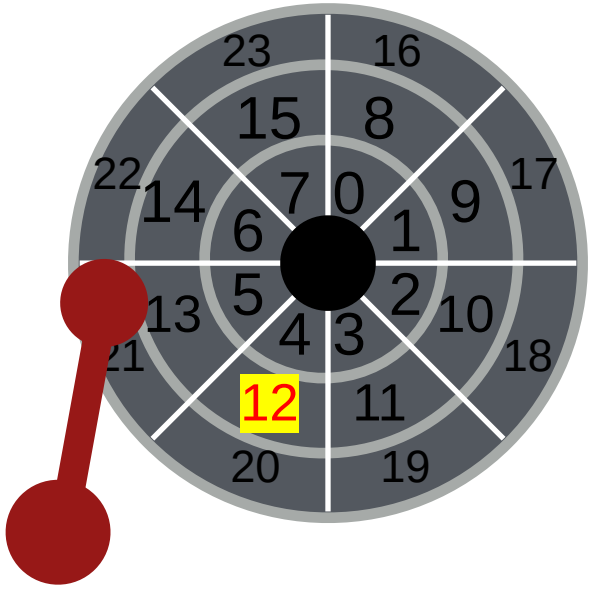
- **Accessible by sector only**
  - ReadSector ( logical_sector_number, buffer )
  - WriteSector( logical_sector_number, buffer )

- Logical_sector_number =
  - Platter
  - Cylinder or track
  - Sector

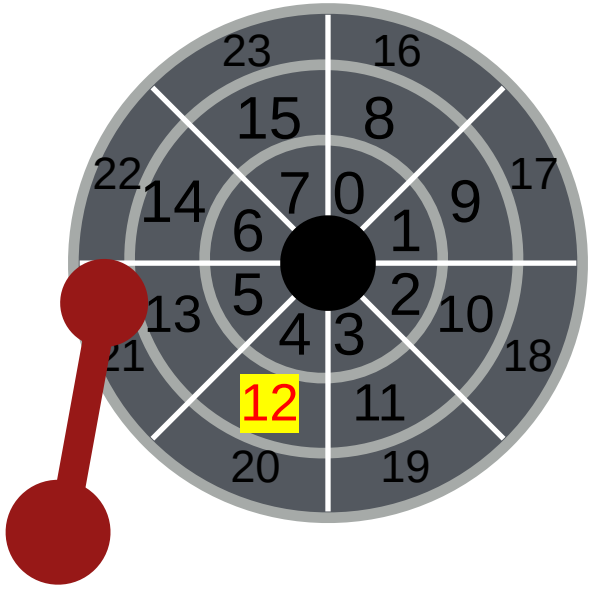
# Disk Access

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

# Let’s Read 12!



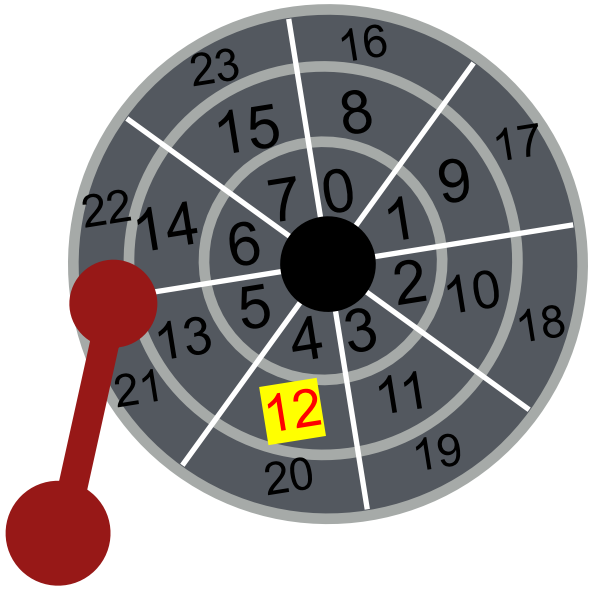
# Select platter.




**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**

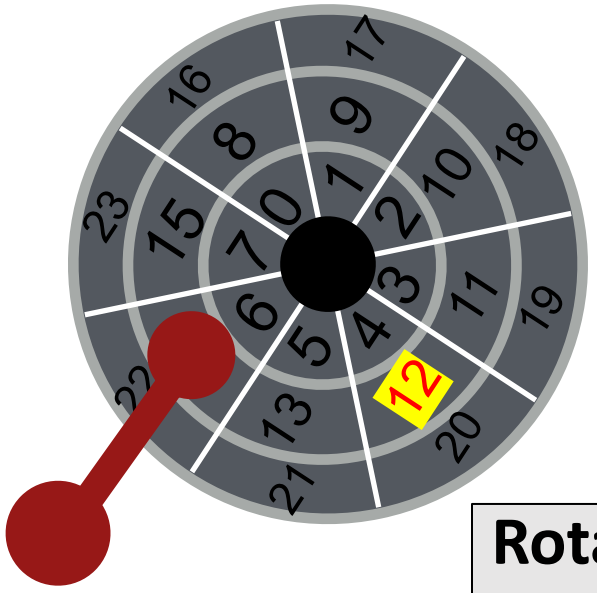
- Approx. linear in the number of cylinders
- 3 to 12 milliseconds

# Seek to right track.


**Seek Time**

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

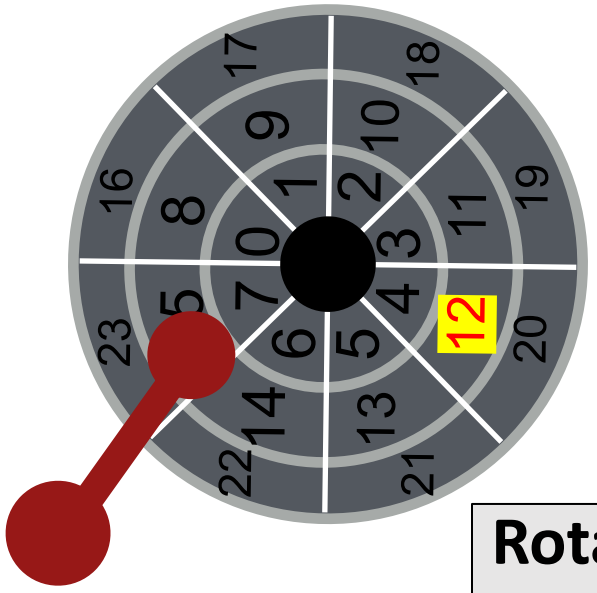
# Wait for rotation.




**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

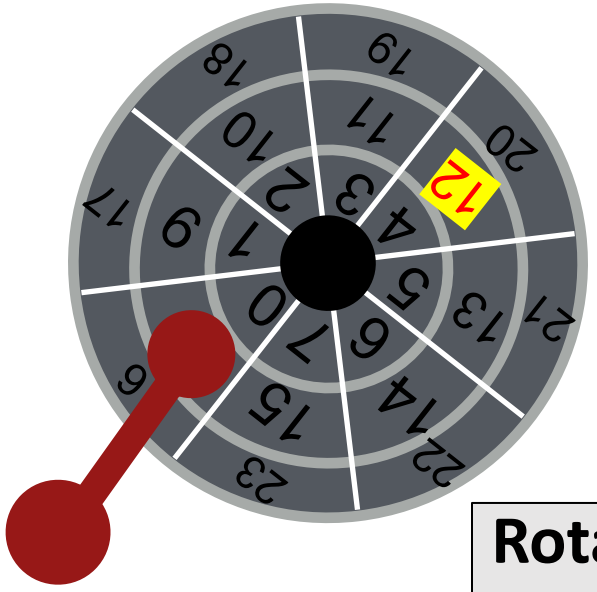
# Wait for rotation.




**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

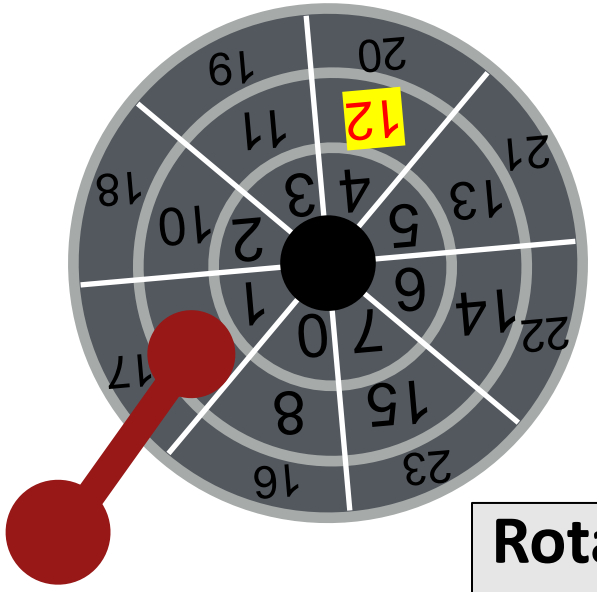
# Wait for rotation.




**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

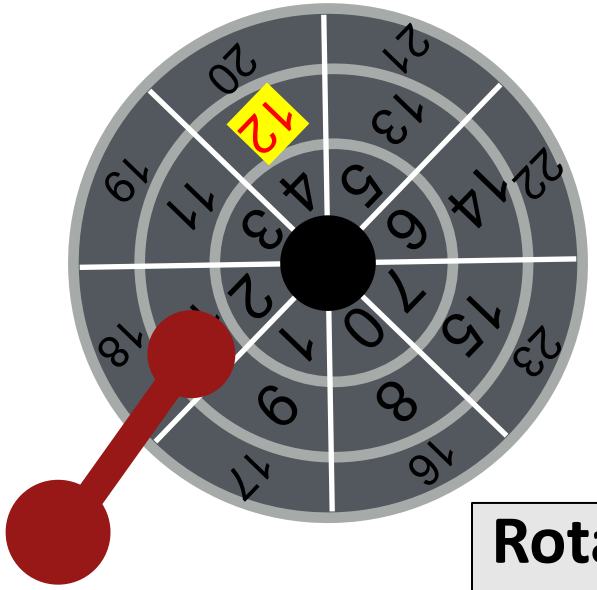
# Wait for rotation.




**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

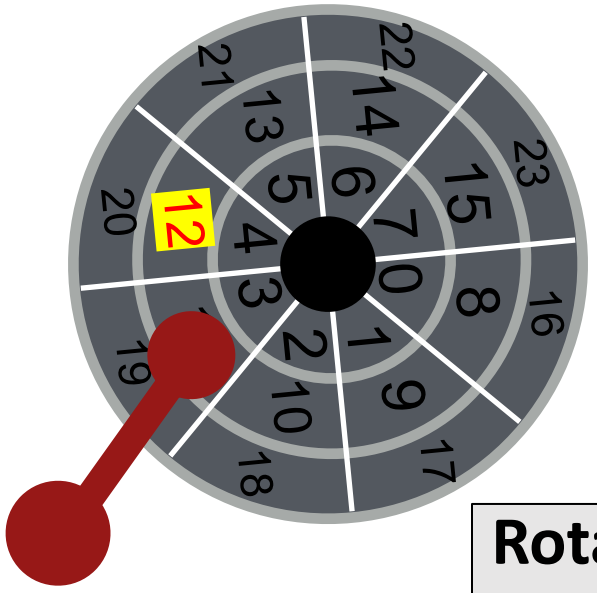
# Wait for rotation.




**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

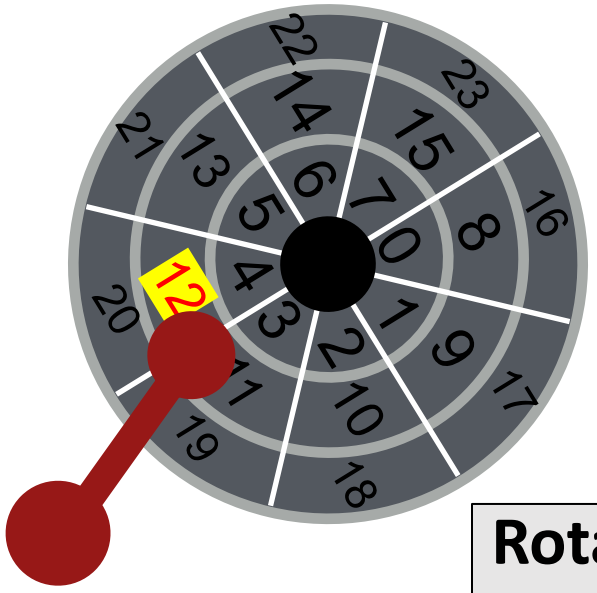
# Wait for rotation.




**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

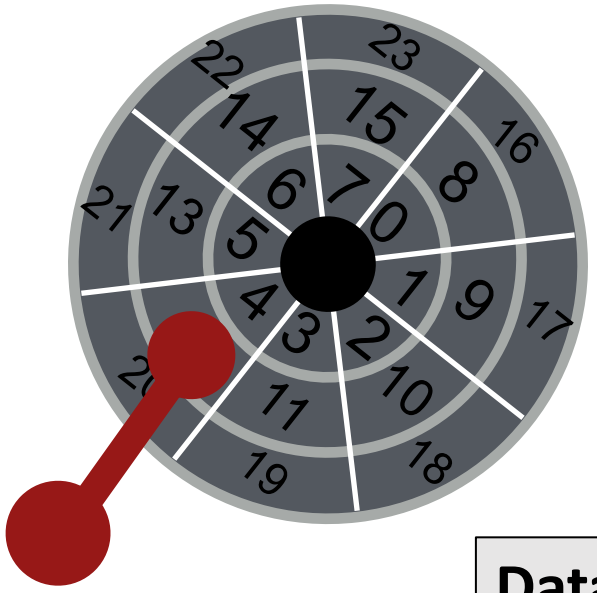
# Transfer data.




**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

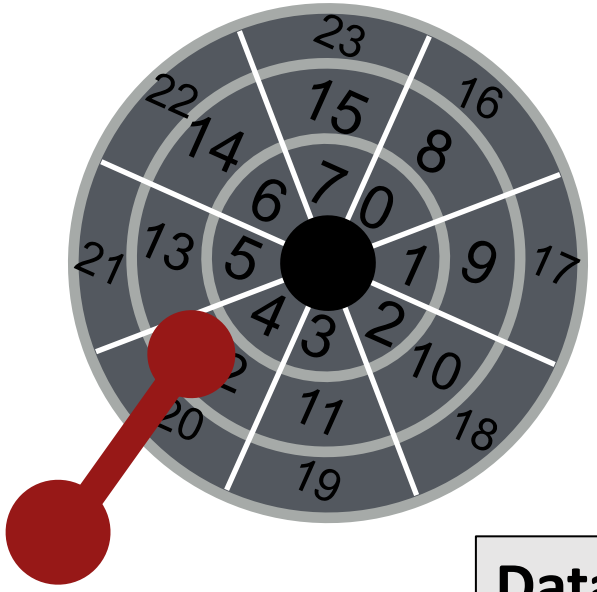
# Transfer data.




**Data Transfer**

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

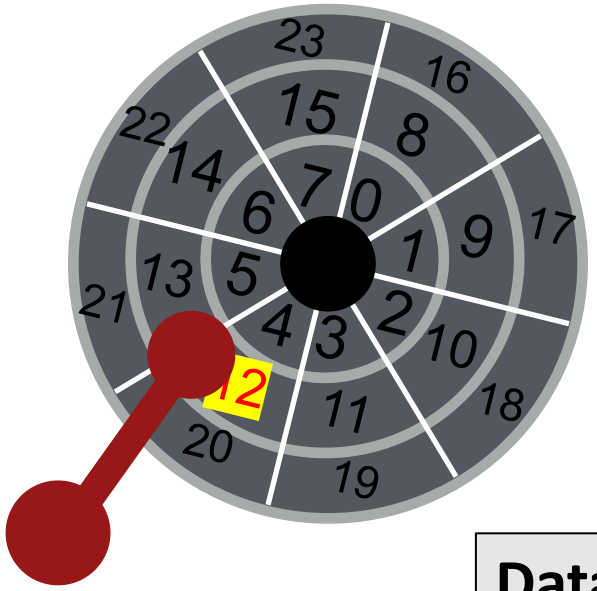
# Transfer data.




**Data Transfer**

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

# Done!




**Data Transfer**

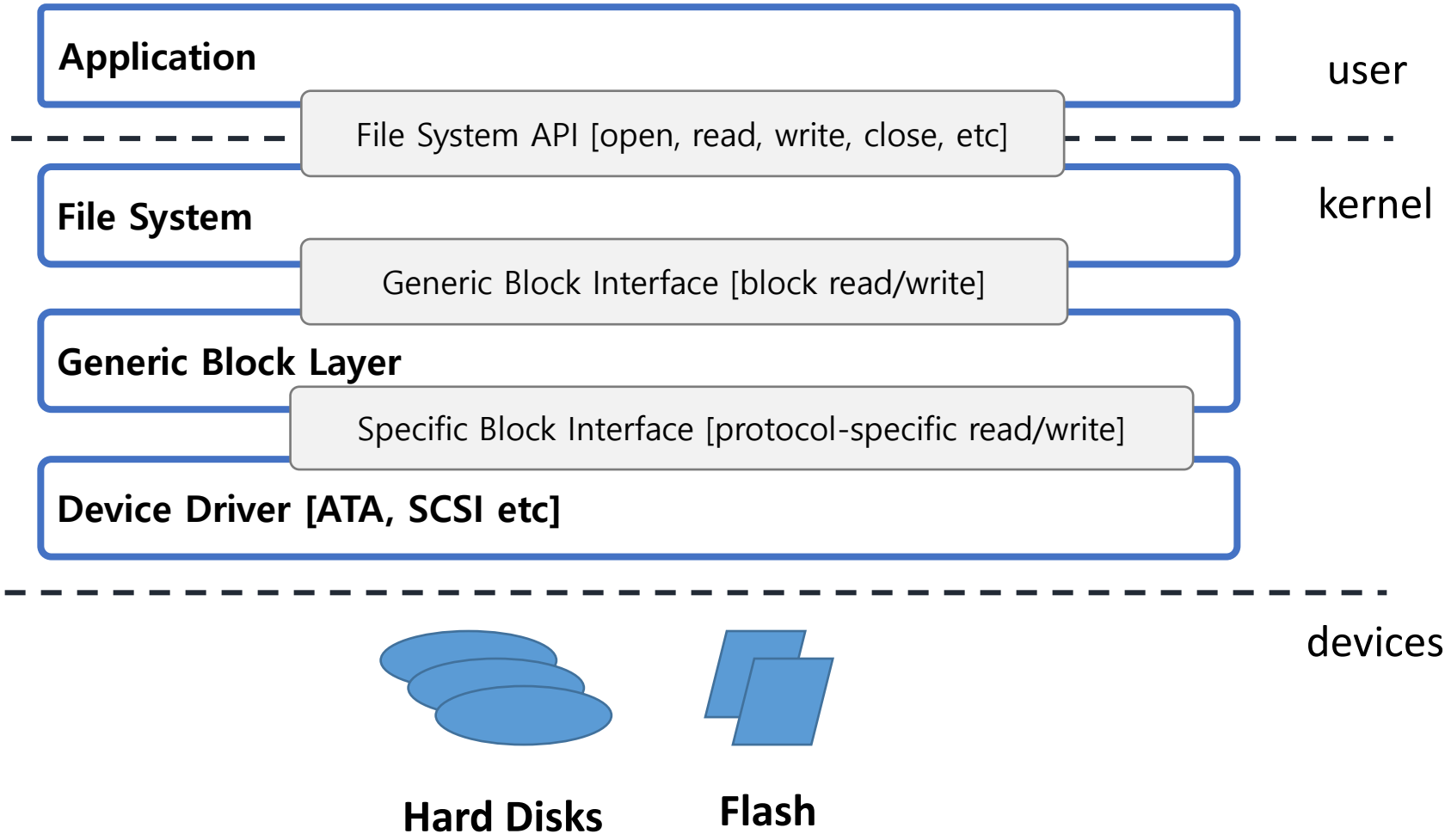
- Effective transfer rate ~ 1 Gbyte per second
- Sector = 512 bytes
- Transfer time ~ 0.5 microseconds

# Week 10

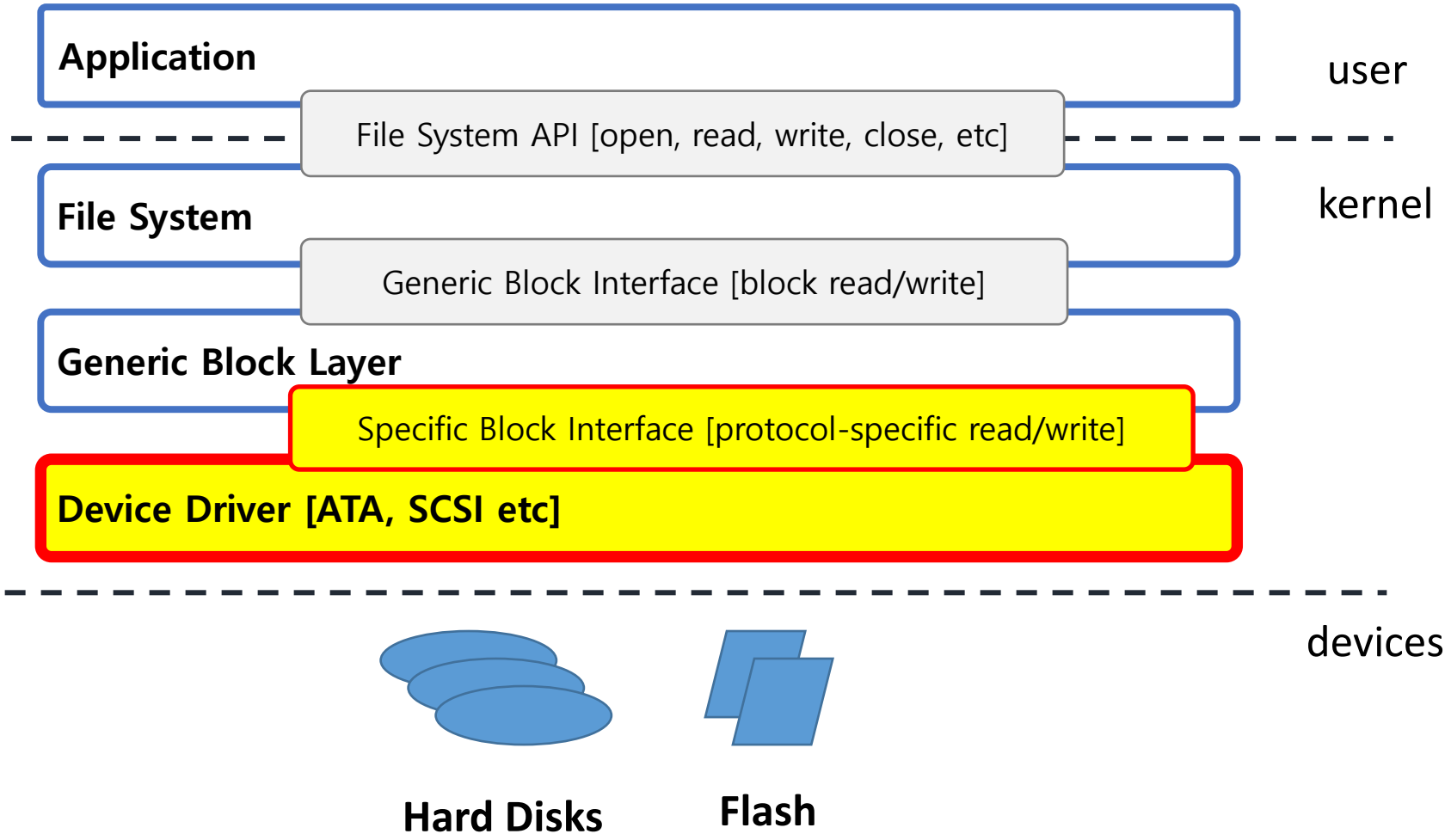
# Persistent Storage: Intro to File Systems

Max Kopinsky
13 March, 2025

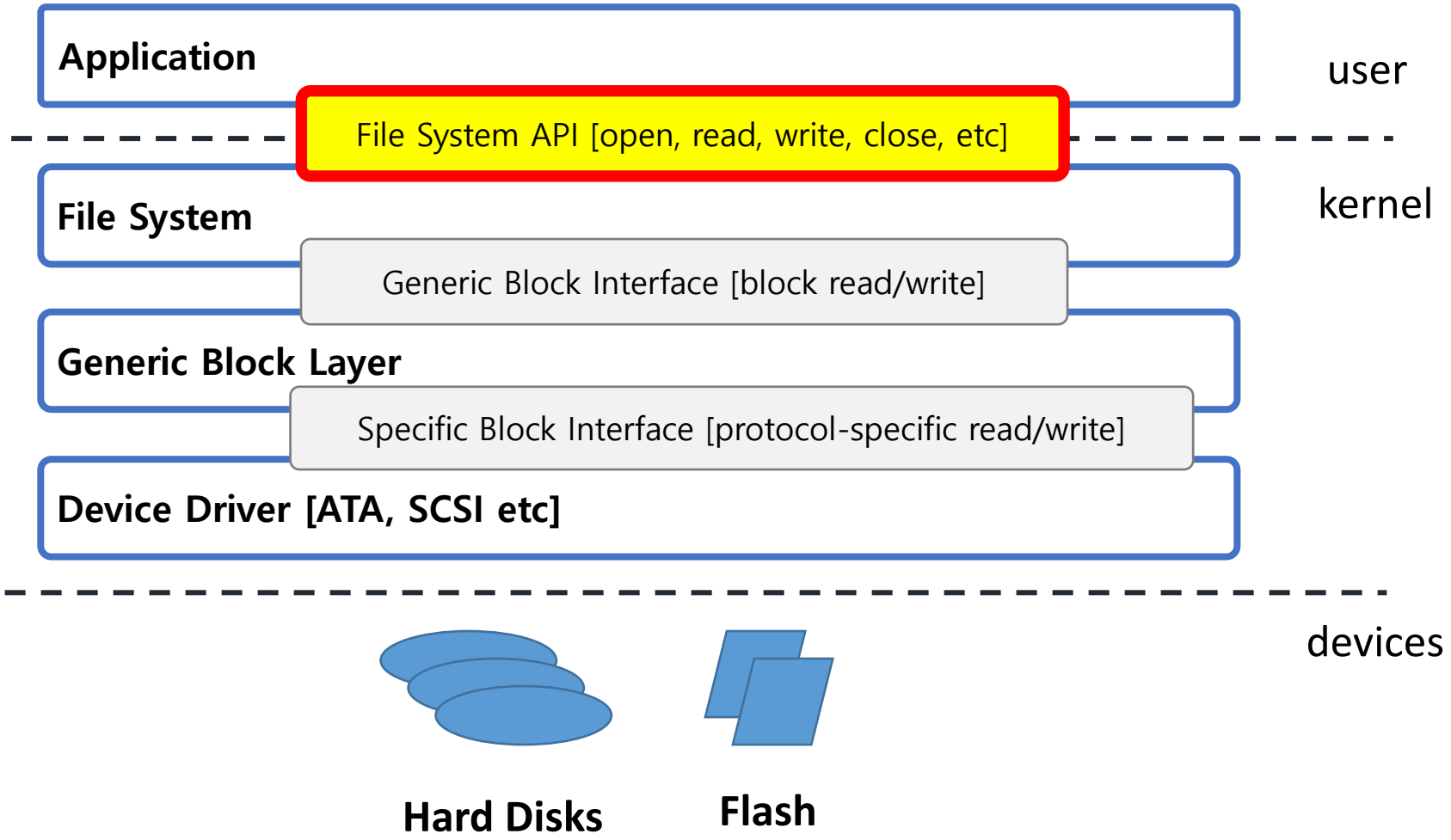
# Recap: Overall Picture



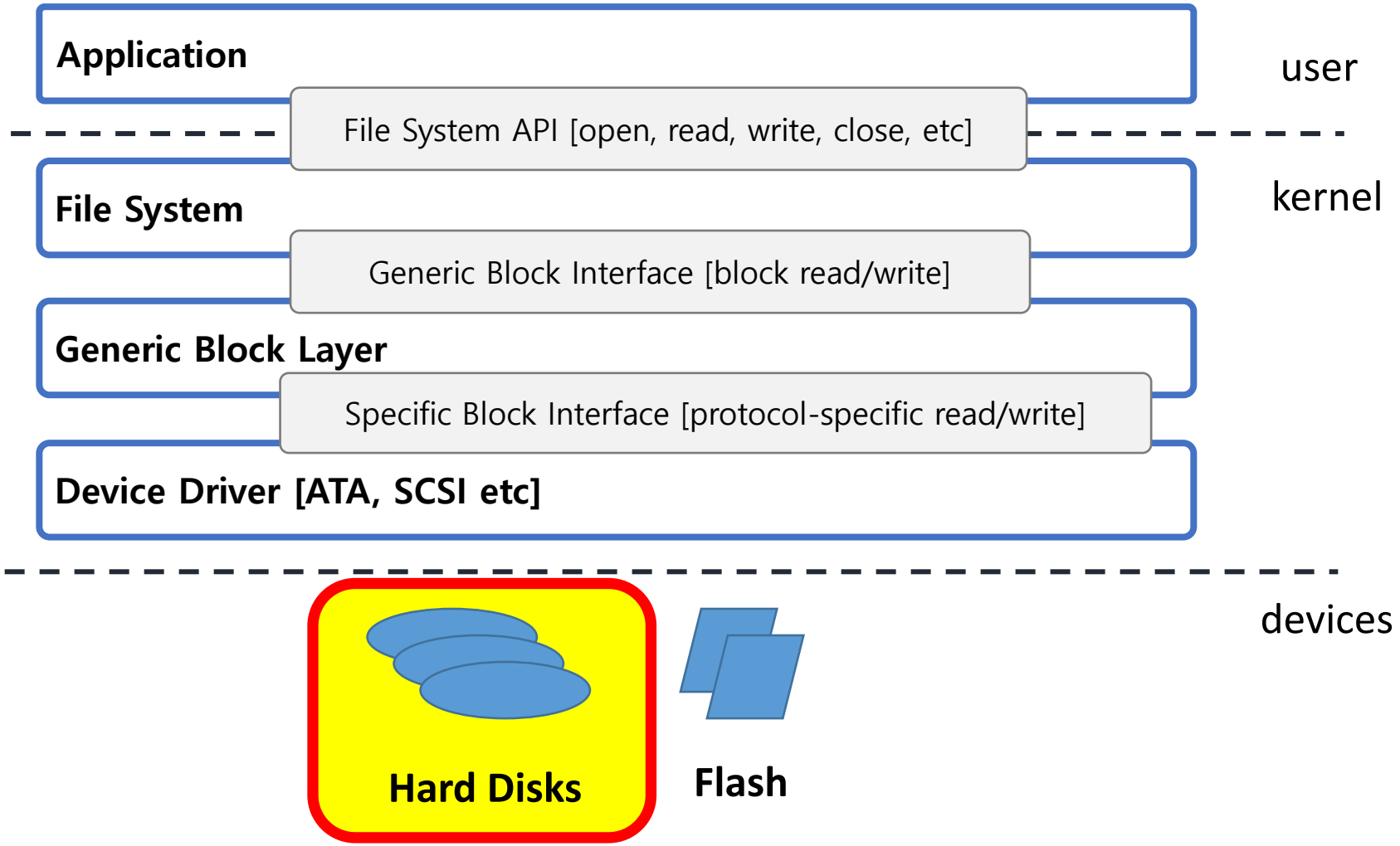
# We talked about this part



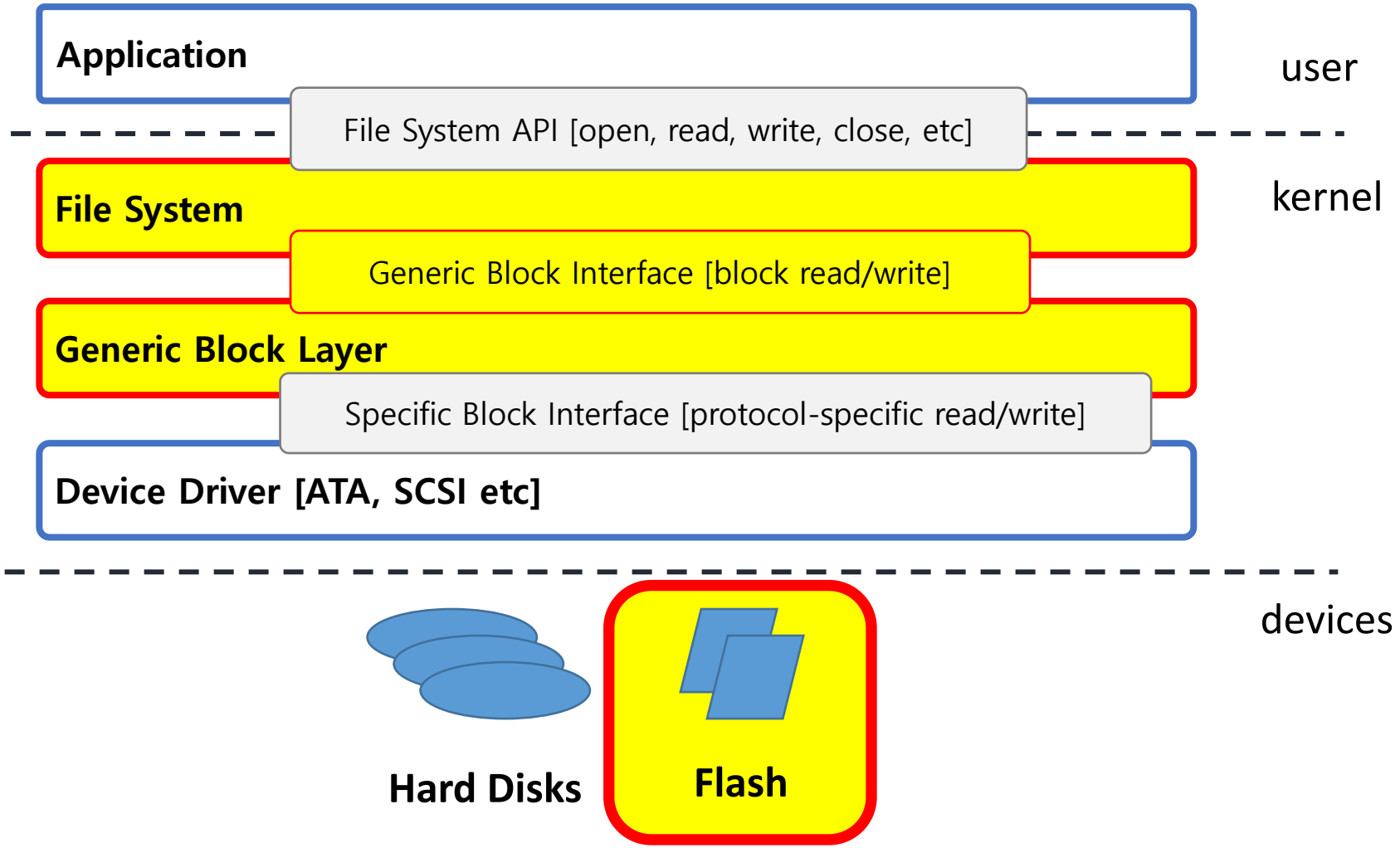
# Then we talked about this part



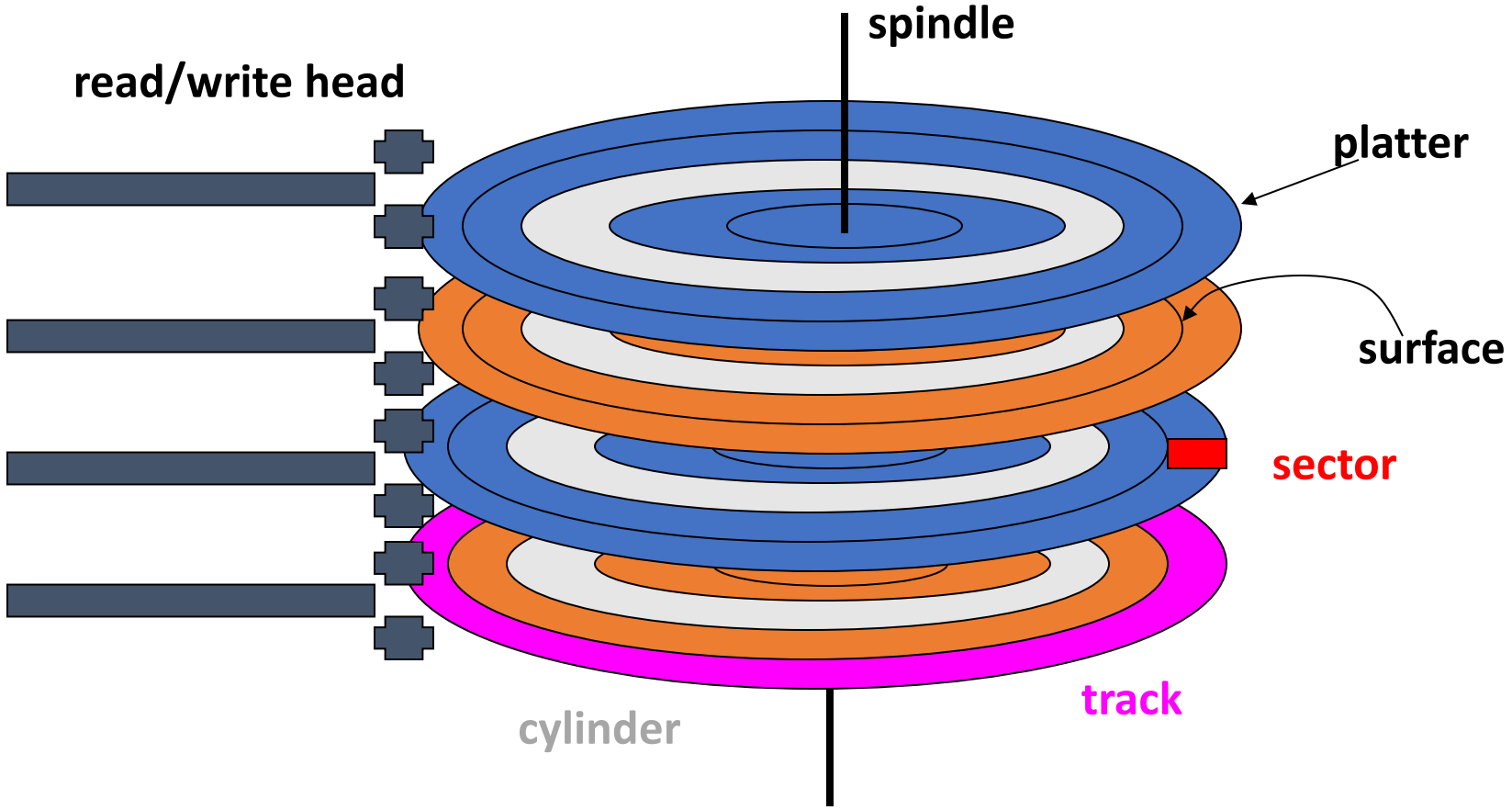
# And this part. Today we continue with this part.



# Next Weeks' Lectures



# Recap: Disk Terminology



# Disk Access Time

| Component | Time |
|---|---|
| Head Selection | nanoseconds |
| **Seek Time** | **3-12 milliseconds** |
| **Rotational Latency** | **2-7 milliseconds** |
| Transfer Time | microseconds |
| Controller Overhead | < 1 millisecond |

# Disk Access Time

**Seek time dominates**

| Component | Time |
|---|---|
| Head Selection | nanoseconds |
| **Seek Time** | **3-12 milliseconds** |
| **Rotational Latency** | **2-7 milliseconds** |
| Transfer Time | microseconds |
| Controller Overhead | < 1 millisecond |

**Note: Disk access time >> memory access time (nanoseconds)**

# Let’s Practice! Disk Access Time

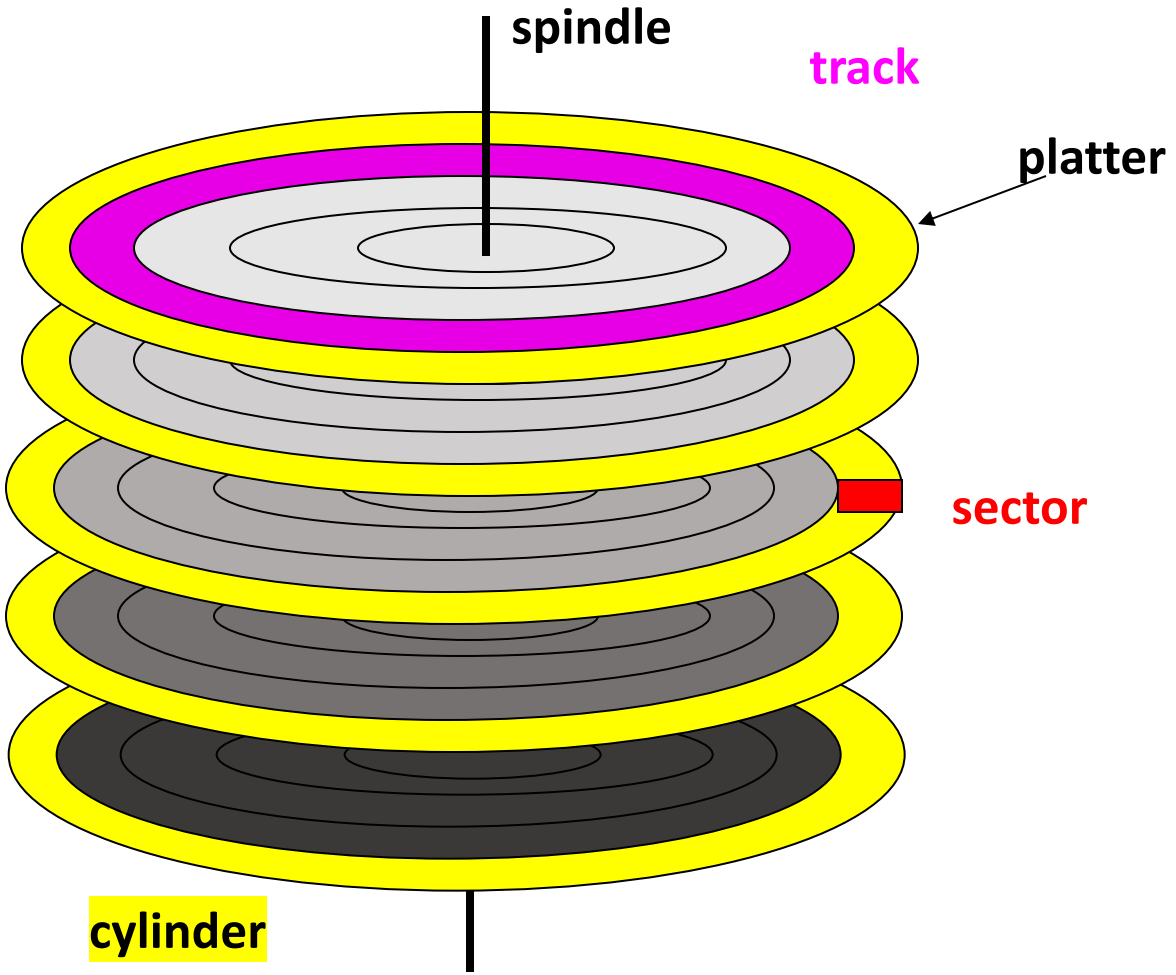
Consider a disk with a sector size of 512 bytes, 1,000 tracks per surface, 100 sectors per track, 5 double-sided platters and a block size of 2,048 bytes. Suppose that the average seek time is 10ms, the average rotational delay is 5 ms, and the transfer rate is 200 MB per second. Suppose that a file containing 1,000,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

A. How many blocks are required to store the entire file?

B. If the file is arranged sequentially on disk, how many cylinders are needed?

C. What is the time required to read the file sequentially?

# Let's Practice! Disk Access Time




sector size of 512 bytes, 1,000 tracks per surface, 100 sectors per track, 5 double-sided platters and a block size of 2,048 bytes.

# Let’s Practice! Disk Access Time

Consider a disk with a sector size of 512 bytes, 1,000 tracks per surface, 100 sectors per track, 5 double-sided platters and a block size of 2,048 bytes. Suppose that the average seek time is 10ms, the average rotational delay is 5 ms, and the transfer rate is 200 MB per second. Suppose that a file containing 1,000,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

A. How many blocks are required to store the entire file?

# Let's Practice! Disk Access Time

Consider a disk with a sector size of 512 bytes, 1,000 tracks per surface, 100 sectors per track, 5 double-sided platters and a block size of 2,048 bytes. Suppose that the average seek time is 10ms, the average rotational delay is 5 ms, and the transfer rate is 200 MB per second. Suppose that a file containing 1,000,000 records of 100 bytes each is to be stored on such a disk and that **no record is allowed to span two blocks.**

A. How many blocks are required to store the entire file?

→ How many records fit in a block?

**20 records** (2048B can fully fit 20 records of 100B each)

→ 1,000,000 records are stored in 1,000,000 / 20 = **50,000 blocks**

# Let’s Practice! Disk Access Time

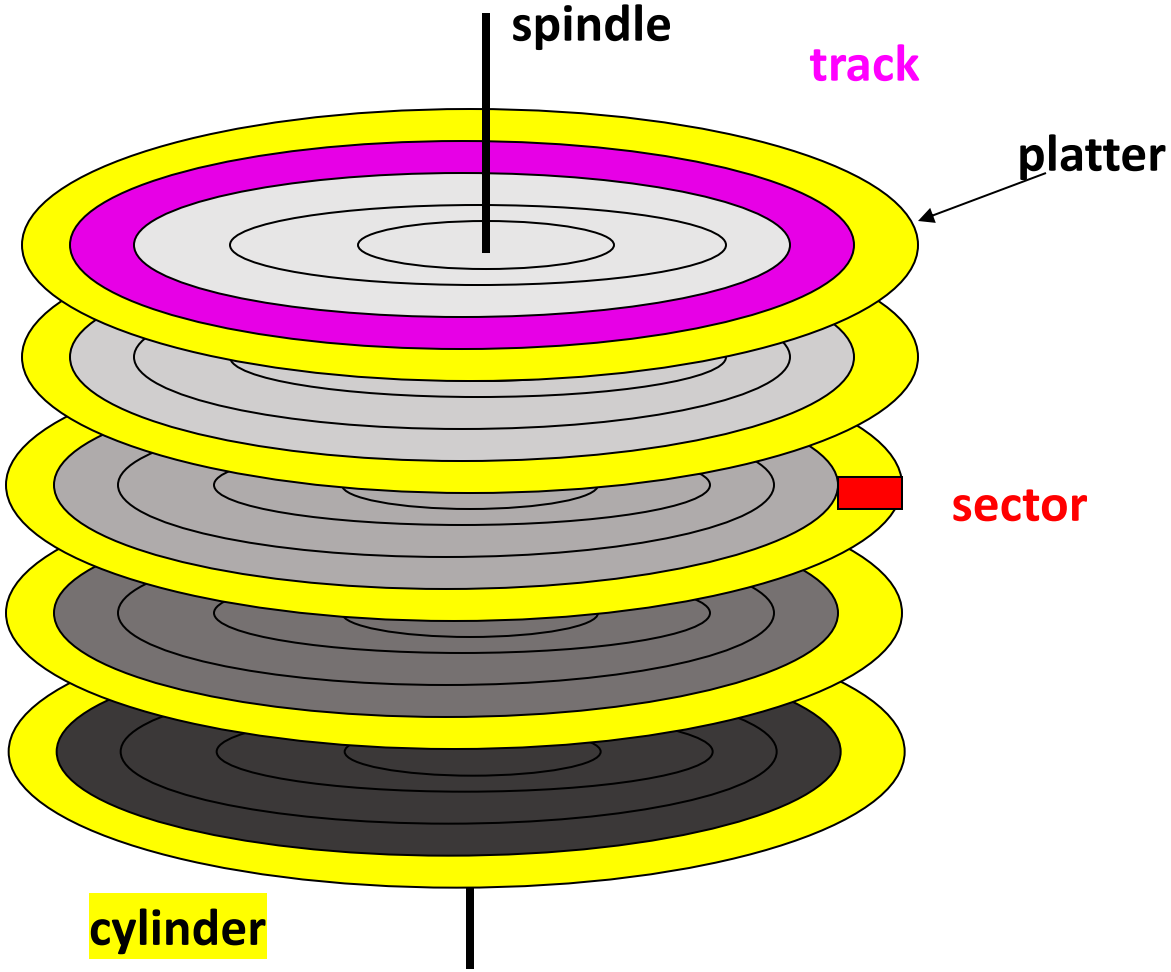
Consider a disk with a sector size of 512 bytes, 1,000 tracks per surface, 100 sectors per track, 5 double-sided platters and a block size of 2,048 bytes. Suppose that the average seek time is 10ms, the average rotational delay is 5 ms, and the transfer rate is 200 MB per second. Suppose that a file containing 100,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

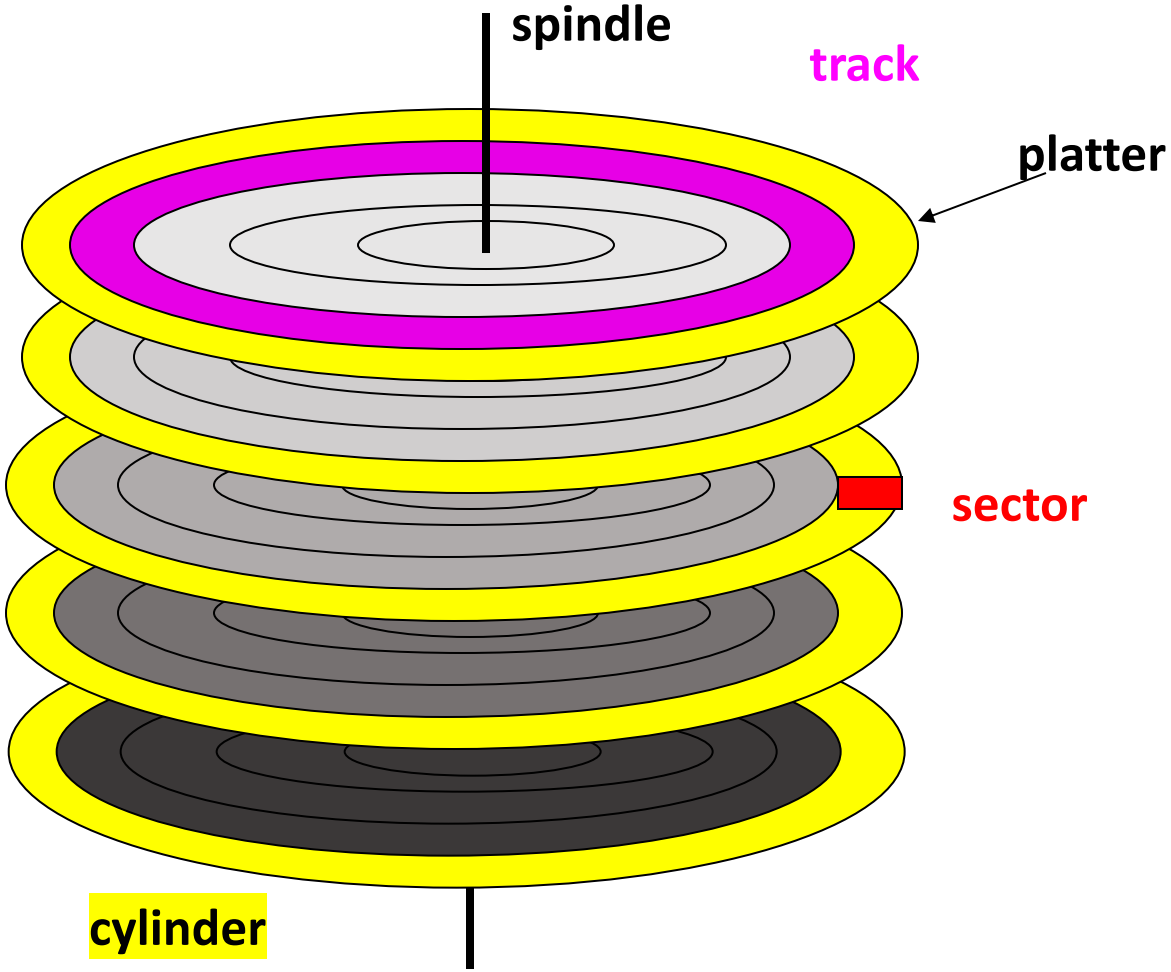
A. How many blocks are required to store the entire file? → **50,000 blocks**

B. If the file is arranged sequentially on disk, how many cylinders are needed?

2.B. If the file is arranged **sequentially on disk**, how many cylinders are needed?

Need to place 50,000 blocks (computed in 2.A).

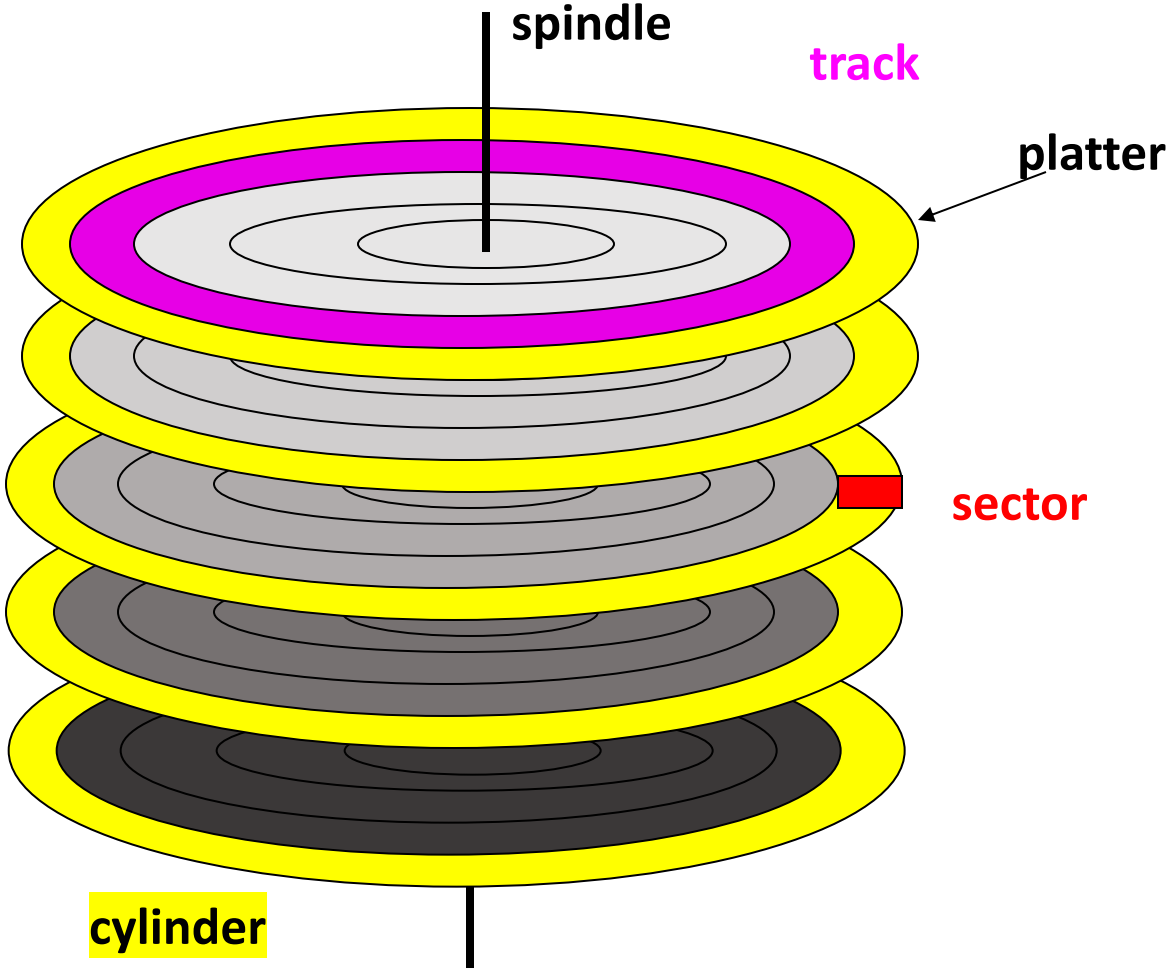





2.B. If the file is arranged **sequentially on disk**, how many cylinders are needed?

Need to place 50,000 blocks (computed in 2.A).

→ **How de we arrange blocks for sequential access?**



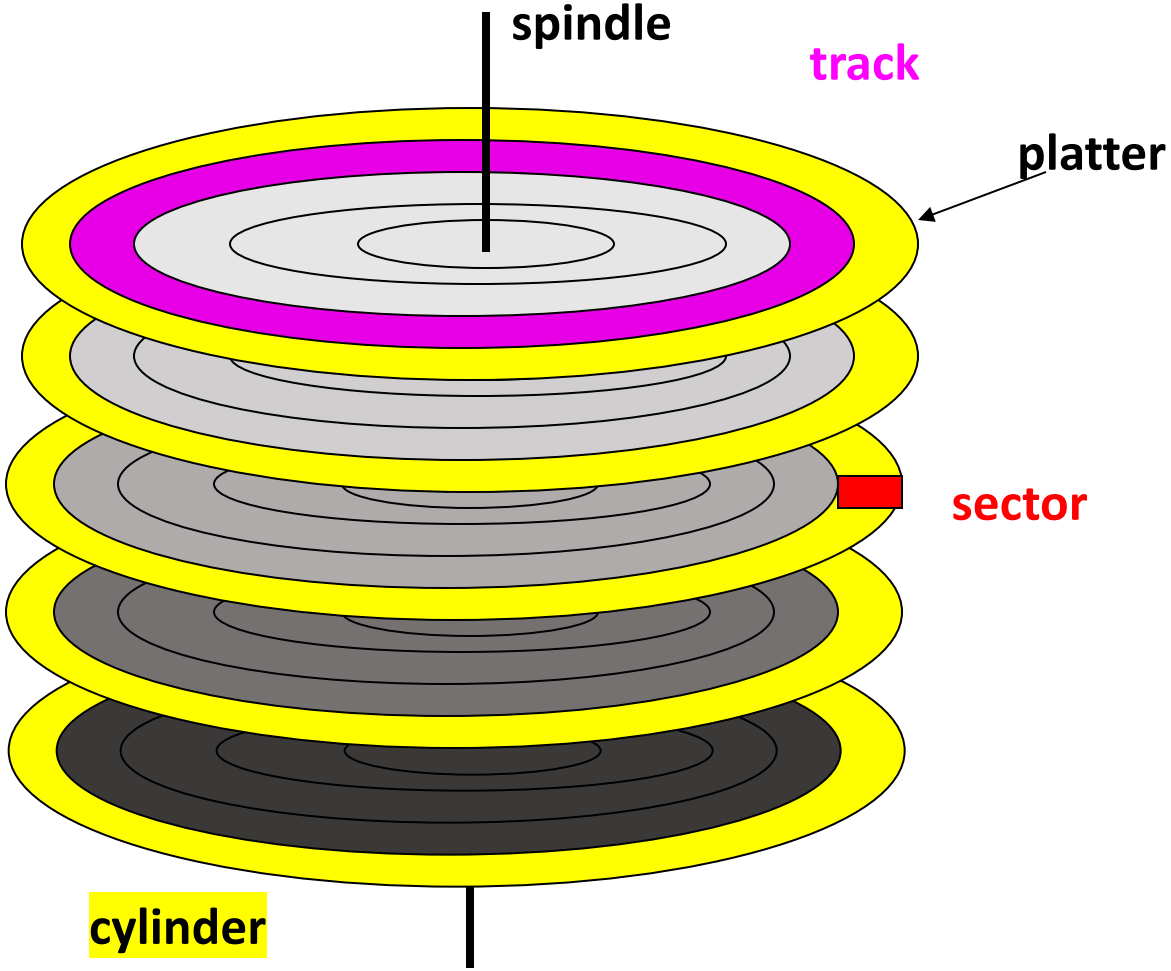

2.B. If the file is arranged **sequentially on disk**, how many cylinders are needed?

Need to place 50,000 blocks (computed in 2.A).

→ **How de we arrange blocks for sequential access?**

Next block concept:

- blocks on same track, followed by
- blocks on same cylinder, followed by
- blocks on adjacent cylinder




2.B. If the file is arranged **sequentially on disk**, how many cylinders are needed?

Need to place 50,000 blocks (computed in 2.A).

→ **How de we arrange blocks for sequential access?**

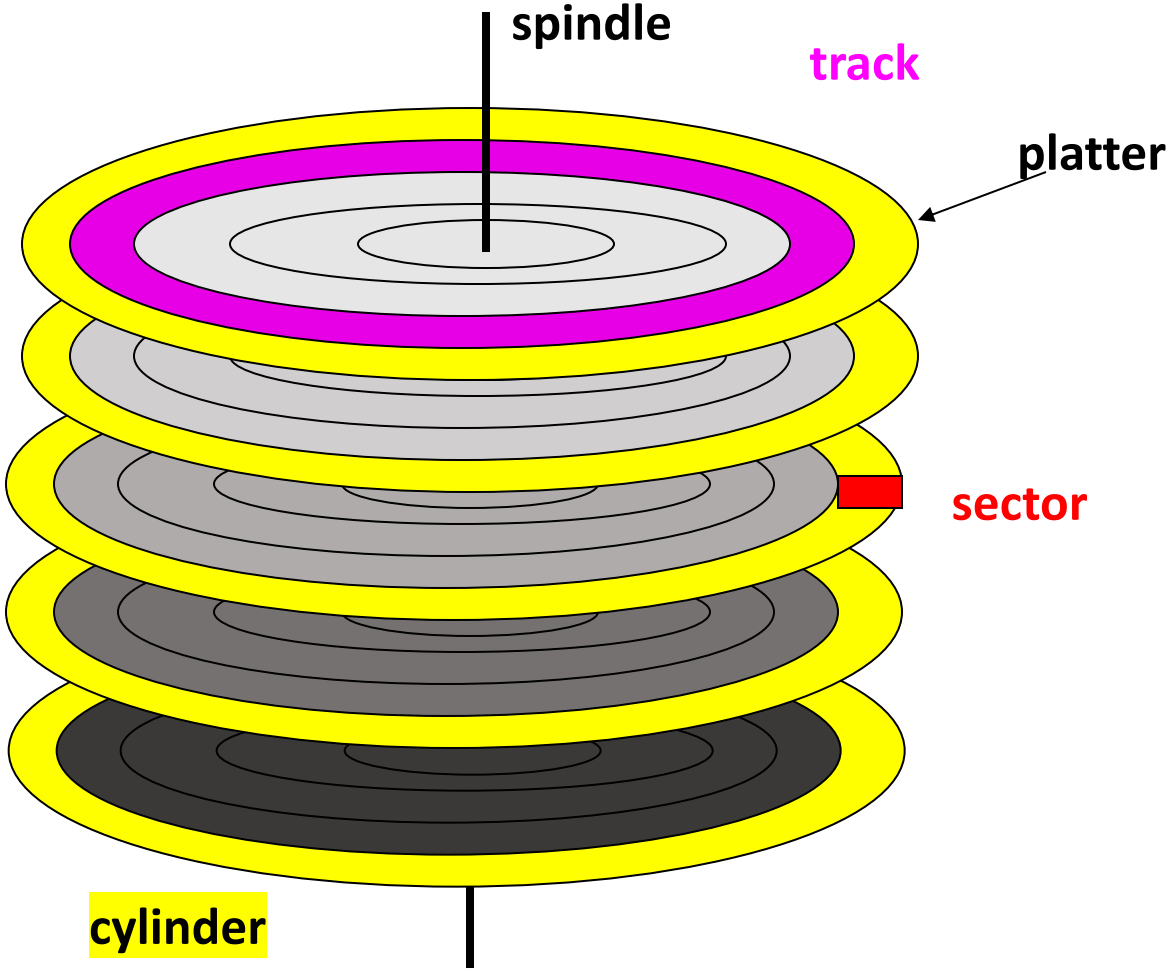
Next block concept:

- blocks on same track, followed by
- blocks on same cylinder, followed by
- blocks on adjacent cylinder

→ **How many blocks can we place in a track?**

→ **How many blocks can we place in a cylinder?**

→ **How many cylinders do we need?**




2.B. If the file is arranged **sequentially on disk**, how many cylinders are needed?

Need to place 50,000 blocks (computed in 2.A).

→ **How de we arrange blocks for sequential access?**

Next block concept:

- blocks on same track, followed by
- blocks on same cylinder, followed by
- blocks on adjacent cylinder

→ **How many blocks can we place in a track?**

Sector size: 512B, 100 sectors per track

→ **25 blocks per track**

→ **How many blocks can we place in a cylinder?**

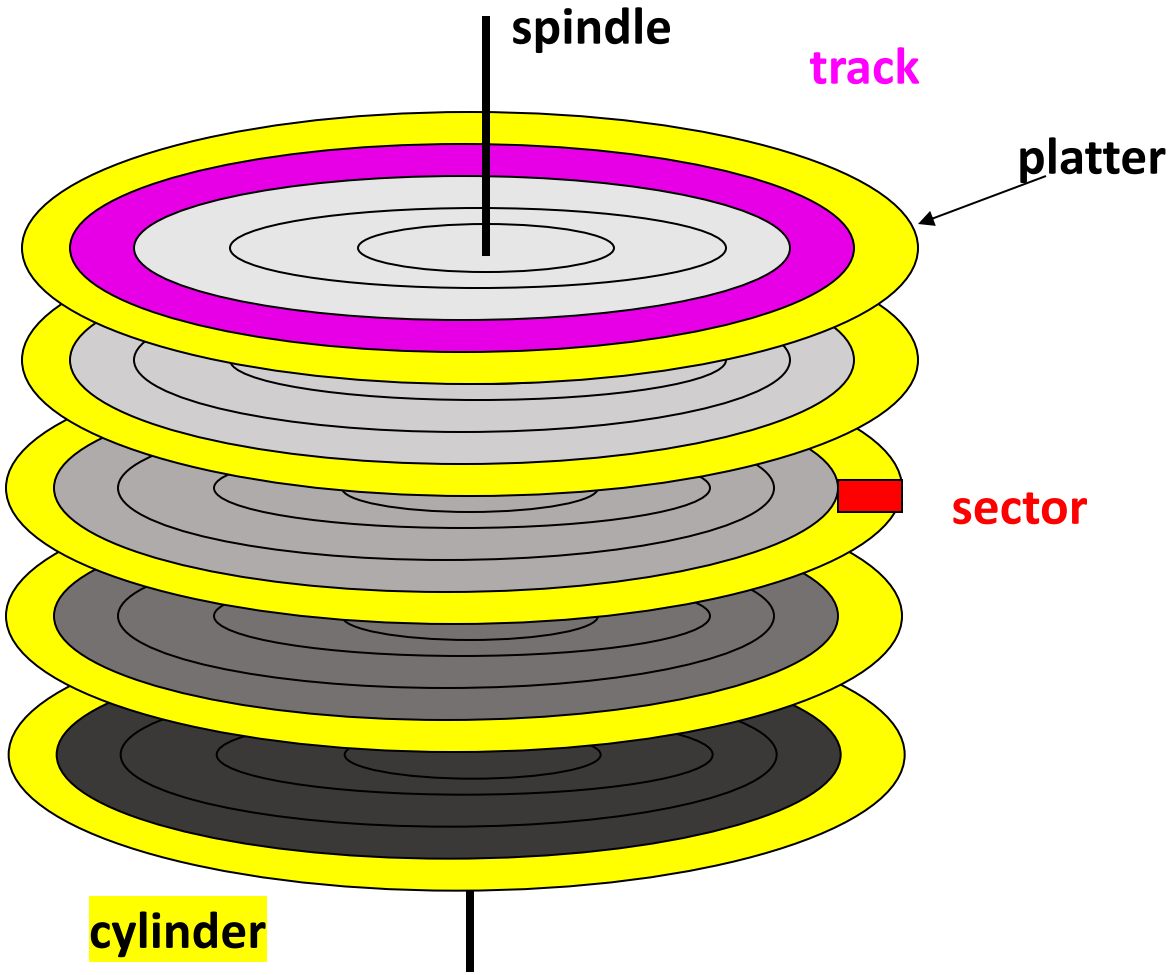
25 x 2 x 5 = **250 blocks per cylinder**

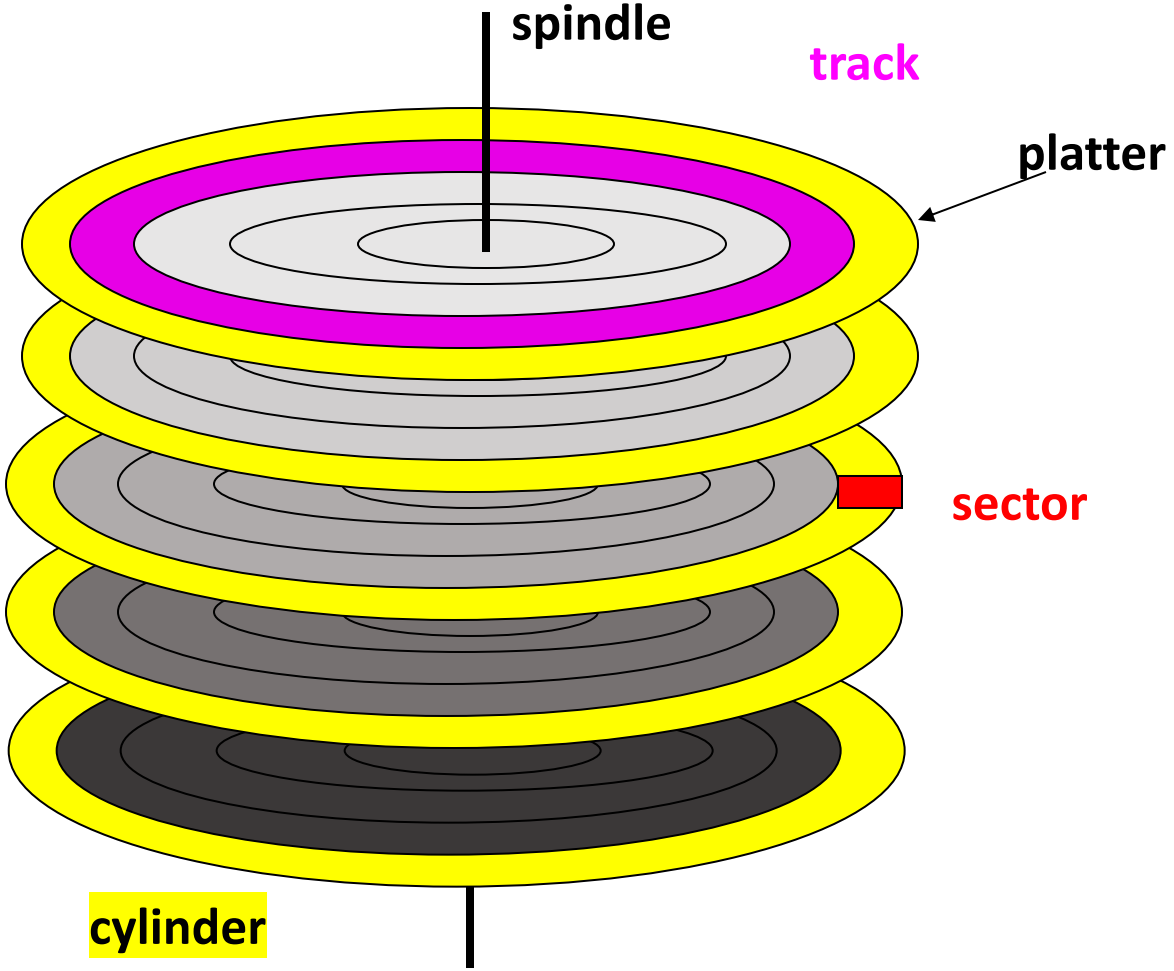
→ **How many cylinders do we need?**

50,000 / 250 = **200 cylinders**

2.C. What is the time required to read the file sequentially?

Need to read 50,000 blocks; each block has size 2048B.





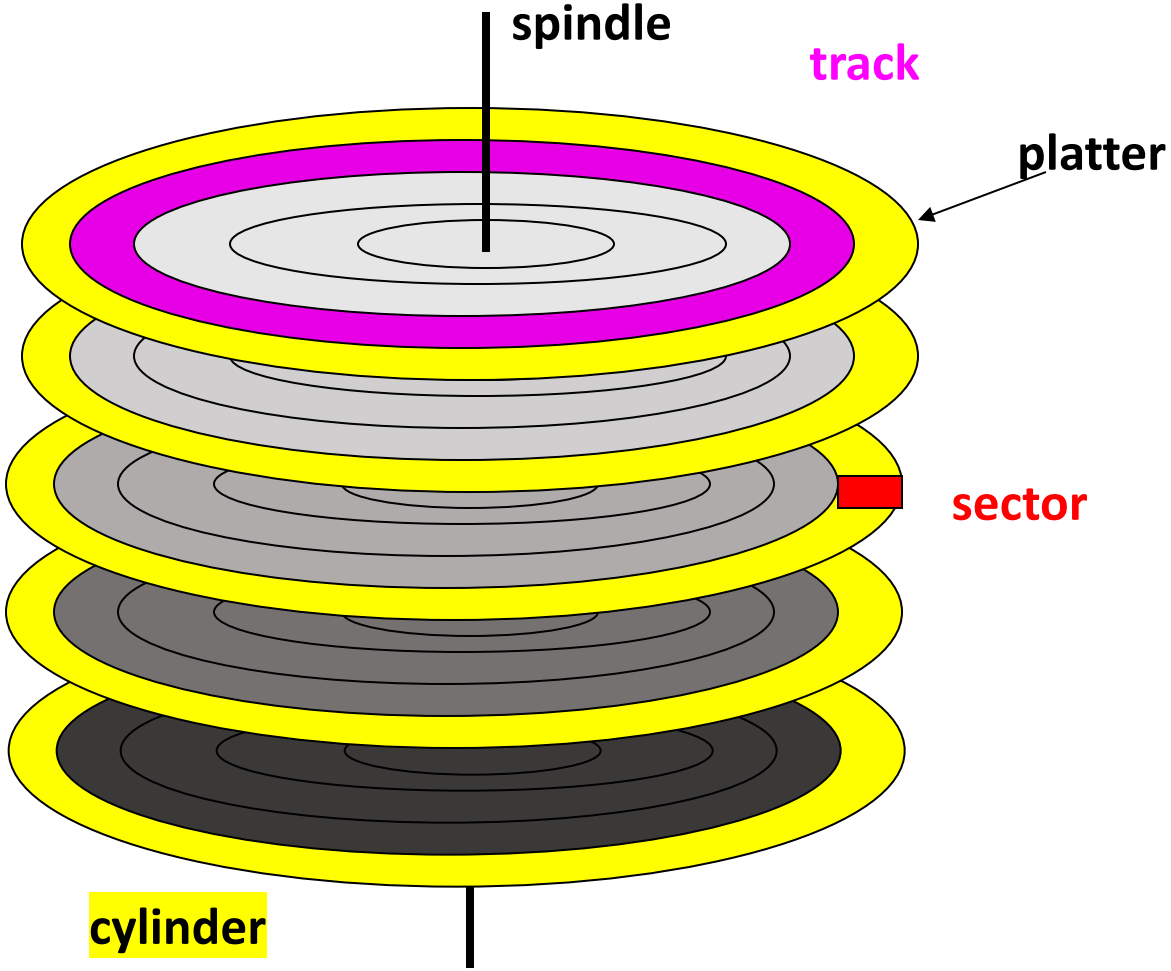

2.C. What is the time required to read the file sequentially?

Need to read 50,000 blocks; each block has size 2048B.

Seek time = 10ms

Avg rotational delay = 5ms

Transfer rate = 200 MB / second.




2.C. What is the time required to read the file sequentially?

Need to read 50,000 blocks; each block has size 2048B.

Seek time = 10ms

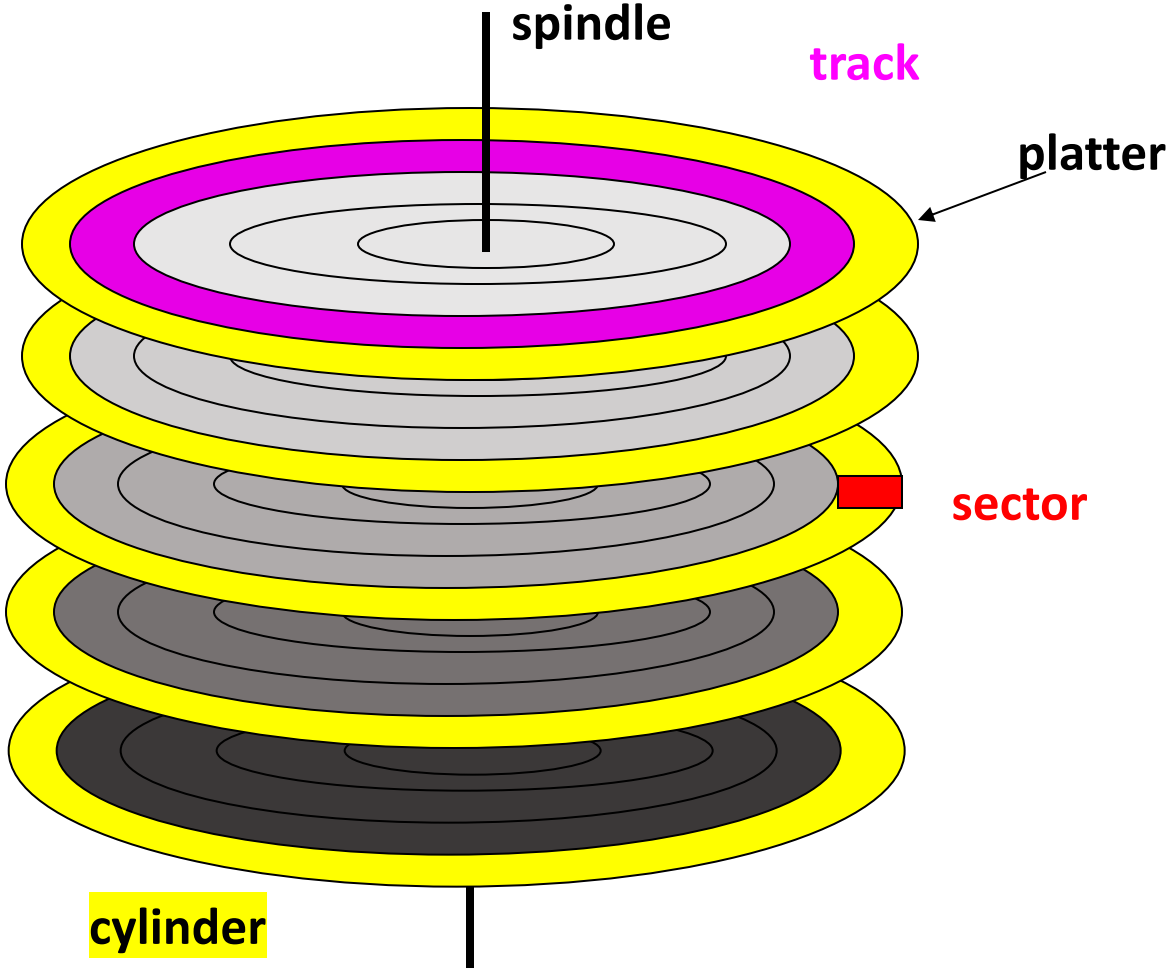
Avg rotational delay = 5ms

Transfer rate = 200 MB / second.

→ **What are the components of disk access?**

→ **How long do each of them take?**

→ **What is the total read time?**




2.C. What is the time required to read the file sequentially?

Need to read 50,000 blocks; each block has size 2048B.

Seek time = 10ms

Avg rotational delay = 5ms

Transfer rate = 200 MB / second.

→ **What are the components of disk access?**

Head selection (negligible)

Seek (one time cost because sequential access)

Rotational Delay (one time cost because sequential access)

Transfer time

Controller overhead (negligible)

→ **What is the transfer time?**

(50000 * 2^11 B) / (200 * 2^20 B/s ) = 0.488 s

= 488 ms

→ **What is the total read time?**

10 + 5 + 488 = **503 ms**

# Optimizing disk access

Remember:

- Disk access time >> memory access time
- If we go to disk, seek time dominates

# Optimize Disk Access

**Rule 1:**

**Do not access disk, use a cache**

# File System Cache (Buffer Cache)

**What?**

- Keep recently accessed blocks in memory

**Why?**

- Reduce latency
- Reduce disk load

**How?**

- Reserve kernel memory for cache
- Cache entries: file blocks (of block size)

# Read with a Cache

**If in cache**

- Return data from cache

**If not**

- Find free cache slot
- Initiate disk read
- When disk read completes, return data

# Write with a Cache

**Always write in cache**

How does it get to disk?

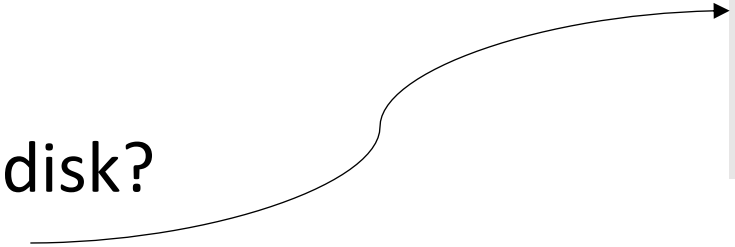
- **Write-through**
- **Write-behind**

# Write with a Cache

**Always write in cache**

How does it get to disk?

- **Write-through**
- **Write-behind**



**Write-through**

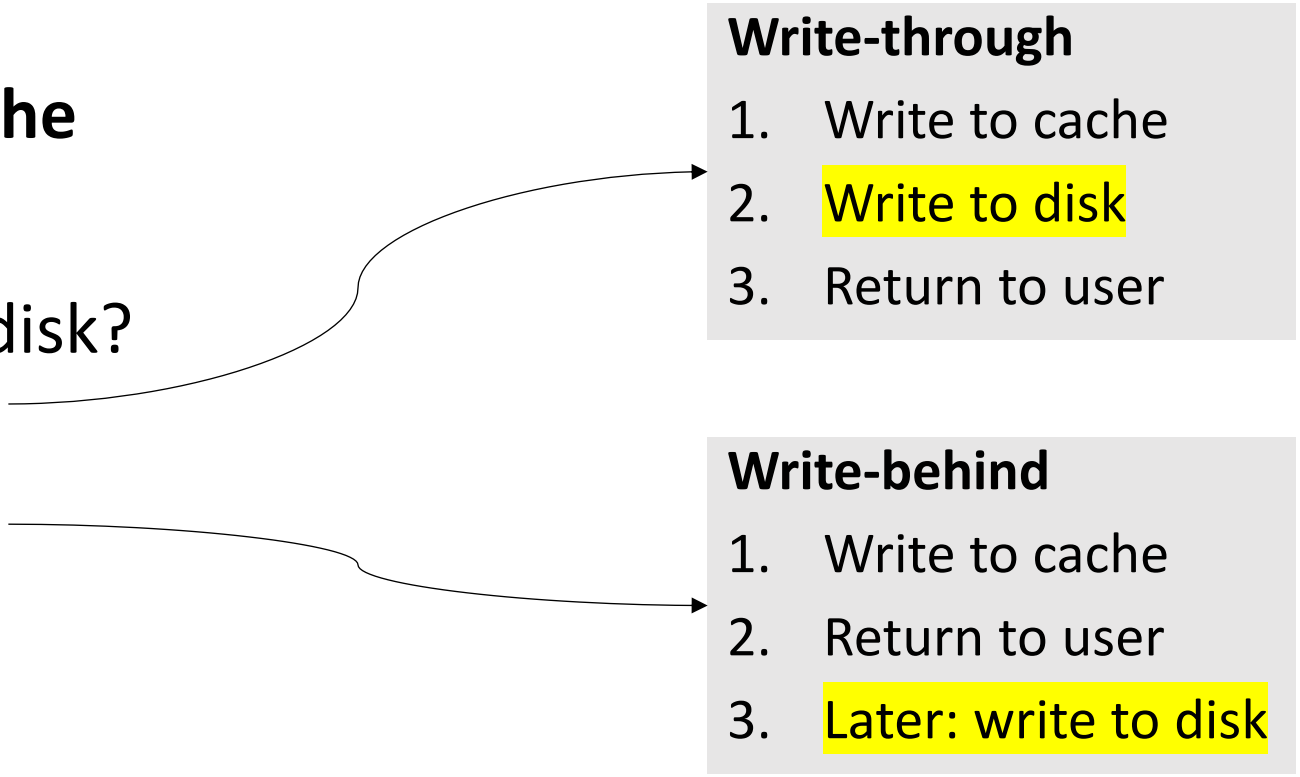
1. Write to cache
2. Write to disk
3. Return to user

# Write with a Cache

**Always write in cache**

How does it get to disk?

- **Write-through**
- **Write-behind**



# Write-Through vs. Write-Behind

**Response time:**

- **Write-behind** is (much) better

**Disk load:**

- **Write-behind** is (much) better
- Much data overwritten before it gets to disk

**Crash:**

- **Write-through** is much better
- No “window of vulnerability”

# Write-Through vs. Write-Behind

**Response time:**

- **Write-behind** is (much) better

**Disk load:**

- **Write-behind** is (much) better
- Much data overwritten before it gets to disk

**Crash:**

- **Write-through** is much better
- No “window of vulnerability”

**In practice:**

- Write-behind
- Periodic cache flush
- User primitive to flush data

# Optimize Disk Access

**Rule 2:**

**Do not wait for disk, read ahead**

- Also called prefetching
- Only for sequential access

# Read-Ahead

**What?**

- User request for block i of a file
- Also read block i+1 from disk

**Why?**

- No disk I/O on (expected) user access to block i+1

**How?**

- Put block i+1 in the buffer cache

→**Remember:** Pre-paging uses read-ahead if neighboring virtual pages are also neighbors in physical memory.

# Read-Ahead

- **Works for sequential access**

- Most access is sequential
- In Linux it is the default

# Caveat about Read-Ahead

- Does not reduce number of disk I/Os
- In fact, could increase them (if not sequential)

- In practice, very often a win
- Linux always reads one block ahead

# Optimize Disk Access

**Rule 3:**

**Minimize seeks**

2 Approaches

- Clever disk allocation
- Clever scheduling

# Clever Disk Allocation

**Idea:**

- Locate related data (same file) on same cylinder
- Allocate “related” blocks “together”

**“together”**

- On the same cylinder
- On a nearby cylinder

**“related”**

- Consecutive blocks in the same file
- Sequential access

# Disk Scheduling

**Idea:** Reorder requests to seek as little as possible

Different disk scheduling policies:

- FCFS – First-Come-First-Served
- SSTF – Shortest-Seek-Time-First
- SCAN
- C-SCAN
- LOOK
- C-LOOK

# Disk Scheduling Illustration

Initial position of the **head = cylinder 53**

Queue of requests:

**98, 193, 37, 122, 14, 124, 65, 67**

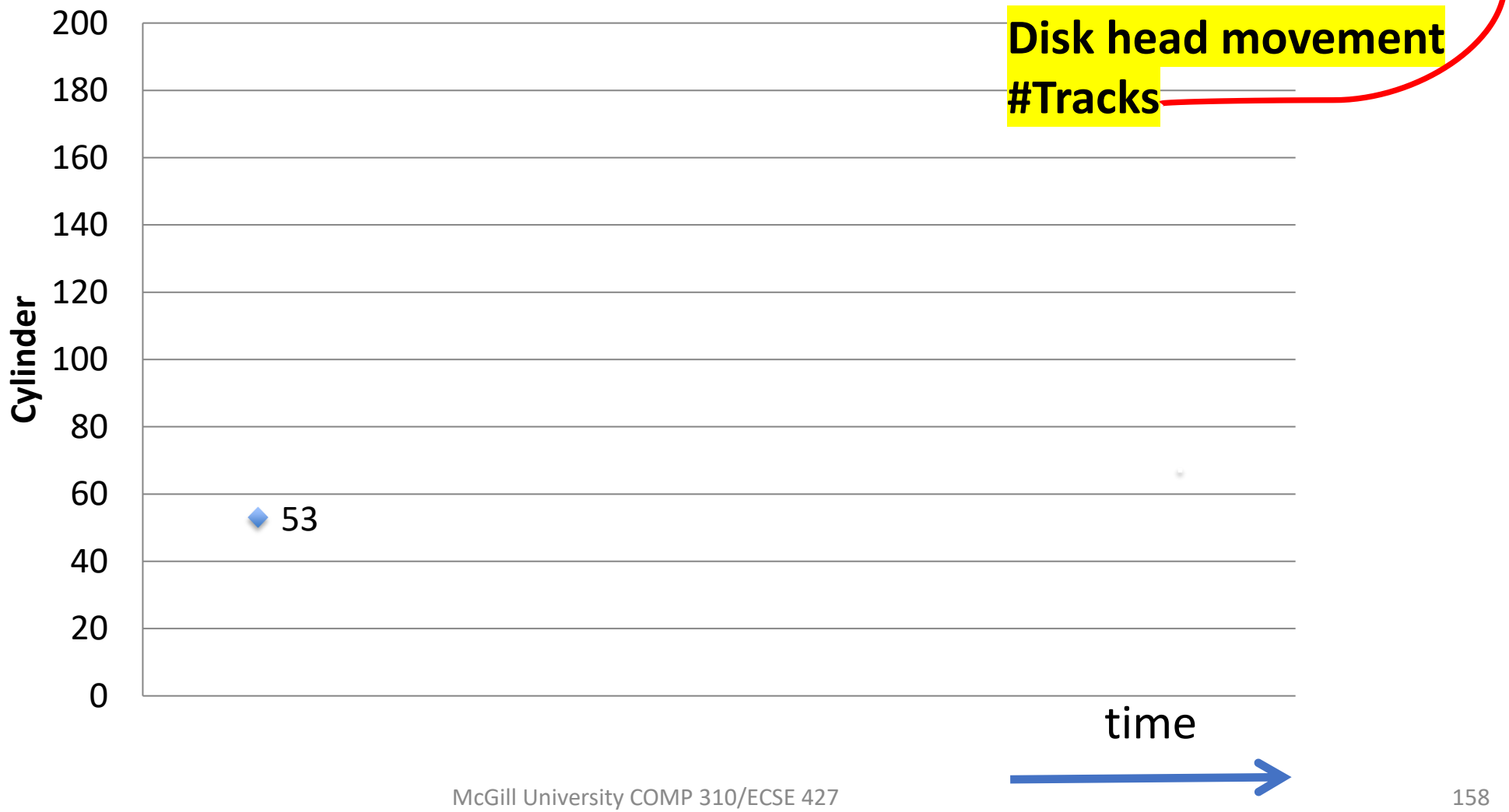
# First Come, First Served (FCFS)

Serve **next request** in the queue

Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

**FCFS**

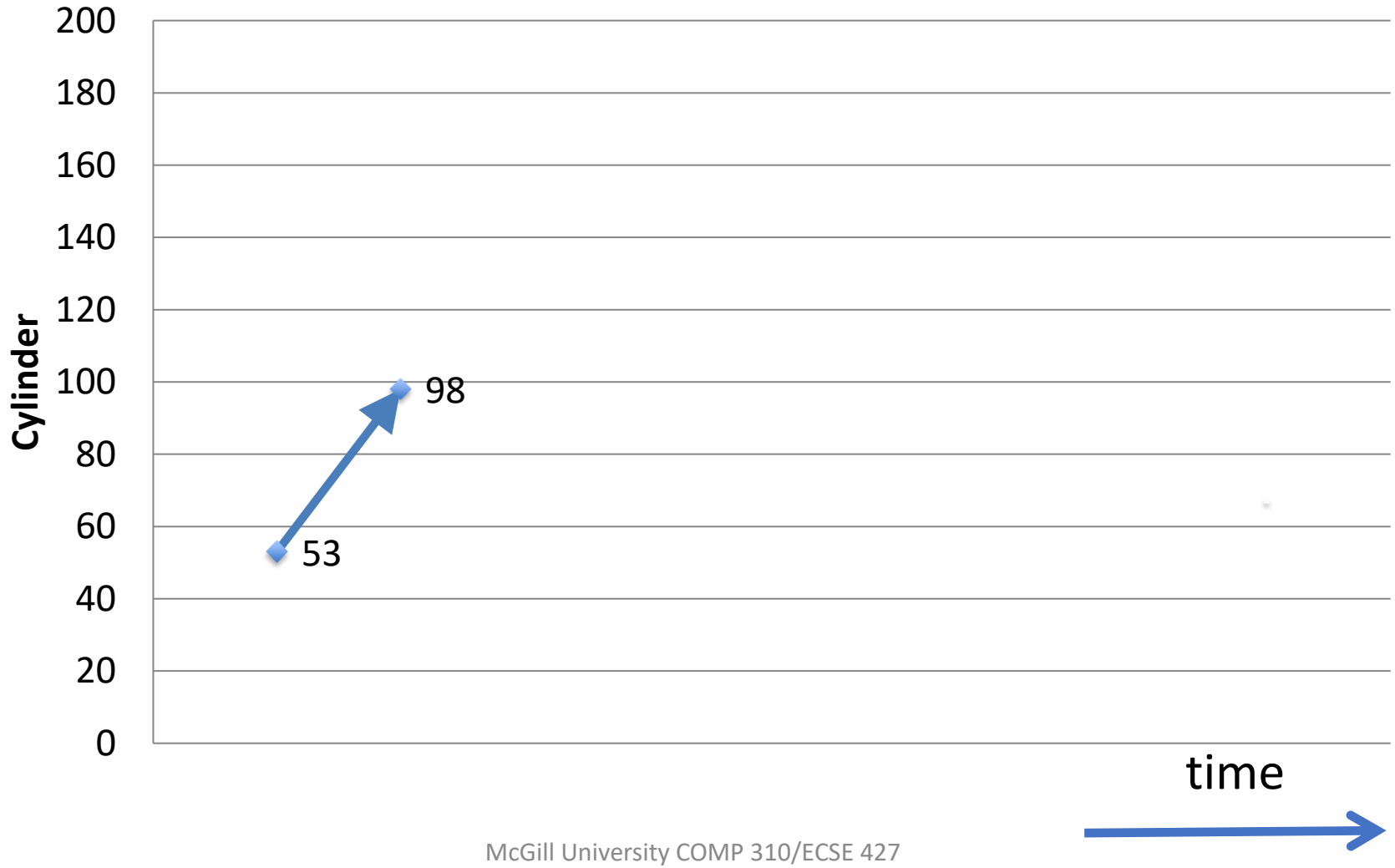


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

45

**FCFS**

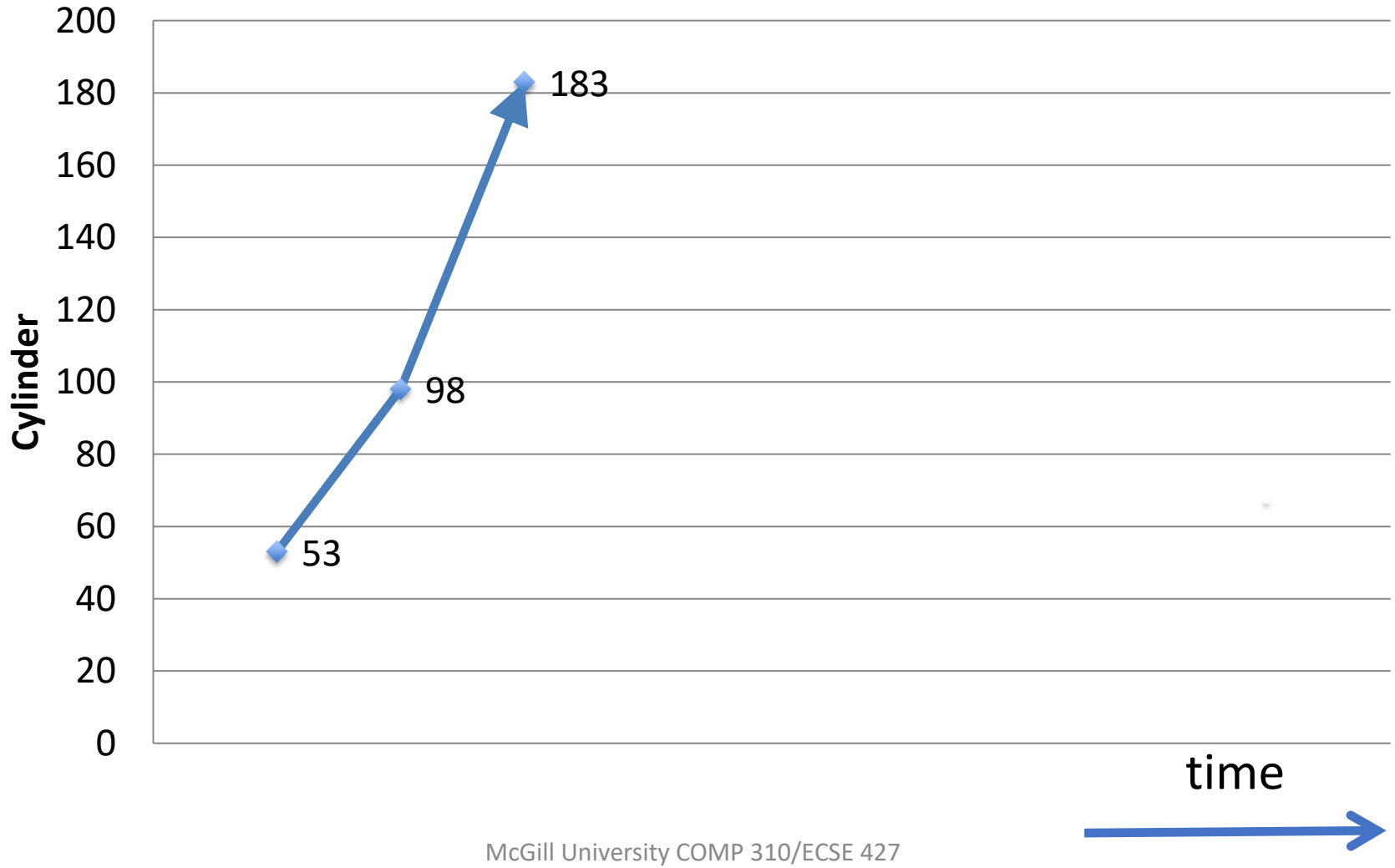


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

130

**FCFS**

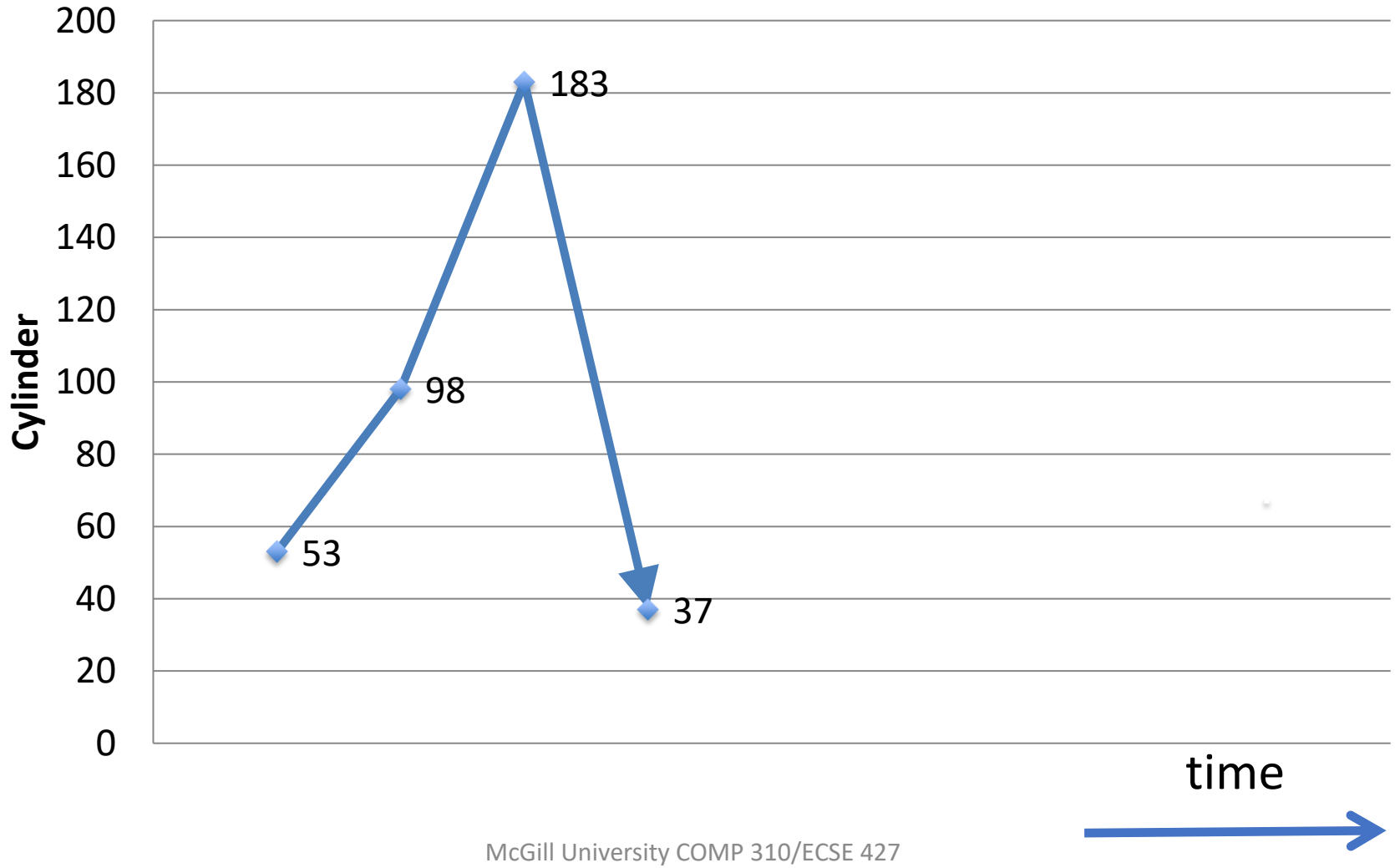


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

276

**FCFS**

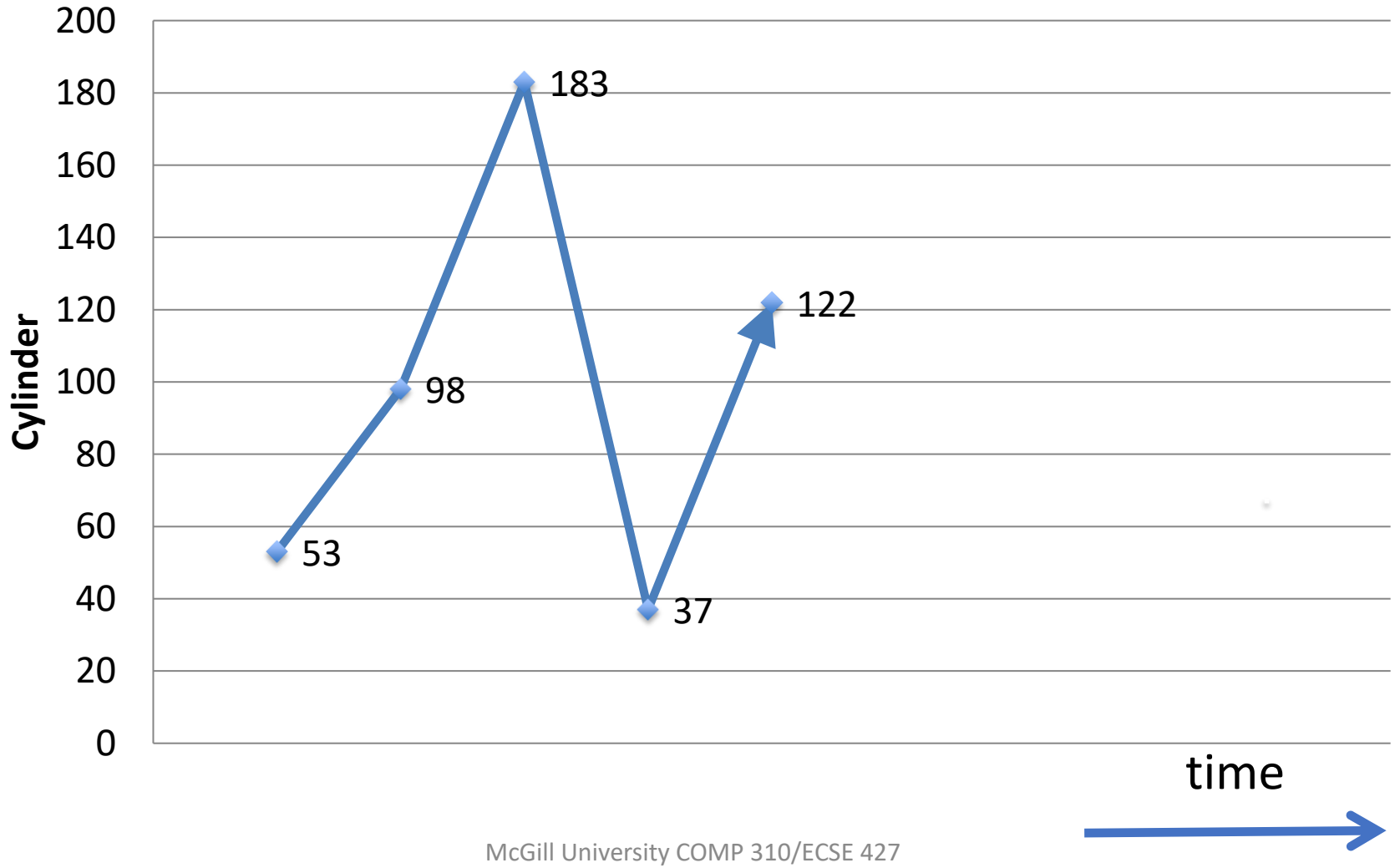


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

361

**FCFS**

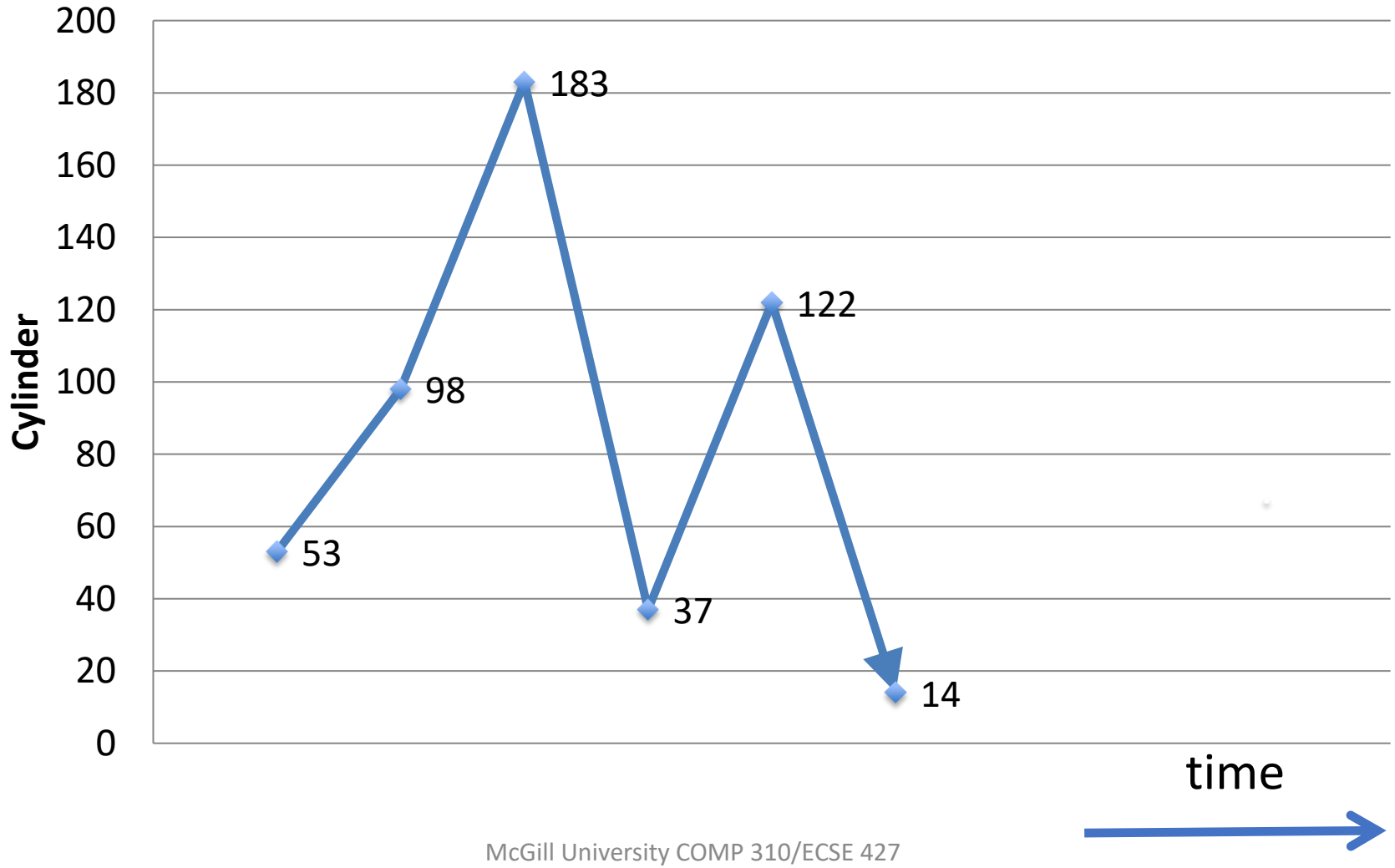


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

469

**FCFS**

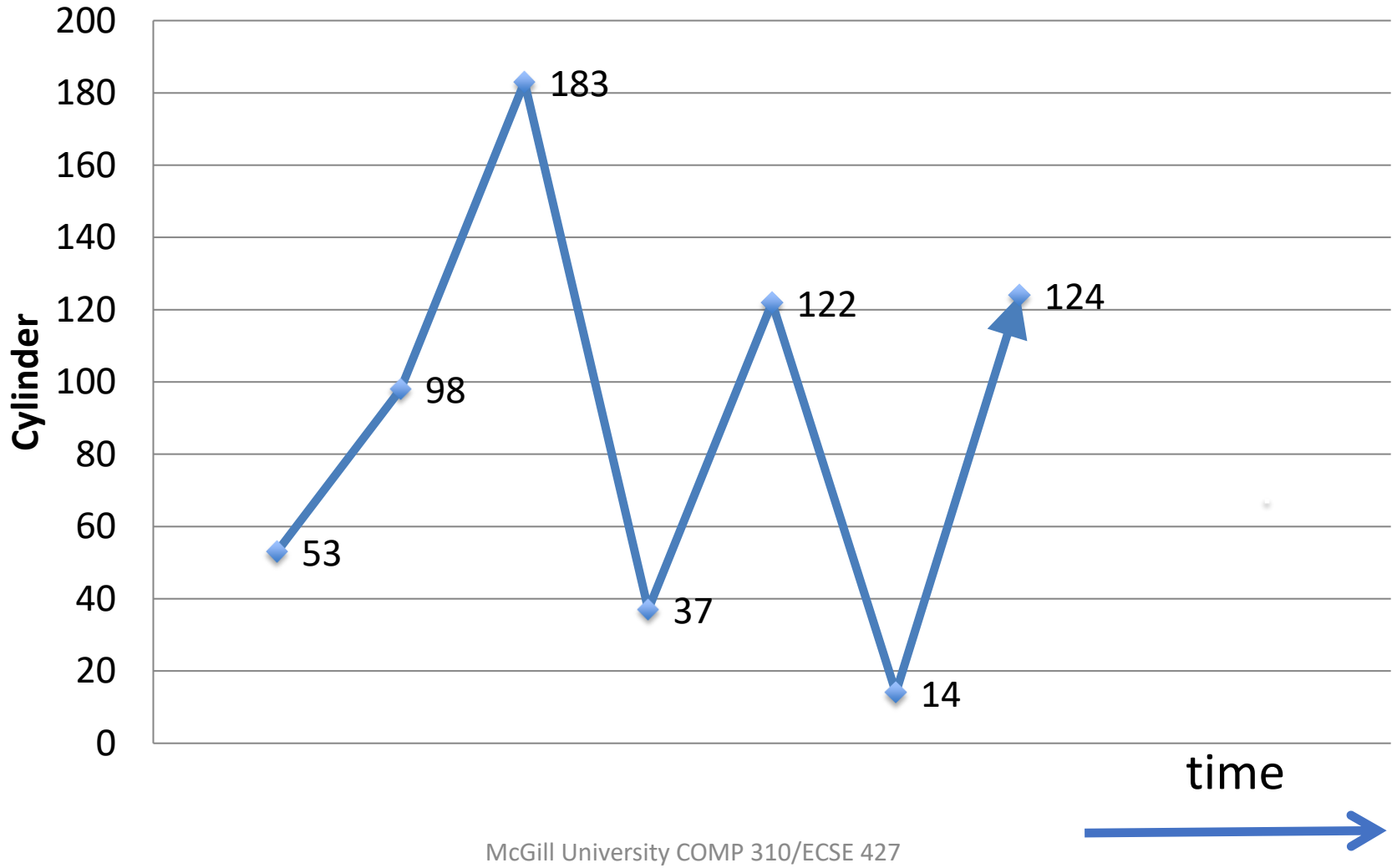


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

579

**FCFS**

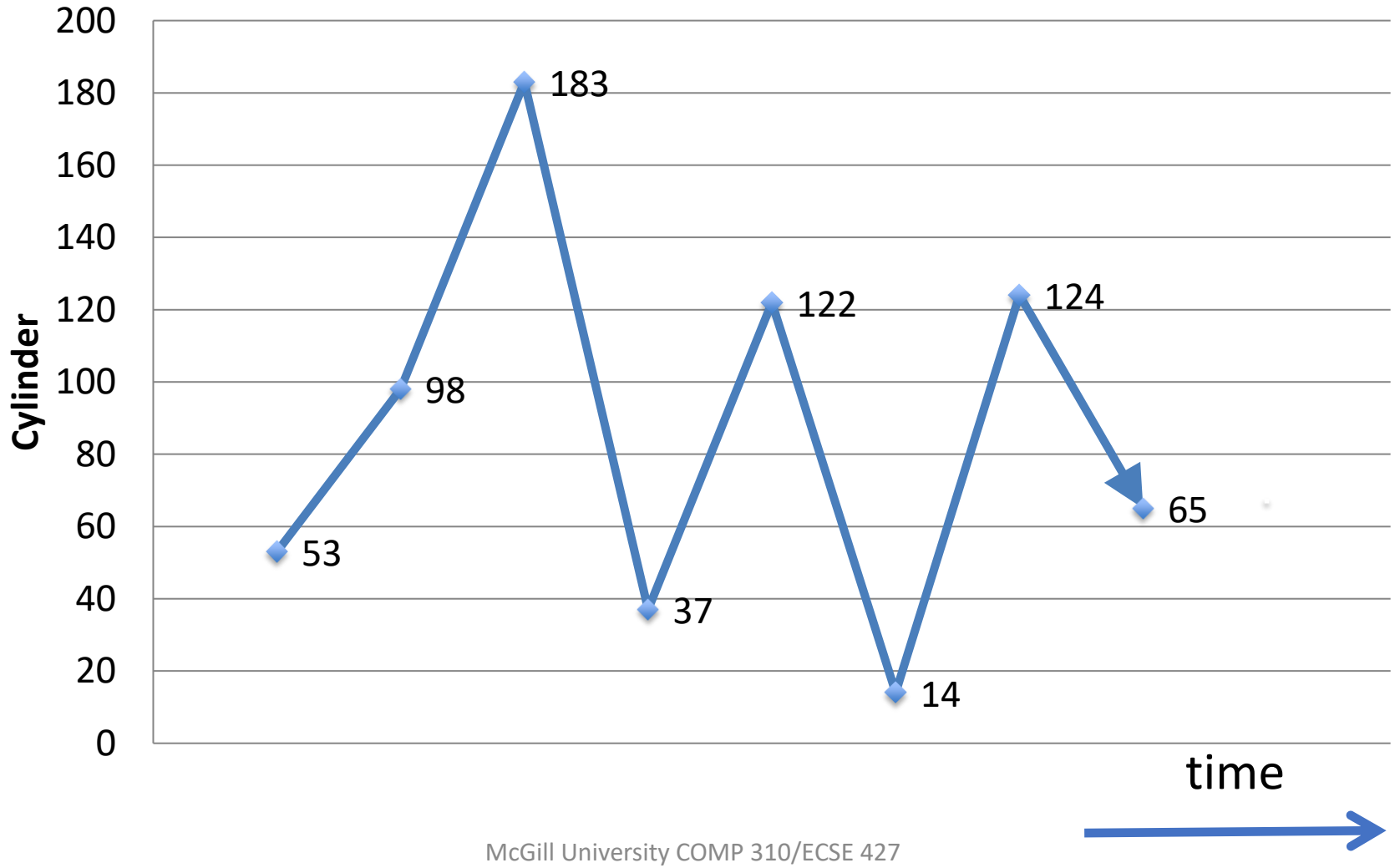


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

638

**FCFS**

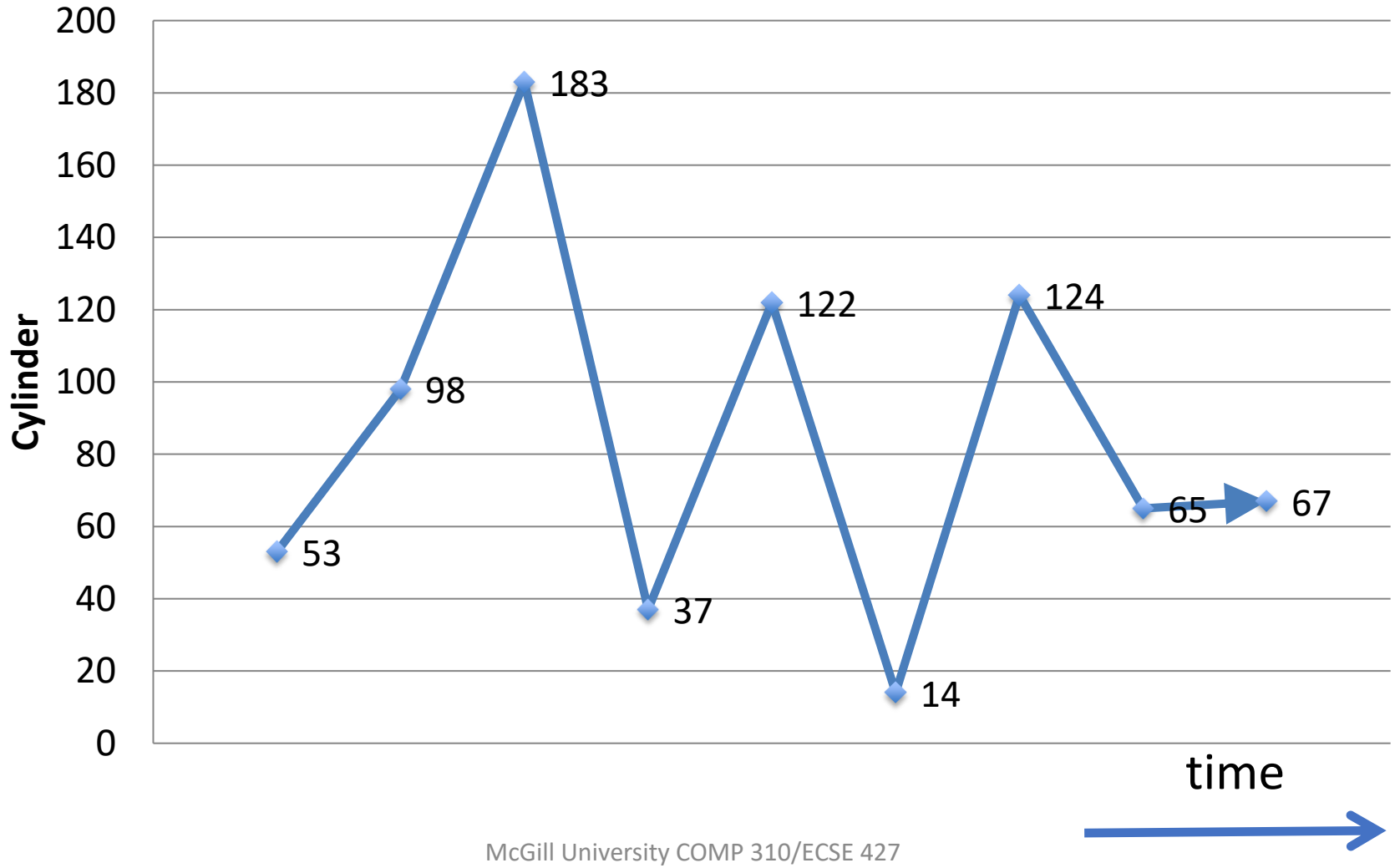


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

640

**FCFS**



# Shortest Seek Time First (SSTF)

Pick **“nearest” request** in queue

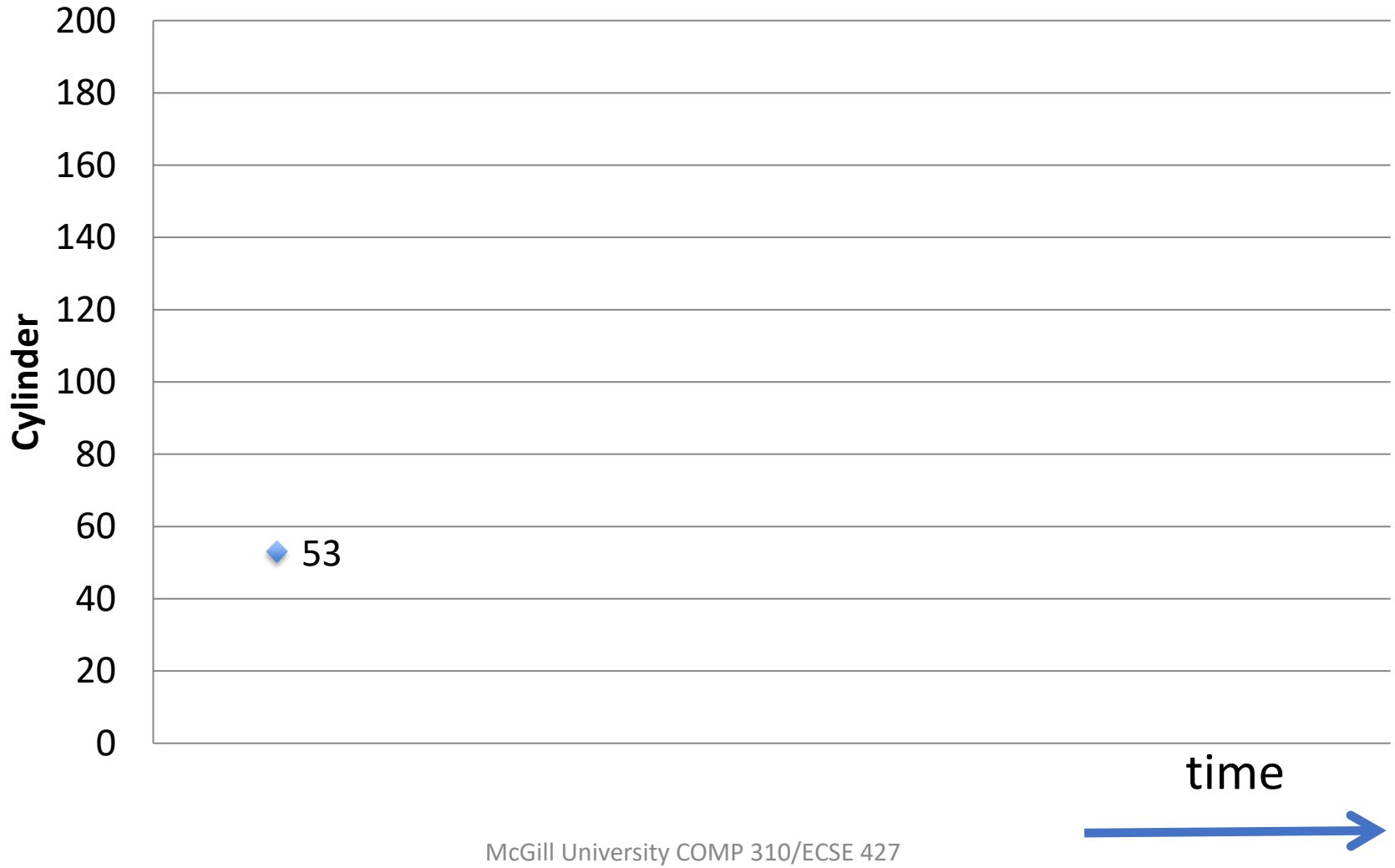
- “nearest” = closest to current head position

Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

0

**SSTF**

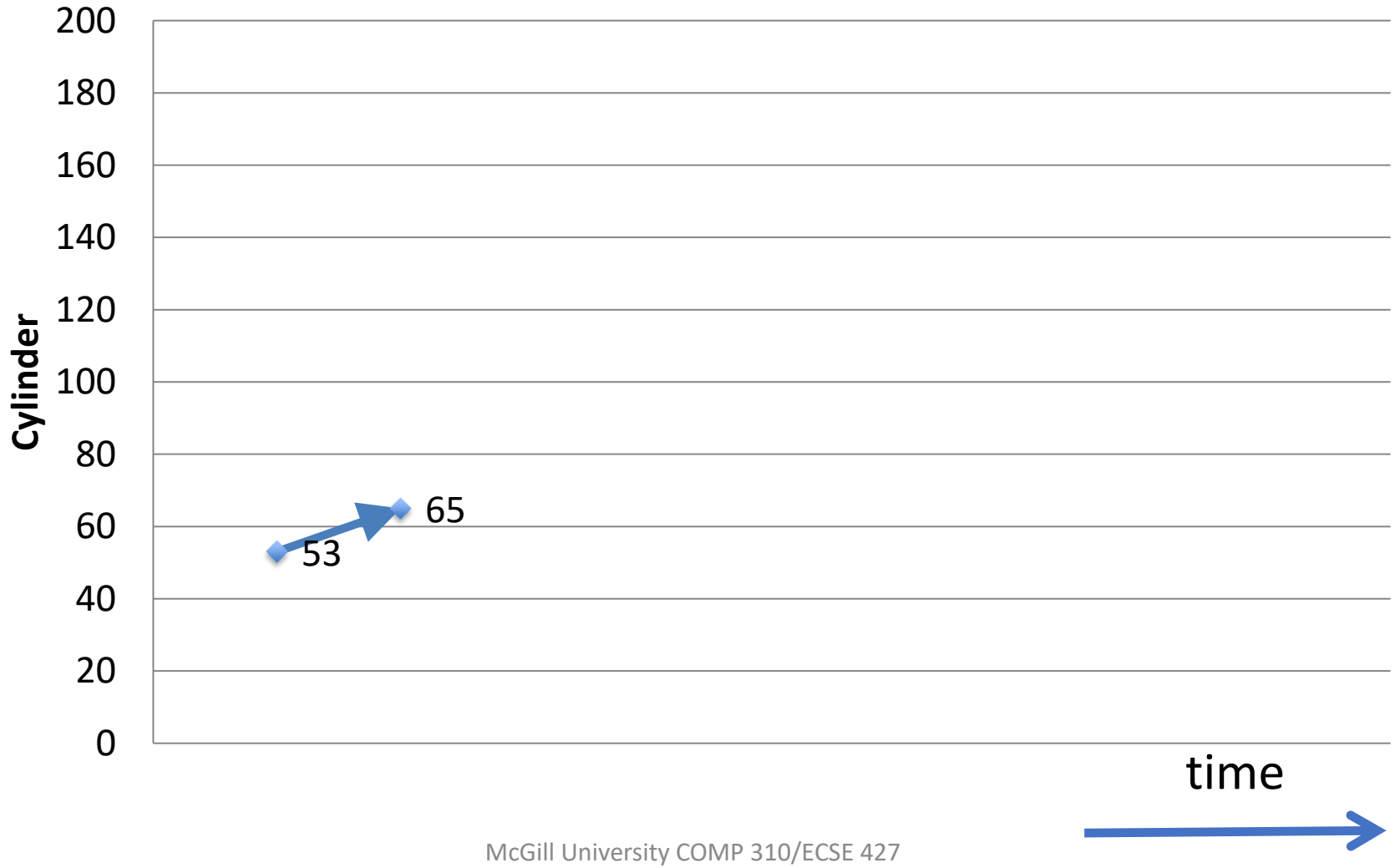


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

12

**SSTF**

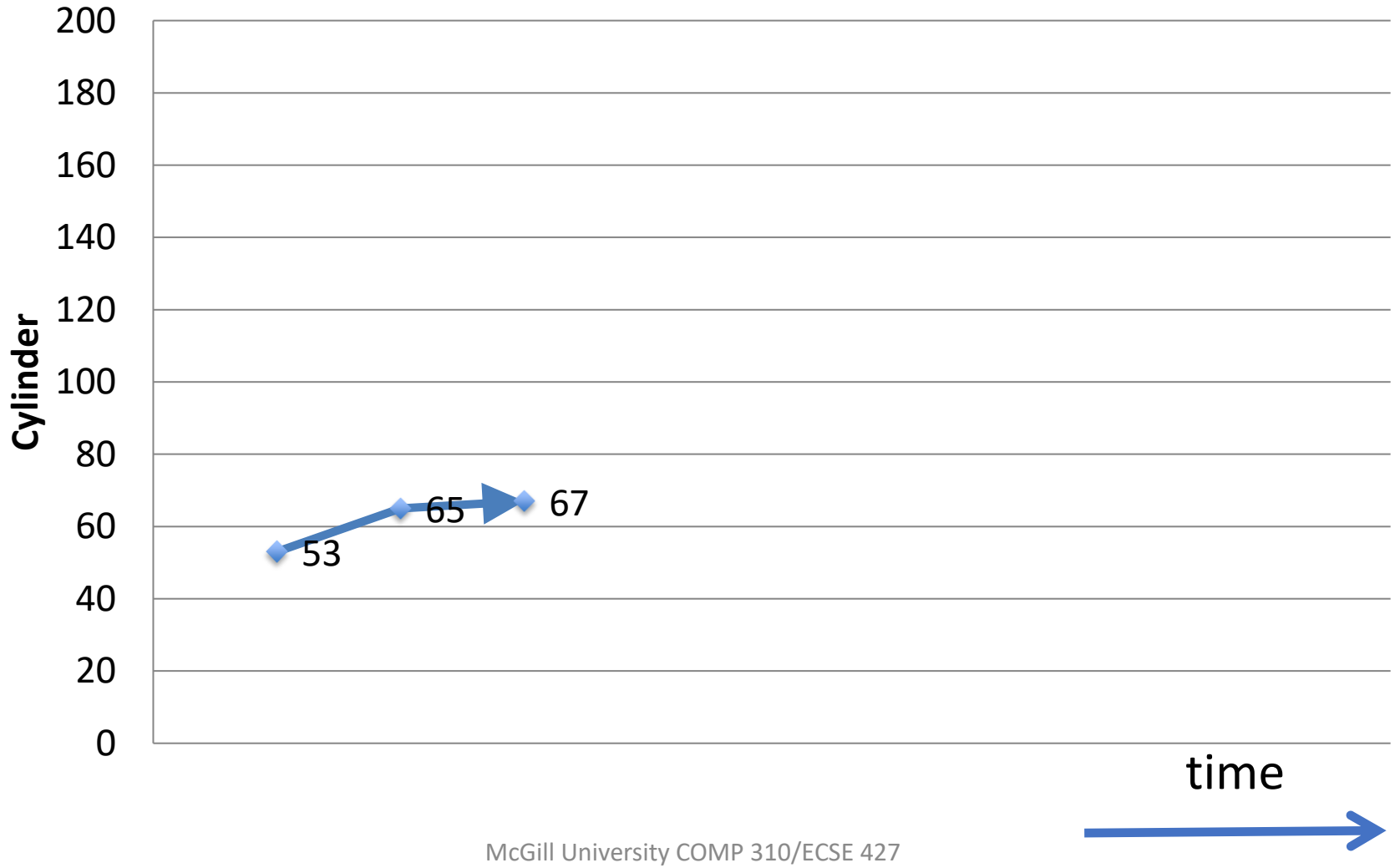


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

14

**SSTF**

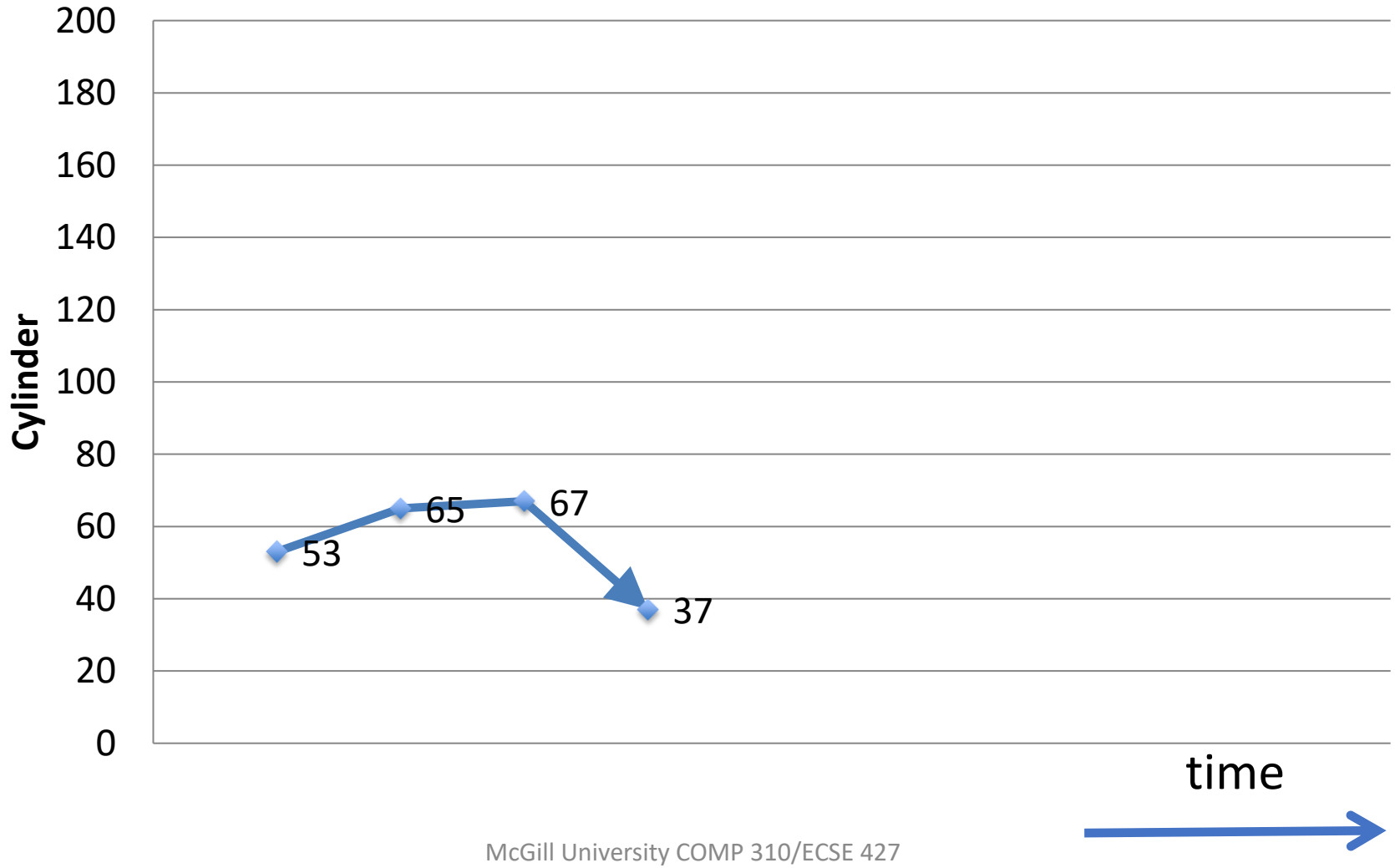


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

44

**SSTF**

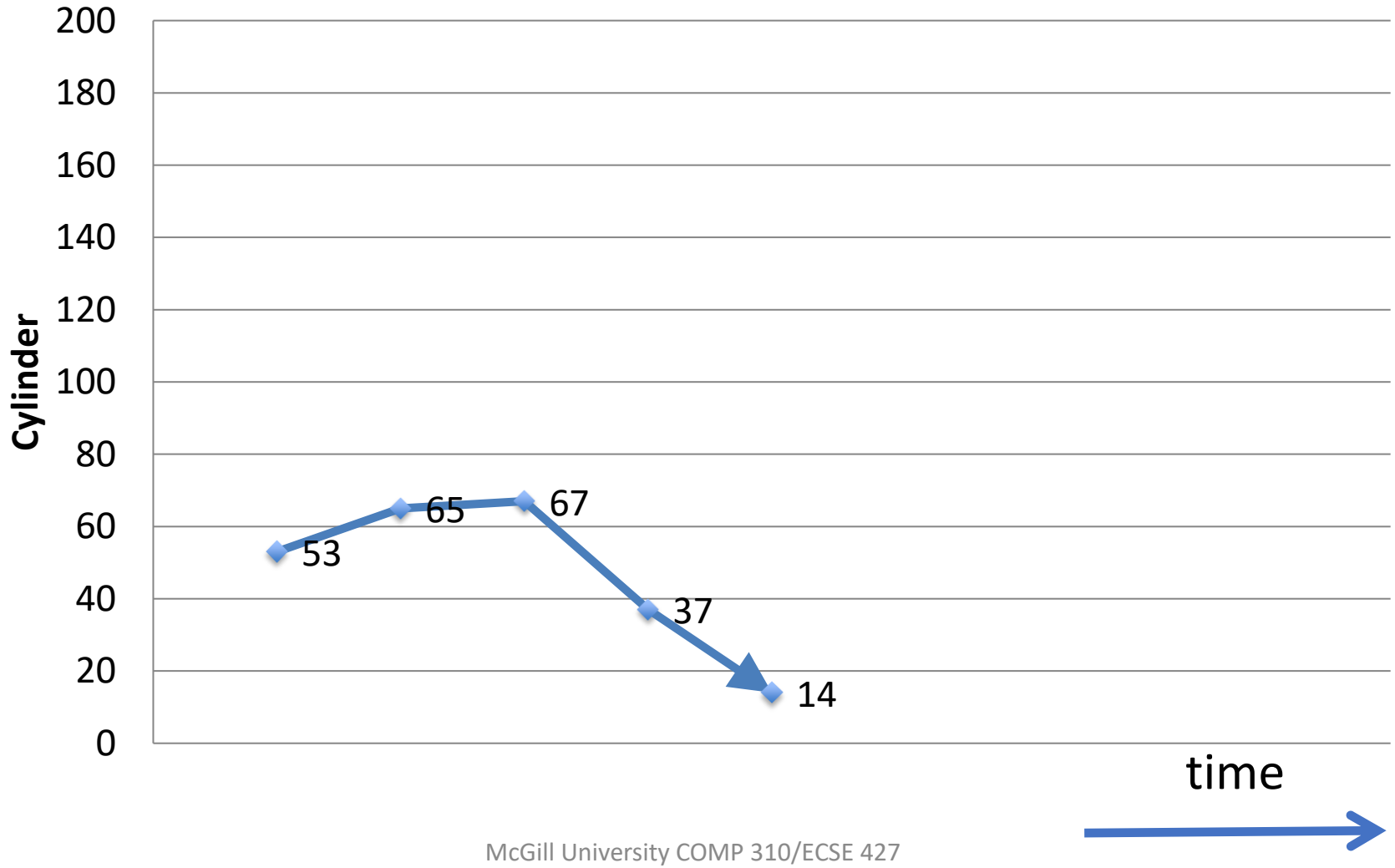


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

67

**SSTF**

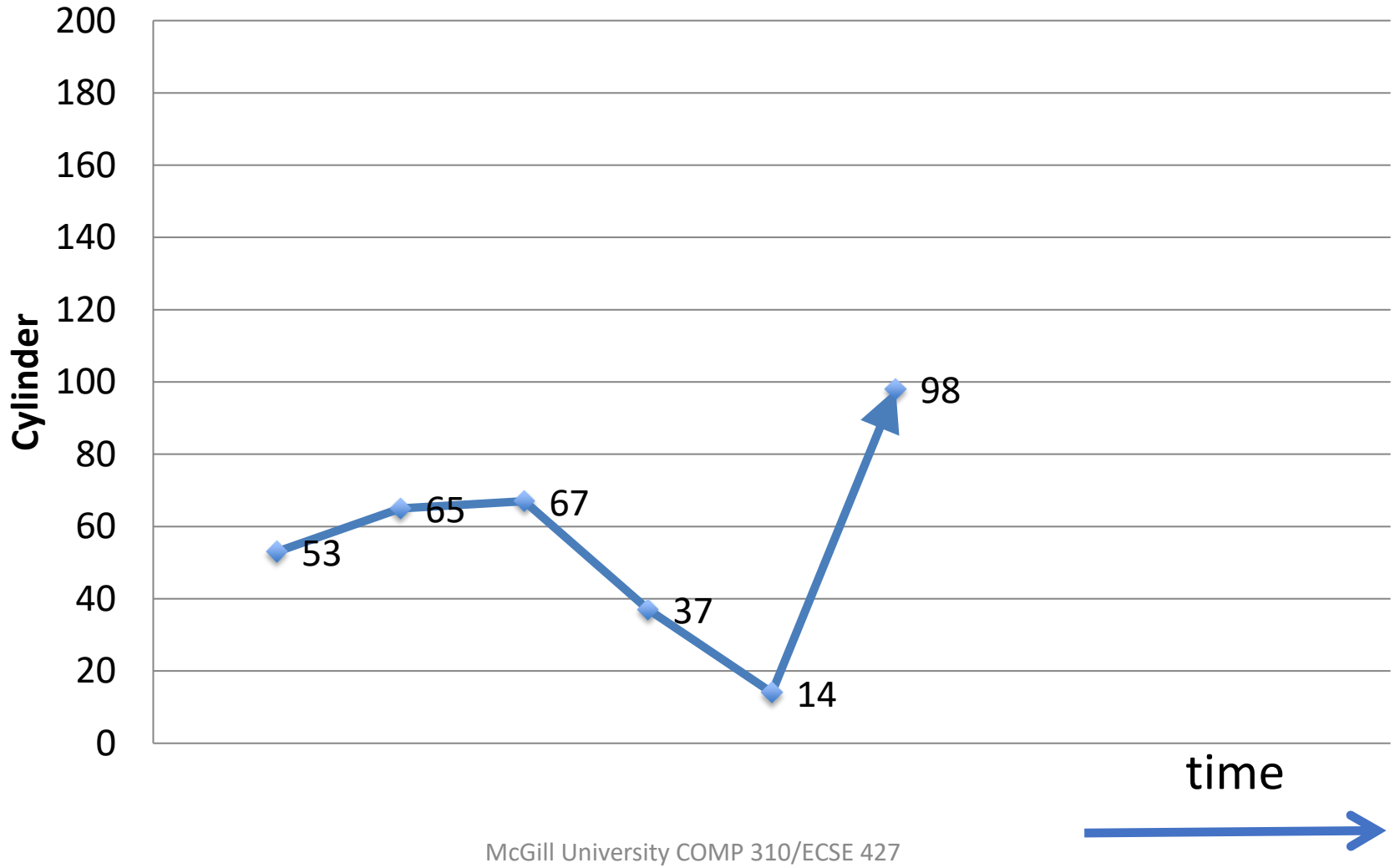


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

151

**SSTF**

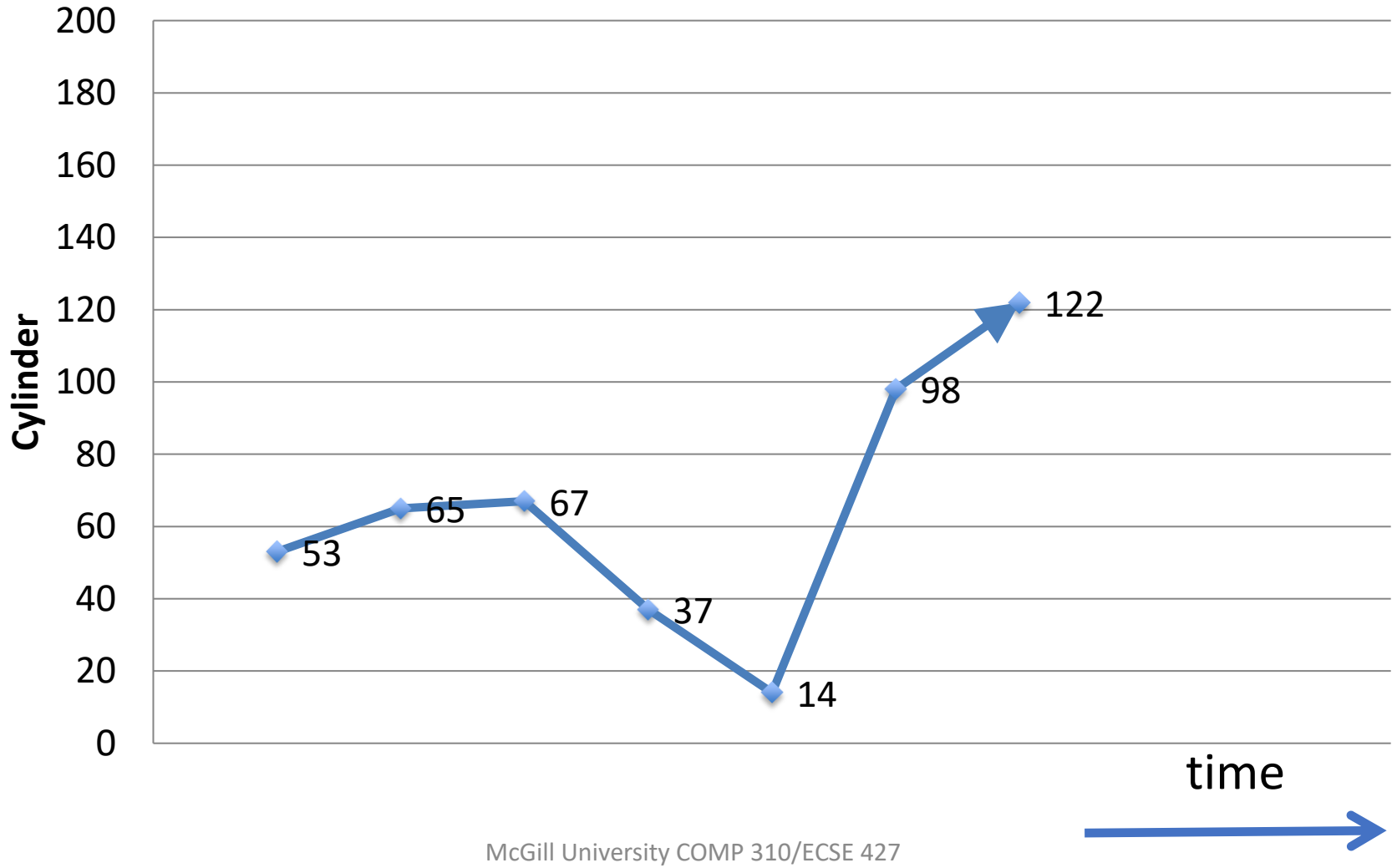


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

175

**SSTF**

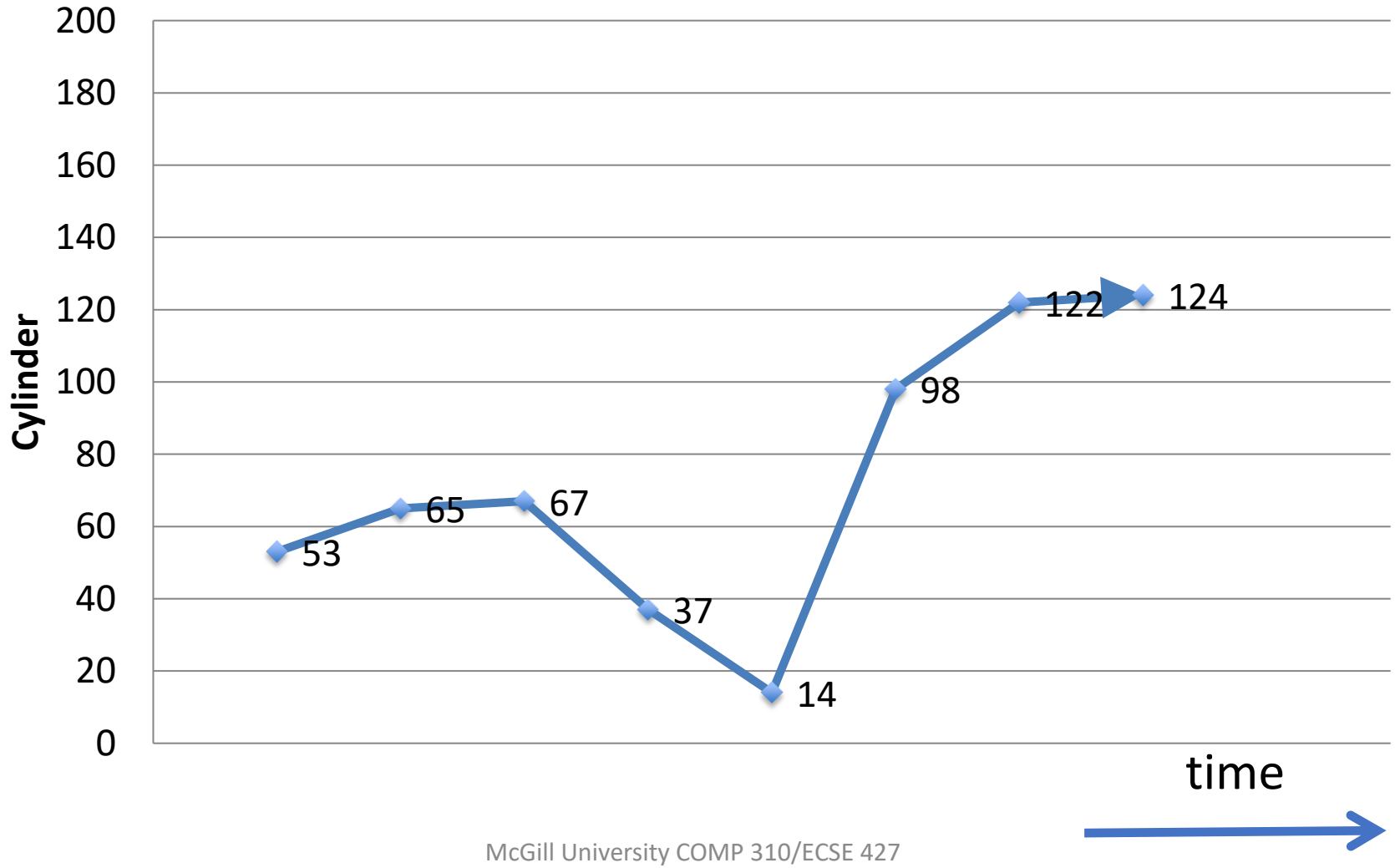


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

177

**SSTF**

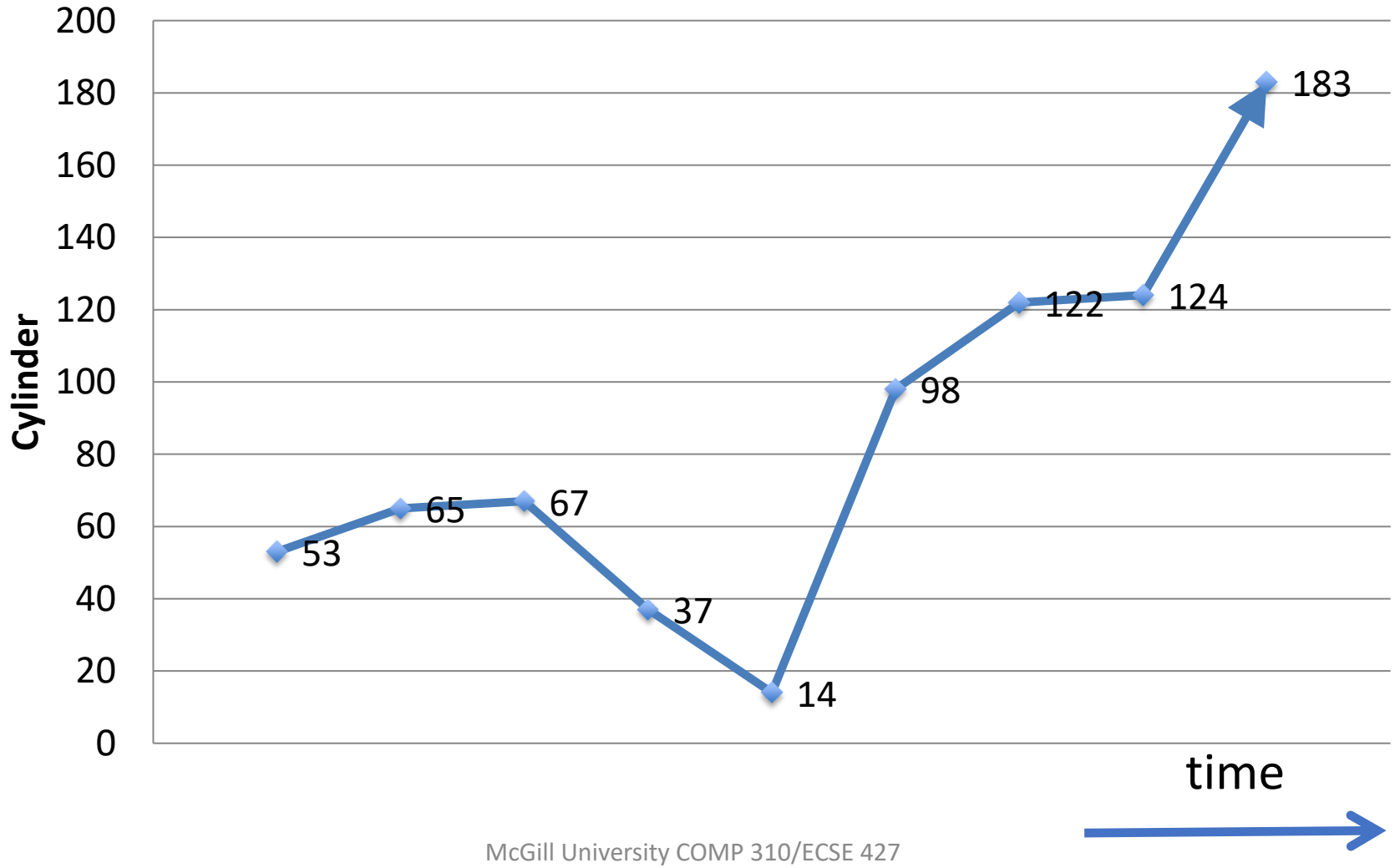


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

236

**SSTF**



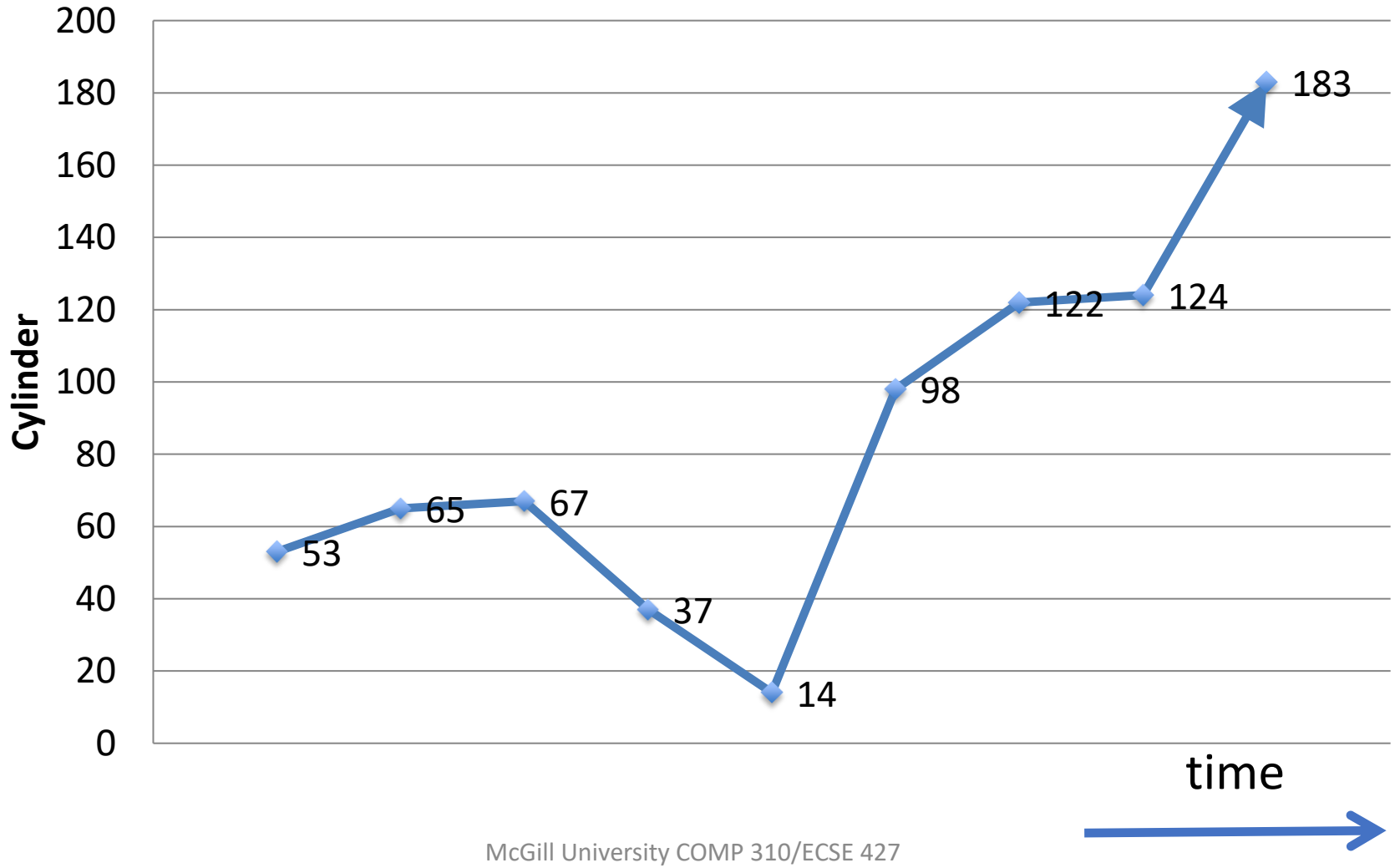
Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

236

**< FCFS (640)**

**SSTF**



# SSTF

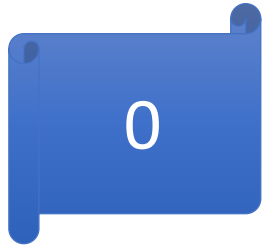
+ Very good seek times
☹ Subject to starvation
  - Request on inside or outside can get starved

# SCAN

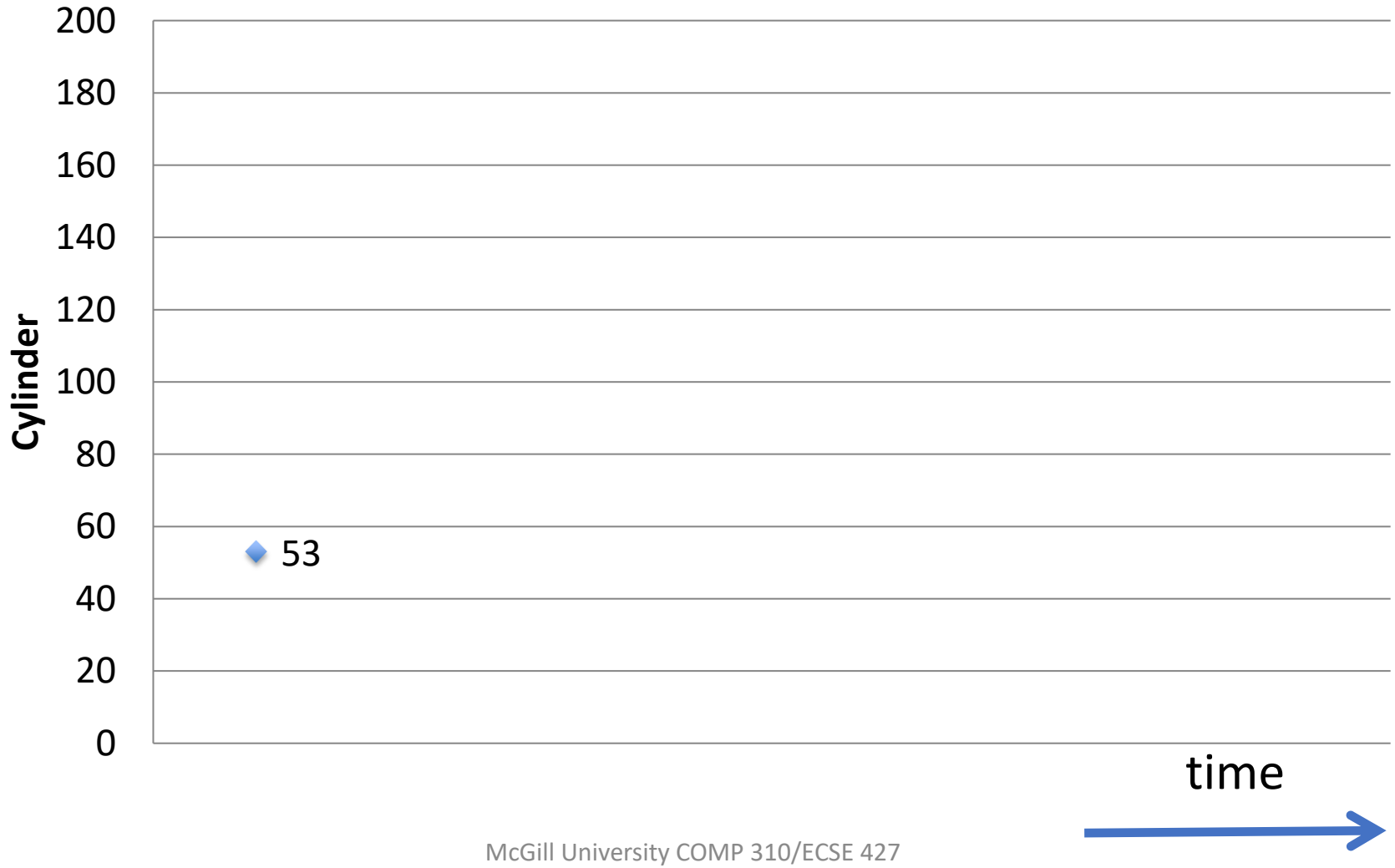
- Continue moving head in one direction
  - From 0 to MAX_CYL
  - Then, from MAX_CYL to 0
- Pick up requests as you move head

Head = 53, moving down

Queue = 98, 183, 37, 122, 14, 124, 65, 67




**SCAN**

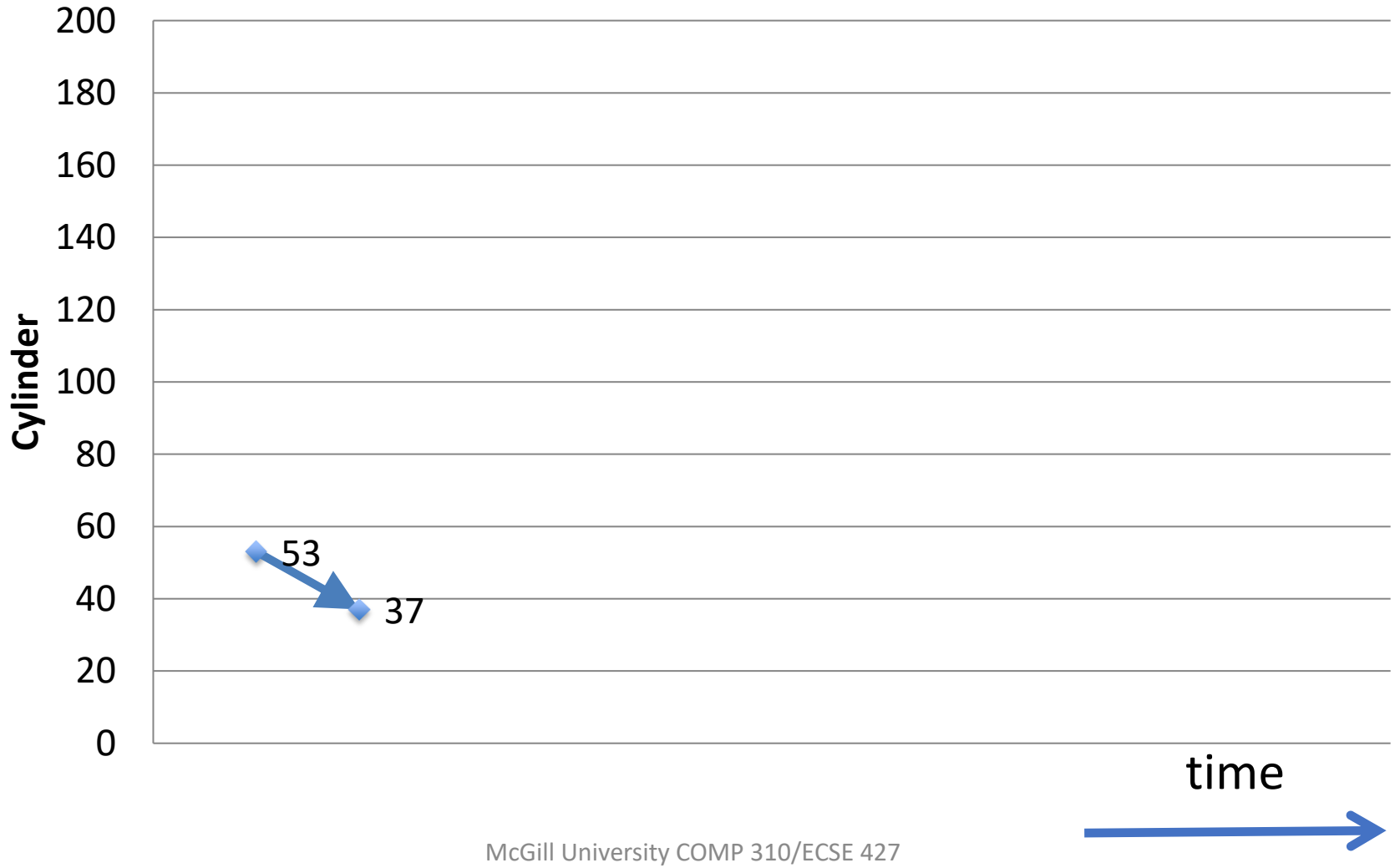


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

16

**SCAN**

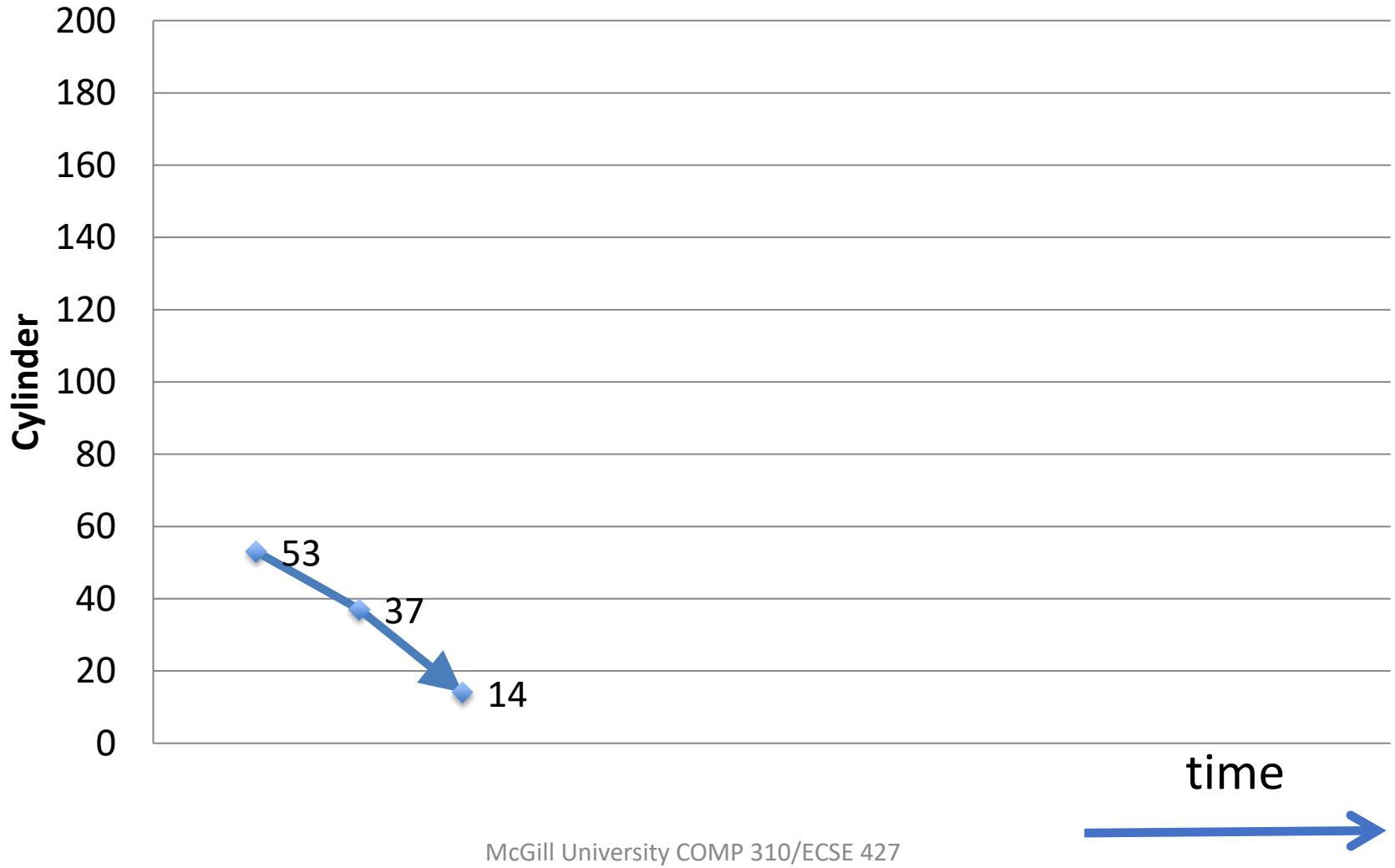


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

39

**SCAN**

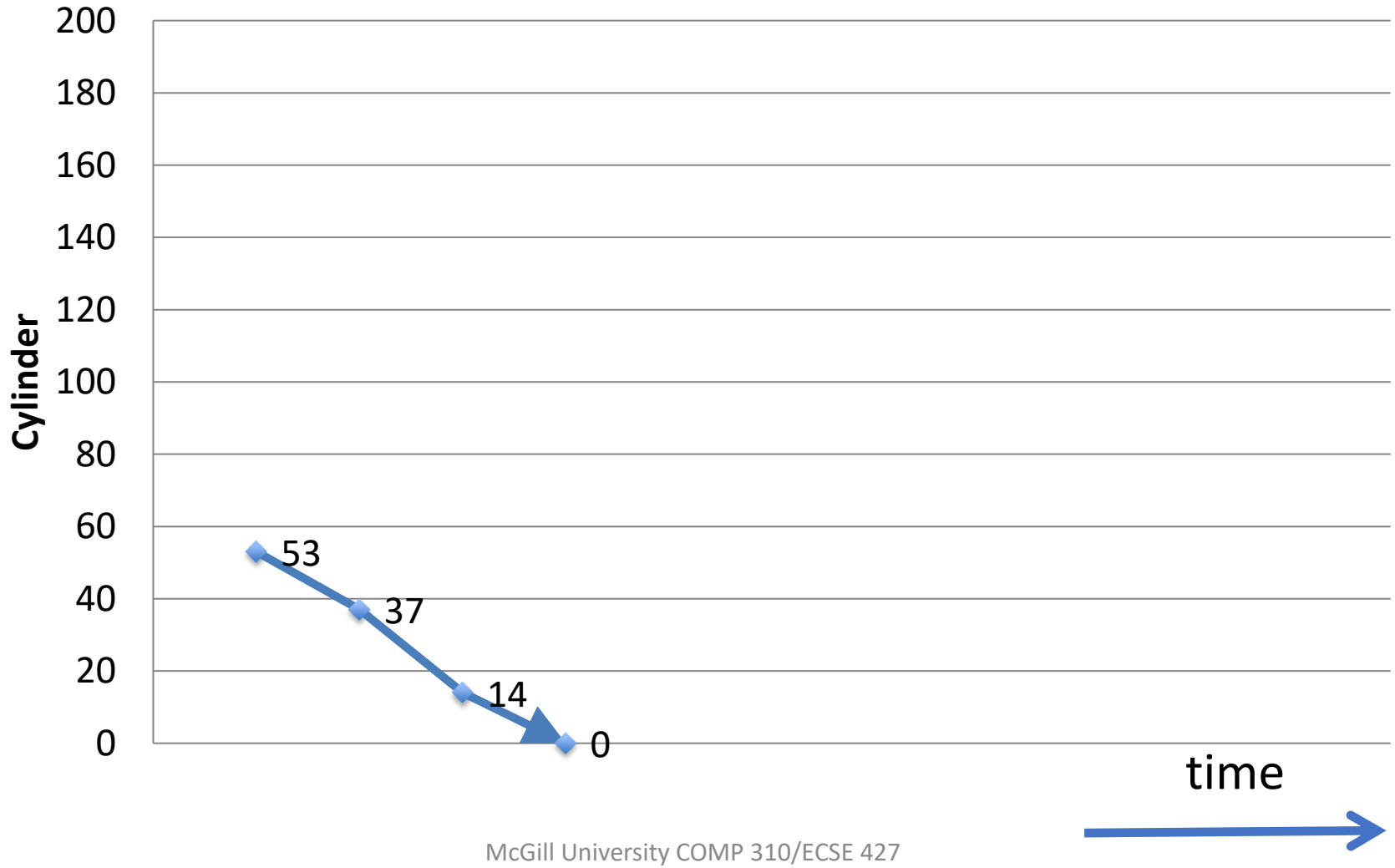


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

53

**SCAN**

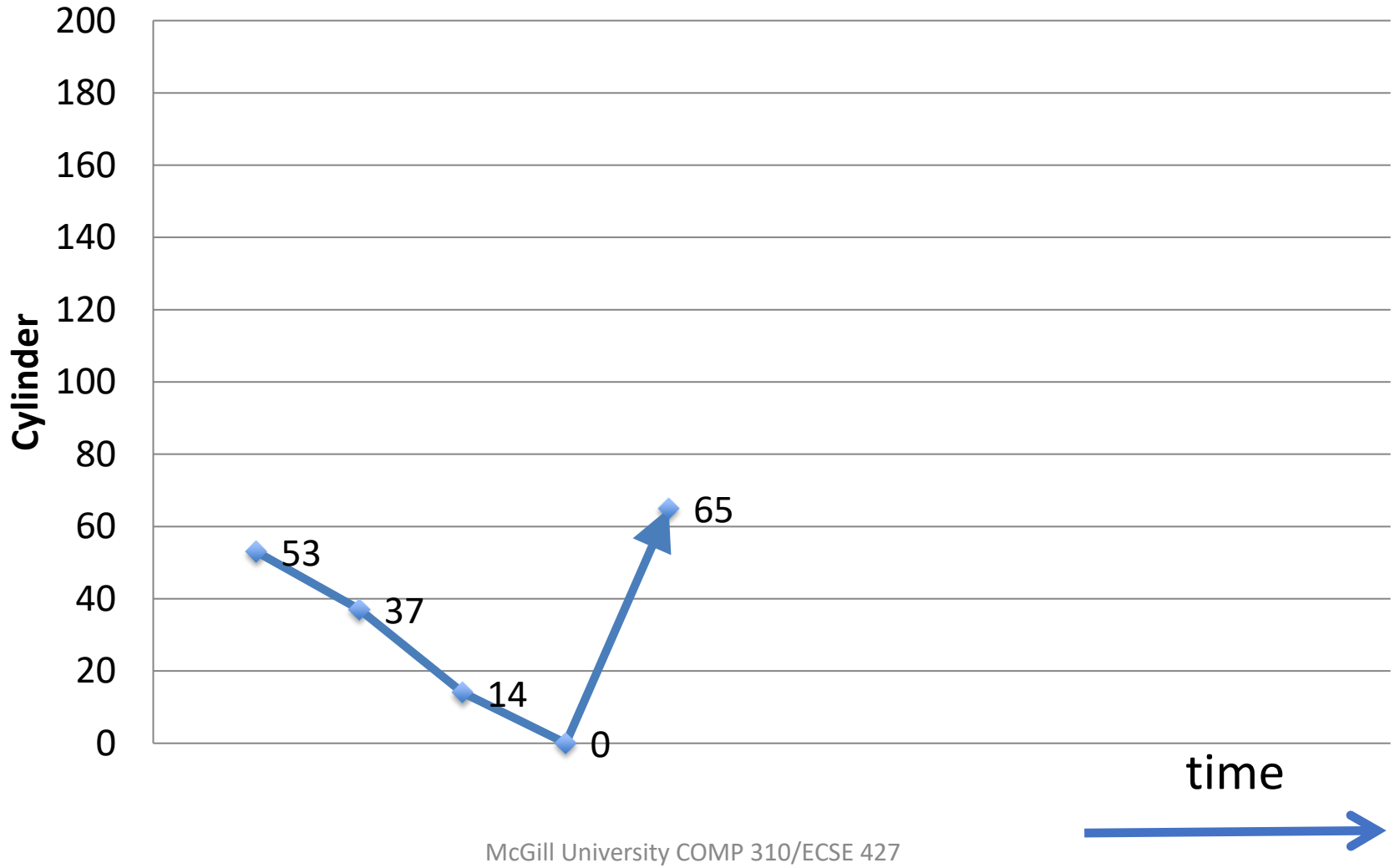


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

118

**SCAN**

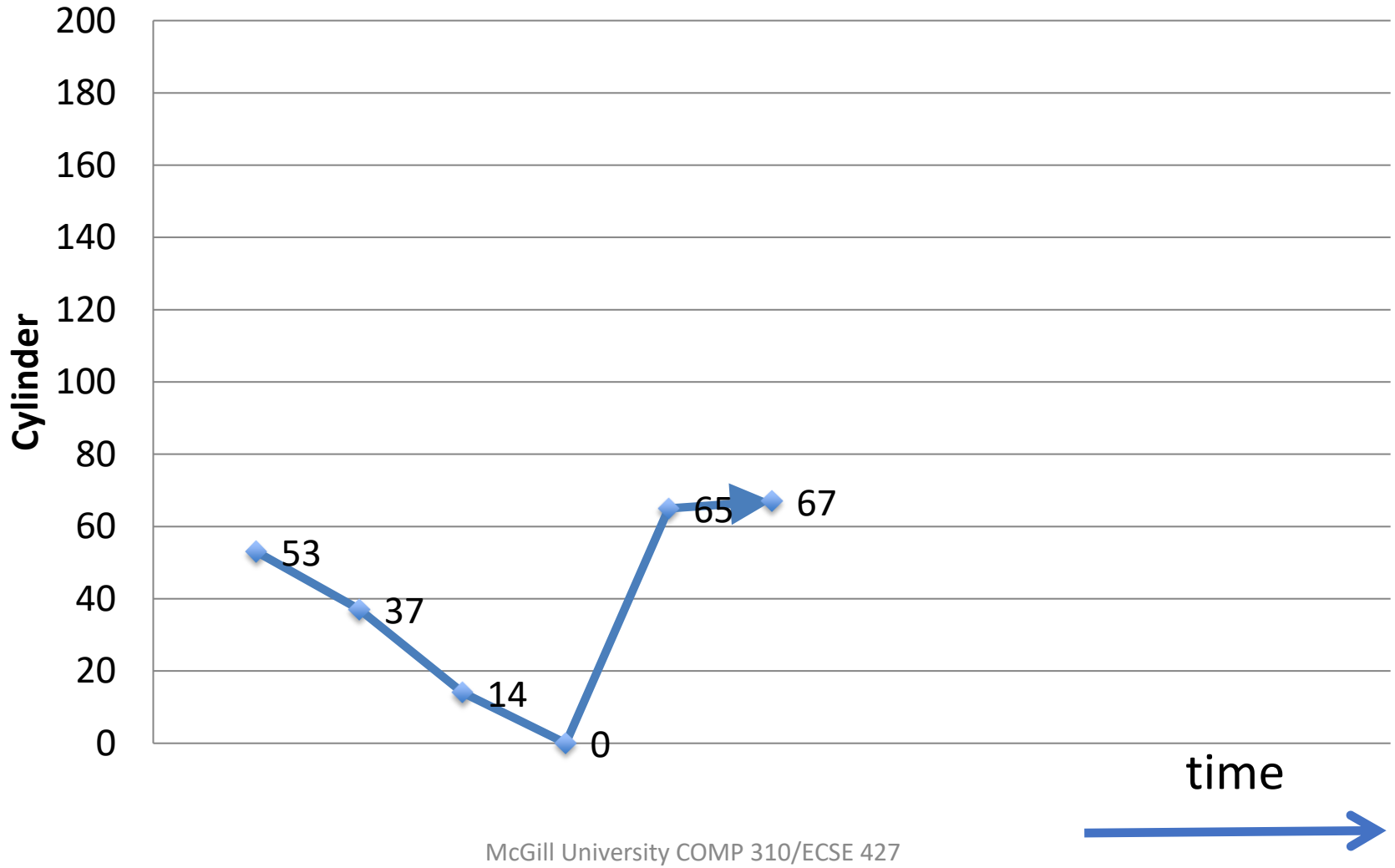


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

120

**SCAN**

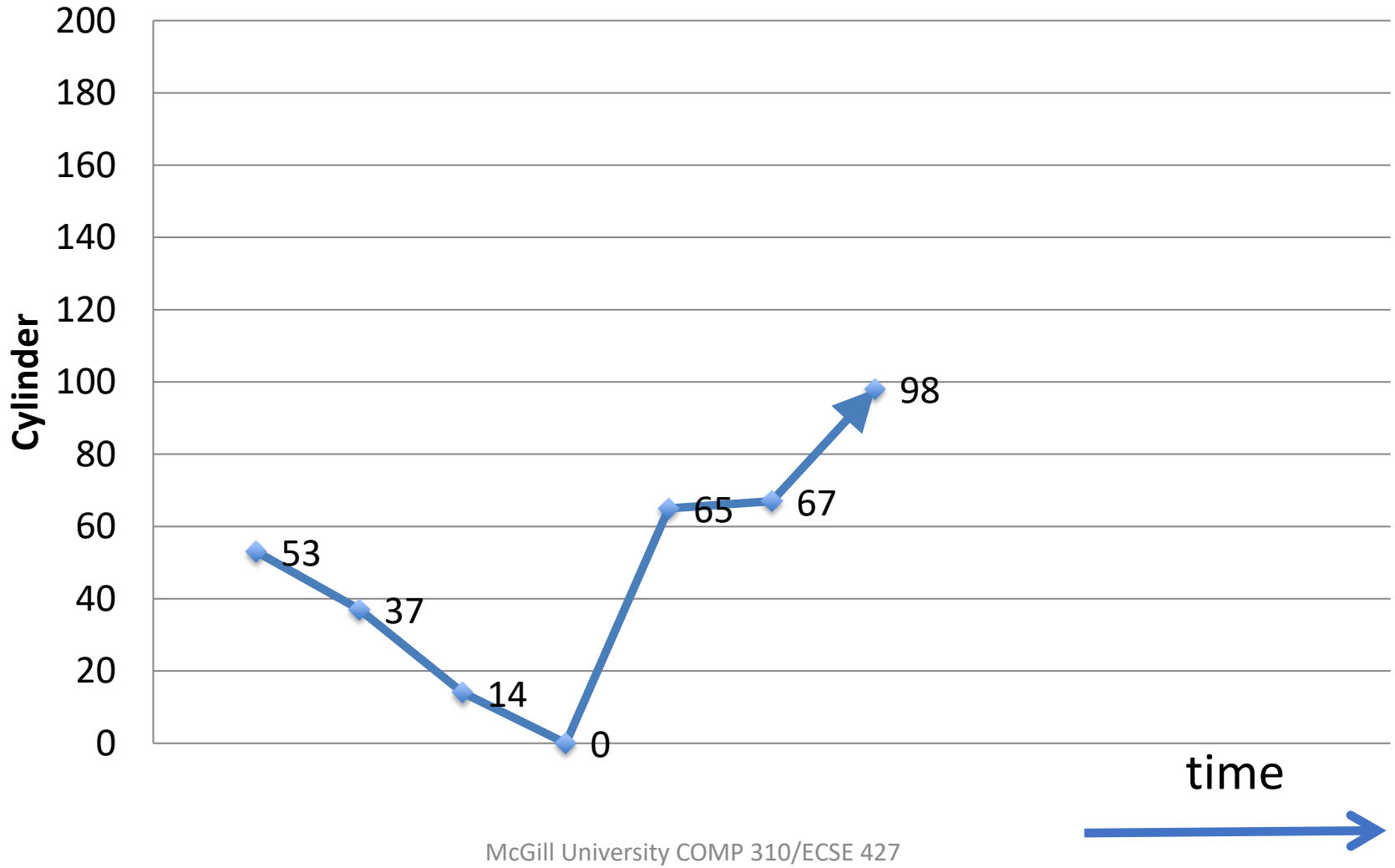


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

151

**SCAN**

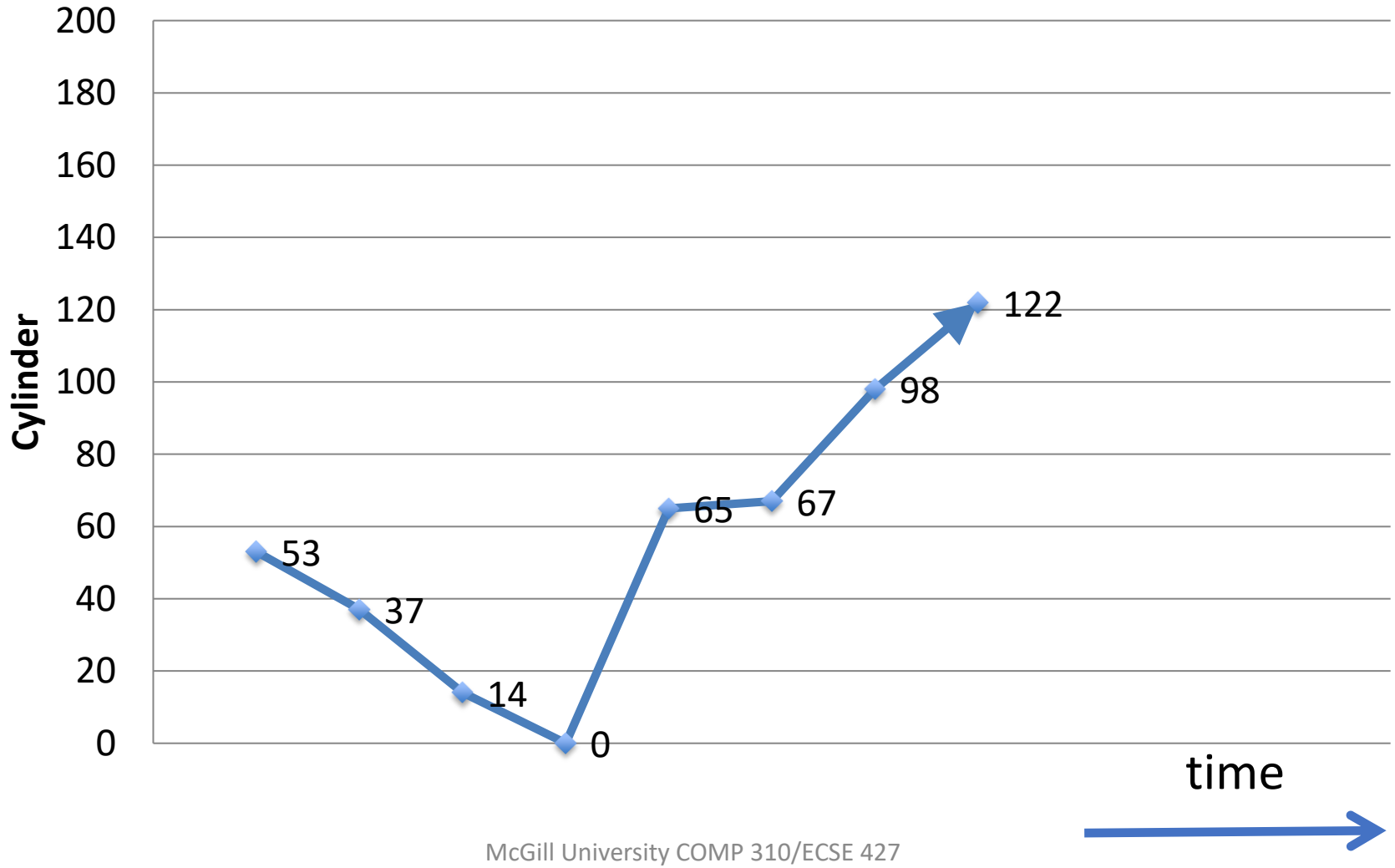


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

175

**SCAN**

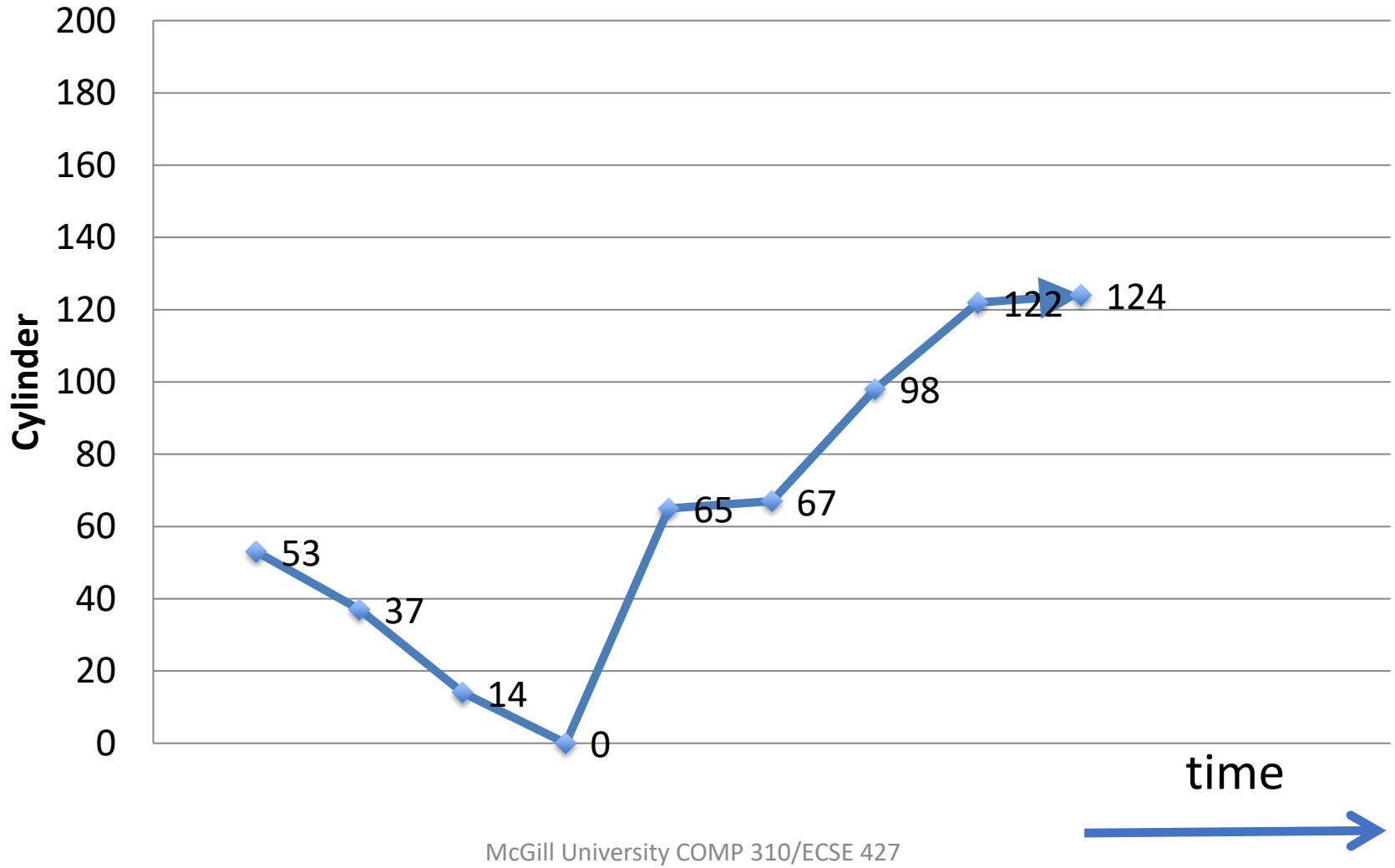


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

177

**SCAN**

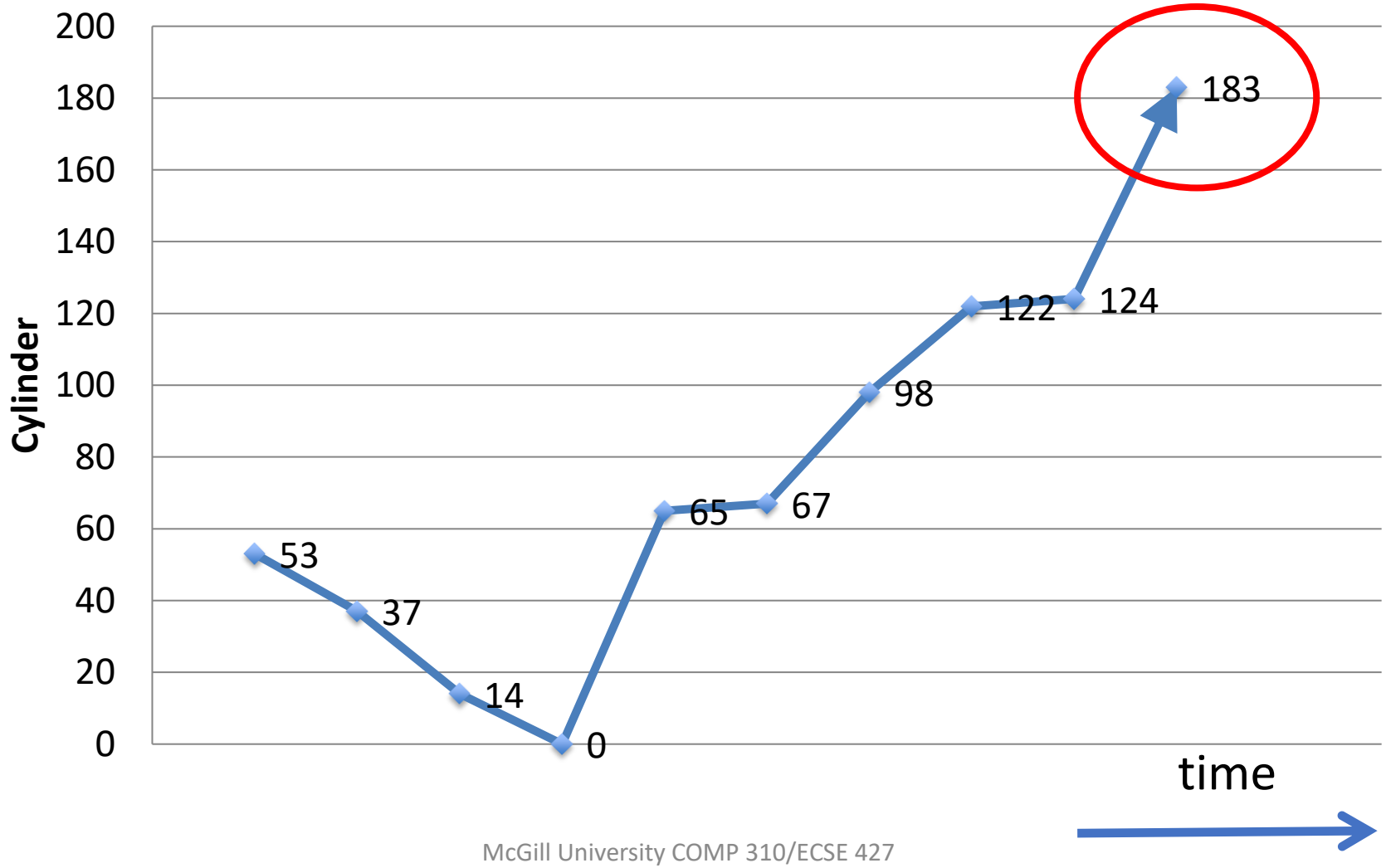


Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

236

**SCAN**




Here we do not go all the way to MAX_CYL, because this is the last request in the queue.

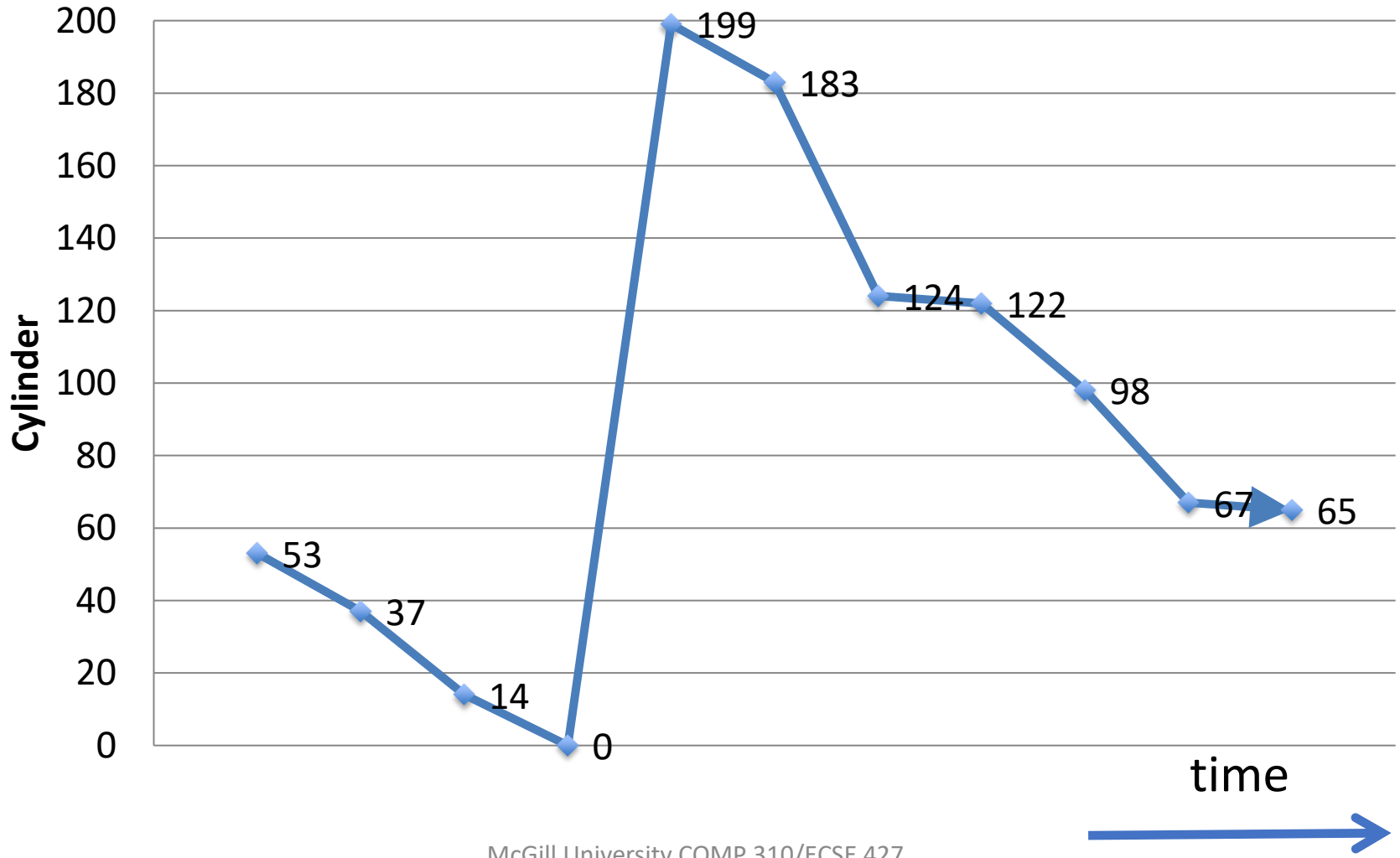
# C-SCAN

- Similar to SCAN
- Always move head
  - From MAX_CYL to 0; pick up requests as head moves
  - From 0 to MAX_CYL; no requests served
- C-SCAN can also be implemented in reverse
  - From MAX_CYL to 0; no requests served
  - From 0 to MAX_CYL; pick up requests as head moves

Head = 53, moving down
Queue = 98, 183, 37, 122, 14, 124, 65, 67

388

**C-SCAN**



# C-SCAN

☹ Number of cylinders slightly higher

\+ More uniform wait time

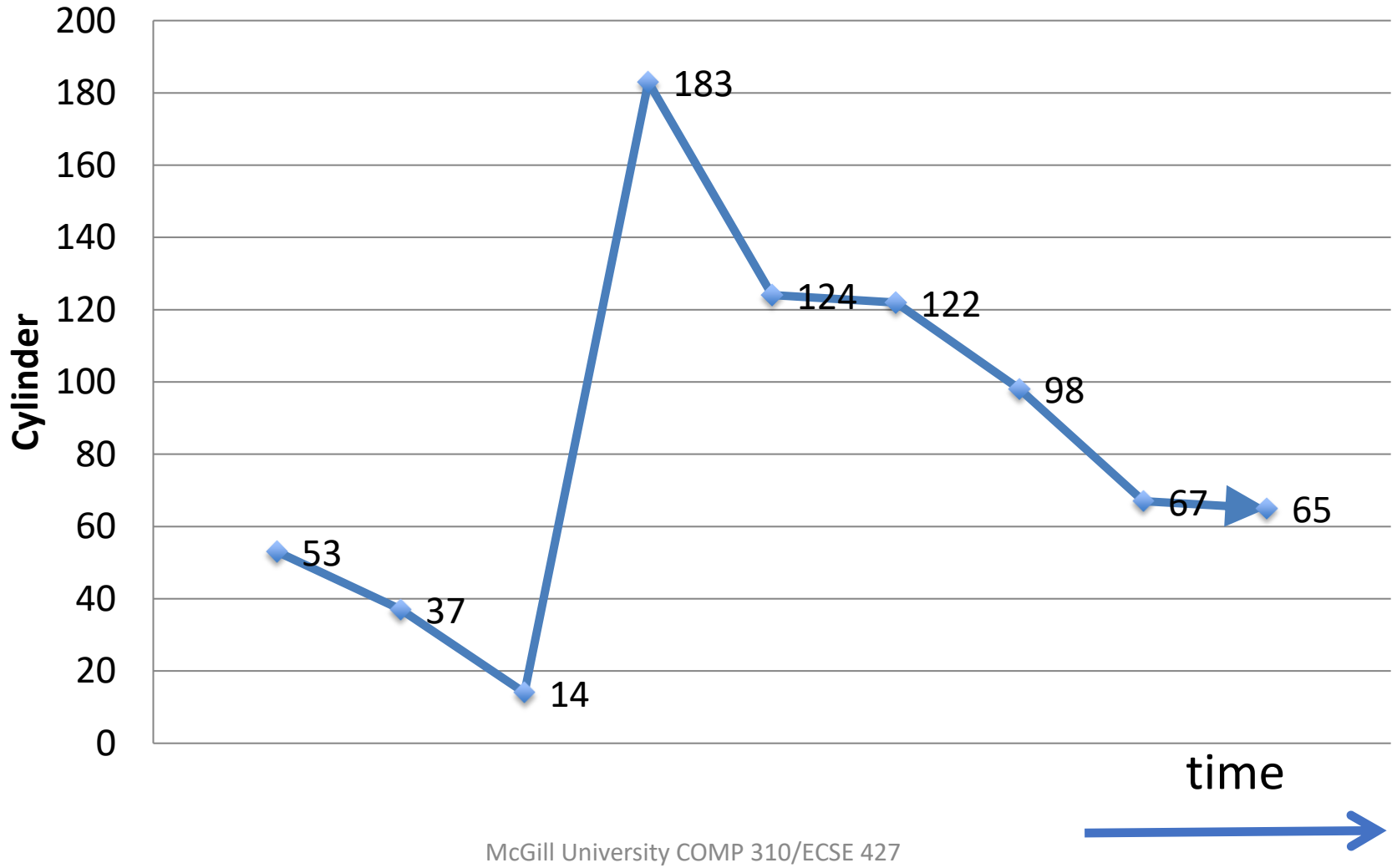
# C-LOOK

- Similar to C-SCAN
- Always move head
  - From MAX_CYL_IN_QUEUE to MIN_CYL; serve requests as head moves
  - From MIN_CYL to MAX_CYL_IN_QUEUE; no requests served
- C-LOOK can also be implemented in reverse

Head = 53, moving down
Queue = 98, 183, 37, 122, 14, 124, 65, 67

326

**C-LOOK**



# In Practice

- Some variation of C-LOOK is pretty common

# Optimize Disk Access

**Rule 4:**

**Avoid rotational latency**

- Clever disk allocation
- Locate consecutive blocks of file on consecutive sectors in a cylinder

# When does what work well?

- Low load: clever allocation
- High load: disk scheduling

# Why? – Under High Load

- Many scheduling opportunities
  - Many requests in the queue
- Allocation gets defeated
  - By interleaved requests for different files

# Why? – Under Low Load

- Not much scheduling opportunity
  - Not many requests in the queue
- Sequential user access -> sequential disk access
- Cache tends to reduce load

# Summary – Disk Management

- Disk characteristics
  - Access disk >> access memory
  - Seek > Rotational Latency > Transfer
- Optimizations
  - Cache
  - Read-ahead
  - Disk allocation
  - Disk scheduling

# Summary – Key Concepts

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

# 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).