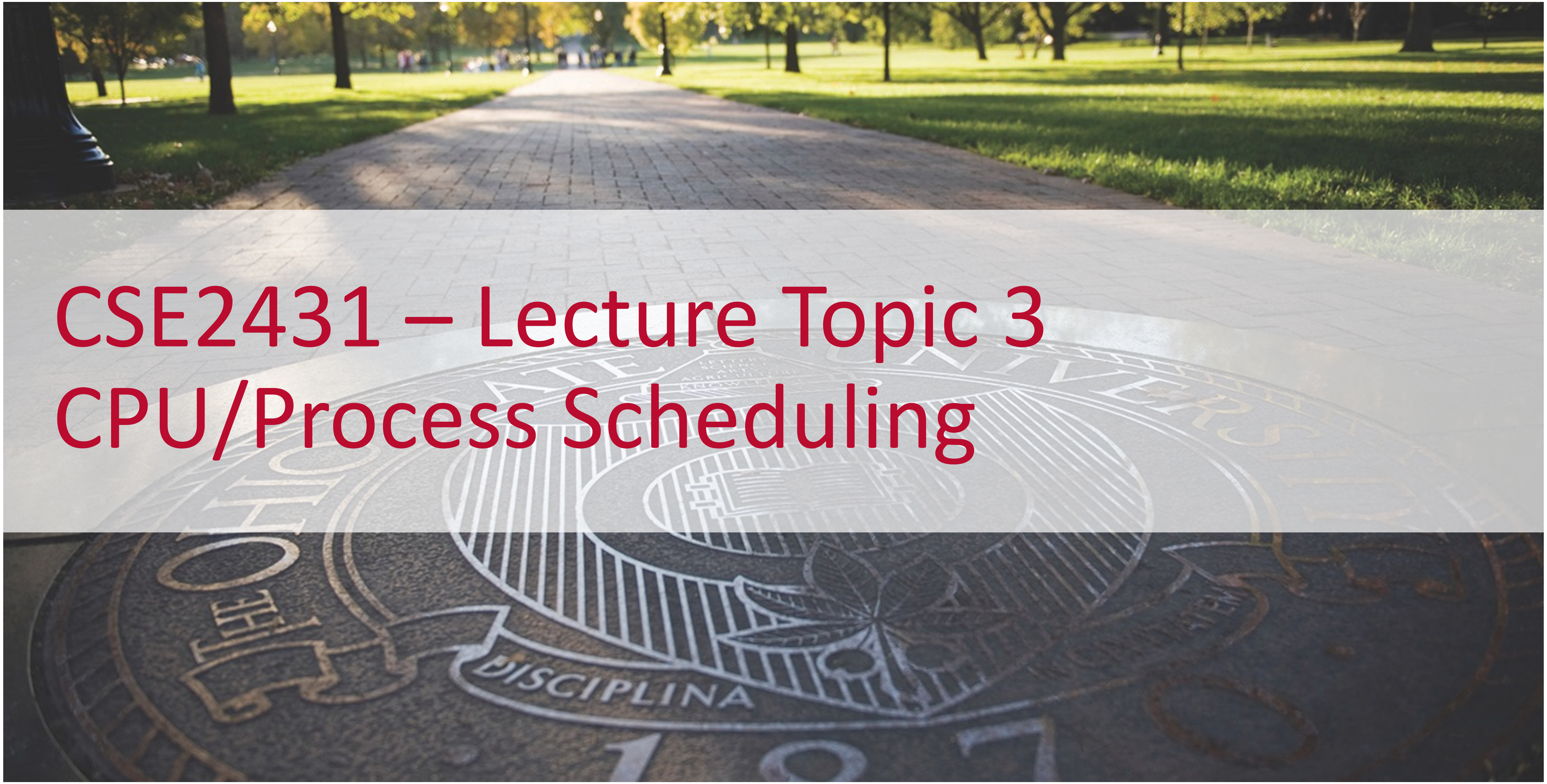


CSE2431 – Lecture Topic 3
CPU/Process Scheduling

THE OHIO STATE UNIVERSITY
COLLEGE OF ENGINEERING

CPU Scheduling

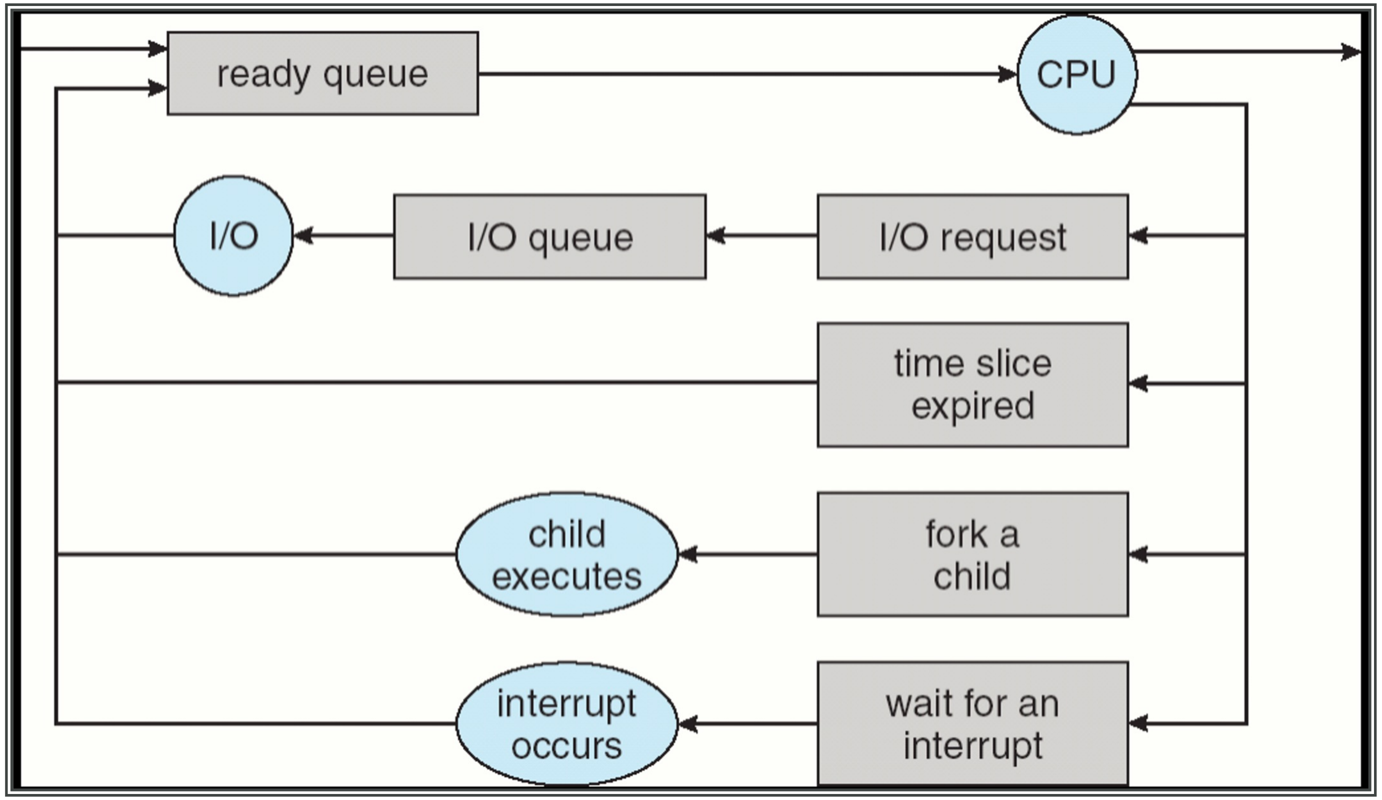
Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

Reading: Chapter 7 to 10 in required textbook

Previous Lecture

- OS has a list of processes to run
  - OS usually maintains them in a queue
  - New processes can be added to the queue at any time
- Computer is equipped with a number of CPUs
  - Let us assume there is only one CPU at this moment for simplicity
- OS needs to decide which process to run on each CPU



Previous Lecture

- **Long-term scheduler** (or job scheduler)
  - Selects which processes should be brought into the ready queue
  - Invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - Controls the degree of multiprogramming
  - Should balance different types of processes:
    - **I/O-bound process** spends more time doing I/O than computation; many short CPU bursts
    - **CPU-bound process** spends more time doing computation; few long CPU bursts
- **Short-term scheduler** (or CPU scheduler)
  - Selects which process should be executed next and allocates CPU
  - Invoked very frequently (milliseconds) $\Rightarrow$ must be fast!

Previous Lecture

- In **<u>system design</u>**, we separate mechanism and policy:
  - **Mechanism:** low-level implementation of a needed piece (i.e. the brake, the accelerator, the engine)
  - **Policy:** high-level intelligence about when and how to use a mechanism (i.e. when to brake a car and how hard – nowadays car like Tesla has automatic braking when the car comes close to another car)
- Similarly, for time-sharing:
  - Mechanism: context switching – a mechanism to stop/pause a process and resume it later
  - Policy: Which process is the chosen one?

Outline: CPU Scheduling (Policies)

- **Why Scheduling?**
- Basic concepts of Scheduling
- Scheduling Criteria and Scheduling Metrics
- Basic Scheduling Algorithm (FIFO)
- More advanced Scheduling Algorithms
- Thread Scheduling

CPU Scheduling

- To decide which process/thread/job should occupy a resource
- Jobs and Process: This lecture, we use “jobs” and “processes” interchangeably. Since a process which is in the schedule is called a “job”
- Which process/job needs to be schedules first? It depends!
    - Length of the process
    - Arrival time of process
    - Behaviors of process
        - Some processes runs by themselves; some interact with users
        - Some processes are CPU-intensive; some perform many I/Os

Scheduling dependability

- Examples?
  - A process that can run on itself: Any process/job that relates to arithmetic calculation (i.e. matrix multiplication)
  - A process that interact with users: Powerpoint slideshow (it has to wait and interact with the users' mouse input)
  - A process that is CPU-intensive: Matrix multiplication when doing inference in neural networks
  - A process that perform many I/Os: Games! (a lot of I/Os from the users)

Scheduling objectives

- Fairness: Ensure fairness among all the processes
- Priority: Prioritized process needs to get schedule first
- Efficiency: Make best use of the resources
- Encourage good behavior: No malicious process allowed
- Support heavy loads: Schedule needs to degrade gracefully (i.e. a heavy job needs to gradually replaced by a small-load job)
- Adapt to different environments: schedule needs to be interactive, real-time, multimedia

Scheduling depends on system type!

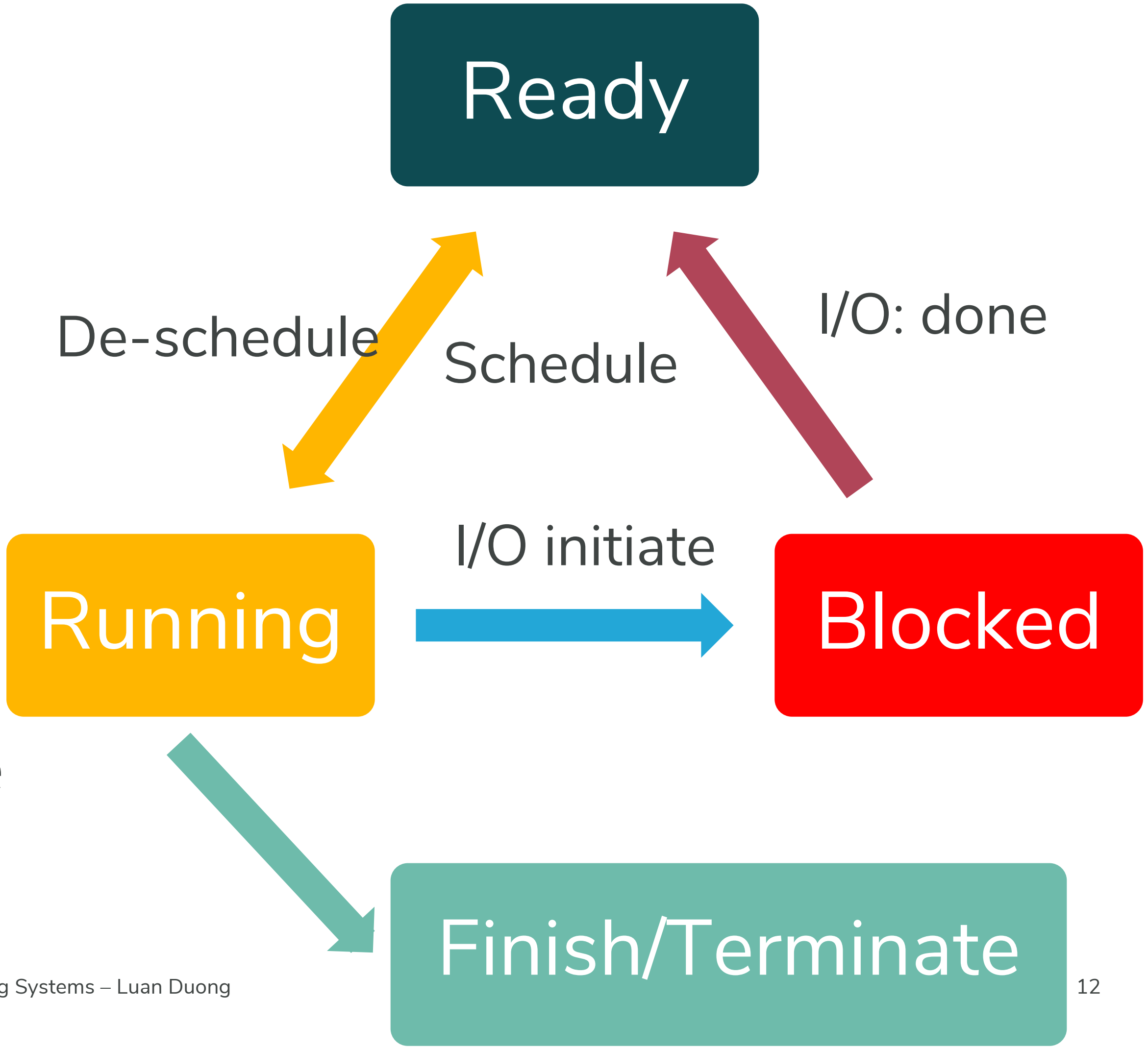
- For all types of systems:
  - Ensure fairness, policy enforcement, resource balancing
- Batch systems
  - Maximize throughput and CPU utilization, minimize turnaround time
- Interactive systems:
  - Minimize response time, achieve best proportionality
- Real-time systems:
  - Meet deadlines, and able to predict what comes next.

Scheduling metrics

- **Fairness:** No starvation
- **Throughput:** Number of processes completed per unit of time
- **Turnaround time** (also called elapsed time): Amount of time to complete a certain process from its start $\left(T_{turnaround} = T_{completion} - T_{arrival}\right)$
  - $\mathrm{T}_{\mathrm{arrival}}$ is the **earliest time** that a particular process arrives (it can arrive much earlier than getting scheduled)
- **Waiting time:** Amount of time process has been waiting in ready queue
- **Response Time:** Amount of time from request submission until first response
- **Proportionality:** Meet users' expectations (i.e. in user-i/o intensive system)
- **Meeting deadlines:** Ensure all jobs can be scheduled (in real-time)

Preemptive vs. Non-preemptive

- <u>**Non-preemptive**</u> scheduling:
  - The running process keeps the CPU until it voluntarily gives up the CPU (when? Process exits, Switches to waiting state) (No <u>**de-schedule**</u>)
- <u>**Preemptive**</u> scheduling:
  - The running process can be interrupted and must release the CPU (it is forced to give up the CPU)



Process Behavior

- **I/O Bound:** Does too much I/O to keep CPU busy
- **CPU-Bound:** Does too much computation to keep I/O busy
- Process Mix:
  - Scheduling should balance between I/O bound and CPU-bound processes
  - Ideally, we would run all equipment at 100% utilization, but that would not necessarily be good for response time

Outline: CPU Scheduling (Policies)

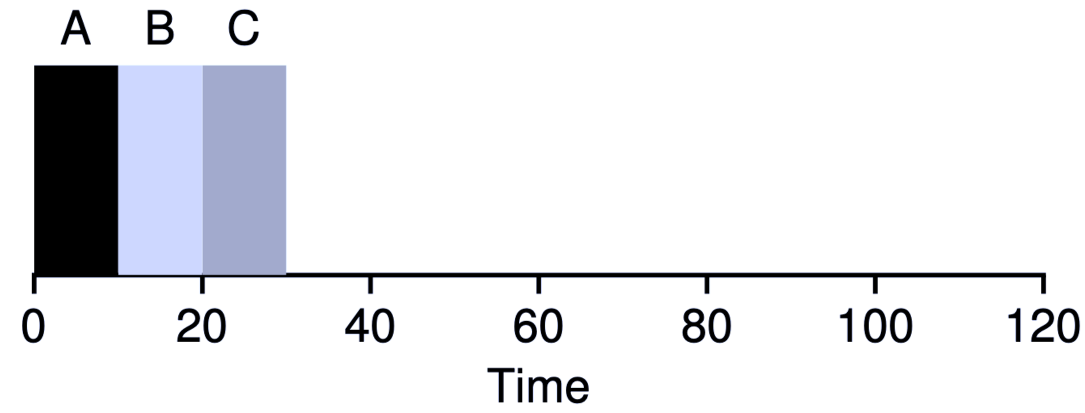
- Why Scheduling?
- Basic concepts of Scheduling
- Scheduling Criteria and Scheduling Metrics
- **Basic Scheduling Algorithm (FIFO)**
- Other Scheduling Algorithms
- Thread Scheduling

Basic Scheduling Algorithms (FIFO)

- What is **FIFO**? **First In First Out**. Sometimes also called "FCFS" algorithm: **First Come First Serve**.
- **Any real-life analogy to FCFS/FIFO scheduling?**
- It is used in batch systems.
- Is it preemptive or non-preemptive? Non-pre-emptive, no time-sharing
- Implementation: Queue
- Performance metrics: Turnaround time (in time units)
- Given parameters: Burst time (in time units), Arrival time, and Order

FIFO: Example 1

- Three processes: A, B, C. Supposed they arrive at roughly **at the same time.**
- Assume: A arrives slightly earlier than B. B slightly earlier than C.
- Each process ru



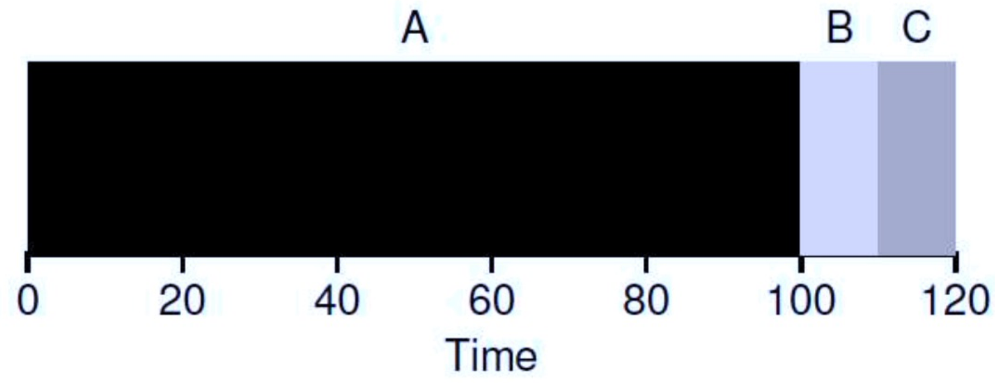
- What is the average turnaround time?

$$\frac{[(10-0)+(20-0)+(30-0)]}{3} = 20(s)$$

- Do you think of any situation when FIFO becomes worse?

FIFO: Example 2

- Three processes: A, B, C. Supposed they arrive at roughly at the same time.
- Assume: A arrives slightly earlier than B. B slightly earlier than C.
- Suppose: A runs



- What is the average turnaround time?

$$\frac{[(100-0)+(110-0)+(120-0)]}{3} = 110(s)$$

FIFO Problems:

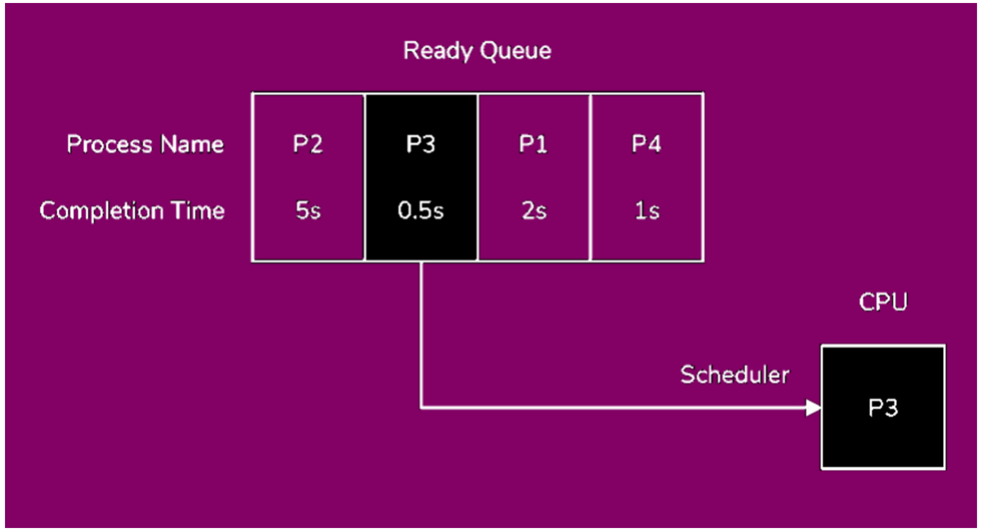
- Since FIFO is non-preemptive, it does not allow any process interrupting the running process.
- Thus, short-term processes have to wait, and can be blocked by long ones → **Convoy Problem**! I am sure you have met this problem in daily life

Why Convoy effects?

- Consider 100 I/O-bound processes, 1 CPU-bound job in system.
- I/O-bound processes pass quickly through the ready queue and suspend themselves waiting for I/O
- A CPU-bound process arrives at head of queue, executes until completion.
- I/O-bound processes rejoin ready queue, however, it needs to wait for CPU-bound process to release CPU.
- I/O device idle until the CPU-bound process completes.
- In general, a convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all the other processes. This causes poor utilization of system resources.

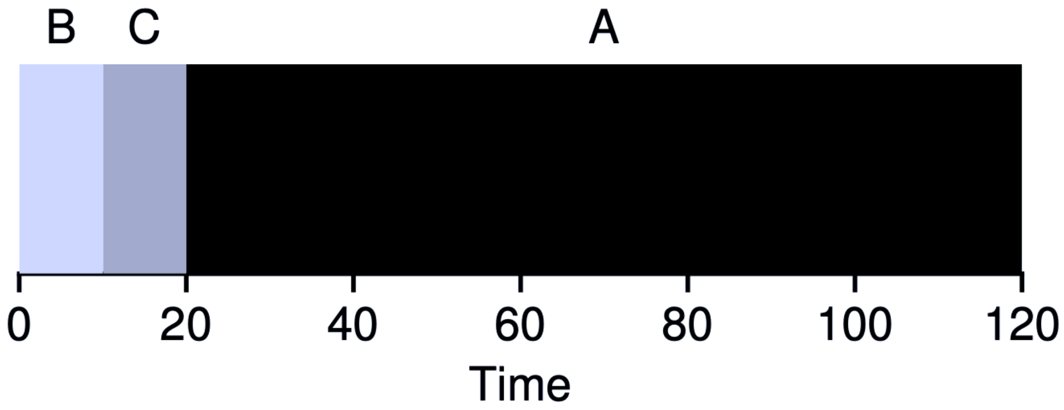
How to improve? Shortest Job First (SJF)

- Always schedule the shortest job and run it until completion
  - Assume we know the length of each process
  - Assume they all come at roughly the same time
  - Assume there are no I/O operations
- With these assumptions: it can be proved that SJF achieves lowest average turnaround time given these assumptions.



Shortest Job First – Example

- Three processes: A, B, C: arrived roughly the same time.
- A runs for 100 seconds, B and C run for 10 seconds.



- What is the average turnaround time now? (Reduced from 110s → 50s)

$$\frac{[(10-0)+(20-0)+(120-0)]}{3} = 50(s)$$

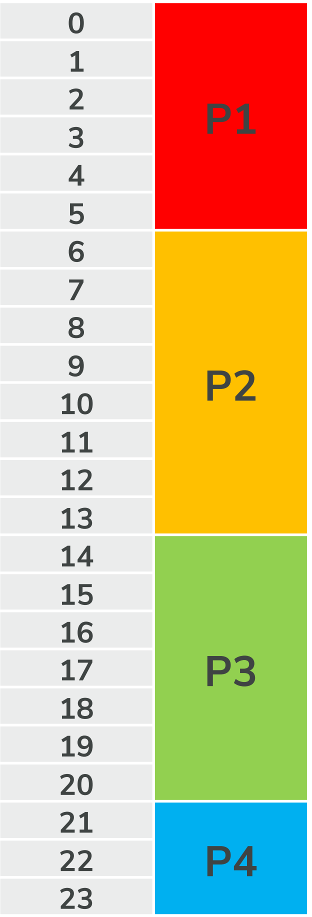
Shortest Job First – Example 2

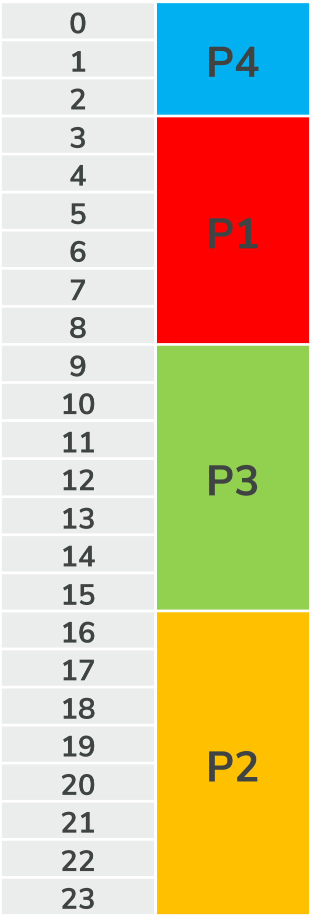
- FIFO compared to SJF– Non-preemptive case

Job	Duration	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0

Shortest Job First – Example 2

- FIFO compared to SJF: Non-preemptive case





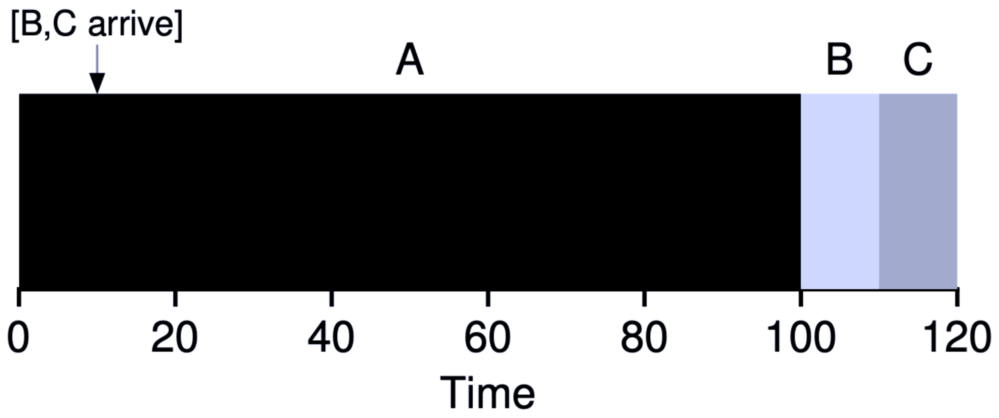
Job	Duration	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0

$$T_{turnaround}^{FIFO} = \frac{[(6-0)+(14-0)+(21-0)+(24-0)]}{4} = \mathbf{16.25}\ (s)$$

$$T_{turnaround}^{SJF} = \frac{[(3-0)+(9-0)+(16-0)+(24-0)]}{4} = \mathbf{13}(s)$$

Shortest Job First – 'Counter'-Example

- Three processes: A arrives at t = 0; B and C arrive at time t = 10. Now what will happen with SJF?
- Suppose: A runs for 100 seconds, B and C run for 10 seconds.



- What is the average turnaround time now?

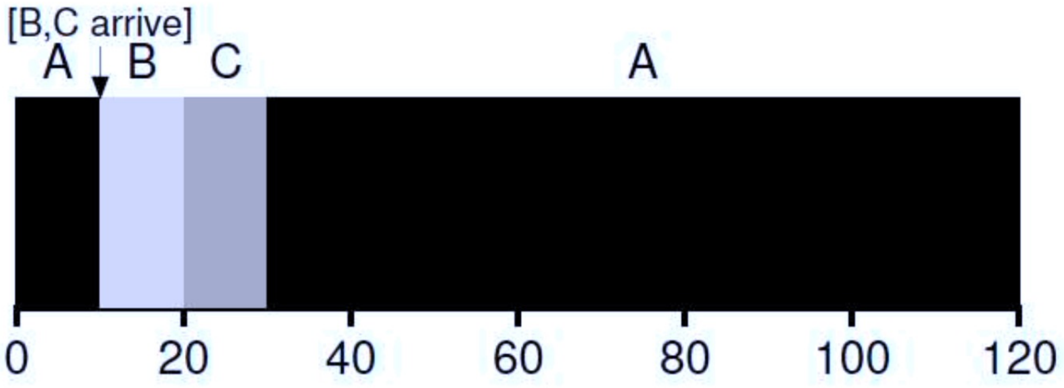
$$\frac{[(100-0)+(110-10)+(120-10)]}{3} = 103.333(s)$$

Further Improvements?
Shortest Time-to-Completion First (STCF)

- Always run the process with the lowest time-to-completion.
- When a new job comes or the current job completes
  - Compute the time-to-completion of all jobs
  - Pause the current job if necessary
  - Switch to the job with lowest time-to-completion
- Introduction of **preemptive scheduling**!
  - FIFO and SJF are **non-preemptive schedulers**
  - STCF is a **preemptive scheduler**

STCF - Example

- Three processes: A arrives at t = 0; B and C arrive at time t = 10. Now what will happen with STCF?
- Suppose: A runs for 100 seconds, B and C run for 10 seconds.



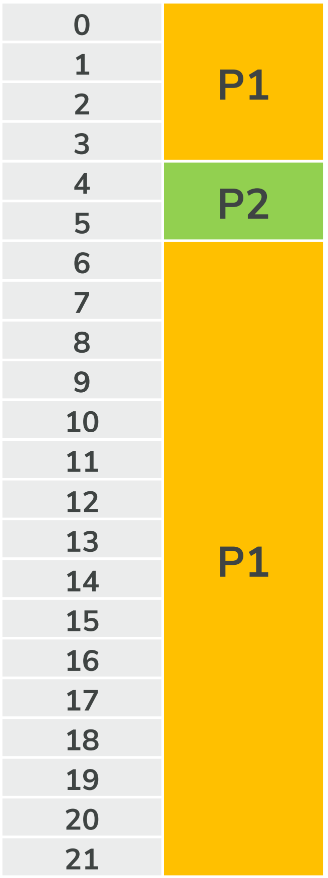
- What is the turnaround time now?

$$\frac{[(120-0)+(20-10)+(30-10)]}{3} = 50(s)$$

- STCF is provably optimal (can achieve lowest turnaround time)

SJF vs. STCF : Comparison

- Let's try this: SJF vs. STCF



Job	Duration	Order	Arrival Time
P1	20	1	0
P2	2	2	4

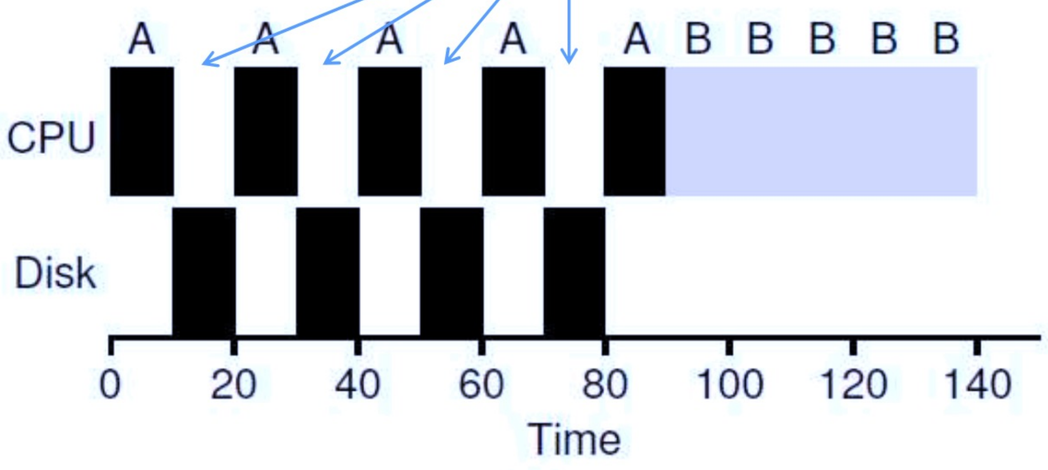
$$\boldsymbol{T^{\text{SJF}}_{turnaround}} = \frac{[(20-0)+(22-4)]}{2} = \mathbf{19}(\boldsymbol{s})$$

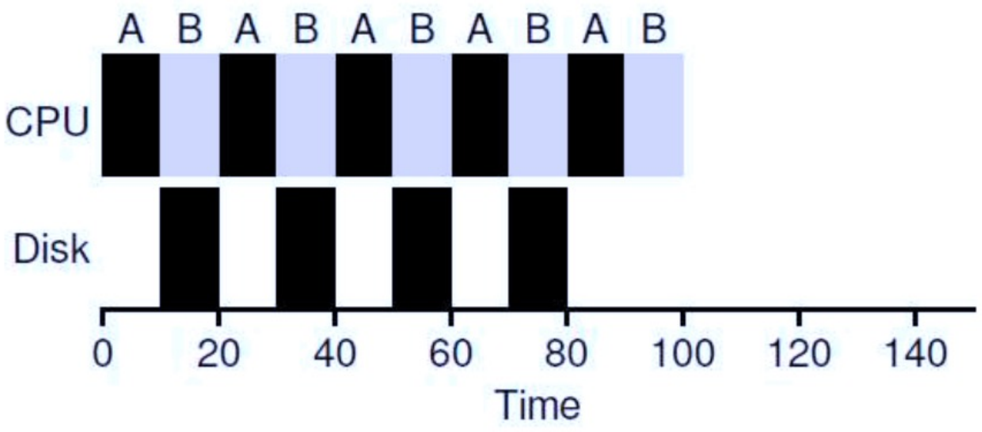
$$\boldsymbol{T^{\text{STCF}}_{turnaround}} = \frac{[(6-4)+(22-0)]}{2} = \mathbf{12}(\boldsymbol{s})$$

What if there is I/O?

- So far we have not considered I/O
- What is special with I/O?
  - Sometimes a process needs to wait for an I/O to complete (i.e. read a disk)
  - I/O is usually slow (a disk I/O can take tens of milliseconds – assume it's the HDD)
- If a process is waiting for an I/O, it's better to give CPU to another process
- A solution: Treat each sub-job of A as an independent job

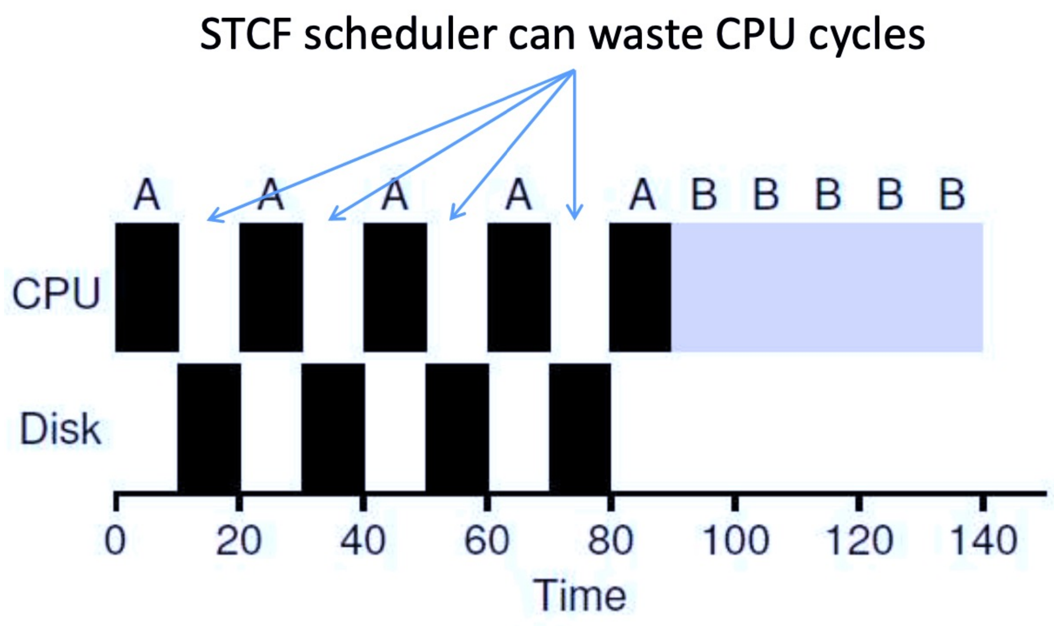
STCF scheduler can waste CPU cycles

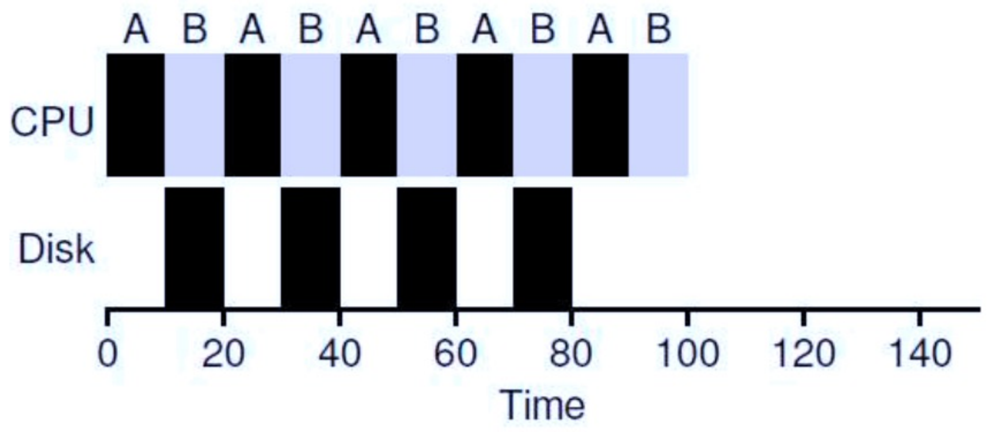




What if there is I/O?

- A's subjobs got scheduled first.
- After completion, B gets scheduled.
- However, when A's disk I/O completes, A submits the second 10ms sub-job
- The new sub-job preempts B and get scheduled...





What if there is I/O?

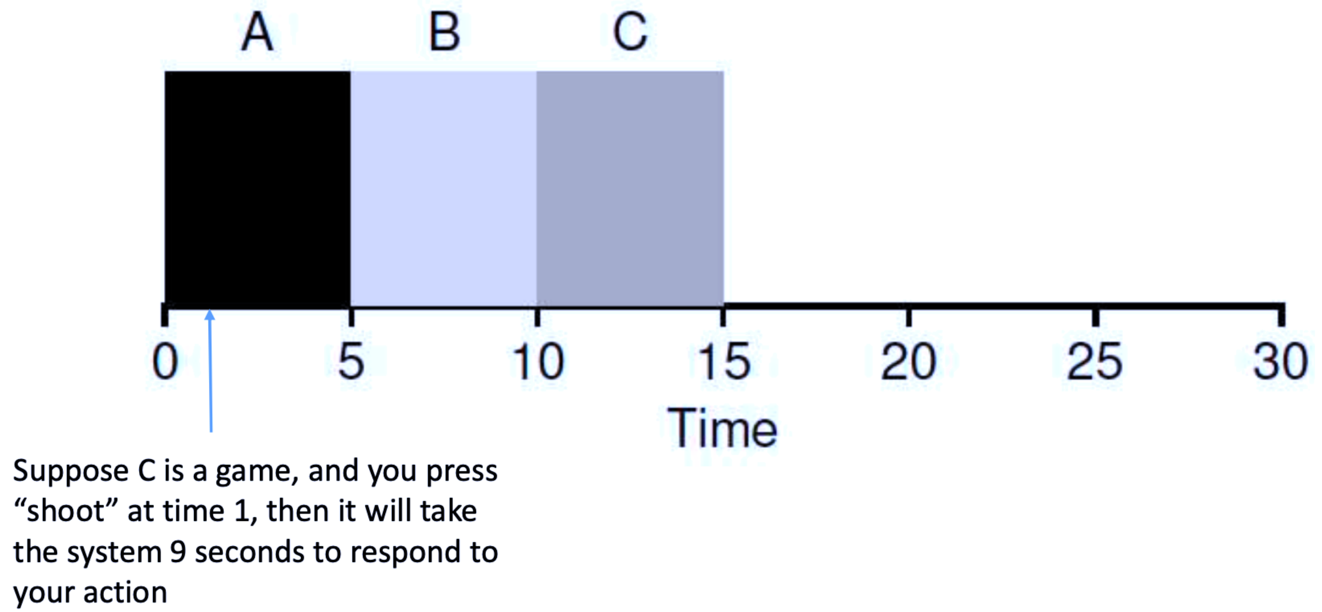
- Have we really solved the problem? No!
- Fairness: A long job can be blocked for long
- Length of a job is usually unknown when it arrives (Remember we have this assumption?)
- Some applications prefer other metrics

A new metric: Response time

- For **computation jobs**, minimizing completion time makes sense
  - Examples: matrix multiplication, weather forecast, ……
- For **interactive jobs**, minimizing completion time is not much useful
  - Examples: Linux terminal, MS Word and PowerPoint, PC games, …
  - They don’t “complete” until users tell them to.
  - Users care more about how quickly the application can react to a user event
- Response time:
  - Obviously, a job cannot respond if it is not scheduled.
  - Text-book definition: $\mathrm{T}_{\mathrm{response}} = \mathrm{T}_{\mathrm{first-run}} - \mathrm{T}_{\mathrm{arrival}}$

A new metric: Response time

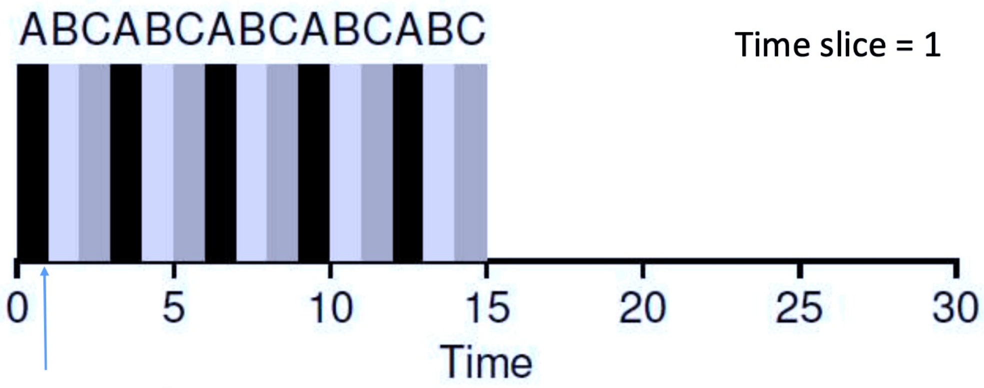
- Now look at this FIFO/SJF schedule



- Suppose C is a game, then if you press "shoot" at time 1 (I/O), C has to wait until A finishes and B finishes, then it will take system 9 seconds to respond to your action!

Round Robin

- Fact: User events can arrive at any time.
- Not a good idea to run a job to completion
- Basic idea: run a job for a time slice, then switch to another job
  - Round Robin is also called Time-slicing
- Example?

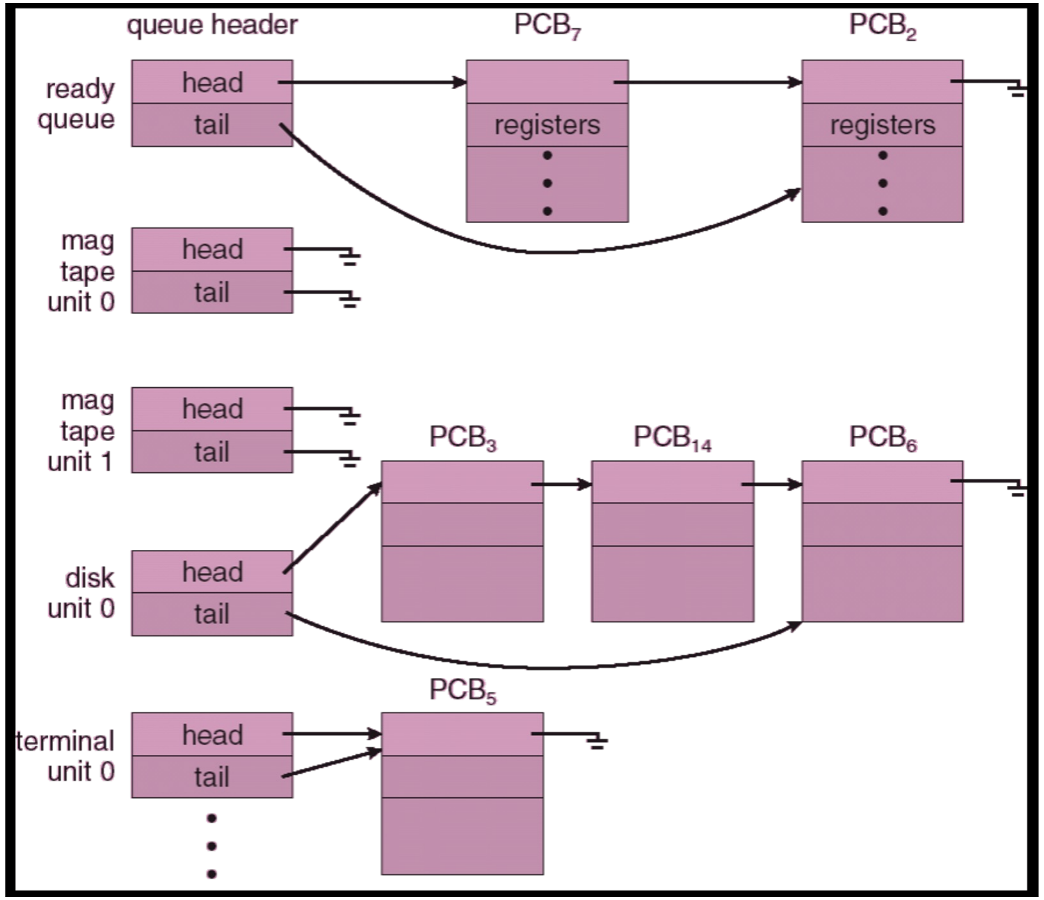


Time slice = 1

Suppose C is a game, and you press "shoot" at time 1, now it will take the system 1 second to respond to your action

How about jobs arriving later?

- The OS usually maintains a queue of all "RUNABLE" processes
  - A new process or a paused process is put at the end of the queue
- Round Robin: Select the head of the queue as the next job to run.
  - Note: If a job terminates before its time slice expires, RR will immediately run the next job
- SJF and STCF need to maintain a priority queue

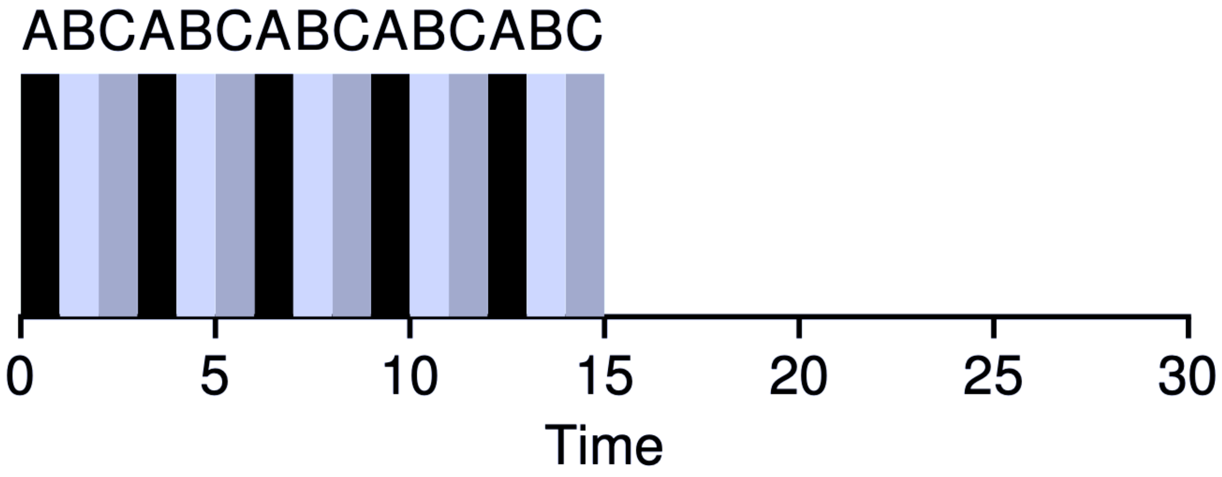


How to set time slice?

- Obviously, short time slice is good for response time.
- BUT! Context switch has an overhead
  - Shorter time slice means more context switches and higher overhead
- Example:
  - Context switch = 1ms and Time slice = 1ms → %time spent in context switch = 50%
  - Context switch = 1ms, Time slice = 10ms → %time spent in context switch = 1/(10+1) = 9%!
- Setting of time slice is a trade-off between response time and overhead: such trade-off is common in OS design

Round Robin problem?

- Did you notice any problem with Round Robin?



- Yes, average turnaround time is not so good:
- $T_{turnaround} = \frac{[(13-0)+(14-0)+(15-0)]}{3} = 14(s)$

Summary

- FIFO/FCFS: Very basic, easiest to implement, no need to know any run-time metrics; but bad performance with high turnaround time, and long processes can take up CPU
- SJF: Solved the problem for high turnaround time, but long processes can take up CPU (still non-preemptive)
- STCF: Optimal for turnaround time, but bad for response time
- RR: Good for response time, but bad for turnaround time
- When I/O exists, it will also complicate situations
- So, which one to use?

Multi-level feedback queue

- Fact:
  - Computation and interactive jobs can co-exist
  - Computation jobs prefer low turn-around time
  - Interactive jobs prefer low response time
  - A job can change pattern (computing or interactive) during its lifetime
  - Jobs can perform I/Os
- Problem?
  - When a new job arrives, we also do not know its properties
- How to schedule new jobs?

Multi-level feedback queue

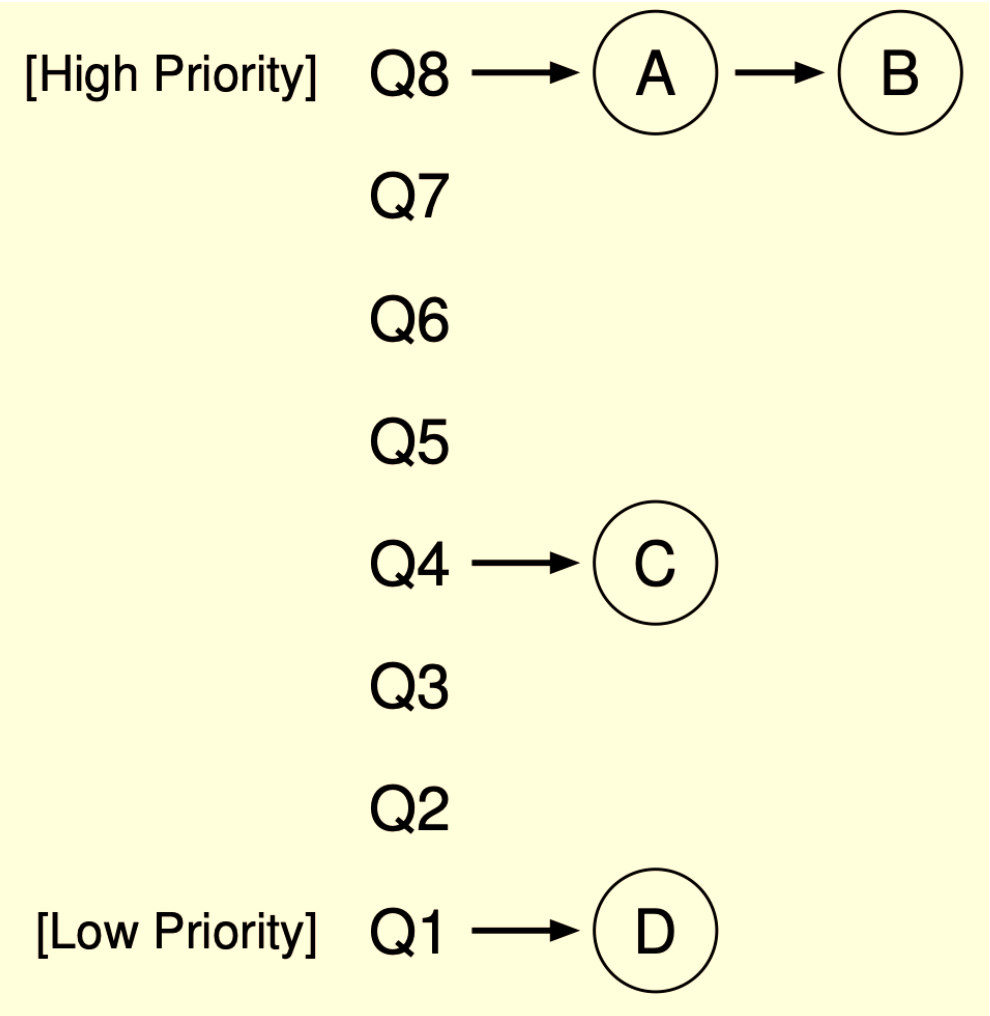
- **Fernando J. Corbató (**1926 – 2019)
- Major contributor to CTSS (Compatible Time-Sharing System) and MULTICS
- **Turing Award in 1990**: “for his pioneering work in organizing the concepts and leading the development of the general-purpose, large-scale, time-sharing and resource sharing computer systems”

Multi-level feedback queue

- MLFQ has a number of distinct queues:
  - Each queue has a priority level
  - Each job is assigned to a queue (and correspondingly, a priority level)
- **Rule 1:** If priority(A) > priority(B), A runs (B doesn't)
- **Rule 2:** if priority(A) = priority(B), A&B run in RR.

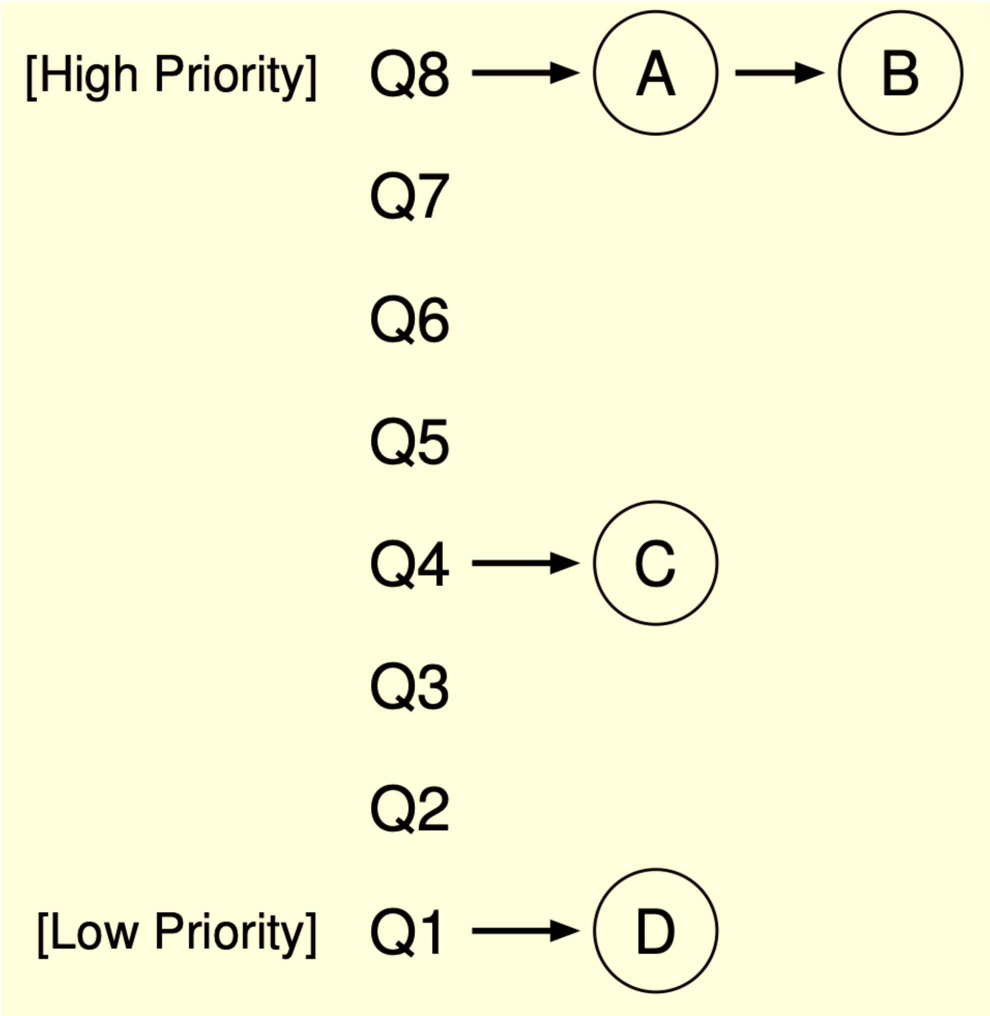
Multi-level feedback queue: Example

- What will happen here?
- A and B execute in RR until completion (or blocked)
- Then C executes until completion (or blocked)
- Then D executes until completion (or blocked)



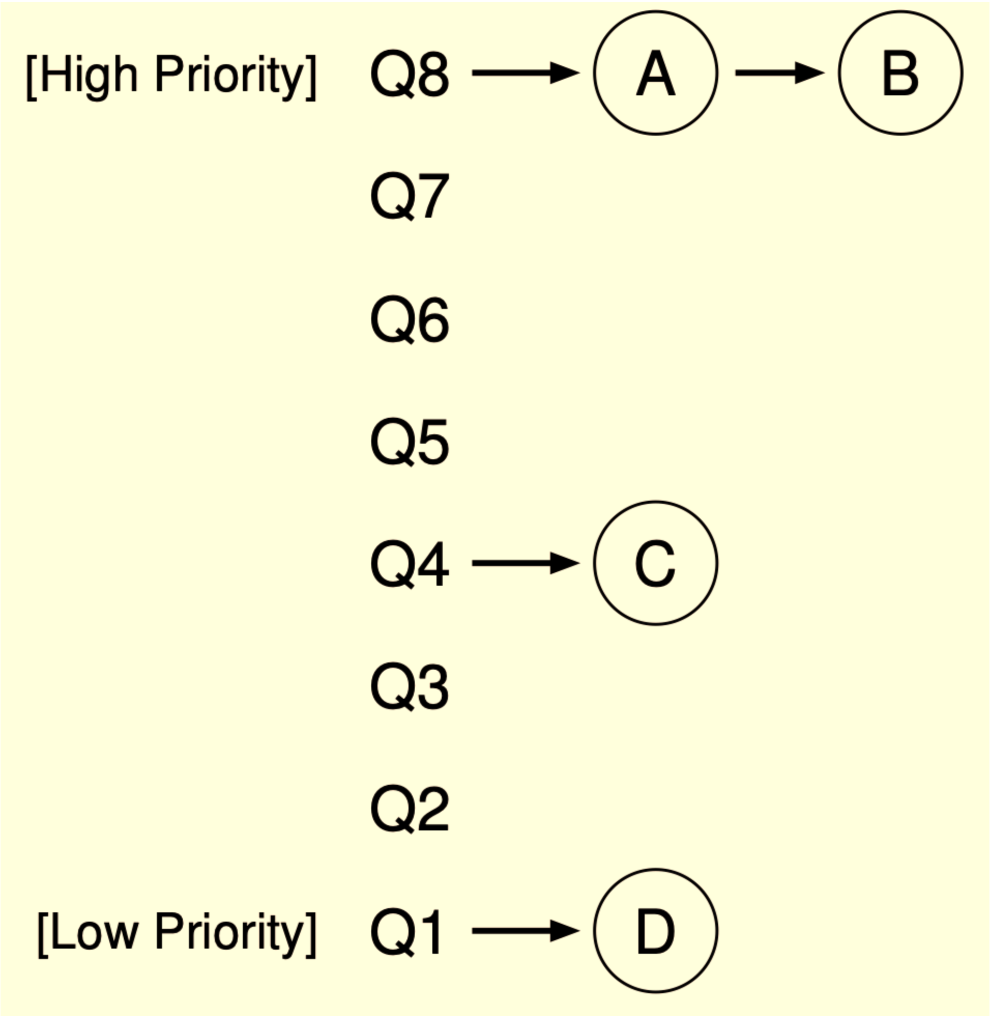
Multi-level feedback queue: Example

- Give high priority to interactive jobs
  - Interactive jobs should get a high priority to respond quickly to users' events
- But (another but), we do not know which job is interactive
  - MLFQ learns about process as they run: if a process repeatedly relinquishes CPU while waiting for input from keyboard, then it is probably an interactive job. At that time, they already had 'reinforcement learning' ;)



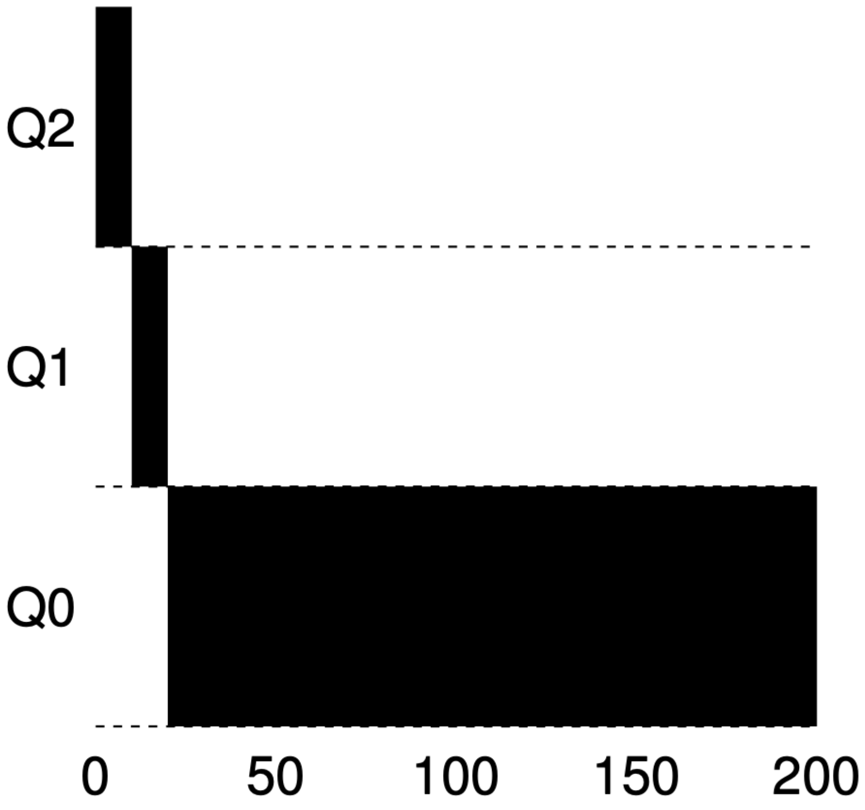
Multi-level feedback queue: Example

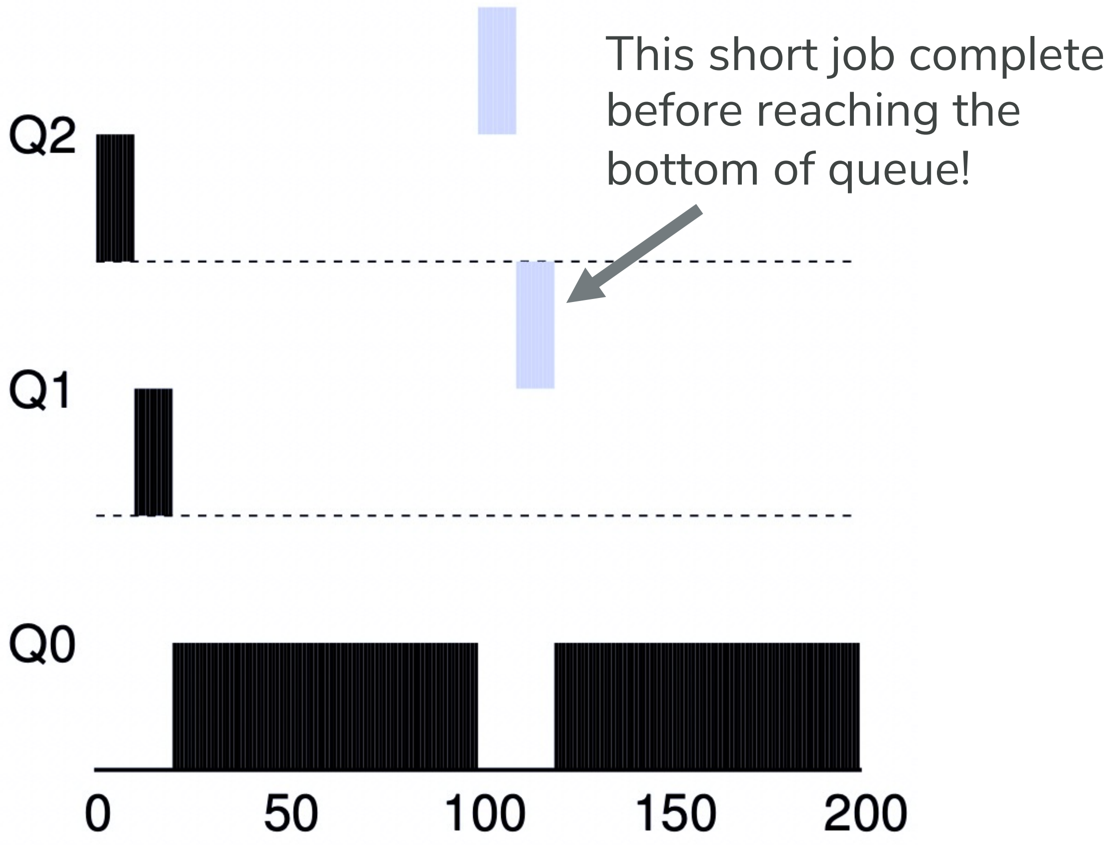
- So how to change priority?
- Rule 3: when a job enters the system, it is placed at the highest priority queue.
- Rule 4a: if a job uses up an entire time slice, it moves down a queue.
  - This is probably not an interactive job
- Rule 4b: if a job gives up the CPU before the time slice is up, it stays in the same queue.
  - Probably an interative job



MLFQ: Example

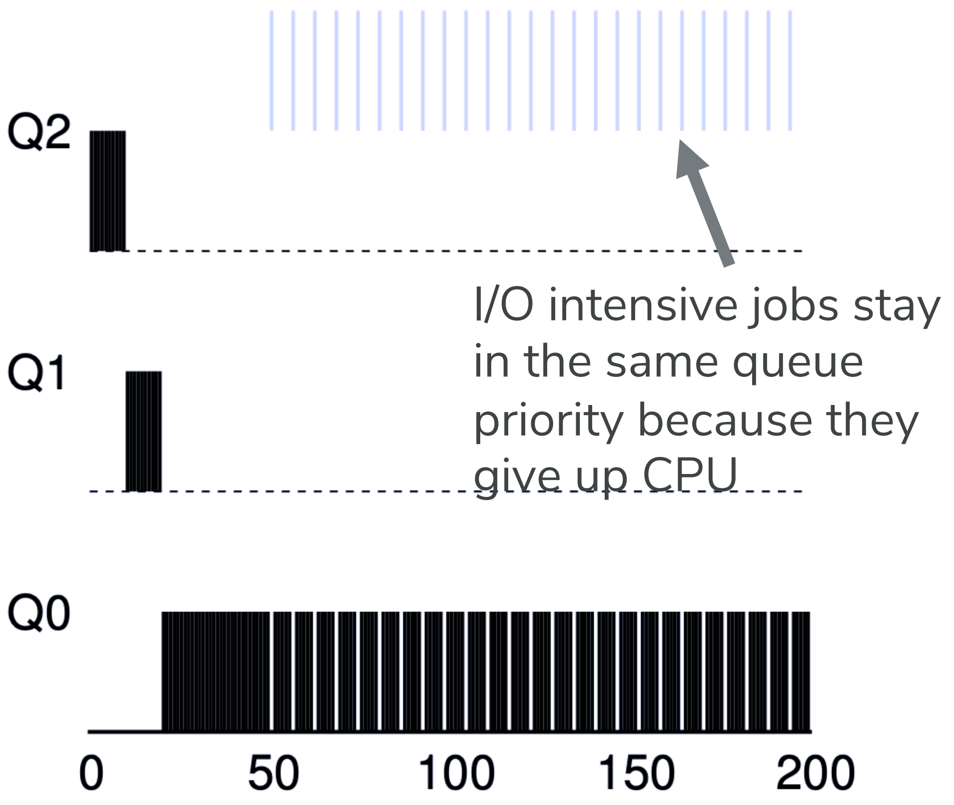
- A single long-running job vs. a long job with a short job





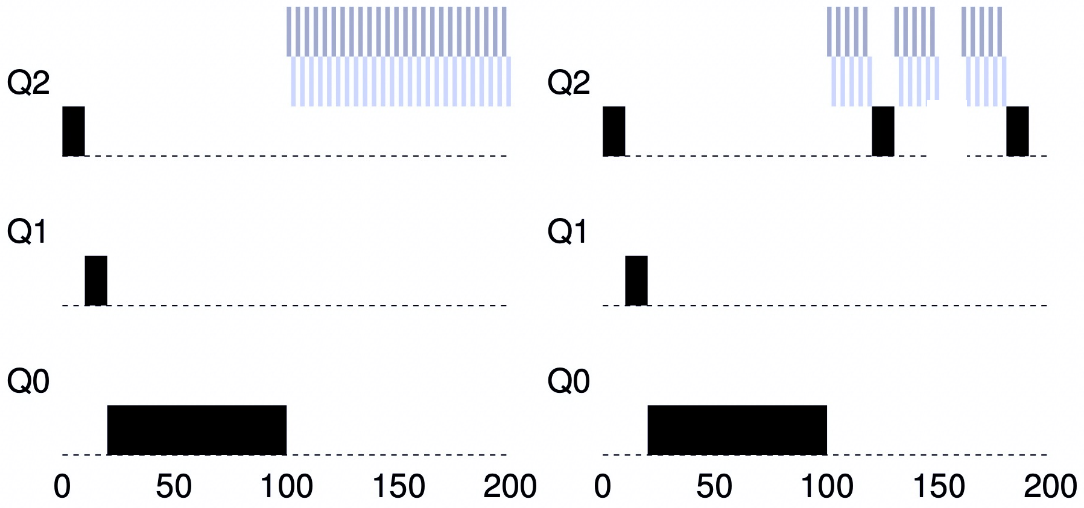
MLFQ: Example: Extensive I/O

- A single long-running job vs. a long job with a short job
- Problem?
  - Starvation: many short interactive jobs can consume all CPU time and block long-running jobs
  - A job may change its behavior: a computation job can transition to an interactive job
  - We forgot about malicious job: they want to take as much CPU as possible. How? A job can consume almost a full time slice and then issue an I/O



MLFQ: Example: Extensive I/O

- Priority boost
  - Rule 5: After some time period: S, move all the jobs in the system to the top-most queue
- It solves 2 problems:
  - No starvation because all jobs will be promoted at some time
  - If a computation job becomes interactive, it will be promoted at some time.



MLFQ: Improvement

- The scheduler keeps track of how much a time slice is used.
- Once a process has used its allotment, it is demoted

- Change Rule 4: once a job uses up its time allotment at a given level, its priority is reduced.
  - ~~Rule 4a: if a job uses up an entire time slice, it moves down a queue.~~
  - ~~Rule 4b: if a job gives up the CPU before the time slice is up, it stays in the same queue.~~

MLFQ: Summary

- Rule 1: If priority(A) > priority(B), A runs (B doesn't)
- Rule 2: if priority(A) = priority(B), A&B run in RR.
- Rule 3: when a job enters the system, it is placed at the highest priority queue.
- Rule 4: once a job uses up its time allotment at a given level, its priority is reduced.
- Rule 5: After some time period S, move all the jobs in the system to the top-most queue

MLFQ: How to set-up the parameters?

- Number of queues, time slice, allotment, when to boost, …
- It is a hard problem
- Typical solution: OS developers provide a default setting, which works fine in most environments, but it may not be optimal. Administrators can further tune these parameters.
- In general, high-priority queues are given shorter time slices.

Another metric: Fairness

- Sometimes we care more about another metric: Fairness
  - Every user should get a fair share of the resource
- Particularly important when users pay for the resource
  - E.g. Cloud, supercomputing clusters
  - If two users pay the same amount of money, they should get the same amount of CPU time, no matter whether their jobs are long or short
- Solution: lottery scheduling (chapter 9 required textbook [OSTEP])

More advanced scheduling (C10 Textbook)

- Multi-processor scheduling

- Real-time scheduling

Rise of multicore processor

- Moore's Law: the number of transistors in a dense integrated circuit has doubled approximately every two years
- Before 2005, people use more transistors to build faster CPUs.
  - The speed of a program can automatically increase with a faster CPU.
- However, CPU speed seems to reach its limit
  - Power consumption and heating
  - Manufacturing
- Today: people use more transistors to build more CPUs
  - The speed of a program does not automatically increase with more CPUs

  → Parallelize our program

Multithreaded programming

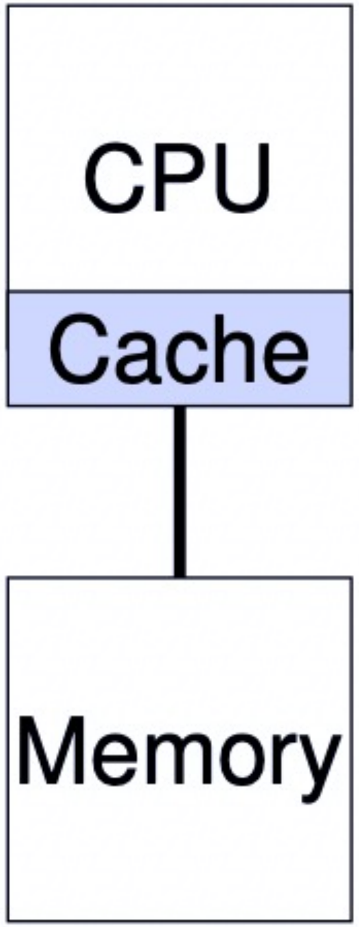
- Questions for developers:
  - How to parallelize an algorithm? --- Take another course
  - How to reason about the ensure correctness? --- Thread
- Questions for OS:
  - How to schedule processes when there are multiple CPUs?

Multi-processor scheduling

- Centralized approach:
  - Maintain a centralized data structure (a queue or a MLFQ) for all processes
  - Whenever a CPU is free, pick up a job from the centralized data structure according to some rule
- Pros: simple. Not much different from single-CPU scheduling
- Cons:
  - Low scalability: the centralized data structure must be protected by locks.
  - May miss other constraints

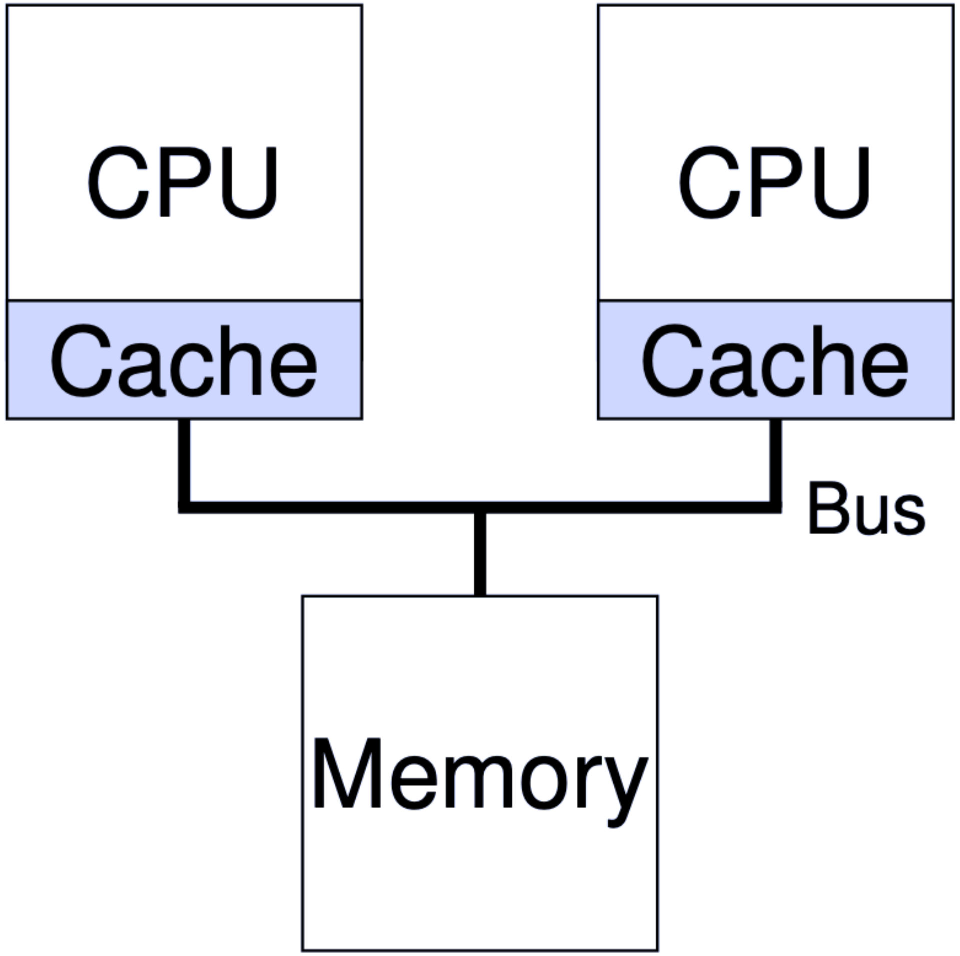
Additional constraint for a multi-core CPU

- From 1 CPU:
  - Consider a program that issues an explicit load instruction to fetch a value from memory, and a simple system with only a single CPU
  - The CPU has a small cache (say 64 KB) and a large main memory. The first time a program issues this load, the data resides in main memory, and thus takes a long time to fetch (perhaps in the tens of ns, or even hundreds).
  - The processor, anticipating that the data may be reused, puts a copy of the loaded data into the CPU cache. If the program later fetches this same data item again, the CPU first checks for it in the cache; if it finds it there, the data is fetched much more quickly (say, just a few nanoseconds), and thus the program runs faster.



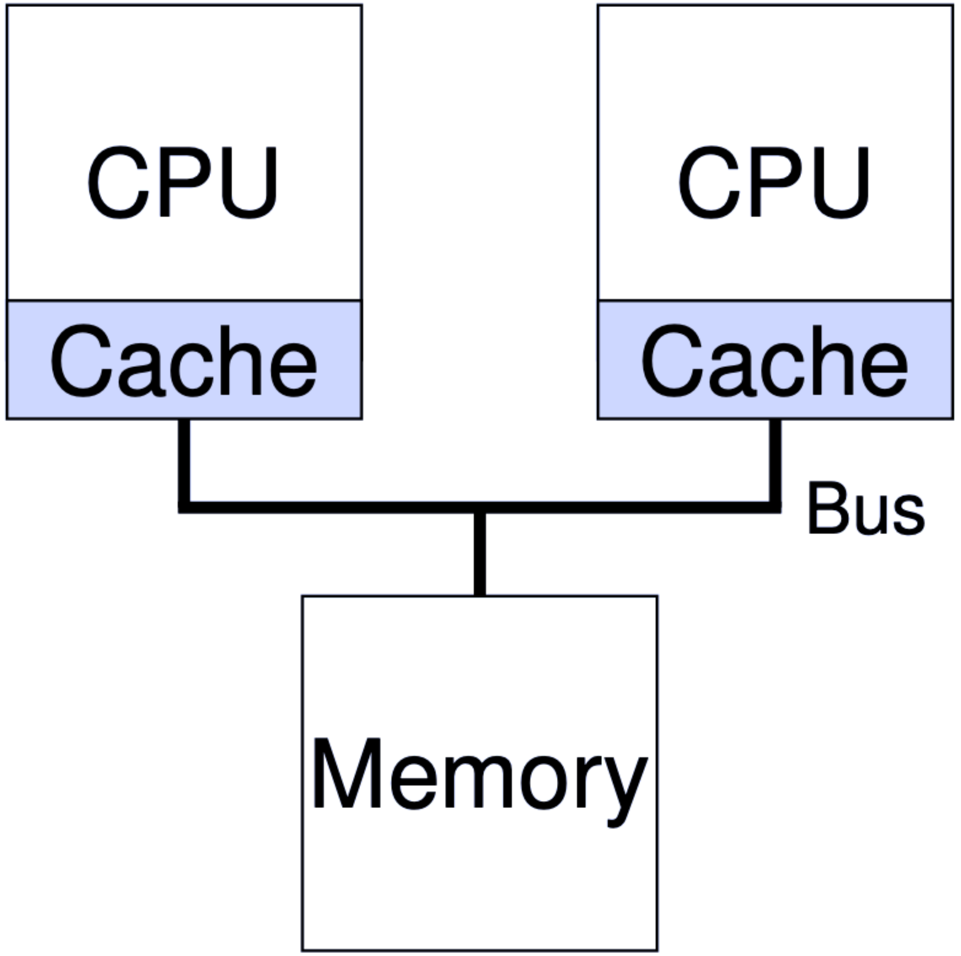
Additional constraint for a multi-core CPU

- To multicore CPUs
  - Each CPU has its own cache
  - However, only one shared main memory.
  - Catching with multiple CPUs is much more complicated.
- Problem 1: Cache Coherence.
  - CPU1 needs data in address A from main memory. Fetch it, then put it in its own cache. However, data will be manipulated, but CPU anticipated that data writing on main memory is slow.
  - CPU2 then needs the same data in address A → Different data now!



Additional constraint for a multi-core CPU

- To multicore CPUs
  - Each CPU has its own cache
  - However, only one shared main memory.
  - Catching with multiple CPUs is much more complicated.
- Problem 2: Cache affinity
  - It is better to schedule a job to the CPU that previously ran it, because its cache may still contain the job's data.
  - Scheduling a previously ran job on a different CPU leads to a lot of state reloading and data fetching.



Additional constraint for a multi-core CPU

- De-centralized approach
  - Each CPU maintains its own data structure (e.g. a queue) for scheduling
  - Assign a job to a CPU
  - Each CPU schedules by itself

- Pros: no scalability bottleneck; take cache affinity into consideration

- Cons: load imbalance

Additional constraint for a multi-core CPU

- Load imbalance example:
  - Assume we have a system where there are two CPUs (CPU0, CPU1).
  - Jobs enter the system: A, B, C, D as follows. OS may do something like this:

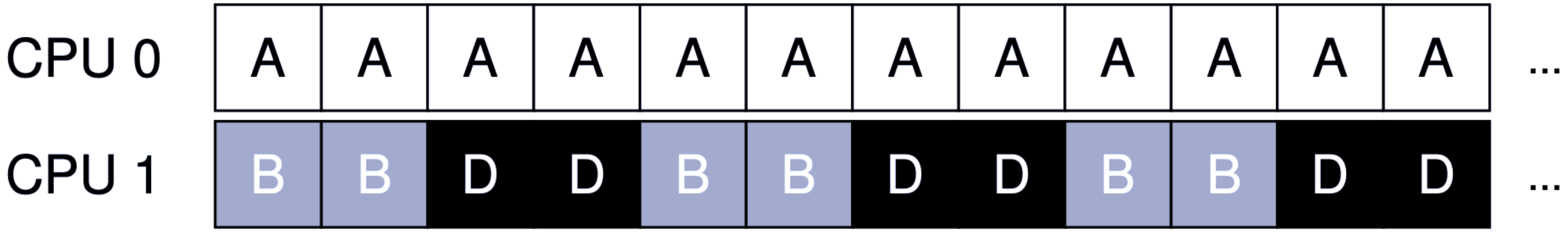
  - Depending on the queue scheduling policy, each CPU now has two jobs to choose from when deciding what should run. E.g. RR

CPU 0	A	A	C	C	A	A	C	C	A	A	C	C	…
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	…

Additional constraint for a multi-core CPU

- Load imbalance example:
  - Assume we have a system where there are two CPUs (CPU0, CPU1).
  - Now what if one of the jobs (job C) finishes. We have now:

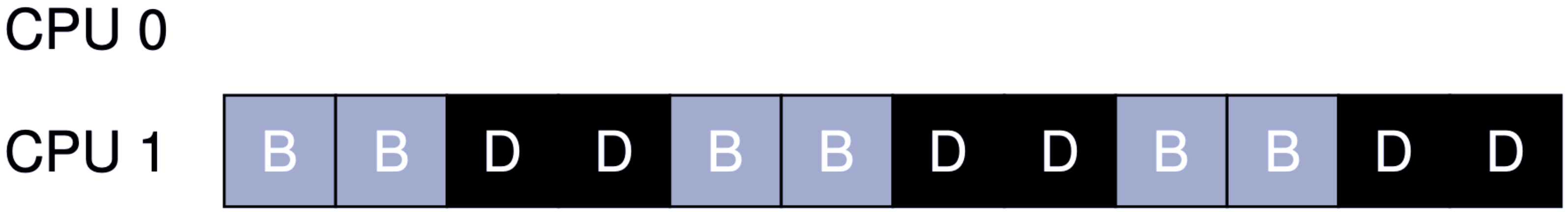
  - Now for RR policy on each queue of the system, we will see this resulting schedule: A gets twice as much CPU as B and D, which is not the desired outcome



Additional constraint for a multi-core CPU

- Load imbalance example:
  - Assume we have a system where there are two CPUs (CPU0, CPU1).
  - Even worse, let's imagine both A and C finish, leaving just jobs B and D in system.

  - Now for RR policy on each queue of the system, we will see this result:



How terrible – CPU 0 is idle! *(insert dramatic and sinister music here)*
And thus our CPU usage timeline looks quite sad.

Centralized or De-centralized?

- Both have pros and cons. Both are used in practice
- People have tried to mitigate their shortcomings:
  - For centralized approach, we can take cache affinity into consideration when picking up the next process to schedule
  - For decentralized approach, we can use work stealing to migrate one job from a busy CPU to a free one.

Real-time scheduling

- Today many devices are controlled by computers:
  - **E.g. cars**, planes, satellites, power plants, …
- Many of them require that the computer must respond to critical tasks within a given amount of time (**hard deadline**)
  - **E.g. brakes on the car**
- The scheduling algorithms we learned so far cannot guarantee hard deadlines. We need new algorithms:
  - It can take another course to discuss this topic.
  - Those devices usually use specific operating systems.

Summary: CPU virtualization

- Process: it is the basic unit of CPU virtualization
- Basic idea: time-sharing
- Time-sharing Mechanism (Context Switch): Limited Directed Execution
  - Users' programs execute restricted operations through system calls
  - OS periodically takes control of CPU with the help of timer
- Time-sharing Policy (CPU scheduling)
  - Metrics: turnaround time, response time, fairness, …
  - Algorithms: FIFO, SJF, STCF, RR, MLFQ, …