Operating Systems

2. Virtualizing CPU

Pablo Prieto Torralbo

DEPARTMENT OF COMPUTER ENGINEERING AND ELECTRONICS

2.1 Virtualizing the CPU -Process

What is a process?

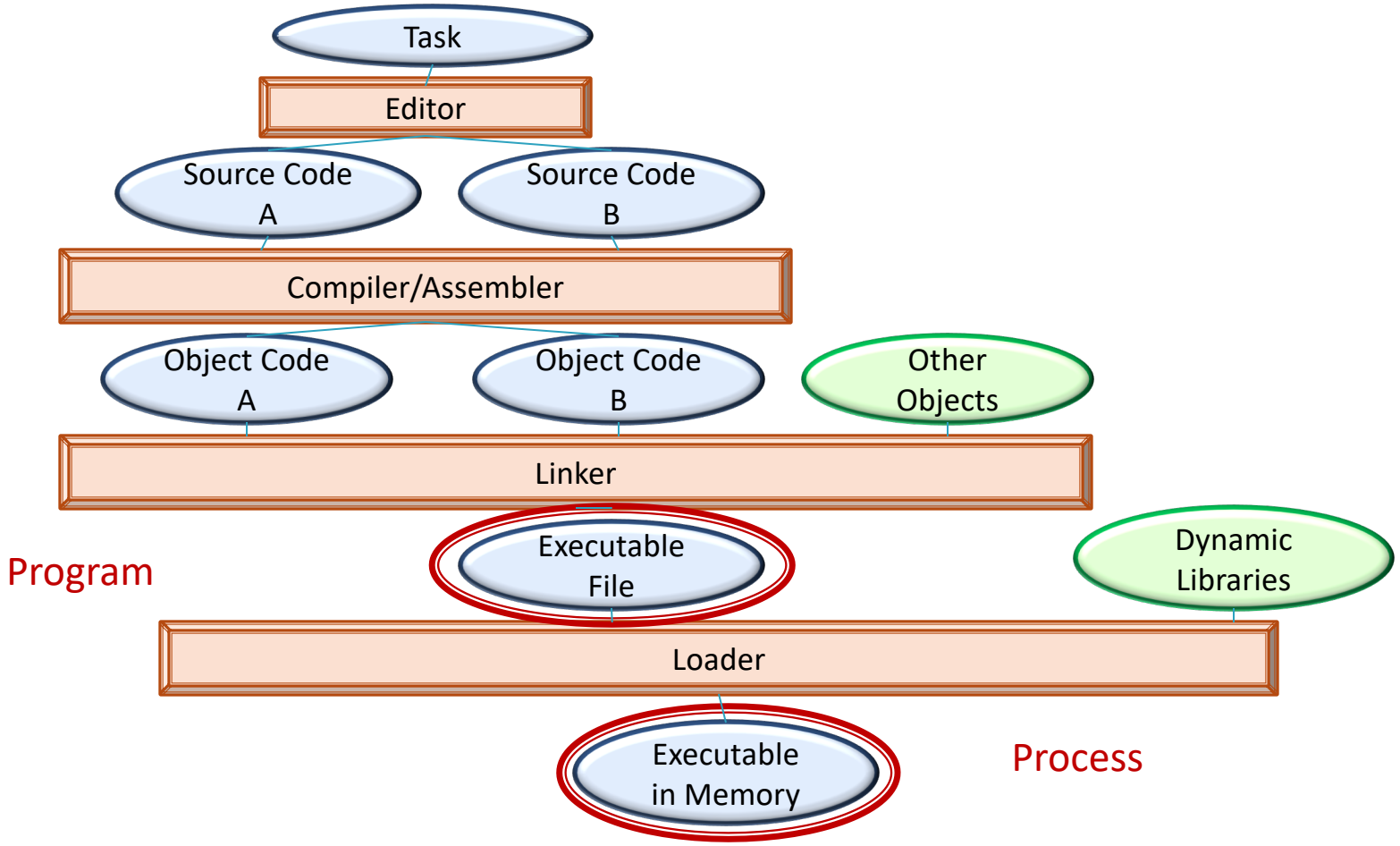
- A running program.
  - Program: Static code and static data sitting on the disk.
  - Process: Dynamic instance of a program.
  - You can have multiple instances (processes) of the same program (or none).
  - Users usually run more than one program at a time.
    - Web browser, mail program, music player, a game…
- The process is the OS's abstraction for execution
  - often called a job, task...
  - Is the unit of scheduling

Program

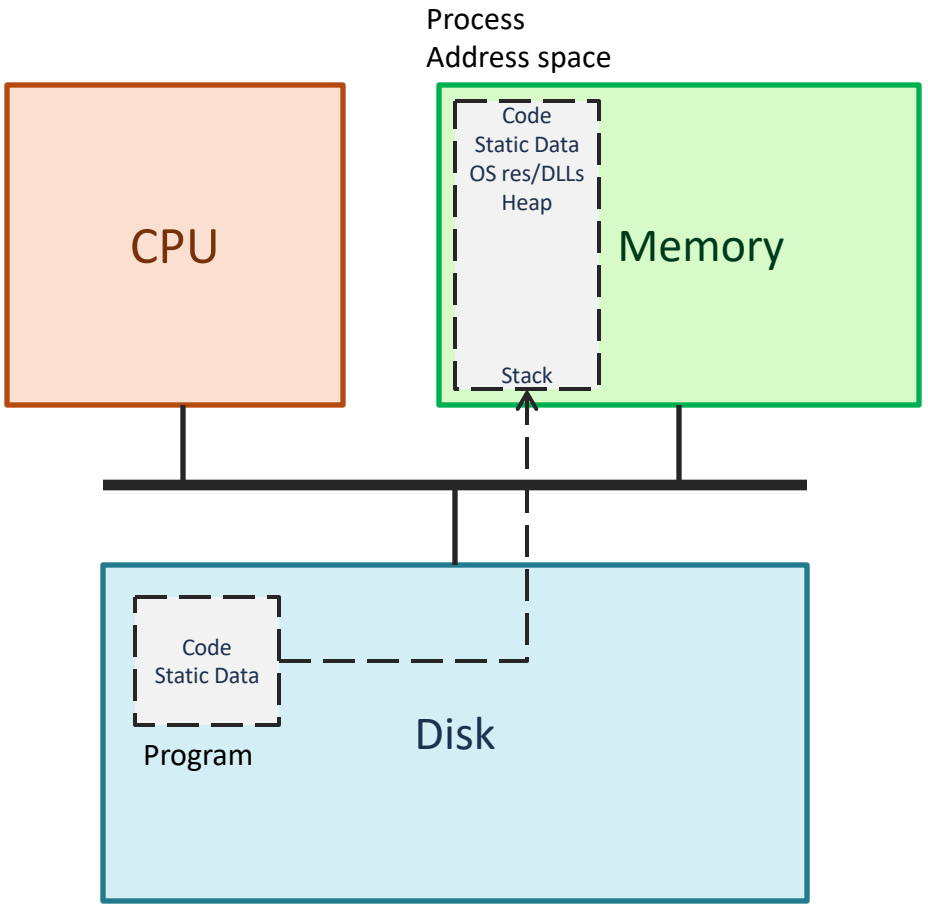
- A program consists of:
  - Code: machine instructions.
  - Data: variables stored and manipulated in memory.
    - Initialized variables (global)
    - Dynamically allocated (malloc, new)
    - Stack variables (function arguments, C automatic variables)
- What is added to a program to become a process?
  - DLLs: Libraries not compiled or linked with the program (probably shared with other programs).
  - OS resources: open files…

Program

- Preparing a program:



Process Creation



Loading:
OS Reads on disk program and places it into the address space of the process

Eagerly/Lazily

Process State

- Each process has an execution state, which indicates what it is currently doing

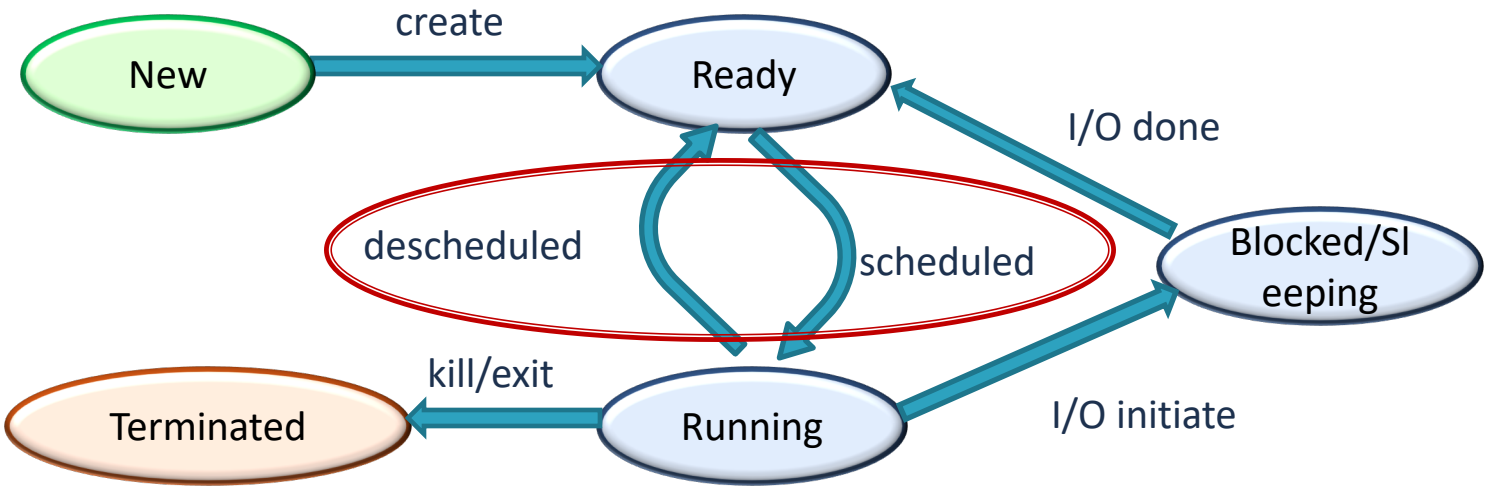
  `ps -l, top`

  - Running (R): executing on the CPU
    - Is the process that currently controls the CPU
    - How many processes can be running simultaneously?
  - Ready/Runnable (R): Ready to run and waiting to be assigned by the OS
    - Could run, but another process has the CPU
    - Same state (TASK_RUNNING) in Linux.
  - Blocked/Sleeping (D/S): Waiting for an event.
    - Has performed some operation (e.g. I/O) and cannot make progress until an event happens (e.g. request fulfilled).

T(Stopped) & Z(Zombie)

Process State

- As a process executes, it moves from state to state



How to transition? → mechanism
Which/When to transition? → policy

Process State

- Example1: Two processes – CPU Only

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	
5	Running	Ready	$Process_0$ done
6		Running	
7		Running	
8		Running	
9		Running	
10		Running	$Process_1$ done

Process State

- Example2: Two processes – CPU and I/O

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is Blocked
5	Blocked	Running	
6	Blocked	Running	$Process_0$ I/O done
7	Ready	Running	
8	Ready	Running	$Process_1$ done
9	Running		
10	Running		$Process_0$ done

Process State

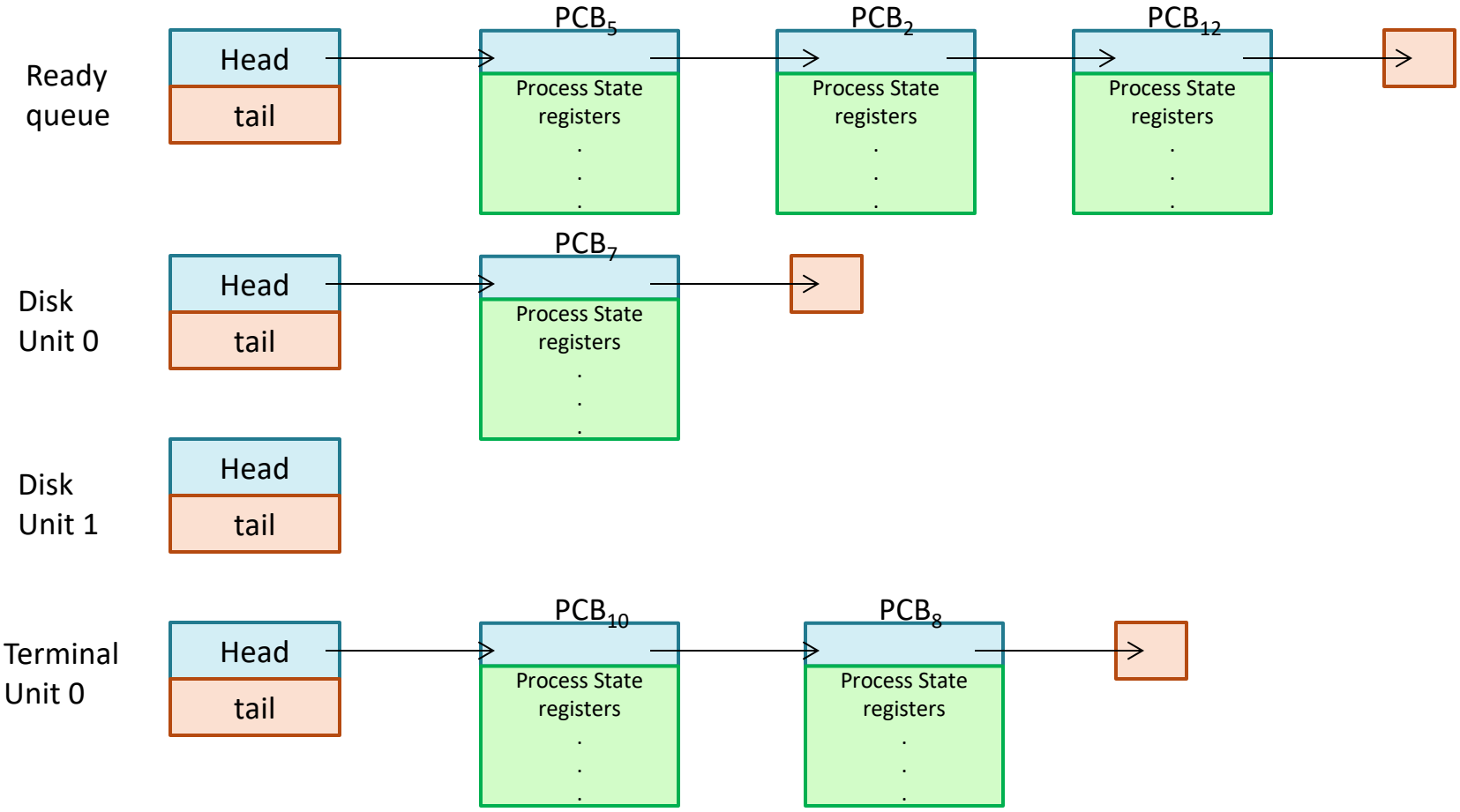
- Process machine state (or context) is the system pieces it accesses or affects and consists of (at least):
  - Address space (`/proc/<PID>/maps`)
    - The code
    - Data (Heap)
    - Stack and stack pointer (SP)
  - Program Counter indicating the next instruction
  - A set of general-purpose registers (stack pointer...)
  - A set of OS resources
    - File descriptors (`/proc/<PID>/fdinfo/*`)
    - network connections
    - sound channels…
- The process is named by the OS with a process ID (PID).
  - `ps, top`…

Process data structures

- OS is a program and has its own data structures.
- At any time there are many processes, each in its own state.
  - How does the OS represent a process in the kernel?
- The OS data structure that represents each process is called Process Control Block (PCB)
- The OS maintains a list (queue) of PCBs for each execution state
  - E.g. Ready queue, Job queue, blocked/waiting queue/s.

Process data structures

- Process State Queues. Example:



Process data structures

- A PCB contains info about the process:
  - Hardware state
    - PC
    - SP
    - Registers
    - Mem address
    - Mem size
  - OS information
    - PID
    - Open Files
    - Other (Parent process, killed, final, cwd, owner…)
- In linux:
  - task_struct (`sched.h`)
  - thread_struct (architecture dependent `processor.h`)
  - Hundreds of fields.

Context Switch

- When a process is running, its hardware state is inside the CPU
  - PC, SP, Registers…
- When the OS stops a running process, it saves the register values in the corresponding PCB
  - When the OS puts a process in the running state, it loads the hardware registers from that process PCB.
- The act of switching the CPU from one process to another is called a context switch
  - Takes a few microseconds on today's hardware

2.2 Virtualizing the CPU
-Limited Direct Execution

Virtualizing the CPU

- Different processes will share the CPU
  - Processes should not know that they are sharing
- How can you give the illusion of multiple CPUs?
  - Giving each process the impression that it is using the CPU alone
  - Time Sharing (Context Switch)
    - Running one process → stopping it → running another …
- Goals:
  - Efficiency:
    - How to implement virtualization without OS adding excessive overhead?
  - Control:
    - How to run processes efficiently while restraining control over the System?
    - OS decides which/when processes run.

Direct Execution

- Direct execution
  - Just run the program directly on the CPU

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute call main	
	Process A runs main
	…
	Execute return from main
Free memory of process	
Remove process from list	

- Direct Execution is fast
  - Running on the hardware CPU natively

Problem #1: Restricted Operations

- How can the OS make sure that the program doesn't do anything it shouldn't...
  - ... While still running the program efficiently?

- Let the application do what it wants:
  - Protection would be lost
  - File system compromised
    - Read or write anywhere on the disk
  - No privacy
  - Memory manipulation (OS, other processes…)

Limited Direct Execution

- Hardware assists the OS by providing modes of execution
  - User mode: applications do not have full access to hardware resources.
    - Cannot issue I/O requests → raise an exception → OS would likely kill the offending process
  - Kernel mode: the OS has full access to the resources of the machine.
    - Including privilege operations, I/O requests…
- What if a process wants to read from disk?
  - System call.

Limited Direct Execution

- System calls allow the kernel to expose key functionalities to user programs.

- To execute a system call a special instruction (trap) is provided.
  - This instruction raises privilege level to kernel mode.
  - Simultaneously jumps into the kernel and executes what is needed.

- When finished, the OS calls another special instruction (return-from-trap)
  - This instruction returns to the calling user program
  - Simultaneously reduces the privilege level back to user mode.

Limited Direct Execution

- How does the trap know which code to run inside the OS?
  - Calling process cannot specify the address to jump to or any code could be executed in kernel mode.
  - Instead, the calling process must specify a system-call number.

- Kernel sets up a trap table at boot time.
  - When the machine boots up, kernel (in privileged mode) sets up the table.
  - The table tells the hardware what code to run when certain events occur (e.g. hard-disk interrupt)
    - Traps, exceptions and interrupts are handled similarly.
    - Trap number n, must look at the n entry of the table.
  - OS informs hardware about the location of these trap handlers with a special instruction.
    - Saves the memory address of the trap table.
    - This is a privileged operation (only in kernel mode).

Limited Direct Execution

OS boot (kernel mode)	Hardware	Program (user mode)
Initialize trap table		
	Remember address of: Trap handler	
Initialize process table		

OS run (kernel mode)	Hardware	Program (user mode)
To start process A: allocate entry in process table allocate memory for process load program into memory setup user stack (argc/argv) setup kernel stack with reg/PC return-from-trap (into A)		
	Restore registers of A from kernel stack Move to user mode Jump to A initial PC (main)	
		Process A runs main() … Call system call trap into OS
	Save regs to kernel stack Move to kernel mode Jump to trap handler	
Handle trap Do work of syscall return-from-trap		
	Restore registers of A from kernel stack Move to user mode Jump to A to PC after trap	
		… (keep execution) Return from main trap (via exit)
Free memory of process Remove from process list		

Problem #2: Switching Processes

- When running a process, how does the operative system stop it and switch to another process?
  - If a process is running on the CPU, OS is not running
- How can the OS regain control of the CPU to switch between processes?
  - Early versions of Macintosh → Cooperative Approach
    - OS trusts processes to behave reasonably.
    - Most long processes transfer control to the OS due to frequently making system calls.
    - OS includes an explicit system call (yield) that does nothing but transfer control to OS → Utopia?
- What if a process (malicious or full of bugs) enters an infinite loop?
  - Reboot?

Problem #2: Switching Processes

- Non-cooperative approach: OS takes control
  - Without hardware help, it cannot.
- Hardware help: Timer interrupt
  - Timer device to raise an interrupt every so many milliseconds (100 – 1000 switches per second).
  - Current process is halted and a preconfigured interrupt handler runs in the OS (kernel mode).
    - Similar process as with trap handler.
    - OS gets control over the CPU and can make decisions (e.g. switch to another process). → scheduler
  - OS must inform hardware of the code to run when the timer interrupt occurs.
    - Privileged operation at boot time. Also start the timer.

LDE with Timer Interrupt

OS boot (kernel mode)	Hardware	Program (user mode)
Initialize trap table	Remember address of: System call handler	
Initialize process table Start interrupt timer		
	Start timer Interrupt CPU in X ms.	

OS run (kernel mode)	Hardware	Program (user mode)
		Process A running...
	Timer interrupt Save regs to kernel stack (A) Move to kernel mode Jump to interrupt handler	
Handle timer interrupt Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to kernel-stack(B) return-from-trap (into B)	Context Switch	
	Restore regs (B) from kernel-stack (B) Move to user mode Jump to B's PC	
		Process B running...

2.3 Virtualizing the CPU -Scheduling

Introduction

- We understand the basic mechanisms of running processes
  - Context switching
  - Timer interrupt
  - Limited direct execution
    - Execution modes
    - Syscalls, Traps
- OS needs to decide when to use them
  - Scheduling policies

Remember…

- Example2: Two processes – CPU and I/O

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked
5	Blocked	Running	
6	Blocked	Running	I/O done
7	Ready	Running	
8	Ready	Running	$Process_1$ done
9	Running	-	
10	Running	-	$Process_0$ done

OS Decides to Run $Process_1$

OS Decides not to Switch back to 0

Workload Assumptions

- Some assumptions about the processes running in the system (workload)
  - Each job runs for the same amount of time
  - All jobs started at the same time
  - Once started, each job runs until completion
  - All jobs only use the CPU (e.g. no I/O)
  - The run-time of each job is known

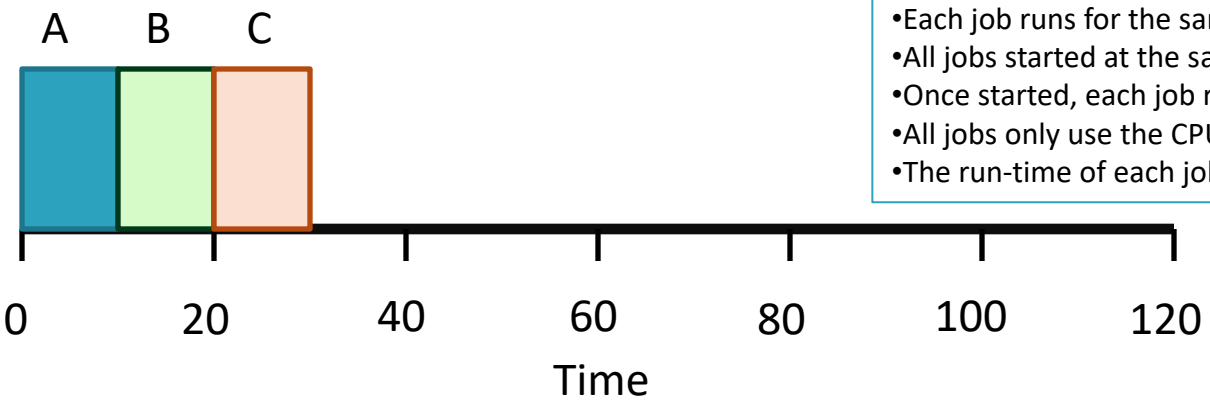
Unrealistic but good to start with…

Scheduling Metrics

- We need metrics to compare different scheduling policies → a scheduling metric
  - Turnaround time:
    - Performance metric
    - $T_{turnaround} = T_{completion} - T_{arrival}$
  - Fairness
    - Jain's Fairness Index (variability of throughput)
    - Max-min
  - Response time
    - Interactive performance
    - $T_{response} = T_{first\_run} - T_{arrival}$
- Usually there is a trade-off between metrics
  - Let's focus on turnaround time.

First In, First Out

- The most basic algorithm is First In, First Out (FIFO) or First Come, First Served (FCFS).
  - Simple
  - Easy to implement
  - Works well with our assumptions
    - Three Jobs (A,B,C) arriving at 0 running for 10 seconds



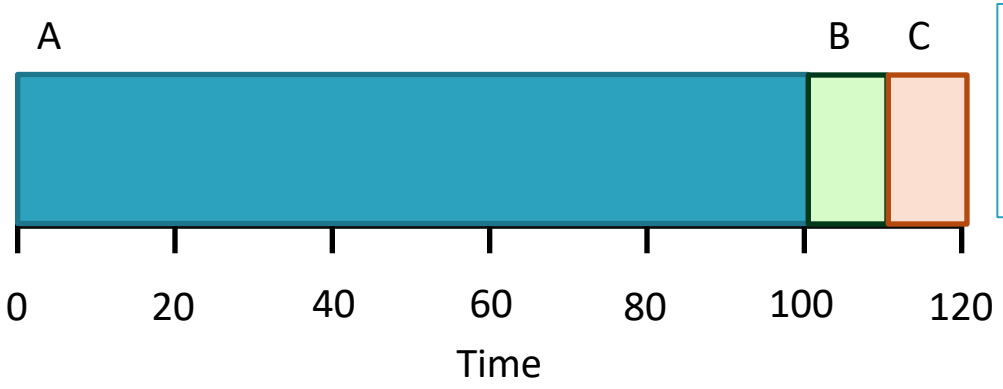
- Each job runs for the same amount of time
- All jobs started at the same time
- Once started, each job runs until completion
- All jobs only use the CPU (e.g. no I/O)
- The run-time of each job is known

    - Average turnaround time

(10+20+30)/3=20seconds

First In, First Out

- Relax assumption #1: Each job runs for the same amount of time
  - How does FIFO perform now?
    - Three jobs (A,B,C): A runs for 100 seconds and B and C 10 each.



~~•Each job runs for the same amount of time~~
•All jobs started at the same time
•Once started, each job runs until completion
•All jobs only use the CPU (e.g. no I/O)
•The run-time of each job is known

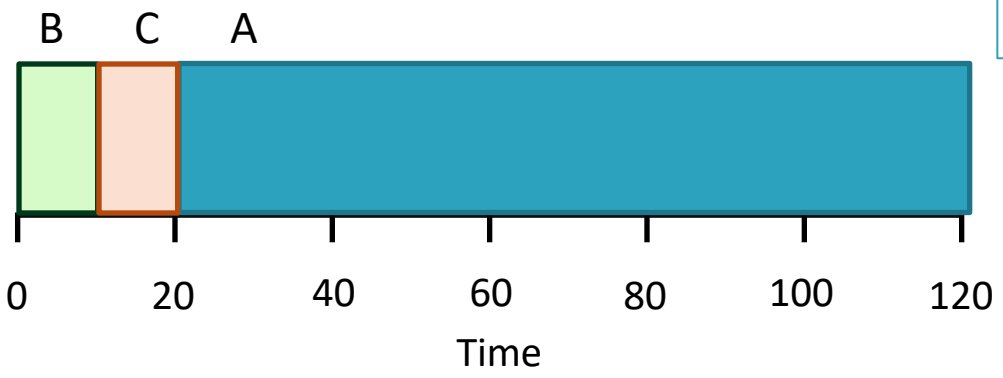
    - Average turnaround time

(100+110+120)/3=110seconds

Shortest Job First

- A new scheduling discipline solves the problem: Shortest Job First (SJF).
  - Runs the shortest job, then the next shortest…
  - Same example as before:

~~•Each job runs for the same amount of time~~
•All jobs started at the same time
•Once started, each job runs until completion
•All jobs only use the CPU (e.g. no I/O)
•The run-time of each job is known

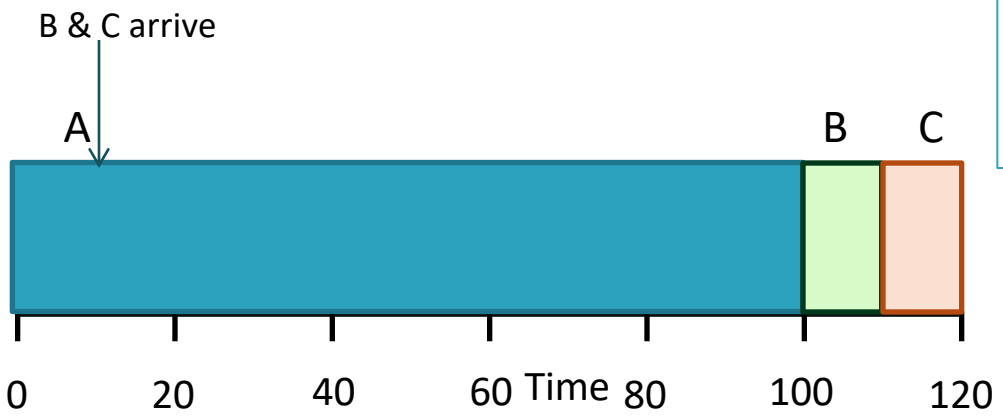


  - Average Turnaround time

(10+20+120)/3=50seconds

Shortest Job First

- Let's relax assumption #2: Jobs can arrive at any time.
  - A arrives at t=0 running for 100 seconds and B and C arrive at t=10 running for 10 seconds each.
  - How does SJF perform now?



~~•Each job runs for the same amount of time~~
~~•All jobs started at the same time~~
•Once started, each job runs until completion
•All jobs only use the CPU (e.g. no I/O)
•The run-time of each job is known

  - Average Turnaround time

(100+(110-10)+(120-10))/3=103.33seconds

Shortest Time-to-Completion First

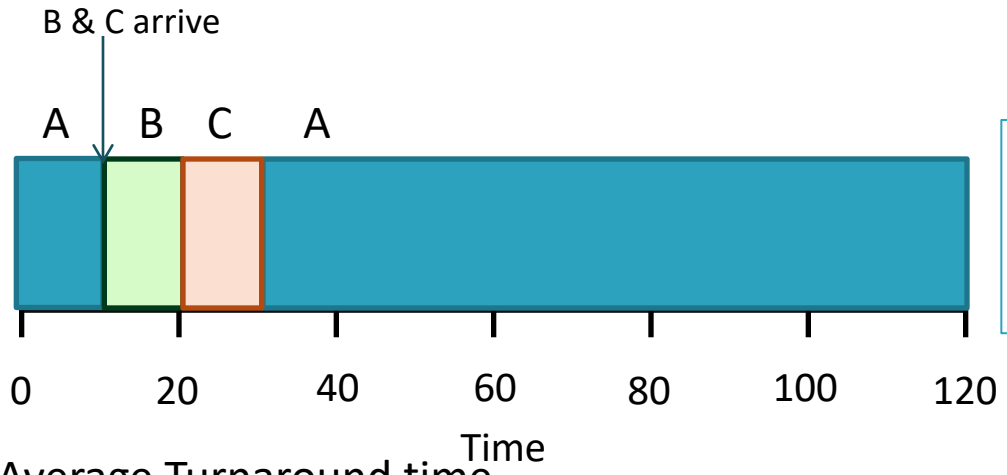
- To address this concern we need to relax assumption #3.

~~•Each job runs for the same amount of time~~
~~•All jobs started at the same time~~
~~•Once started, each job runs until completion~~
•All jobs only use the CPU (e.g. no I/O)
•The run-time of each job is known

- Scheduler needs to make some decisions:
  - Using timer interrupts and context switching, preempt job A and run another job.
  - Previous scheduling algorithms were non-preemptive (cooperative).

Shortest Time-to-Completion First

- Scheduler that adds preemption to SJF: Shortest Time-to-Completion First (STCF) or Preemptive Shortest Job First (PSJF)
  - Let's run previous example: A arrives at t=0 and runs for 100 seconds, B and C arrive at t=10 and run for 10 seconds each



- ~~Each job runs for the same amount of time~~
- ~~All jobs started at the same time~~
- ~~Once started, each job runs until completion~~
- All jobs only use the CPU (e.g. no I/O)
- The run-time of each job is known

  - Average Turnaround time

(120+(20-10)+(30-10))/3=50seconds

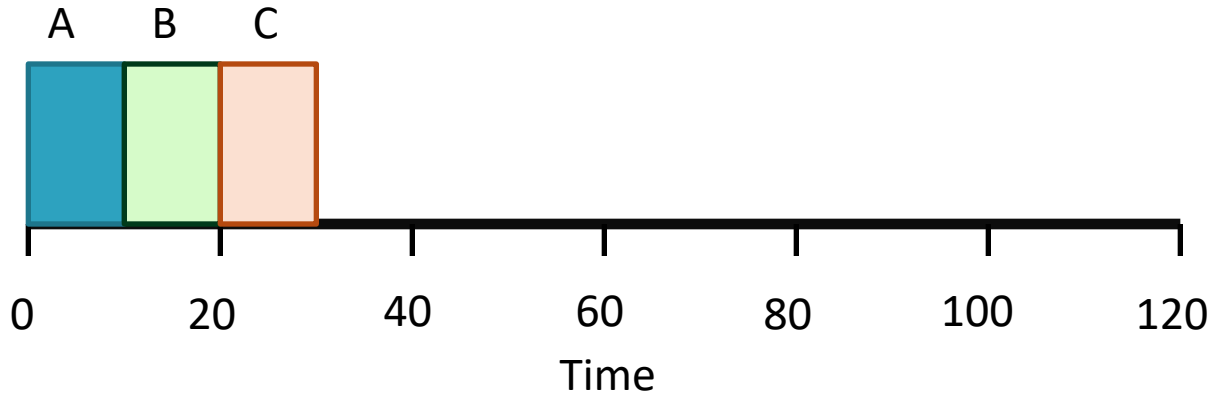
Response Time

- Considering just turnaround time, STCF is a great policy.
  - For early batch computing systems → made sense
  - When users demand interaction → not so much

- New metric: Response Time
  - $T_{response} = T_{first\_run} - T_{arrival}$

Response Time

- Imagine first example: A, B and C arriving at t=0 and running for 10 seconds each.
  - All previous policies behave equally



  - Response Time in seconds for A: 0 , B: 10 and C:20
    - Average: 10 seconds

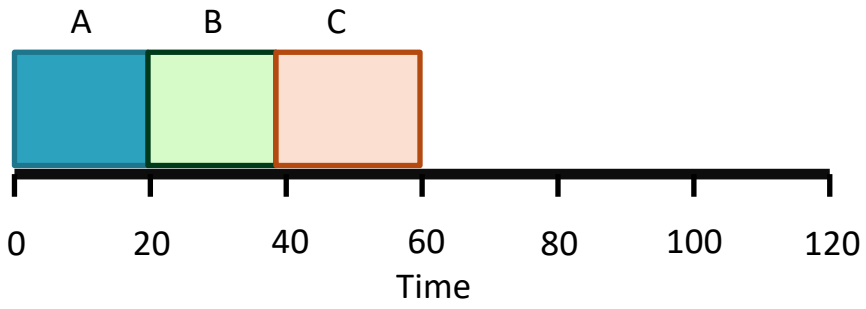
Round Robin

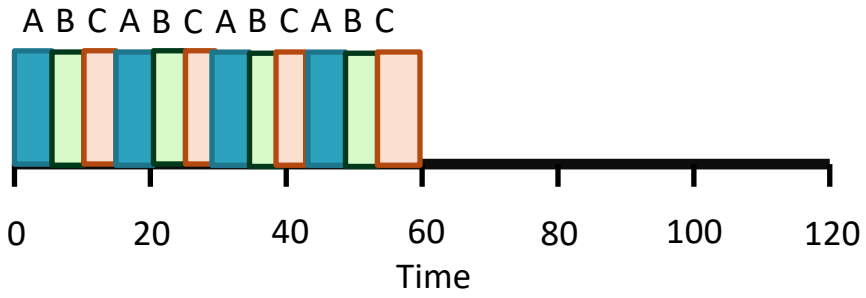
- How can we build a scheduler sensitive to response time?

- Round-Robin scheduling (RR)
  - OS Instead of running jobs to completion, runs each job for a time slice (scheduling quantum) and switches to the next job in the run queue.
  - Round-Robin is sometimes called time-slicing.
  - Length of the time slice must be multiple of timer-interrupt period.
    - E.g. If the timer interrupts every 10 milliseconds, a time slice could be 10, 20 or any multiple of 10 ms.

Round Robin

- Let's look at an example:
  - A, B, and C arrive at the same time (t=0) and each one runs for 20 seconds.
  - RR time slice = 5 second





SJF:

Average Response Time:

(0+20+40)/3=20 seconds

RR:

Average Response Time:

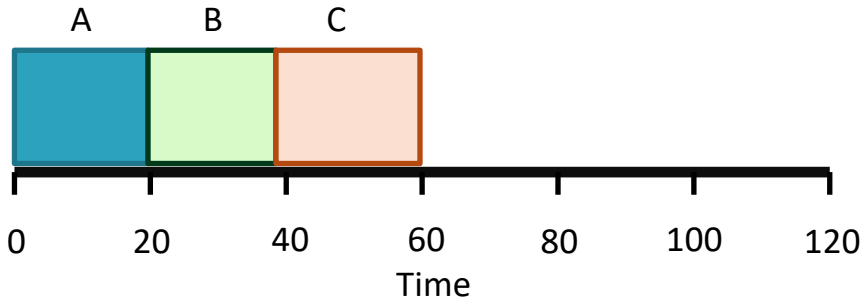
(0+5+10)/3=5 seconds

Round Robin

- Length of the time slice is critical for RR.
  - Shorter = better for interactive performance (response time)
  - Context switching could dominate overall performance
    - Remember: current hardware about 5 microseconds.
    - Context switching also flushes the program's HW state (caches, TLB, branch predictors…).
  - Trade-off:
    - Long enough to amortize the cost of switching.
    - Short enough so that the system is still responsive.

Round Robin

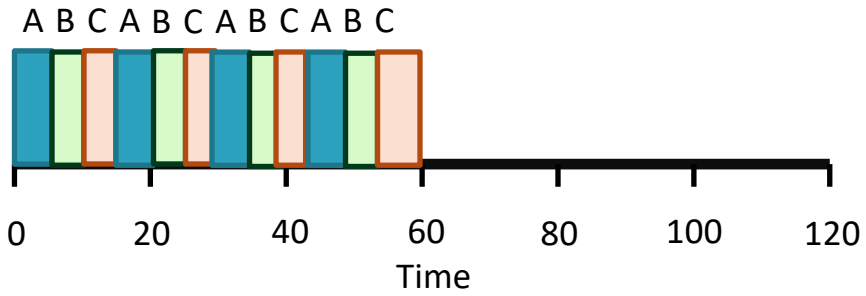
- What about turnaround metric?



SJF:

Average Turnaround Time:

(5+10+15)/3=10 seconds



RR:

Average Turnaround Time:

(13+14+15)/3=14 seconds

- RR is one of the worst policies in performance (turnaround time)
  - Generally, a fair policy will behave poorly in performance metrics.

Incorporating I/O

- We will relax assumption #4: Programs do perform I/O.

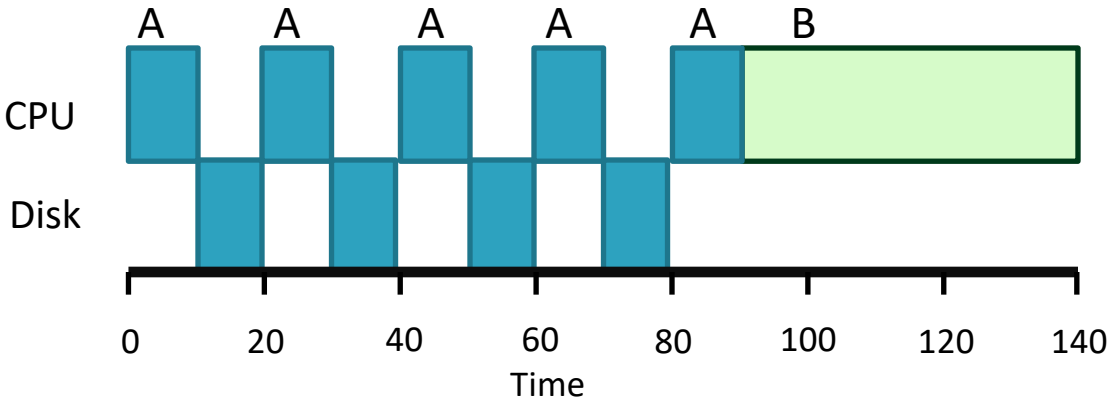
~~•Each job runs for the same amount of time~~
~~•All jobs started at the same time~~
~~•Once started, each job runs until completion~~
~~•All jobs only use the CPU (e.g. no I/O)~~
•The run-time of each job is known

- When a job initiates an I/O request, OS has a decision to make.
  - Currently running process won't be using the CPU.
  - It will be blocked waiting for I/O completion.
  - Usually I/O operations take a long time.
    - E.g. Hard disk drive might block the process for a few milliseconds or longer.
- There is also a decision to make when the I/O completes.
  - The requesting job returns to ready state → should the OS run the job at that point?

Incorporating I/O

- Let's see an example:
  - Two jobs A and B arriving at t=0.
  - Both need 50ms of CPU time.
  - A issues an I/O request every 10ms.

-> STCF runs A

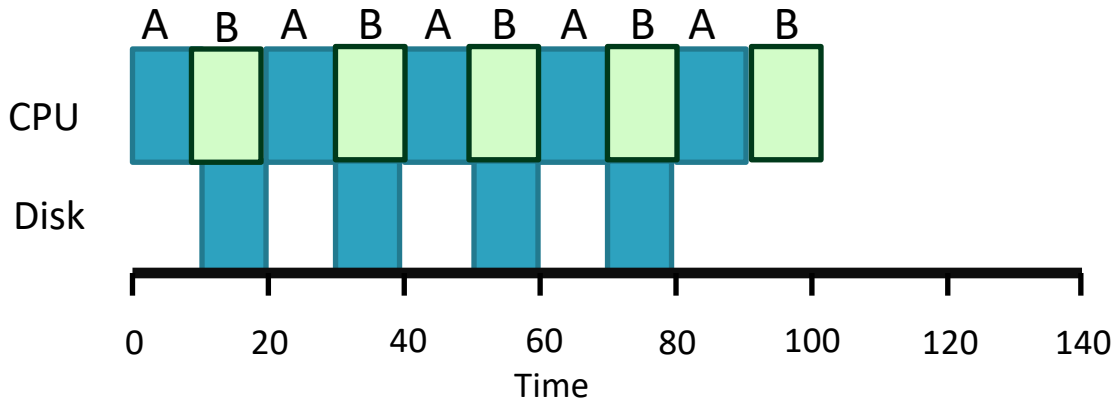


$T_{turnaround}$=(90+140)/2=110 ms
$T_{response}$=(0+90)/2=45 ms

- Just running one job after the other without considering I/O makes little sense.

Incorporating I/O

- A common approach → treat each 10ms A sub-job as an independent job. → apply STCF
  - Doing so allows overlapping another job with the I/O



$T_{turnaround}$=(90+100)/2=95 ms
$T_{response}$=(0+10)/2=5 ms

Workload Assumptions

- Some assumptions about the processes running in the system (workload)
  - ~~Each job runs for the same amount of time~~
  - ~~All jobs started at the same time~~
  - ~~Once started, each job runs until completion~~
  - ~~All jobs only use the CPU (e.g. no I/O)~~
  - ~~The run-time of each job is known~~

- No more Oracle.
  - The OS knows little about the length of each job.
  - How to behave like SJF/STCF without a priori knowledge?
  - How can we incorporate RR scheduler ideas to improve response time?
- OS still needs to decide which job to run at any given moment?

2.4 Virtualizing the CPU
Multi-level Feedback Queue

Multi-Level Feedback Queue

- How to schedule without perfect knowledge?
  - Priority Scheduling.

- One of the most well-known approaches to scheduling
  → Multi-level Feedback Queue (MLFQ)
  - First described by Corbato et. al. in 1962.
  - Subsequently been refined through the years.
  - Many different implementations in some modern systems.
  - We will cover the basic algorithms.

- MLFQ learns from the past to predict the future.

Multi-Level Feedback Queue

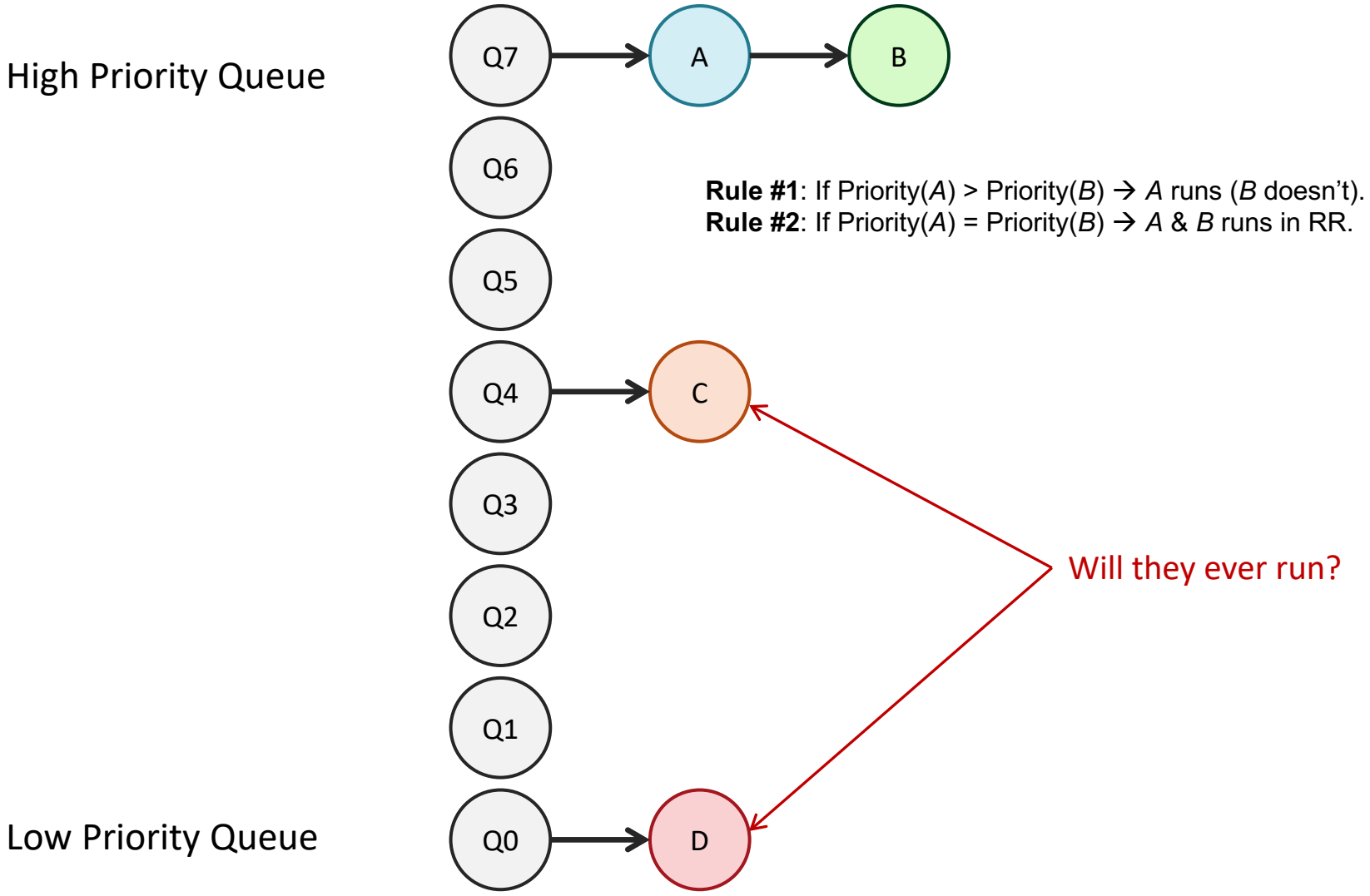
- MLFQ has a number of distinct queues, each assigned a different priority level.
- At any given time, a job that is ready to run is on a single queue.
- MLFQ uses priorities to decide which process to run at a given time:
  - A job with higher priority is chosen to run.
- More than one job could be in a given queue:
  - Same priority → e.g. Round Robin.

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

- How to assign priorities?
  - Not fixed → Observed behavior.
    - E.g. A job continuously requesting Keyboard input keeps its priority high (interactive process)
    - E.g. A CPU demanding job reduces its priority
  - Use history to predict future behavior.

Multi-Level Feedback Queue



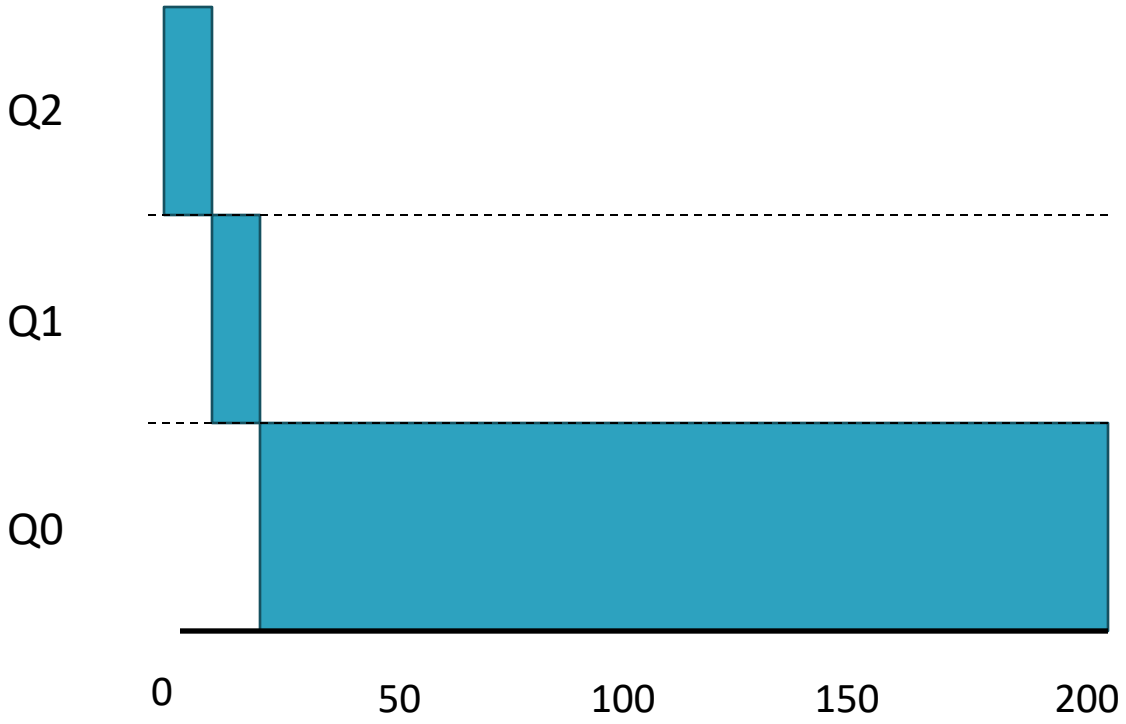
Multi-Level Feedback Queue

- Priority Should Change over time
  - We must decide how MLFQ changes the priority level of a job (which queue it is on) over its lifetime.
  - Keep in mind our workload:
    - Interactive jobs → short-running (may release CPU frequently)
    - CPU-bound jobs → long-running (response time not important)
  - First attempt:

Rule #3: When a job **enters** the system → assign **highest** priority
Rule #4a: If a job uses the CPU an entire time slice → **reduce** priority
Rule #4b: If a job gives up the CPU before time slice is up → **keep** priority

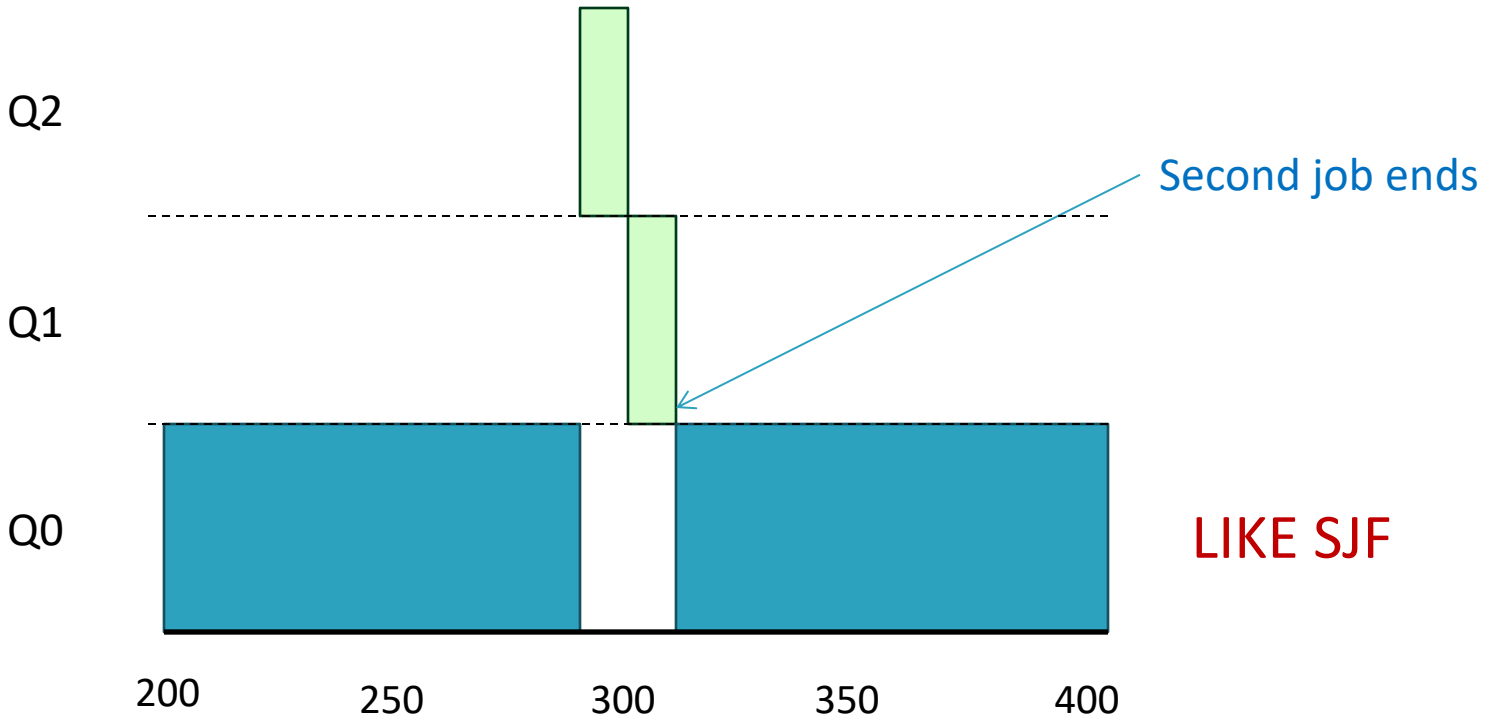
Multi-Level Feedback Queue

- Example #1: Single Long-Running Job



Multi-Level Feedback Queue

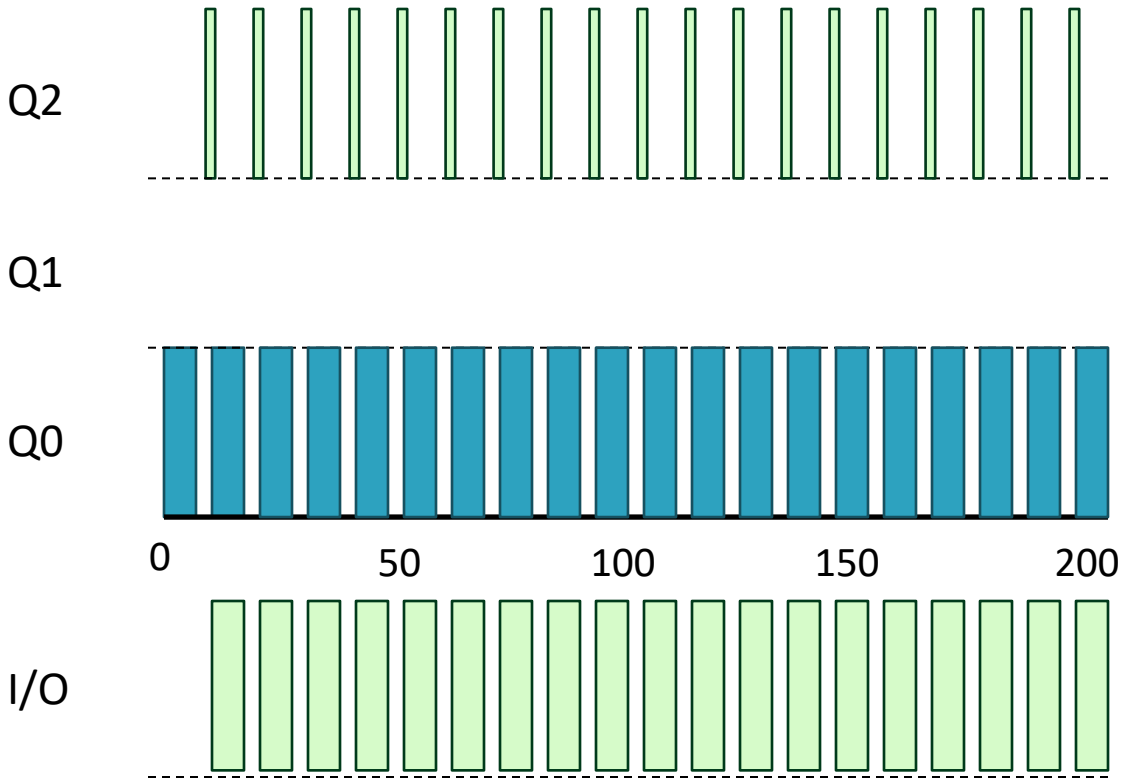
- Example #2: Along Came a Short Job



Multi-Level Feedback Queue

- Example #3: What About I/O

Rule #4b: If a job gives up the CPU before time slice is up → **keep** priority



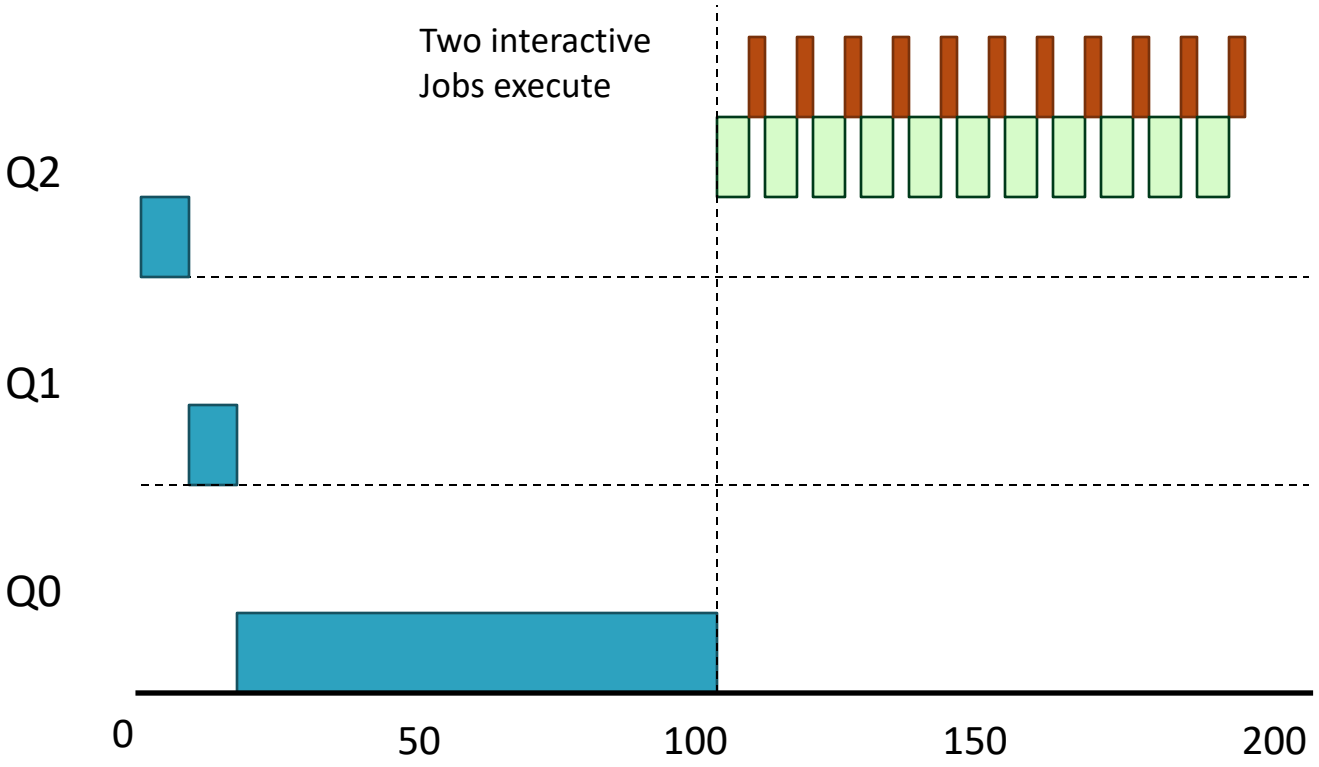
Interactive Job is handled quickly

Multi-Level Feedback Queue

- What happens if there are “too many” interactive jobs in the system?
  - Starvation: They will combine and never give up the CPU to long-running jobs.
- Smart users could rewrite their programs to game the scheduler → Trick the scheduler to give them more resources than the fair share.
  - E.g. Run 99% of the time slice and give up the CPU.
- A program may change its behavior over time
  - E.g. a long running program becomes interactive

Multi-Level Feedback Queue

- Example #4a:



Multi-Level Feedback Queue

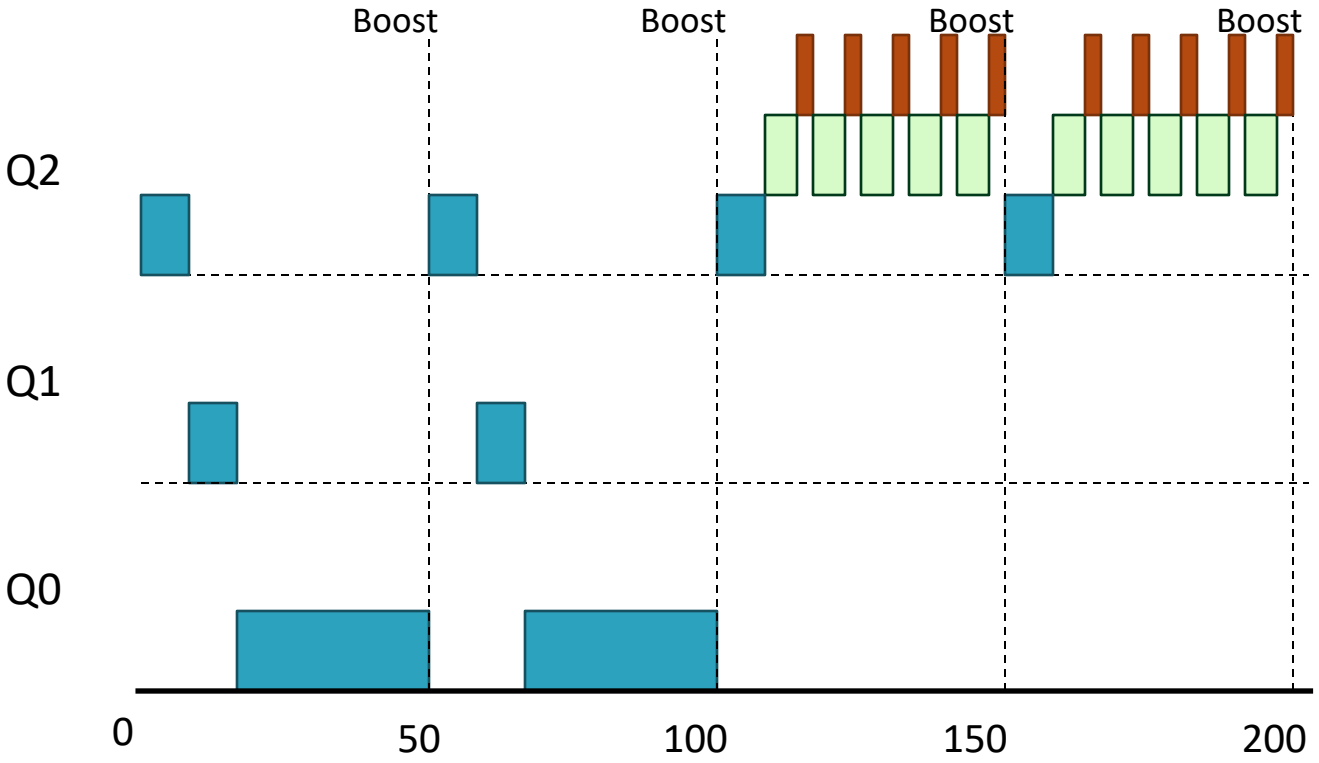
- Attempt #2: Priority Boost
  - Periodically give a boost to all the jobs. E.g.:

    Rule #5: After some time period *S* → move all jobs to topmost queue

- Solves two problems at once:
  - Processes do not starve.
  - If a job changes behavior, scheduler treats it properly.

Multi-Level Feedback Queue

- Example #4b: With Priority Boost



Multi-Level Feedback Queue

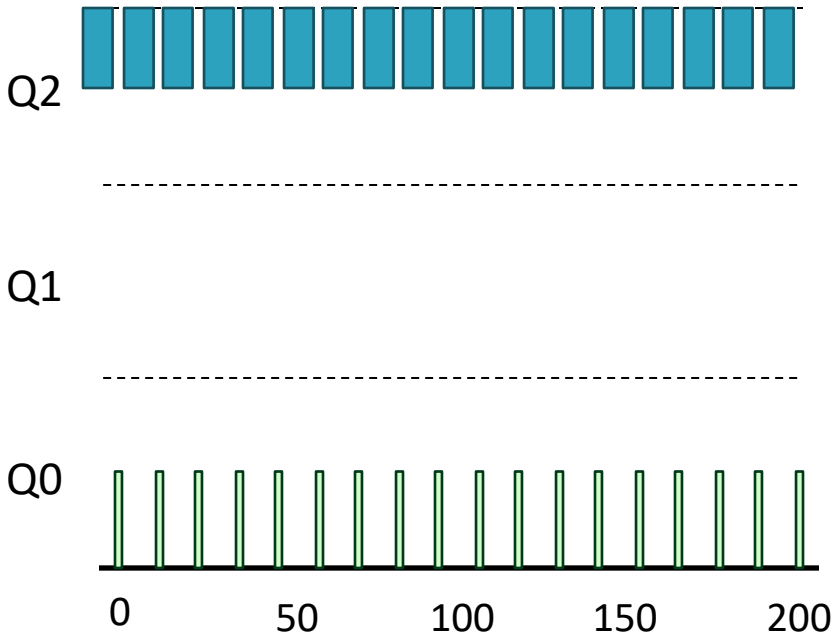
- Attempt #3: Better Accounting
  - How to prevent gaming of our scheduler?
    - Rules 4a and 4b let a job retain its priority even using the CPU 99% of the time slice.
  - Solution: Perform a better accounting of CPU time:
    - Scheduler does not forget how much of the time slice a process used → keeps track
    - Once a process used its allotment, it is demoted

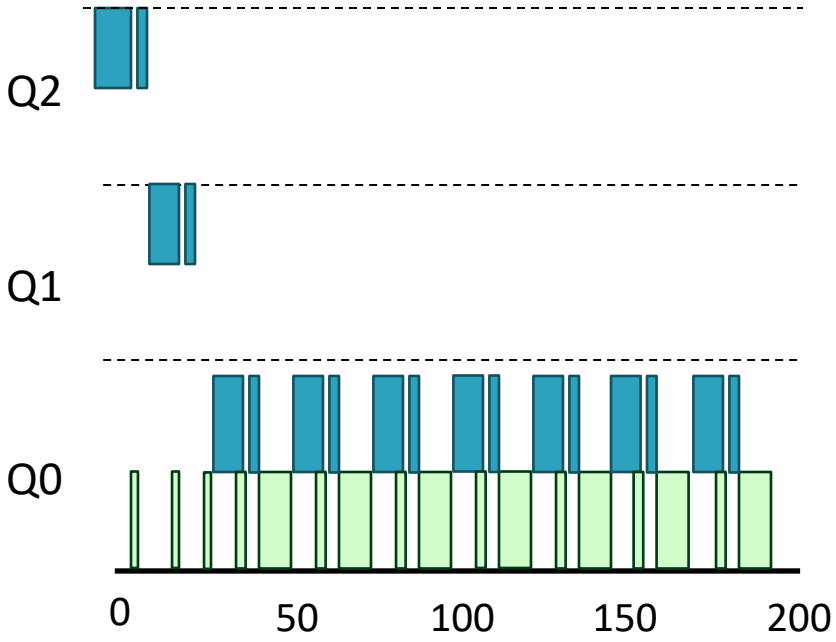
Rule #4: Once a job uses its time allotment at a given level → reduce priority

  - CPU usage accumulated, regardless of how many times it gives up the CPU.
  - Each Queue can have its own time quanta

Multi-Level Feedback Queue

- Example #5: Without (Left) and With (Right) Gaming Tolerance





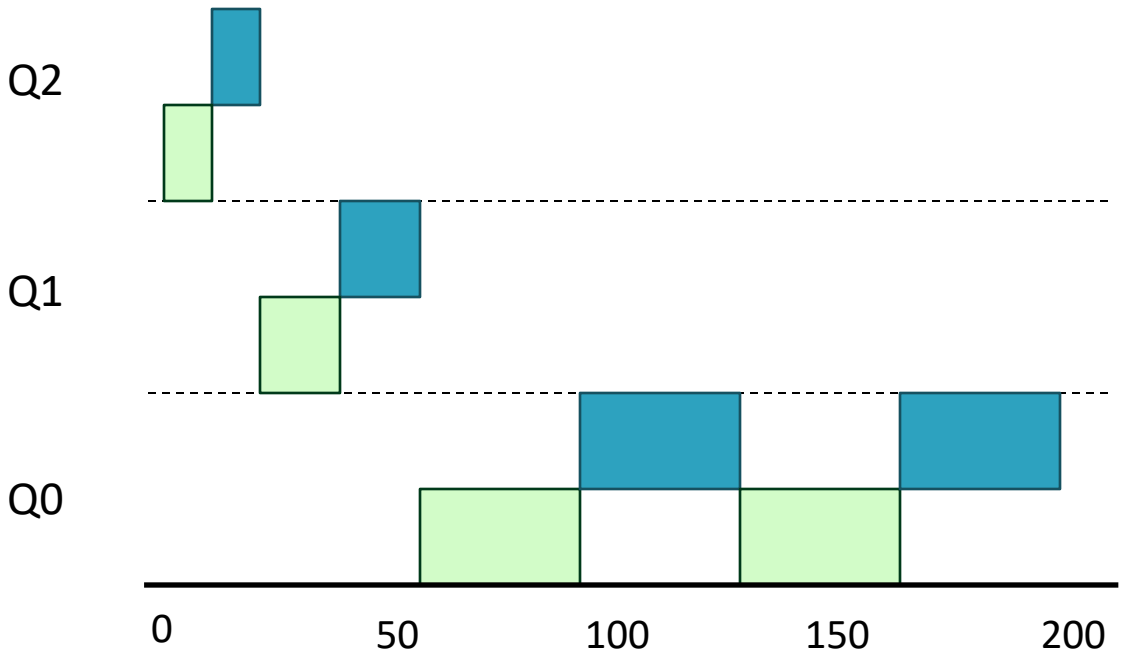
Multi-Level Feedback Queue

- MLFQ Parameterization
  - How many queues?
  - How big the time-slice of each queue?
  - How often should we boost priority?

- Most MLFQ allow varying time-slice length across queues:
  - High-priority queues → short time slices
    - interaction → fast switching
  - Low-priority queues → longer time slices
    - CPU-bound jobs

Multi-Level Feedback Queue

- Example #6: Lower Priority, Longer Quanta



Multi-Level Feedback Queue

- Rules set 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 gets highest priority.
  Rule #4: Once a job uses its time allotment at a given level → reduce priority.
  Rule #5: After some period S, all jobs move to highest priority queue.

- Multi-Level Feedback Queue:
  - Does not need a priori knowledge, but uses history.
  - Can deliver excellent overall performance (similar to SJF/STCF)
  - Is fair and all jobs make progress.

MLFQ Examples

- Solaris MLFQ provides a set of tables with the rules explained, modifiable by administrator.
  - Default: 60 queues, 20 ms time-slice high priority, 100s ms low priority, priority boost every second…
- FreeBSD MLFQ scheduler uses a mathematical formula to calculate the priority of a job.
- Windows uses a priority-based preemptive scheduling
  - Higher priority threads preempt lower running threads.
  - Increases priority of the job selected on the screen (foreground process) from others (background processes).
  - Windows 7 allows user-mode scheduling (UMS): Applications manage thread scheduling without involving kernel.
- Many schedulers offer other features:
  - Highest priority devoted to system (no user)
  - User advice to set priority (nice)

2.5 Virtualizing the CPU
-Other Scheduling

Proportional Share Scheduling

- Instead of Turnaround or response time → Proportional use of CPU.

- Proportional-share scheduling (Lottery scheduling)
  - Hold a “lottery” to determine the running process.
  - Tickets → a share of the resource.
  - Random choice
    - Probability leads to proportionality (no guarantee).
    - Avoid corner-case behavior.
    - Random could be fast.

Proportional Share Scheduling

- Example #7: Lottery Scheduler
  - Two processes A and B. A has 75 tickets, B only 25.
  - Lottery from 0 to 99. A wins from 0 to 74

Lottery result	63	85	70	39	76	17	29	41	36	39
A	X		X	X		X	X	X	X	X
B		X			X					

  - B gets to run 2 out of 10 time slices (20%), which is a bit far from the 25% we expect.
  - The longer the processes run, the more likely they match the percentages.

Stride Scheduling

- Why do it randomly?
  - Deterministic Proportional-share Scheduling.

- Divide a large number by the number of tickets → stride.
  - Whenever a process executes, an execution counter (pass) is incremented by the stride.

Stride Scheduling

- Example #8: Three processes A, B and C with 100, 50 and 250 tickets respectively.
  - Divide 1000 by tickets → Strides: 10, 20 and 4.

Pass (A) (stride=10)	Pass (B) (stride=20)	Pass (C) (stride=4)	Who runs?
0	0	0	A
10	0	0	B
10	20	0	C
10	20	4	C
10	20	8	C
10	20	12	A
20	20	12	C
20	20	16	C
20	20	20	-

Lottery vs. Stride

- Lottery Scheduler gets proportion probabilistically over time.
- Stride Scheduler achieves proportion (exactly) just at the end of each scheduling cycle.
- Why use lottery scheduling at all?
  - There is no global state.
  - What if a process arrives in the middle of a scheduling cycle of the stride scheduler?
    - If it gets pass value = 0, it will monopolize the CPU for a while.
  - It is easier.
- Both cases: How to assign tickets?

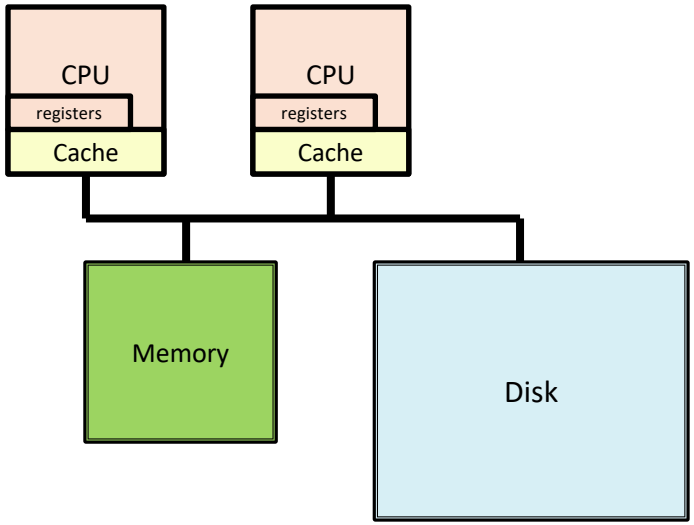
Real-Time Scheduling

- Event-driven system
  - E.g. Car collision detector.
  - Response time is critical (predictable)→ event latency.
  - Hard vs Soft real-time systems.
- Interrupt latency
  - Arrival of interrupt to start of service routine.
  - Critical on hard real-time systems.
- Dispatch Latency
  - Amount of time it takes to stop a process and start another one.
  - Preemptive kernel.
  - Preemption of processes running.
  - Release of resources needed by priority process.
- Keep the number of running processes low.

2.5 Virtualizing the CPU -Multiprocessor Scheduling

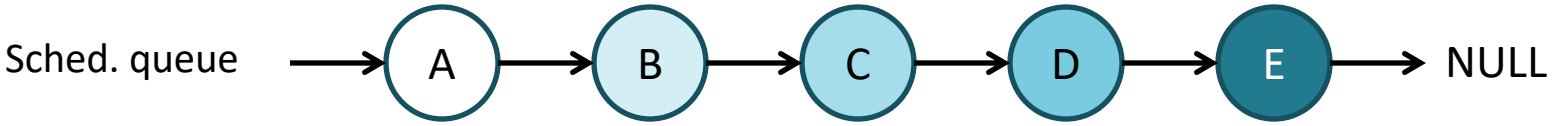
Mutliprocessor Issues

- Cache Hierarchy



- Data Coherence
  - Moving processes to different CPUs, what happens to the data?
- Processor Affinity
  - Cache affinity
  - TLB affinity

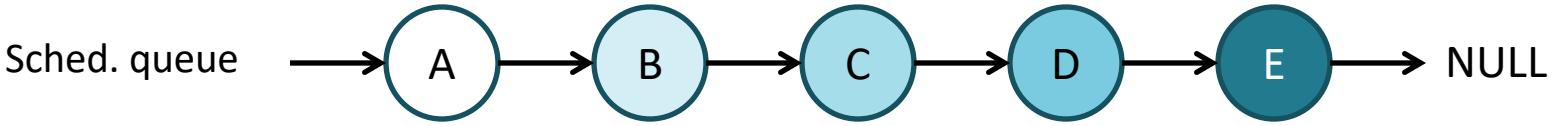
Single-Queue Scheduling



CPU 0	A	E	D	C	B	...
CPU 1	B	A	E	D	C	...
CPU 2	C	B	A	E	D	...
CPU 3	D	C	B	A	E	...

CPU Affinity!

Single-Queue Scheduling

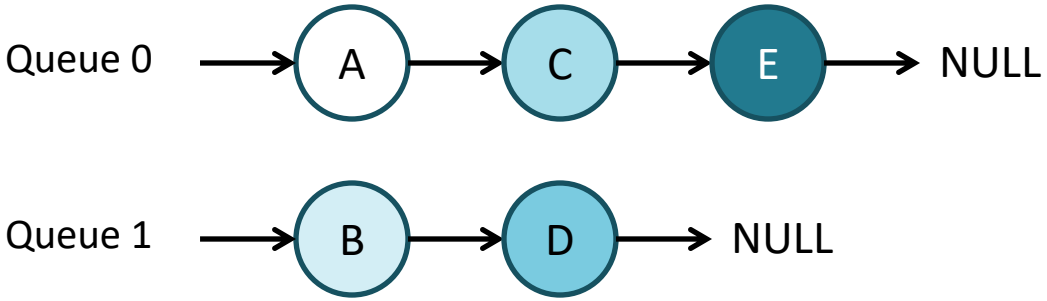


CPU 0	A	E	A	A	A	...
CPU 1	B	B	E	B	B	...
CPU 2	C	C	C	E	C	...
CPU 3	D	D	D	D	E	...

Affinity fairness...?

SQ shared -> concurrency

Multi-Queue Scheduling

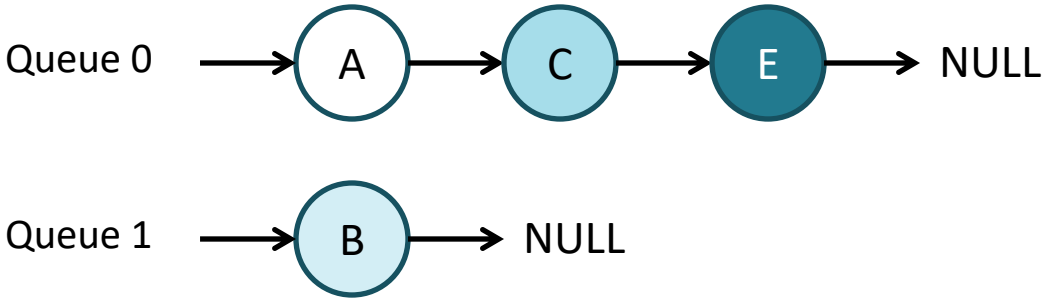


CPU 0	A	A	C	C	E	E	A	A	C	C	E	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	...

Load Imbalance

There is not much to do

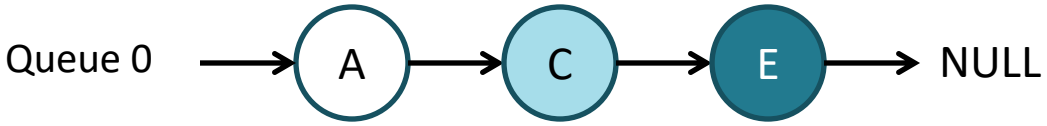
Multi-Queue Scheduling

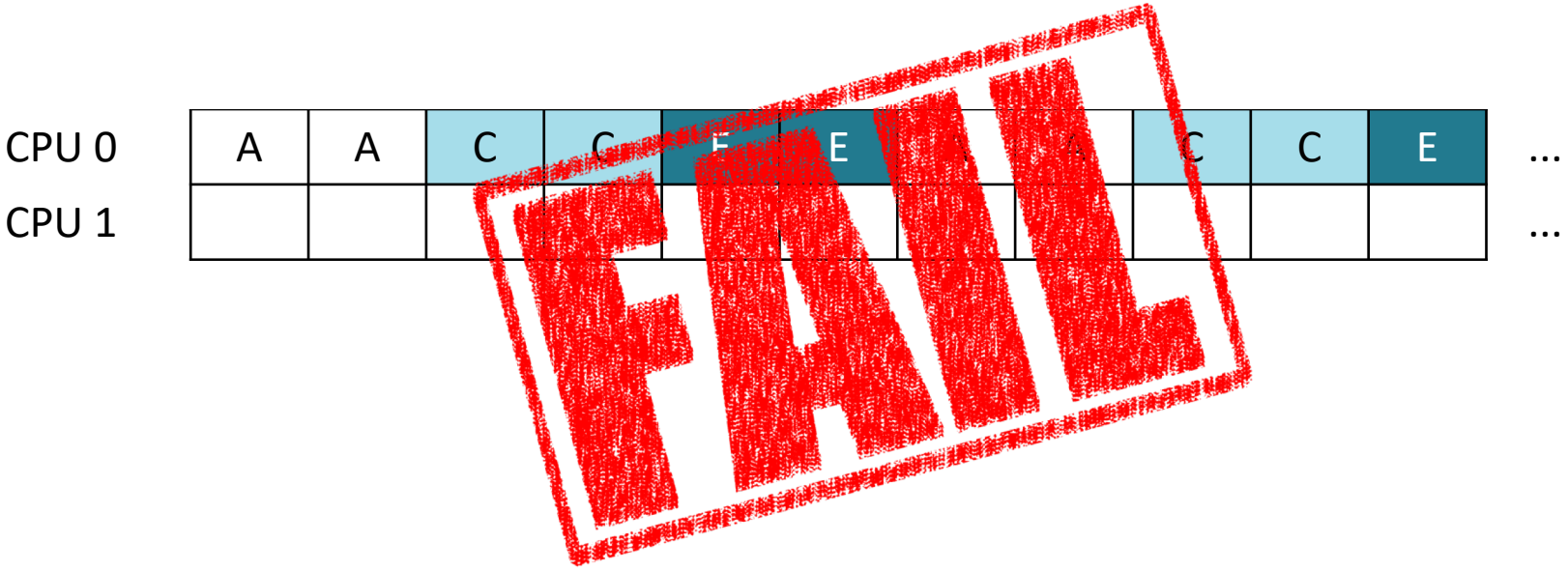


CPU 0	A	A	C	C	E	E	A	A	C	C	E	...
CPU 1	B	B	B	B	B	B	B	B	B	B	B	...

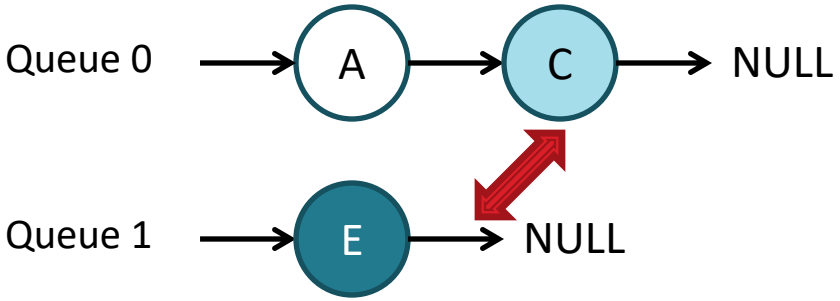
Even Worse!

Multi-Queue Scheduling





Multi-Queue Scheduling



CPU 0	A	A	C	C	A	A	A	A	C	C	A	...
CPU 1	E	E	E	E	C	C	E	E	E	E	C	...

Job Migration / Work Stealing

Multiprocessor Scheduling

- Asymmetric vs Symmetric → mostly SMP.
- Single-Queue Scheduling (SQMS)
  - Simplicity.
  - Lack of Scalability.
  - Concurrency → locks in SQMS accesses.
  - Migration →← Processor Affinity (cache).
- Multi-Queue Scheduling (MQMS)
  - Flexible: Each queue can have its own algorithm.
  - More scalable.
  - Load imbalance → work stealing (migration).
  - More complex.

Multiprocessor Scheduling

- Linux Multiprocessor Schedulers:
  - O(1)
    - Priority based (similar to MLFQ).
    - Focus on interactivity/performance → good response time.
    - MQMS with load balancing.
  - Completely Fair Scheduler (CFS)
    - Deterministic Proportional-share (uses nice).
    - Accumulates virtual run time (decay based on priority)
    - MQMS.
  - BF* Scheduler (BFS)
    - Single-Queue approach.
    - Proportional-share → Earliest Eligible Virtual Deadline First

2.7 Virtualizing the CPU -Process API

Application Programming Interface

- Application Program Interface (API) is what programs use to access OS system calls.
  - Most common APIs are:
    - Win32 for Windows;
    - POSIX for POSIX-based systems (almost all UNIX, Linux and Mac OS X);
    - JAVA API for the Java virtual Machine (JVM);
    - Graphic APIs like: DirectX, OpenGL…
  - Usually come in the form of libraries.

- Why should we use APIs rather than system calls?
  - The caller doesn't need to know anything about how the system call is implemented.
    - Just needs to obey the API and understand what the OS will do as a result.
    - Most details of the OS are hidden to the programmer by the API.

Application Programming Interface

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shmget()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Process API

- We will discuss the UNIX POSIX API for process creation and control.
  - fork system call: creates a new process.
  - exec system call: replaces the process' memory space with a new program.
  - wait system call: waits for a process to finish.

The fork() System Call

- The `fork()` system call is used to create a new process.

```
#include <stdio.h>
pid_t fork(void);
```

- The new process is almost an exact copy of the calling process (parent).
- The created process (child) doesn't start running at main(), instead it starts running as if it had called the fork() itself.
- To the OS, now there are two copies of the same process that both return from the `fork()` system call.
- Each process has its own copy of the address space, its own registers and different PIDs.
  - The return value of the `fork()` is different for each process: the child is simply returned a 0, while the parent receives the PID of the newly-created process.
    - `pid_t getpid(void)` returns the PID of the calling process.
    - `pid_t getppid(void)` returns the PID of the parent of the calling process.

The fork() System Call

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  printf("Hello world (pid:%d)\n", (int)getpid());
  int proc_id = fork(); // process cloning
  if (proc_id < 0) //error at process creation
  {
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (proc_id == 0) //the child (new process)
  {
    printf("hello, I am the child (pid:%d)\n), (int)getpid());
  } else //the parent goes this path
  {
    printf("hello, I am the parent of %d (pid:%d)\n), proc_id, (int)getpid());
  }
  return 0;
}
```

The wait() System Call

- The `wait()` system call waits for termination of a child of the calling process:

```
#include <sys/wait.h>
pid_t wait(int *status);
```

  - In case of child termination, wait allows OS to release the resources associated (otherwise it becomes a "zombie").
    - If the parent terminates, they become adopted by init(1), which performs a wait.
  - If the child has already changed state, the call returns immediately. Otherwise, the calling process suspends execution until one of its process terminates.

- Other system calls (`waitpid()` and `waitid()`) allow tracking of state changes in child processes: child terminated, stopped or resumed.
- `waitpid()` system call allows specifying the pid of the child you want to wait for.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

The wait() System Call

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
  printf("Hello world (pid:%d)\n", (int)getpid());
  int proc_id = fork(); // process cloning
  if (proc_id < 0) //error at process creation
  {
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (proc_id == 0) //the child (new process)
  {
    printf("hello, I am the child (pid:%d)\n), (int)getpid());
  } else //the parent goes this path
  {
    int wc = wait(NULL);
    printf("hello, I am the parent of %d (wc:%d) (pid:%d)\n),
           proc_id, wc, (int)getpid());
  }
  return 0;
}
```

The exec() System Call

- The `exec()` system call is useful when you want to run a program different from the calling program.
- It loads code (and static data) from the executable and overwrites the current code segment (and current static data).
  - The heap, stack and other parts of the memory space are re-initialized.
- Then the OS runs the program passing the corresponding arguments.
  - It does not create a new process, it transforms the current process into a different running program.
  - A successful call to `exec()` never returns.
- There is a family of `exec()` functions:

```
#include <unistd.h>

int execl(const char *path, const char *arg, …);
int execlp(const char *file, const char *arg, …);
int execle(const char *path, const char *arg, …, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

The exec() System Call

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char *argv[])
{
  printf("Hello world (pid:%d)\n", (int)getpid());
  int proc_id = fork(); // process cloning
  if (proc_id < 0) //error at process creation
  {
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (proc_id == 0) //the child (new process)
  {
    printf("hello, I am the child (pid:%d)\n), (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");     //program: wc (word count)
    myargs[1] = strdup("exec_call.c"); // argument: file
    myargs[2] = NULL;             // marks end of array
    execvp(myargs[0], myargs); //runs word count
    printf("this should not print out");
  } else //the parent goes this path
  {
    int wc = wait(NULL);
    printf("hello, I am the parent of %d (wc:%d) (pid:%d)\n),
            proc_id, wc, (int)getpid());
  }
  return 0;
}
```

Process API and the Shell

- The shell is just a user program. It shows a prompt and waits for a command from the user.

- In most cases:
  - The shell figures out where the command is in the file system;
  - Calls `fork()` to create a new child process to run the command;
  - Calls some variant of `exec()` to run the command;
  - Then waits for the command to complete by calling `wait()` and prints out the prompt again.

- The separation of `fork()` and `exec()` allows the shell to do work in between.

2 Virtualizing the CPU
-Exercises

Exercise #1

Three jobs with the following characteristics are running in a system:

Job	Start time	CPU time	I/O frequency	Duration of I/O
A	0ms	60ms	every 20ms of CPU	10ms
B	5ms	50ms	no I/O	-
C	15ms	40ms	every 10ms of CPU	5ms

a) Draw the behavior of the scheduling algorithm for First-In First-Out (**FIFO**), Sortest Time to Completion First (**STCF**) and Round Robin (**RR**) with 10 ms of time slice. Bear in mind that **STCF** is preemptive and that each time a process comes into ready state, the algorithm is reevaluated.

b) What is the turnaround time of each job for the different algorithms? What is the average turnaround time of each algorithm?

c) What is the response time of each job for the different algorithms? What is the average response time of each algorithm?

Exercise #2

There is an Operating System with an MLFQ (Multi-Level Feedback Queue) Scheduler with three priority queues (Q1, Q2 and Q3). The **time slice** for each queue is 4ms in Q1, 8ms in Q2 and 12ms in Q3 respectively. Every 60ms there is a priority boost. The three jobs are using the same I/O device that can only be used by one job at a time. Assume that the context switch time is zero for this exercise. If there are three jobs running (A, B and C) with the following characteristics, draw the execution trace and obtain the turnaround time and the response time of each job.

A:	B:	C:
Starts at t=0ms. Requires 20 ms of CPU. I/O every 4ms of CPU and each I/O operation lasts for 12ms.	Starts at t=2ms. Requires 40ms of CPU. I/O every 12ms of CPU and each I/O operation lasts for 12ms.	Starts at t=6ms. Requires 60ms of CPU. No I/O.

Exercise #3

Three jobs with the following characteristics are running in an operating system with a Deterministic Proportional-share Scheduler (Stride Scheduler):

Job A runs for 60 seconds with priority 20

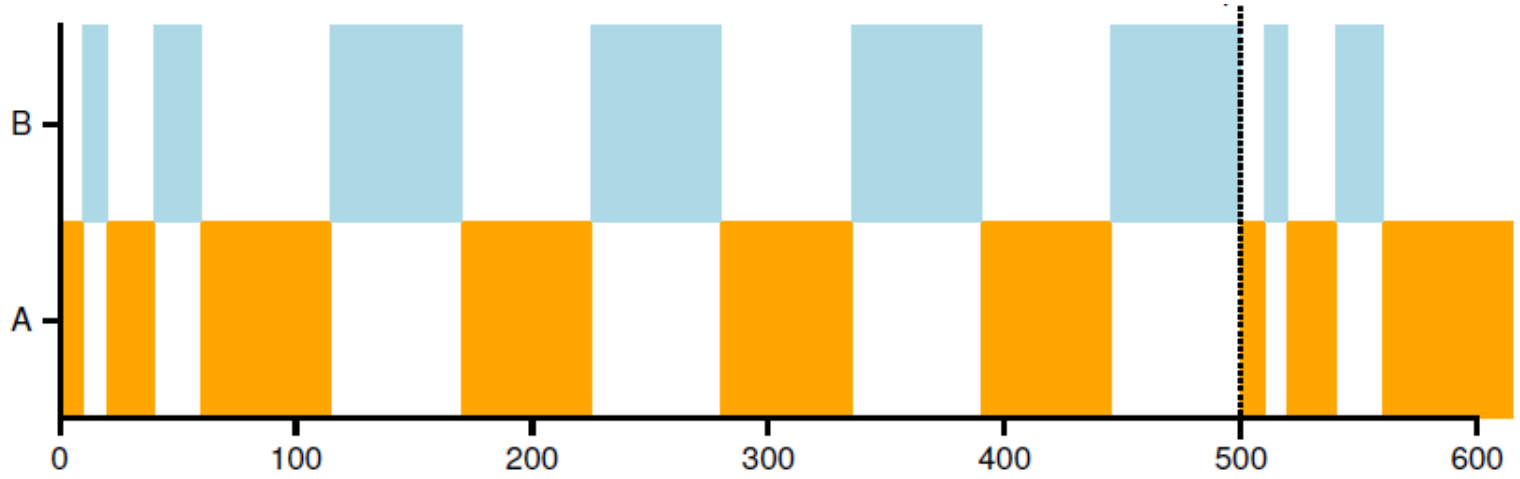
Job B runs for 80 seconds with priority 15

Job C runs for 90 seconds with priority 40

The scheduler runs a job for a time slice of 10 seconds before starting the next job. Draw the execution trace and keep track of the execution time accumulated (pass) for each process. Explain how the scheduler works and the steps you take.

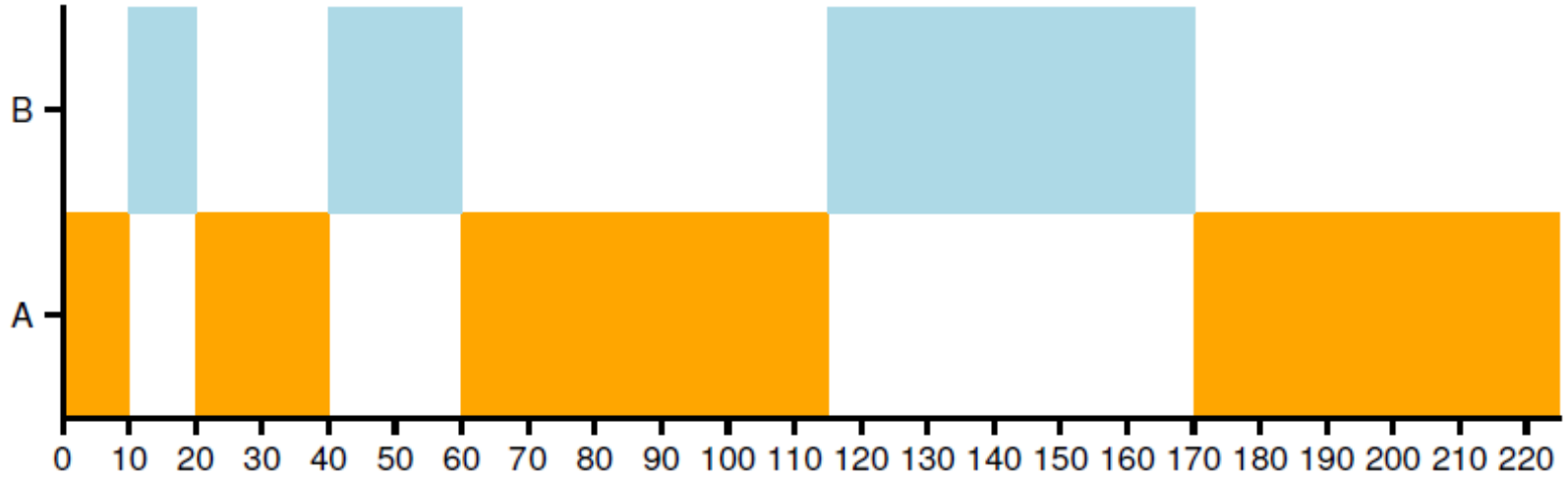
Exercise #4

There is an Operating System with an MLFQ (Multi-Level Feedback Queue) Scheduler. Unfortunately, all of its parameters have been lost. To obtain the most important ones, we execute two processes simultaneously, which only make use of the CPU (no I/O). Here is the execution trace of the two processes (A and B) running:



The graph shows the CPU usage of A and B over time (there is just one CPU in the system). Time scale (horizontal) is in milliseconds, and we can see how every 500 milliseconds the behaviour repeats indefinitely (until the jobs are finished). To help you further, a close-up of the first part of the graph is shown here in detail:

Exercise #4 (cont.)



a) How many queues do you think there are in this MLFQ scheduler?

b) How long is the time slice of the top-most (high priority) queue?

c) How long is the time slice of the bottom-most (lower priority) queue?

d) How often do processes get moved back to the top-most queue (priority boost)?

e) What is the point of the priority boost in the MLFQ scheduler?