CSE 323: Operating System Design

Virtual CPU (Scheduling)

Salman Shamil

North South University (NSU)

Fall 2025

Original slides by Mathias Payer and Sanidhya Kashyap [EPFL]

Lecture Topics

Scheduling has two aspects:

1. How to switch from one process to another?
2. What process should run next?

Lecture Topics

Scheduling has two aspects:

1. How to switch from one process to another?
2. What process should run next?

Divide-and-conquer by OS:

- Mechanism: context switch (how to switch)
- Mechanism: preemption (keeping control)
- Policy: scheduling (where to switch to)
    - [***we discuss this first...***]

This slide deck covers chapters 7–10 in OSTEP.

What is a Scheduling ***Policy***?

The context switch *mechanism* will take care of ***how*** the kernel switches from one process to another, namely by storing its context and restoring the context of the other process.

The scheduling policy determines ***which*** process should run next. If there is only one "ready" process then the answer is easy. If there are more processes then the policy decides in which order processes execute.

Scheduler Metrics

When analyzing scheduler policies, we use the following terms:

Metric	Definition	Goal
Utilization		
Turnaround time		
Response time		
Fairness		
Progress		

Scheduler Metrics

When analyzing scheduler policies, we use the following terms:

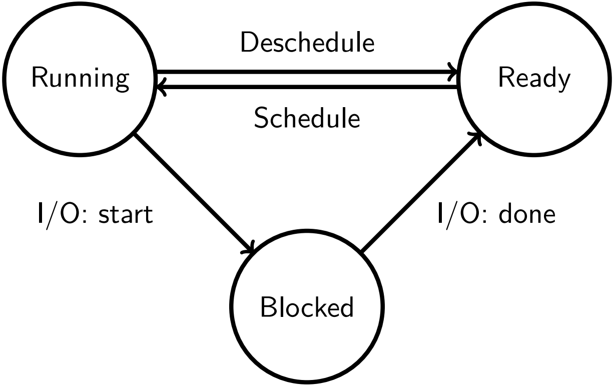
Metric	**Definition**	**Goal**
Utilization	what fraction of time is the CPU executing a program	
Turnaround time	total global time from process creation to process exit	
Response time	time from becoming ready to being scheduled	
Fairness	all processes get a fair share of CPU over time	
Progress	allow processes to make forward progress	

Scheduler Metrics

When analyzing scheduler policies, we use the following terms:

Metric	Definition	Goal
Utilization	what fraction of time is the CPU executing a program	maximize
Turnaround time	total global time from process creation to process exit	minimize
Response time	time from becoming ready to being scheduled	minimize
Fairness	all processes get a fair share of CPU over time	no starvation
Progress	allow processes to make forward progress	minimize kernel interrupts

Reminder: Process States



Scheduling Assumptions

Let's understand scheduler policies step by step. We start with some simplifying assumptions

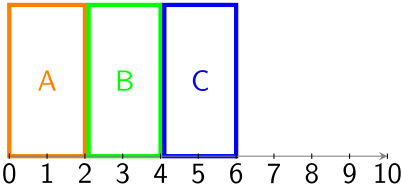
- Each job runs for the same amount of time
- All jobs arrive at the same time
- All jobs only use the CPU (no I/O)
- Run-time of jobs is known
- For now, we assume a single CPU

First In, First Out (FIFO)

- Tasks A, B, C of `len=2` arrive at `T=0` (0,2)

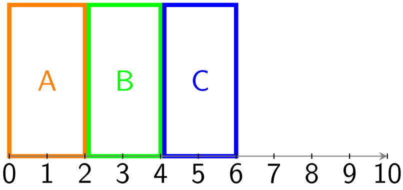
First In, First Out (FIFO)

- Tasks A, B, C of `len=2` arrive at `T=0` (0,2)
- Average turnaround
  - `(2+4+6)/3 = 4`
- Average response
  - `(0+2+4)/3 = 2`



First In, First Out (FIFO)

- Tasks A, B, C of `len=2` arrive at `T=0` (0,2)
- Average turnaround
  - `(2+4+6)/3 = 4`
- Average response
  - `(0+2+4)/3 = 2`



Finding: easy, simple, straight forward. What are drawbacks?

Scheduling Assumptions

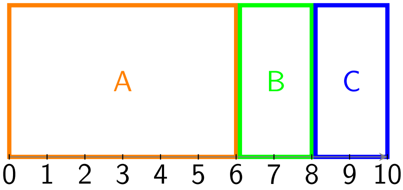
- ~~***Each job runs for the same amount of time***~~
- All jobs arrive at the same time
- All jobs only use the CPU (no I/O)
- Run-time of jobs is known

FIFO challenge: long running task

- Task A is now of `len=6`

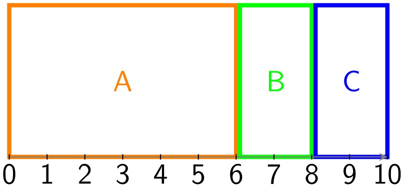
FIFO challenge: long running task

- Task A is now of `len=6`
- Average turnaround
  - `(6+8+10)/3 = 8`
- Average response
  - `(0+6+8)/3 = 4.7`



FIFO challenge: long running task

- Task A is now of `len=6`
- Average turnaround
  - `(6+8+10)/3 = 8`
- Average response
  - `(0+6+8)/3 = 4.7`

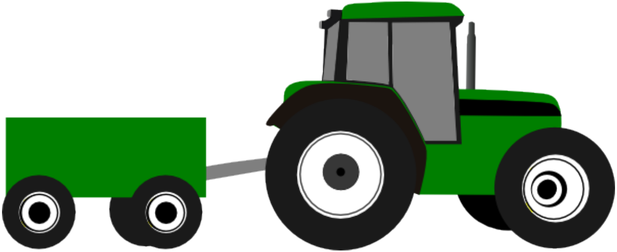


Finding: long jobs delay short jobs, turnaround/response time suffer!

SJF: Shortest Job First

- Long running tasks delay other tasks (convoy effect: one long running task delays many short running tasks like a truck followed by many cars)
- Short jobs must wait for completion of long task

New scheduler: choose ready job with shortest runtime!

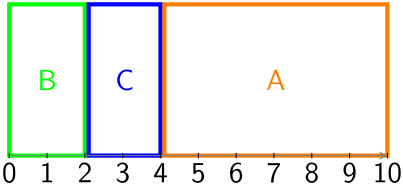


SJF: turnaround

- Task A is now of `len=6`

SJF: turnaround

- Task A is now of `len=6`
- Average turnaround
  - `(2+4+10)/3 = 5.3`
- Average response
  - `(0+2+4)/3 = 2`



Scheduling Assumptions

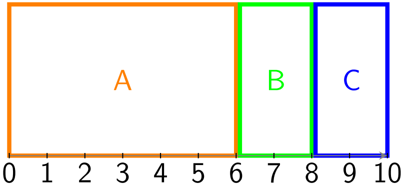
- ~~Each job runs for the same amount of time~~
- ~~***All jobs arrive at the same time***~~
- All jobs only use the CPU (no I/O)
- Run-time of jobs is known

SJF: another convoy!

- Tasks B, C now arrive at 1

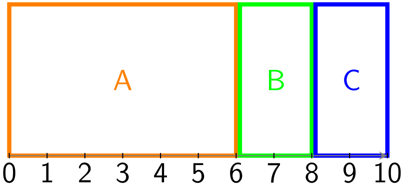
SJF: another convoy!

- Tasks B, C now arrive at 1
- Average turnaround
  - `(6+7+9)/3 = 7.3`
- Average response
  - `(0+5+7)/3 = 4`



SJF: another convoy!

- Tasks B, C now arrive at 1
- Average turnaround
  - `(6+7+9)/3 = 7.3`
- Average response
  - `(0+5+7)/3 = 4`



Finding: long running jobs cannot be interrupted, delay short jobs

Preemptive Scheduling

- Previous schedulers (FIFO, SJF) are non-preemptive.
  Non-preemptive schedulers only switch to another process if the current process gives up the CPU voluntarily.
- Preemptive schedulers may take CPU control at any time, switching to another process according to the scheduling policy.

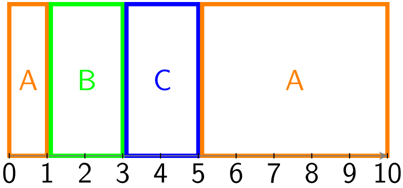
Preemptive Scheduling

- Previous schedulers (FIFO, SJF) are non-preemptive. Non-preemptive schedulers only switch to another process if the current process gives up the CPU voluntarily.
- Preemptive schedulers may take CPU control at any time, switching to another process according to the scheduling policy.
- New scheduler: Shortest Time to Completion First (**STCF**), always run the job that will complete the fastest.

- Tasks B, C now arrive at 1

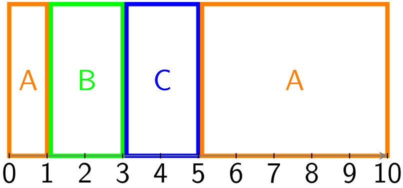
Preemptive Scheduling: STCF

- Tasks B, C now arrive at 1
- Average turnaround
  - `(2+4+10)/3 = 5.3`
- "First" response
  - `(0+0+2)/3 = 0.7`
  - Task A takes a break!



Preemptive Scheduling: STCF

- Tasks B, C now arrive at 1
- Average turnaround
  - `(2+4+10)/3 = 5.3`
- "First" response
  - `(0+0+2)/3 = 0.7`
  - Task A takes a break!



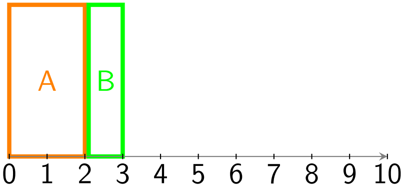
Finding: reschedule whenever new jobs arrive, prioritize short jobs

Next Metric: Response Time

- So far, we have optimized for turnaround time (i.e., completing the tasks as fast as possible).
- On an interactive system, response time is equally important, i.e., how long it takes until a task is scheduled.

Turnaround vs Response Time

- Tasks A (2,0) and B (1, 1)
- B turnaround: 2
- B response time: 1



Round Robin (RR)

- Previous schedulers optimize for turnaround.
- Optimize response time: alternate ready processes every fixed-length time slice.

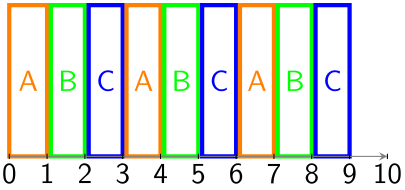
Round Robin (RR)

- Tasks A, B, C (0, 3)

Round Robin (RR)

- Tasks A, B, C (0, 3)
- Average response time
  - $(0+1+2)/3 = 1$
- Compare to FIFO where average response time is 3
- Turnaround increases

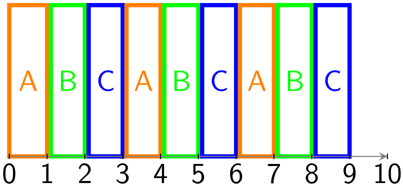
- $(7+8+9)/3 = 8$ for RR
  $(3+6+9)/3 = 6$ for SJF



Round Robin (RR)

- Tasks A, B, C (0, 3)
- Average response time
  - $(0+1+2)/3 = 1$
- Compare to FIFO where average response time is 3
- Turnaround increases

- $(7+8+9)/3 = 8$ for RR
  $(3+6+9)/3 = 6$ for SJF



Finding: responsiveness increases turnaround (for equally long tasks)

Scheduling Assumptions

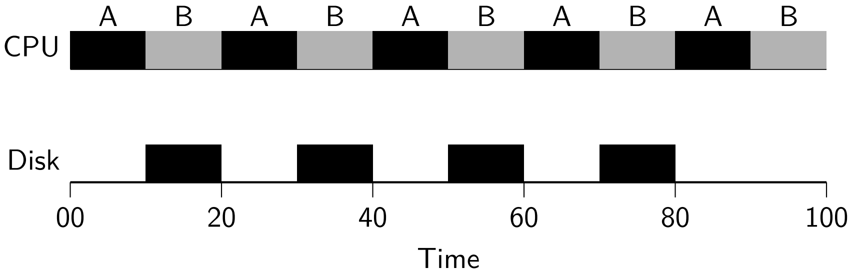
- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- ***~~All jobs only use the CPU (no I/O)~~***
- Run-time of jobs is known

I/O Awareness

- So far, the scheduler only considers preemptive events (i.e., the timer runs out) or process termination/creation to reschedule.
- I/O is usually incredibly slow and can be carried out asynchronously

I/O Awareness

- So far, the scheduler only considers preemptive events (i.e., the timer runs out) or process termination/creation to reschedule.
- I/O is usually incredibly slow and can be carried out asynchronously



Finding: scheduler must consider I/O, unused time used by others

Scheduling Assumptions

- ~~Each job runs for the same amount of time~~
- ~~All jobs arrive at the same time~~
- ~~All jobs only use the CPU (no I/O)~~
- ~~***Run-time of jobs is known***~~

Advanced Scheduling: Multi-Level Feedback Queue (MLFQ)

- Goal: general purpose scheduling

Challenge: The scheduler must support both long running background tasks (batch processes) and low latency foreground tasks (interactive processes).

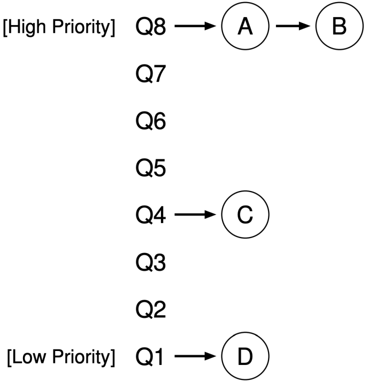
- Goal: general purpose scheduling

Challenge: The scheduler must support both long running background tasks (batch processes) and low latency foreground tasks (interactive processes).

- Batch process: response time not important, cares for long run times (reduce the cost of context switches, cares for lots of CPU, not when)
- Interactive process: response time critical, short bursts (context switching cost not important, not much CPU needed but frequently)

MLFQ: Basics

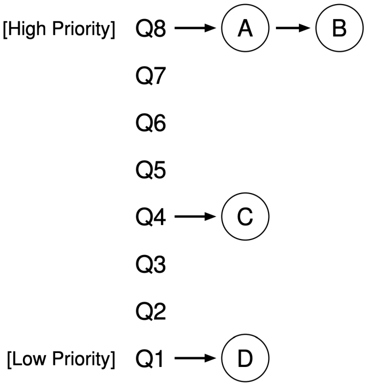
Approach: multiple levels of round robin (one queue per level)



- Each level has higher priority and preempts all lower levels
- Process at higher level will always be scheduled first
- Set of rules adjusts priorities dynamically

MLFQ: Basics

Approach: multiple levels of round robin (one queue per level)



- Each level has higher priority and preempts all lower levels
- Process at higher level will always be scheduled first
- Set of rules adjusts priorities dynamically

- Rule 1: if `prio(A) > prio(B)` then A runs.
- Rule 2: if `prio(A) == prio(B)` then A, B run in RR.

MLFQ: Priority Adjustments

Goal: use **past behavior** as predictor for future behavior.

MLFQ: Priority Adjustments

Goal: use **past behavior** as predictor for future behavior.

- Rule 3: processes start at top priority
- Rule 4: if process uses up full time slice, lower its priority
    - *keep at same level if it voluntarily yields (e.g., for I/O)*

MLFQ: Priority Adjustments

Goal: use **past behavior** as predictor for future behavior.

- Rule 3: processes start at top priority
- Rule 4: if process uses up full time slice, lower its priority
  - *keep at same level if it voluntarily yields (e.g., for I/O)*

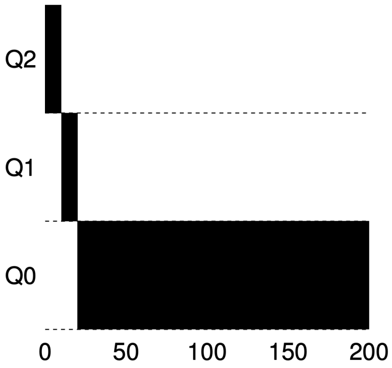


Figure 1: CPU-intensive job getting to the bottom queue over time

MLFQ: Serving Interactive Jobs

A short or interactive job may come later. Automatically gets higher priority with Rules 3-4 in place.

A short or interactive job may come later. Automatically gets higher priority with Rules 3-4 in place.

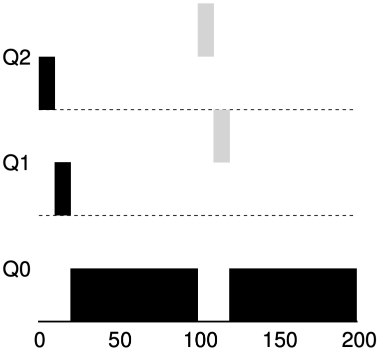


Figure 2: MLFQ Serving short or interactive jobs

MLFQ: Serving Interactive Jobs

A short or interactive job may come later. Automatically gets higher priority with Rules 3-4 in place.

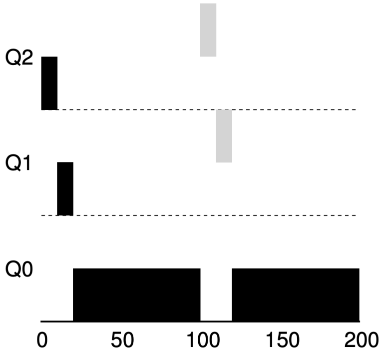


Figure 2: MLFQ Serving short or interactive jobs

All good? Do you see any problem?

MLFQ Challenges: Starvation

Low priority (long-running) tasks may never run on a busy system.

MLFQ Challenges: Starvation

Low priority (long-running) tasks may never run on a busy system.

- Rule 5: periodically move all jobs to the topmost queue

MLFQ Challenges: Starvation

Low priority (long-running) tasks may never run on a busy system.

- Rule 5: periodically move all jobs to the topmost queue

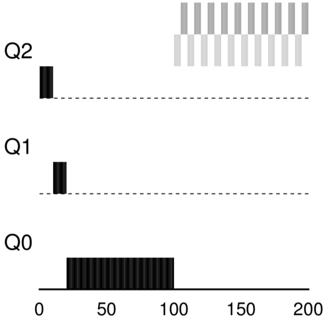


Figure 3: MLFQ prevents starvation via periodic priority boosts

MLFQ Challenges: Gaming the Scheduler

High priority process could yield before its time slice is up, remaining at high priority.

MLFQ Challenges: Gaming the Scheduler

High priority process could yield before its time slice is up, remaining at high priority.

- [Updated] Rule 4: account for total time at priority level (and not just time of the last time slice)

MLFQ Challenges: Gaming the Scheduler

High priority process could yield before its time slice is up, remaining at high priority.

- [Updated] Rule 4: account for total time at priority level (and not just time of the last time slice)

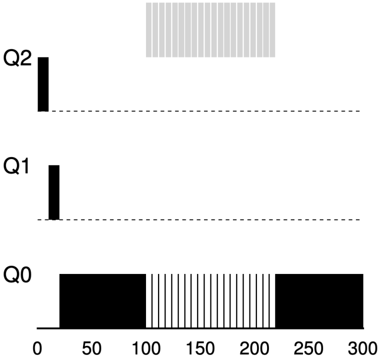


Figure 4: Impact of incorporating `Gaming Tolerance`

MLFQ: Serving CPU-bound and IO-bound Processes

- Interactive Processes: require quick responses and have short CPU bursts.
- Batch Processes: can tolerate delays but need long & uninterrupted CPU time.

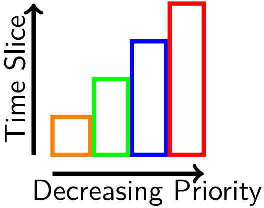
Remember where *context switching* can become costly?

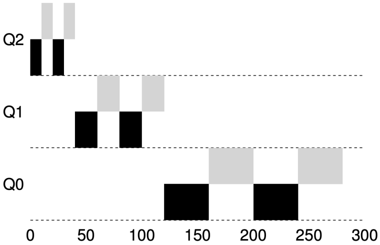
MLFQ: Serving CPU-bound and IO-bound Processes

- Interactive Processes: require quick responses and have short CPU bursts.
- Batch Processes: can tolerate delays but need long & uninterrupted CPU time.

Remember where *context switching* can become costly?

High levels have short time slices, lower levels run for longer





MLFQ Summary

- Rule 1: if `prio(A) > prio(B)` then A runs.
- Rule 2: if `prio(A) == prio(B)` A, B run in RR
- Rule 3: new processes start with top priority
- Rule 4: lower process' priority when whole time slice is used
- Rule 5: periodically move all jobs to the topmost queue

[Self-Study] More Scheduling Algorithms

Due to time constraints, we will stop with scheduling policies here.

For interested readers, I recommend exploring the following chapters.

- Scheduling: Proportional Share
  - Lottery Scheduling
  - Stride Scheduling
  - Completely Fair Scheduler (CFS)
- Multiprocessor Scheduling
  - Single-Queue Multiprocessor Scheduling (SQMS)
  - Multi-Queue Multiprocessor Scheduling (MQMS)

Scheduling Mechanisms

How does the kernel switch from one process to another?

- Context switch saves running process' state in kernel structure
- Context switch restores state of next process
- Context switch transfers control to next process and "returns"

Scheduling Mechanisms

How does the kernel switch from one process to another?

- Context switch saves running process' state in kernel structure
- Context switch restores state of next process
- Context switch transfers control to next process and "returns"

How does the kernel stay in control?

- Processes may `yield()` or execute I/O
- Configurable timer interrupts let OS take control

Scheduling Mechanisms

How does the kernel switch from one process to another?

- Context switch saves running process' state in kernel structure
- Context switch restores state of next process
- Context switch transfers control to next process and "returns"

How does the kernel stay in control?

- Processes may `yield()` or execute I/O
- Configurable timer interrupts let OS take control

Note: a context switch is ***transparent*** to the process

Mechanism: Context Switch

A context switch is a mechanism that allows the OS to store the current process state and switch to some other, previously stored context.

Reasons for a context switch:

- The process completes/exits
- The process executes a slow H/W operation (loading from disk) and the OS switches to another task that is ready
- The hardware requires OS help and issues an interrupt
- The OS decides to preempt the task and switch to another task (i.e., the processes has used up its time slice)

Mechanism: Preemption

If a task never gives up control (`yield()`), exits, or performs I/O then it could run forever and the OS could not gain control.

Mechanism: Preemption

If a task never gives up control (`yield()`), exits, or performs I/O then it could run forever and the OS could not gain control.

The OS therefore sets a timer before scheduling a process. If the timer expires, the hardware interrupts the execution of the process and switches to the kernel. The kernel then decides if the process may continue.

Summary

- Context switch and preemption are fundamental mechanisms that allow the OS to remain in control and to implement higher level scheduling policies.
- Schedulers need to optimize for different metrics: utilization, turnaround, response time, fairness and forward progress
    - FIFO: simple, non-preemptive scheduler
    - SJF: non-preemptive, prevents process jams
    - STCF: preemptive, prevents jams of late processes
    - RR: preemptive, great response time, bad turnaround
    - MLFQ: preemptive, more realistic
- Insight: past behavior is good predictor for future behavior