IE 411: Operating Systems

CPU Scheduling

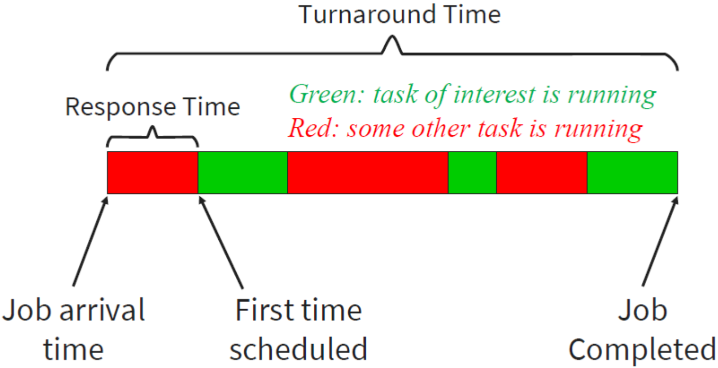
The scheduling problem

- Which process should the OS run?
  - if no runnable process (i.e. no process in the ready state), run the idle task
  - if a single process runable, run this one
  - if more than one runnable process, a scheduling decision must be taken
- Scheduler: code (logic) that decides which process to run as per some policy
- Today: What are the basic scheduling policies? When do they work well?

Standard usage

- Refer to schedulable entities as jobs — could be processes, threads, etc.
- Job: a task that needs a period of CPU time
- Job arrival time
  - when the job was first submitted
- Job run time
  - Time needed to run the task without contention

Scheduling metrics



- Execution time: sum of green periods
- Waiting time: sum of red periods
- Turnaround time: sum of both

Performance terminology

- Turnaround time: How long?
  - User-perceived time to complete some job (completion_time - arrival_time)
- Response time:
  - User-perceived time before first output (initial_schedule_time - arrival_time)
- Waiting time: How much thumb-twiddling?
  - Time on the ready queue (not running)

Performance terminology

- Throughput: How many jobs over time?
  - The rate at which jobs are completed
- Overhead: How much useless work?
  - Time lost due to switching between jobs
- Fairness: How equally are jobs treated?
  - Jobs get same amount of CPU time over some interval

Workload assumptions (to be relaxed later)

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

Scheduling basics

Workloads:

arrival_time
run_time

Scheduling Policies:

FIFO
SJF (SJN, SPN)
STCF
RR

Metrics:

turnaround_time
response_time

FIFO

- Run jobs to completion in arrival_time order
  - aka FCFS (first come first served)
- simple, minimal context switch overhead

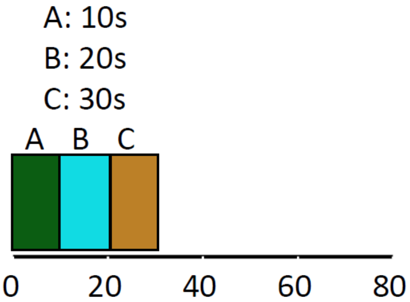
JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10

Time	
0	A arrives
0	B arrives
0	C arrives
0	run A
10	complete A
10	run B
20	complete B
20	run C
30	complete C

FIFO: Identical jobs

- Gantt chart: illustrates how jobs are scheduled over time on a CPU

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10



- What is the average turnaround time?
    - $(10+20+30)/3 = 20$s

Scheduling basics

Workloads:

- arrival_time
- run_time

Scheduling Policies:

- FIFO
- SJF (SJN, SPN)
- STCF
- RR

Metrics:

- turnaround_time
- response_time

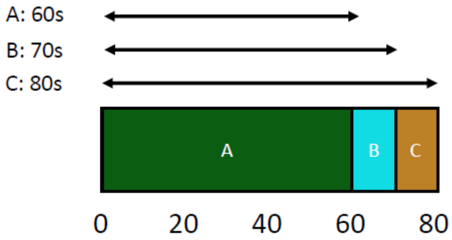
Workload assumptions

1. ~~Each job runs for the same amount of time~~
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

FIFO: Convoy effect

- Problem: turnaround time can suffer when short jobs must wait for long jobs

JOB	arrival_time (s)	run_time (s)
A	~0	60
B	~0	10
C	~0	10



Average turnaround time: 70s

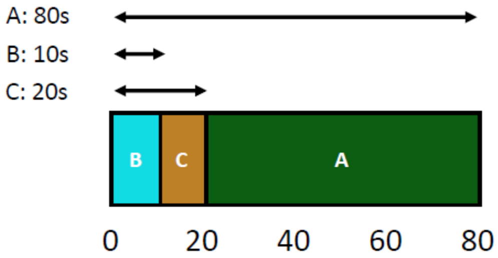
Shortest Job First (SJF)

- Idea: choose job with the smallest run_time
  - aka shortest job next (SJN), shortest process next (SPN)
- Consider our previous example

JOB	arrival_time (s)	run_time (s)
A	~0	60
B	~0	10
C	~0	10

What is the average turnaround time with SJF?

SJF turnaround time



- What is the average turnaround time?
    - $(80+10+20)/3 = 36.7$s
- Average turnaround with FIFO: 70s

Scheduling basics

Workloads:

- arrival_time
- run_time

Scheduling Policies:

- FIFO
- SJF (SJN, SPN)
- STCF
- RR

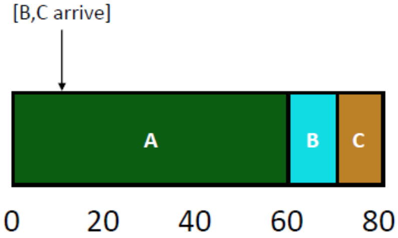
Metrics:

- turnaround_time
- response_time

Workload assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

SJF with late arrivals



JOB	arrival_time (s)	run_time (s)
A	~0	60
B	~10	10
C	~10	10

- What is the average turnaround time?
  - (60 + (70-10) + (80-10)) / 3 = 63.3s

Preemptive scheduling

- FIFO and SJF are non-preemptive
  - Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)
- Preemptive: potentially schedule different job at any point by taking CPU away from running job

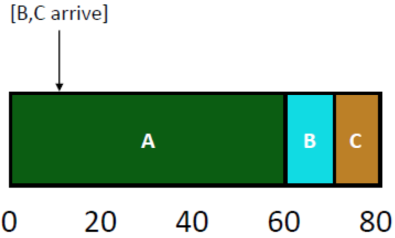
Shortest Time-to-Completion First (STCF)

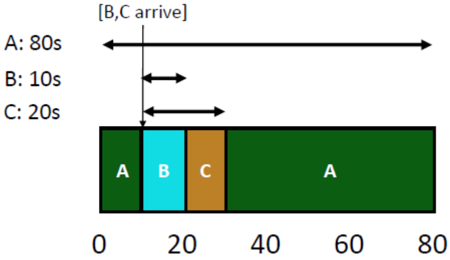
- SJF + preemption
- Select job with the least remaining time to run next
- Consider our previous example

JOB	arrival_time (s)	run_time (s)
A	~0	60
B	~10	10
C	~10	10

- Average turnaround time with STCF?

SJF vs. STCF





- Average turnaround time with STCF
  - $((80\text{-}0) + (20\text{-}10) + (30\text{-}10))/3 = 36.7s$
- Average turnaround time with SJF
  - 63.3s

Scheduling basics

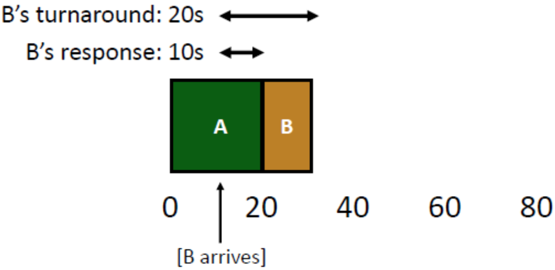
Workloads:
- arrival_time
- run_time

Scheduling Policies:
- FIFO
- SJF (SJN, SPN)
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Response vs. turnaround

- Response time: first run time - arrival time

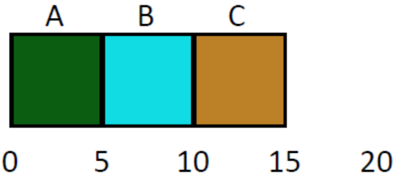


- FIFO, SJF, and STCF can have poor response time

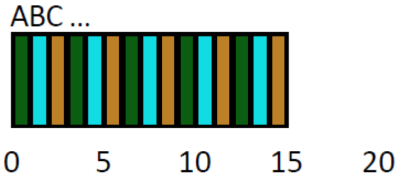
Round-Robin (RR)

- Each job allowed to run for a quantum
    - quantum = some configured period of time
- Context is switched (at the latest) at the end of the quantum – premption!
- Next job is the one on the ready queue that hasn't run for the longest amount of time

FIFO vs RR



Avg Response Time?
(0+5+10)/3 = 5



Avg Response Time?
(0+1+2)/3 = 1

RR Pros and Cons

- In what way is RR worse?
  - Avg. turnaround time with equal job length is horrible
- Other reasons why RR could be better?
  - If don't know run-time of each job, gives short jobs a chance to run and finish fast

Time quantum

- What is a good quantum size?
- Too long, and it morphs into FIFO
- Too short, and time is wasted on context switching
- Typical quantum: about 100X cost of context switch ($\sim$ 100 ms vs. $<<$ 1 ms)

Scheduling basics

Workloads:
- arrival_time
- run_time

Scheduling Policies:
- FIFO
- SJF (SJN, SPN)
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workload assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. Run-time of each job is known

Incorporating I/O

- When a job initiates an I/O request
  - The job is blocked waiting for I/O completion
  - The scheduler should schedule another job on the CPU
- When the I/O completes
  - An interrupt is raised
  - The OS moves the process from blocked back to the ready state

Workload assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. ~~Run-time of each job is known~~

(Need smarter, fancier scheduler)

Multi Level Feedback Queue (MLFQ)

- Goals: To schedule jobs, without knowing their lengths, so that we
  - minimize turnaround time
  - minimize response time (for interactive jobs)

Multi Level Feedback Queue (MLFQ)

- Goals: To schedule jobs, without knowing their lengths, so that we
  - minimize turnaround time
  - minimize response time (for interactive jobs)
- Invented by Fernando J. Corbato in 1962 – won ACM Turing Award
- Used by FreeBSD, Mac OS X, Solaris, Linux 2.6, Windows NT

MLFQ Basic Setup

- The 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 job should run at a given time

MLFQ Basic Setup

- The 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 job should run at a given time
  - a job with higher priority (i.e., a job on a higher queue) is chosen to run

MLFQ Basic Setup

- The 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 job should run at a given time
  - a job with higher priority (i.e., a job on a higher queue) is chosen to run
  - for jobs in the same queue (priority), Round Robin is used

MLFQ Rules

- First two basic rules for MLFQ

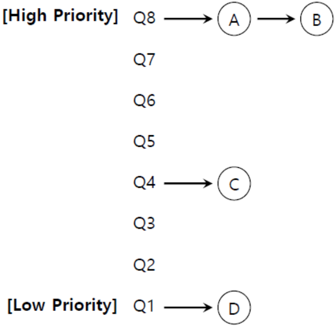
MLFQ Rules

- First two basic rules for MLFQ
  - Rule 1: If Priority(A) $>$ Priority(B), A runs (B doesn't)

MLFQ Rules

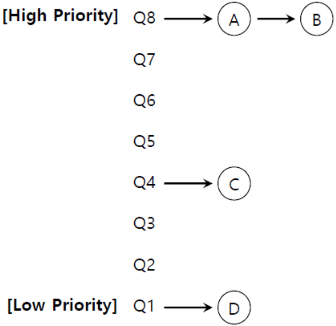
- First two basic rules for MLFQ
  - Rule 1: If Priority(A) $>$ Priority(B), A runs (B doesn't)
  - Rule 2: If Priority(A) $=$ Priority(B), A & B run in RR

MLFQ Example



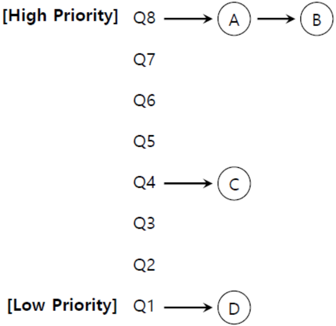
- Would C and D ever run?

MLFQ Example



- Would C and D ever run? No!

MLFQ Example



- Would C and D ever run? No!
- So how could we make C and D to run?

MLFQ: Changing job priority

- The key to MLFQ scheduling is to property set priorities
- For a job that uses the CPU intensively for long periods of time, what should we do with its priority?

MLFQ: Changing job priority

- The key to MLFQ scheduling is to property set priorities
- For a job that uses the CPU intensively for long periods of time, what should we do with its priority?
  - Should reduce its priority

MLFQ: Changing job priority

- The key to MLFQ scheduling is to property set priorities
- For a job that uses the CPU intensively for long periods of time, what should we do with its priority?
  - Should reduce its priority
- For a job that repeatedly relinquishes CPU and waits for I/O, what should we do with its priority?

MLFQ: Changing job priority

- The key to MLFQ scheduling is to property set priorities
- For a job that uses the CPU intensively for long periods of time, what should we do with its priority?
  - Should reduce its priority
- For a job that repeatedly relinquishes CPU and waits for I/O, what should we do with its priority?
  - Should keep its priority high

MLFQ: Changing job priority

- The key to MLFQ scheduling is to property set priorities
- For a job that uses the CPU intensively for long periods of time, what should we do with its priority?
  - Should reduce its priority
- For a job that repeatedly relinquishes CPU and waits for I/O, what should we do with its priority?
  - Should keep its priority high
- MLFQ uses history of a job to predict its future behavior

Changing priority

- Rules on changing priority
  - Rule 3: When a job enters the system it is place at the highest priority

Changing priority

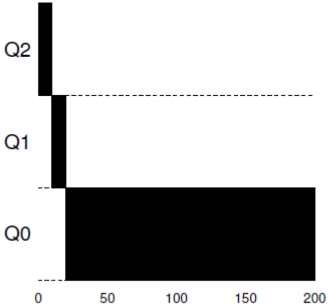
- Rules on changing priority
  - Rule 3: When a job enters the system it is place at the highest priority
  - Rule 4a: If a job uses an entire time slice (of some defined length), its priority is reduced (move down one queue)

Changing priority

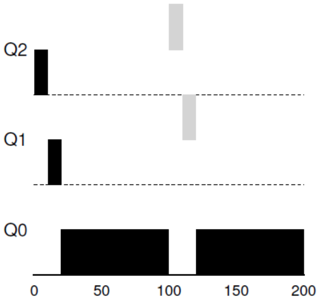
- Rules on changing priority
  - Rule 3: When a job enters the system it is place at the highest priority
  - Rule 4a: If a job uses an entire time slice (of some defined length), its priority is reduced (move down one queue)
  - Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level

Example

- One long-running job
- Time slice length $=$ 10 ms

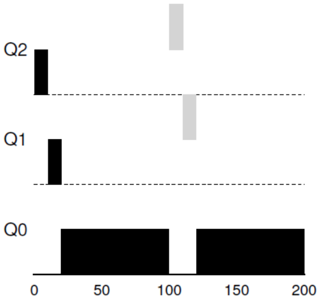


Example



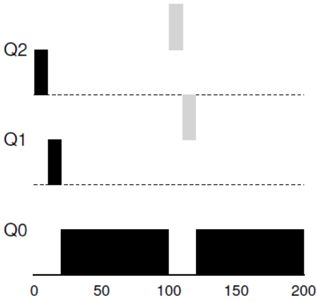
- Two jobs: A (shown in black), and B (shown in gray)
- A is CPU-intensive; it is running in the the lowest-priority queue after two time slices

Example



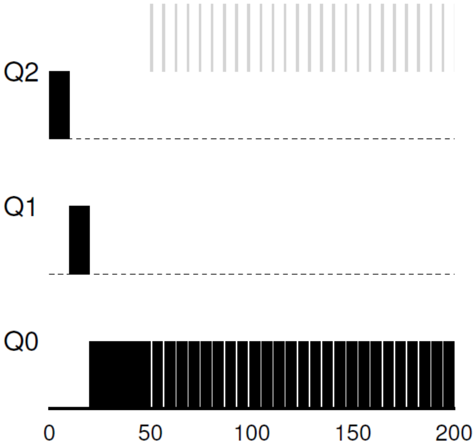
- Two jobs: A (shown in black), and B (shown in gray)
- A is CPU-intensive; it is running in the the lowest-priority queue after two time slices
- B arrives at t = 100 ms and its run-time is short (only 20 ms)

Example



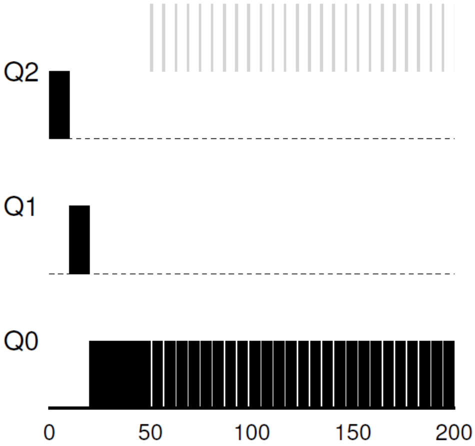
- Two jobs: A (shown in black), and B (shown in gray)
- A is CPU-intensive; it is running in the the lowest-priority queue after two time slices
- B arrives at t = 100 ms and its run-time is short (only 20 ms)
- MLFQ approximates SJF for job B in this case

Example



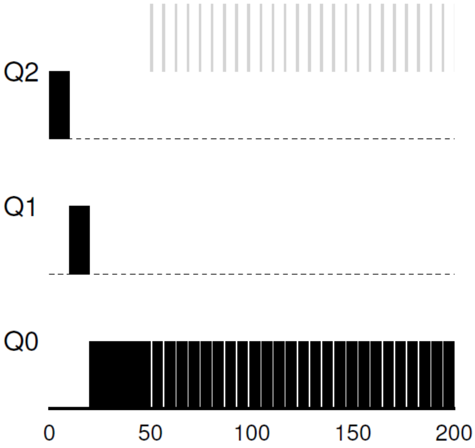
- Job A has run some time $\rightarrow$ drops to the lowest priority queue

Example



- Job A has run some time $\rightarrow$ drops to the lowest priority queue
- Job B is an interactive job: it needs CPU only for 1 ms before performing an I/O

Example



- Job A has run some time $\rightarrow$ drops to the lowest priority queue
- Job B is an interactive job: it needs CPU only for 1 ms before performing an I/O
- Because B keeps releasing the CPU before its time slice is complete, MLFQ keeps B at the highest priority $\rightarrow$ minimize response time

MLFQ Rules

- Rules thus far
  - 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 place at the highest priority
  - Rule 4a: If a job uses an entire time slice (of some defined length), its priority is reduced (move down one queue)
  - Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level

Problems

- Starvation
  - Too many interactive jobs, and thus long-running jobs will never receive any CPU time

Problems

- Starvation
  - Too many interactive jobs, and thus long-running jobs will never receive any CPU time
- Game the scheduler
  - After running 99% of a time slice, issue an I/O
  - The job gains higher percentage of CPU time

Problems

- Starvation
  - Too many interactive jobs, and thus long-running jobs will never receive any CPU time
- Game the scheduler
  - After running 99% of a time slice, issue an I/O
  - The job gains higher percentage of CPU time
- What if a CPU-bound job becomes I/O-bound?

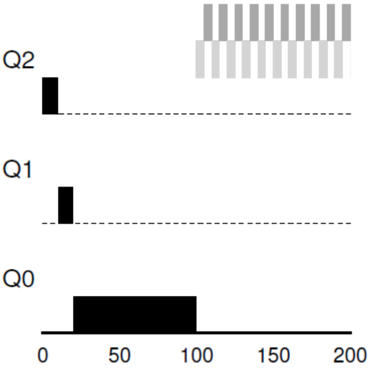
How can we fix these problems?

MLFQ: With priority boost

- Priority boost
  - Reset all jobs to topmost queue after time interval S
- Solves two problems
  - no starvation: low priority jobs will eventually become high priority, acquire CPU time
  - dynamic behavior: a CPU bound job that has become interactive will now be high priority

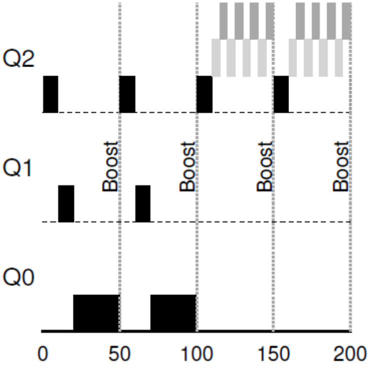
MLFQ: Without priority boost

- A long-running job A
- At time $T = 100$, two interactive jobs show up (B and C)
- B and C each run on CPU for 10 ms, followed by 10 ms of I/O



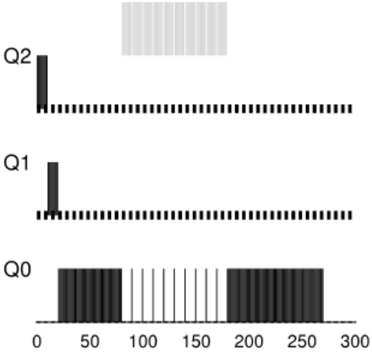
MLFQ: With priority boost

- Assume there is a priority boost every 50 ms



Gaming of scheduler

- Rules 4a and 4b let a job game the scheduler
- Trick: repeatedly relinquish just before the time slice expires



Rule 4 (anti-gaming)

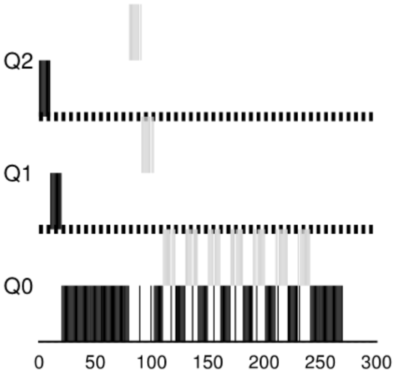
- Scheduling policy becomes a security concern (in datacenter)

Rule 4 (anti-gaming)

- Scheduling policy becomes a security concern (in datacenter)
- The scheduler keeps track of how much of a time slice a job used at a given level
- Revised Rule 4 (replace old 4a and 4b): Once a job has used its time allotment, it is demoted to the next priority queue

Rule 4 (anti-gaming)

- Scheduling policy becomes a security concern (in datacenter)
- The scheduler keeps track of how much of a time slice a job used at a given level
- Revised Rule 4 (replace old 4a and 4b): Once a job has used its time allotment, it is demoted to the next priority queue



Tuning MLFQ

- The high-priority queues $\rightarrow$ Short time slices
  - E.g., 10 or fewer milliseconds
- The Low-priority queue $\rightarrow$ Longer time slices
  - E.g., 100 milliseconds