Scheduling (ch 7+8+9)

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Scheduling Policy

- How to switch processes? **mechanism**
- When to switch? **policy**

Scheduling Policy

- How to switch processes? **mechanism**
- When to switch? **policy**

- **Scheduling policy**

On context switch, which process to run next?

Definitions

- **Job**: what we schedule (i.e., processes)
- **Workload**: set of job descriptions
- **Scheduler**: logic that decides when jobs run
- **Metric**: measurement of scheduling quality

Scheduling Metrics

- What are we trying to optimize?

Scheduling Metrics

- What are we trying to optimize?
- **Turnaround time**
  - Time from job **arrival** to **completion**
  - $T_{turnaround} = T_{completion} - T_{arrival}$

Scheduling Metrics

- What are we trying to optimize?
- **Turnaround time**
  - Time from job **arrival** to **completion**
  - $T_{turnaround} = T_{completion} - T_{arrival}$
- **Fairness**
  - Jobs get same amount of CPU
  - Performance and fairness are often at odds

Workload Assumptions

(Assumption are not realistic)

1. Each job runs for the same amount of time

Workload Assumptions

(Assumption are not realistic)

1. Each job runs for the same amount of time
2. All jobs arrive at the same time

Workload Assumptions

(Assumption are not realistic)

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion

Workload Assumptions

(Assumption are not realistic)

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e., no I/O)

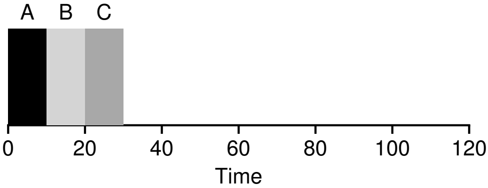
Workload Assumptions

(Assumption are not realistic)

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e., no I/O)
5. The run-time of each job is known

First In, First Out (FIFO)

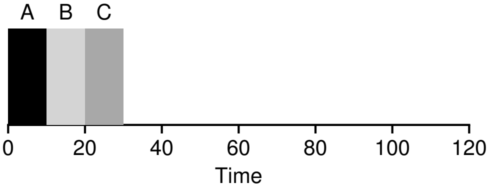
- Also: **First Come, First Served (FCFS)**
  - Simple and easy to implement
- Example: A, B, and C run for 10 seconds each



- What is the average turnaround time?

First In, First Out (FIFO)

- Also: **First Come, First Served (FCFS)**
  - Simple and easy to implement
- Example: A, B, and C run for 10 seconds each



- What is the average turnaround time?

Average turnaround $= \frac{10+20+30}{3} = 20$ sec

First In, First Out (FIFO)

- Relax assumption 1 (~~each job runs for the same amount of time~~)
- In what kind of workload does FIFO perform poorly?

First In, First Out (FIFO)

- Relax assumption 1 (~~each job runs for the same amount of time~~)
- In what kind of workload does FIFO perform poorly?
- Example: three jobs, but A runs for 100 seconds (B & C for 10)

Average turnaround $= \frac{100+110+120}{3} = 110$ sec

First In, First Out (FIFO)

- Relax assumption 1 (~~each job runs for the same amount of time~~)
- In what kind of workload does FIFO perform poorly?
- Example: three jobs, but A runs for 100 seconds (B & C for 10)

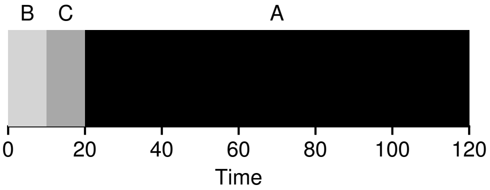
Average turnaround $= \frac{100+110+120}{3} = 110$ sec

Ideal turnaround $= \frac{10+20+120}{3} = 50$ sec

- What should we do?

Shortest Job First (SJF)

- Run shortest job first, then next shortest, and so on
- Previous example:



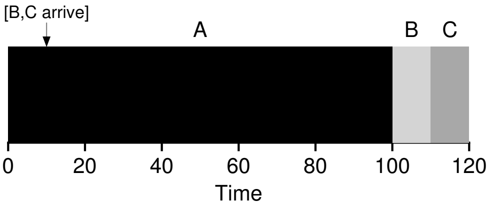
Average turnaround $= \frac{10+20+120}{3} = 50$ sec

Shortest Job First (SJF)

- Relax assumption 2 (~~all jobs arrive at the same time~~)
- Example:

Shortest Job First (SJF)

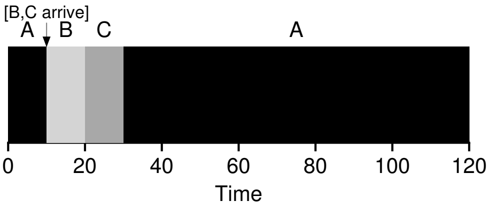
- Relax assumption 2 (~~all jobs arrive at the same time~~)
- Example:
  - A arrives at $t = 0$, needs to run for 100 seconds
  - B & C arrive at $t = 10$, need to run for 10 seconds each



Average turnaround $= \frac{100+100+110}{3} = 103.33$ sec

Shortest Time-to-Completion First (STCF)

- Relax assumption 3 (~~once started, each job runs to completion~~)
- When a new job arrives: schedule job with least time left
- STCF is a **preemptive** scheduler
  - Can **preempt** A to run another job, continuing A later
  - In contrast, SJF is **non-preemptive** (by definition)



Average turnaround $= \frac{120+10+20}{3} = 50$ sec

New Scheduling Metric

- **Response time**
  - Time from job arrival to first scheduling
  - $T_{response} = T_{firstrun} - T_{arrival}$
  - Important for interactive performance (user interaction)
- STCF: not good for response time

How can we build a scheduler that is sensitive to response time?

Round Robin

- Run job for a **time slice** (**scheduling quantum**)
- Then switch to the next job
- Repeat until jobs are finished

Good for response time and fairness,
bad for turnaround time

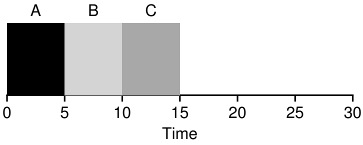
Round Robin

- Run job for a **time slice** (**scheduling quantum**)
- Then switch to the next job
- Repeat until jobs are finished

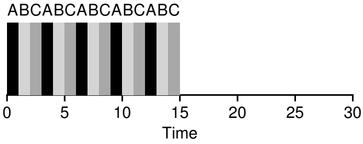
Good for response time and fairness,
bad for turnaround time

- Example: A, B and C arrive at the same time
- Each wish to run for 5 seconds

Round Robin



SJF (Bad for Response Time)



RR (Good for Response Time)

Round Robin

- Shorter time slice
  - Better response time
  - Cost of context switch dominates performance
- Longer time slice
  - Amortize cost of switching
  - Worse response time

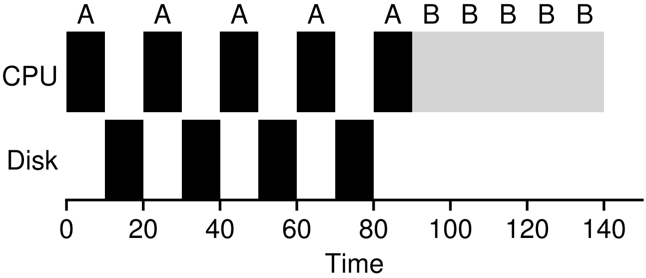
Incorporating I/O

- Relax assumption 4 (~~All jobs only use the CPU~~)
- A job initiates an I/O request
  - Won't be using the CPU; it is **blocked**
  - Scheduler should probably schedule another job
- When the I/O completes
  - An interrupt is raised
  - Moves from **blocked** to **ready**

- Example:
  - A and B need 50ms of CPU time each
  - Each 10ms, A issues an I/O request of 10ms
  - B performs no I/O

Incorporating I/O

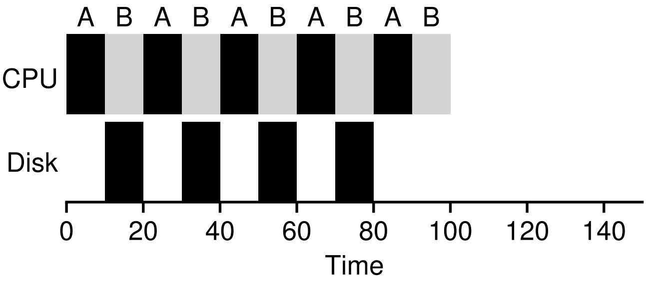
- A runs first, then B after:



Poor use of resources

Incorporating I/O

- STCF scheduler
  - Common approach: treat each sub-job as independent job
  - A is broken up into five 10ms jobs
  - Allows for **overlap** of jobs



Overlap allows better use of resources

No More Oracle

- Relax final assumption (~~the run-time of each job is known~~)
  - Likely the worst assumption we made
  - OS usually knows very little about the length of each job

  - Use recent past to predict the future

Multi-Level Feedback Queue

- Optimize **turnaround time**
  - By running shorter jobs first
  - But the OS doesn't know how long a job will run for
- Make the system feel responsive
  - Minimize **response time**
  - But Round Robin is terrible for turnaround time

How can we design a scheduler without *a priori* knowledge of job length?

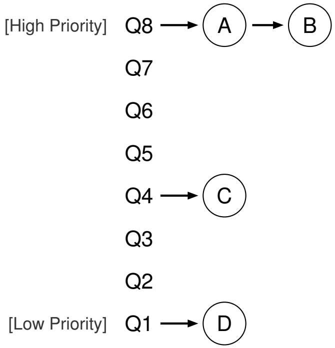
Multi-Level Feedback Queue

- A number of **queues**
- Each assigned a different **priority level**
- Each **ready** job is on a single queue
- MLFQ chooses to run a job with higher priority
  - On a higher queue
  - Round-robin scheduling among jobs with same priority

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

- The key: how the scheduler sets priorities
  - Job repeatedly waits for keyboard input:
    - Priority high, interactive process
  - Job uses the CPU intensively for long periods
    - Reduce its priority

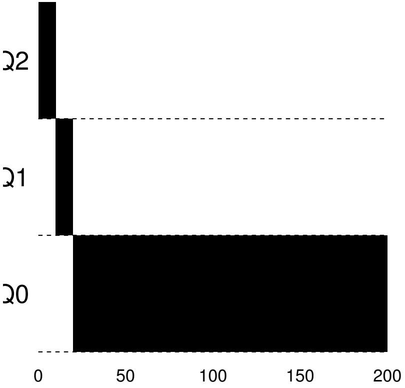


How To Change Priority

- Let's attempt the following:
  - **Rule 3**: A job enters the system $\rightarrow$ highest priority
  - **Rule 4a**: Job uses entire time slice $\rightarrow$ reduce priority
  - **Rule 4b**: Job gives up CPU $\rightarrow$ stay at the same priority

A Single, Long-Running Job

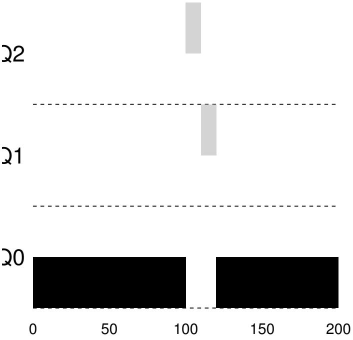
- Three-queue scheduler, 10ms time slice



Along Came A Short Job

MLFQ approximates SJF:

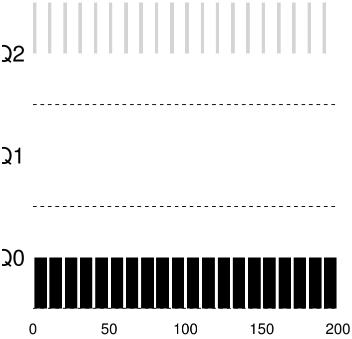
- B: short-running interactive job



- Assume short job, slowly move down the queues if not

What About I/O?

- B needs the CPU for 1ms before performing I/O
  - Same priority by Rule 4b



Problems

- This approach has serious flaws

Problems

- This approach has serious flaws
  - **Starvation**
    - Too many interactive jobs consume all CPU time

Problems

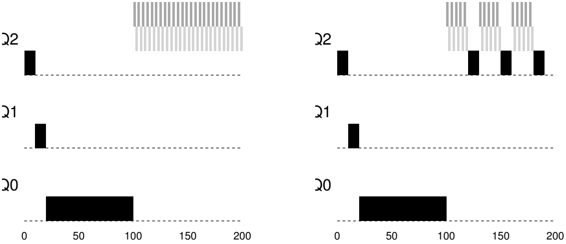
- This approach has serious flaws
  - **Starvation**
    - Too many interactive jobs consume all CPU time
  - **Game the scheduler**
    - Before time slice is over, issue I/O operation
    - Remain in same queue, gain higher percentage of CPU time

Problems

- This approach has serious flaws
  - **Starvation**
    - Too many interactive jobs consume all CPU time
  - **Game the scheduler**
    - Before time slice is over, issue I/O operation
    - Remain in same queue, gain higher percentage of CPU time
  - Change behavior over time
    - Start as CPU-bound, transition to interactivity

The Priority Boost

- Let's make another attempt:
  - **Rule 5**: After some time period, all jobs $\rightarrow$ topmost queue
- Solves two problems at once:
  - Guaranteed not to starve
  - CPU-bound job that becomes interactive is treated properly

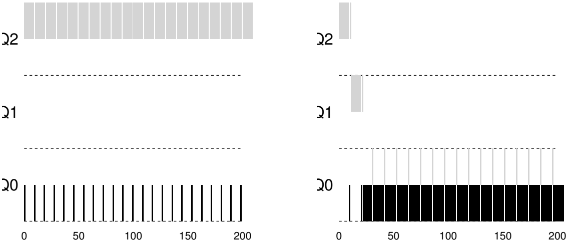


Without (Left) and With (Right) Priority Boost

Better Accounting

- How to prevent gaming of our scheduler?
  - Rules 4a and 4b are the culprits
  - Perform better **accounting** of CPU time

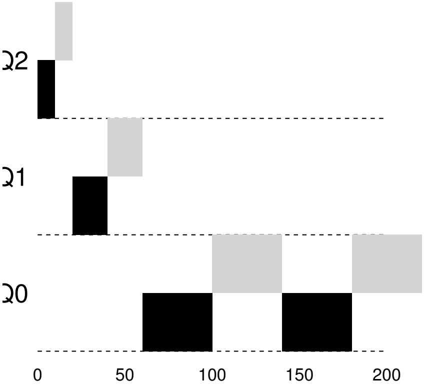
  - **Rule 4**: Job used up time allotment → reduce priority
    - Regardless of how many times it has given up the CPU



Without (Left) and With (Right) Gaming Tolerance

Tuning

- Varying time-slice lengths
  - High priority queues: short time slices
  - Low priority queues: longer time slices



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**: A job enters the system $\rightarrow$ highest priority
- **Rule 4**: Job used up time allotment $\rightarrow$ reduce priority
- **Rule 5**: After some time period, all jobs $\rightarrow$ topmost queue

Proportional Share Scheduler

- **Fair-share** scheduler
  - Instead of optimizing turnaround and response time
  - Try to guarantee each job a certain percentage of CPU time
- An early example: **lottery scheduling**
  - Every so often, hold a lottery to determine next process

How to design a scheduler to share the CPU in a proportional manner?

Tickets

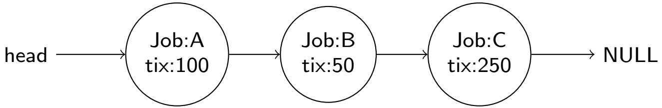
- Each job has a number of **tickets**
- Percent of tickets represents share of CPU it should receive:
  - A has 75 tickets, B has 25 tickets
  - A receives 75% of CPU, B receives 25%
- Lottery scheduling: probabilistic
  - Hold a lottery every time slice

Tickets

- **Ticket currency**
  - User allocates tickets among its jobs how they would like
  - The system converts the currency to the global value
- **Ticket transfer**
  - A job can temporarily hand off its tickets to another job
  - e.g., a client handing its tickets to a server
- **Ticket inflation**
  - A job temporarily raises or lower the number of tickets it owns
  - Only for non-competitive scenarios

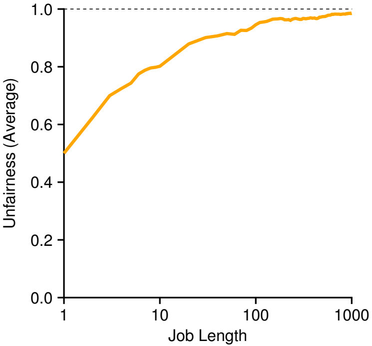
Implementation

- Example: keep processes in a list



Implementation

- **Unfairness metric**
  - The time the first job completes divided by the time the second job completes
  - For example: two jobs with runtime 10 in FCFS: $\frac{10}{20} = 0.5$



Why Not Deterministic?

- For determinism: **stride scheduling**
  - Each job has a **stride**, inverse in proportion to number of tickets
  - Each job has a **pass** value, incremented by stride each run
  - Scheduler picks job with lowest pass value
- For example:
  - A, B, C with 100, 50, 250 tickets respectively
  - Divide some large number by it, e.g., 10,000
  - A: B, C with 100, 200, 40 stride

Why Not Deterministic?

Pass(A)	Pass(B)	Pass(C)	Who runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	160	C
200	200	200	...

- Why use lottery scheduling?
  - New job with pass value 0: monopolizes the CPU
  - Stride scheduler requires global state

Summary

- Scheduler efficiency is very important
  - In Google datacenters, scheduling uses 5% of CPU time
  - Even after aggressive optimization
- **FCFS** and **SJF** are simple, but high turnaround
- **STCF** is good for turnaround, but bad for response time
- **RR** solves is good for response time and fairness, but bad for turnaround
- **STCF** with **overlap** (sub-jobs) allows better use of resources
- Without oracle: **multi-level feedback queue**
  - Number of **queues** each assigned a different **priority level**
- **Proportional share scheduler**
  - Guarantee percentage of CPU instead of optimizing turnaround and response times