CPE334
Operating Systems
Lecture 3: Scheduling

Lecturer:
Stephen John Turner
stephen.tur@kmutt.ac.th
Tel: 02-470-9378 Office: 1027

Scheduling

- Scheduling = The process of deciding how to commit resources between a variety of possible tasks
- It is not a new concept – Assembly lines and many other human constructions require scheduling
- For a computer system, scheduling is a matter of sharing resources among multiple processes
- Context-switch between processes
  - low-level mechanisms (previous lecture)
  - high-level policies (this lecture)

Workload

- Processes running in the system, collectively called the workload
- Assume the following for the processes/jobs running in the system (for now)
  - All jobs only use the CPU (i.e. they perform no I/O)
  - The run-time of each job is known

Scheduling Metrics

- Metrics are used to measure scheduling policies' performance and fairness
- Sometime optimizing performance may prevent a few jobs from running, thus decreasing fairness
- For now, let's assume that we will use just one metric, turnaround time
- The turnaround time is defined as the time at which the job completes minus the time at which the job arrived in the system

$$T_{turnaround} = T_{completion} - T_{arrival}$$

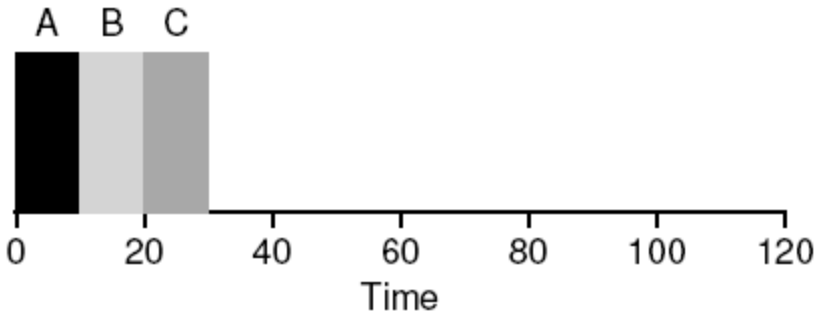
- If all jobs arrive at the same time

$$T_{arrival} = 0 \text{ and } T_{turnaround} = T_{completion}$$

First In, First Out (FIFO)

(First Come, First Served)

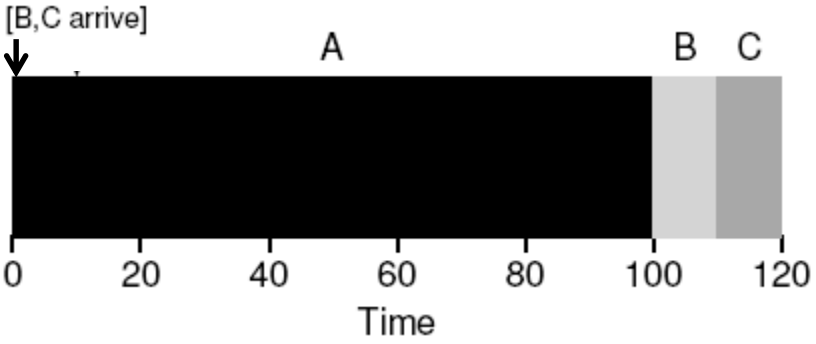
- Imagine three jobs arrive in the system, A, B, and C, at the same time ($T_{arrival}$ = 0)
- FIFO has to put some job first, let's assume A -> B -> C
- Assume also that each job runs for 10 seconds
- What will the average turnaround time be for these jobs?



(10+20+30)/3 = 20 units time

FIFO's Drawback

- Let's again assume three jobs (A, B, and C), but this time A runs for 100 seconds while B and C run for 10 each

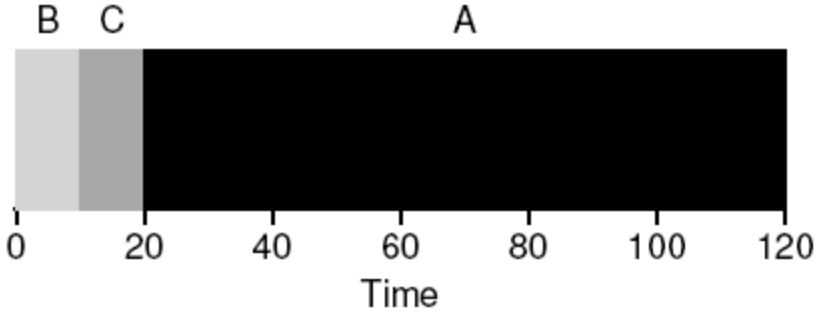


(100+110+120)/3 = 110

- Problem: a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer

Shortest Job First (SJF)

- Policy: Run the shortest job first, then the next shortest, and so on



(10+20+120)/3 = 50

- Average turnaround time becomes 50
- Given assumptions about jobs all arriving at the same time, SJF is indeed optimal

More Realistic Situations

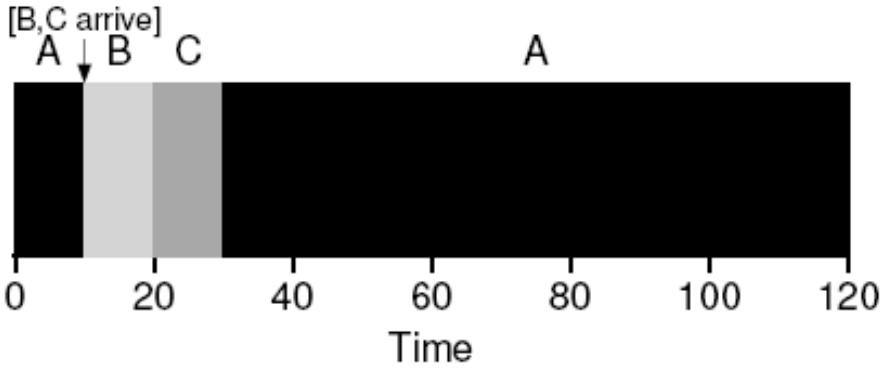
- What if jobs can arrive at any time instead of all at once and can run for different lengths
  - A arrives at t = 0 and needs to run for 100 seconds
  - B and C arrive at t = 10 and each needs to run for 10 seconds
- (100+(110-10)+(120-10))/3= 103.33
- What can a scheduler do?

Shortest Time-to-Completion First (STCF)

- With mechanisms like timer interrupts and context switching, the scheduler can preempt job A and decide to run another job
- SJF is a non-preemptive scheduler
- All modern schedulers are preemptive
- STCF also known as Preemptive Shortest Job First (PSJF) – any time a new job enters the system
  - STCF/PSJF determines of the remaining jobs and new job, which has the least time left
  - then schedules that one

STCF's example

- A arrives at t = 0 and needs 100 secs, B and C arrive at t = 10, each need 10 secs



- Ave turnaround time (120+10+20)/3 = 50
- In batch scheduling, this make sense
- However, in time-shared machines where users sit at a terminal and demand interactive performance, a new metric was introduced: response time

Response Time

- Response time is the time from when the job arrives in a system to the first time it is scheduled

$$T_{response} = T_{firstrun} - T_{arrival}$$

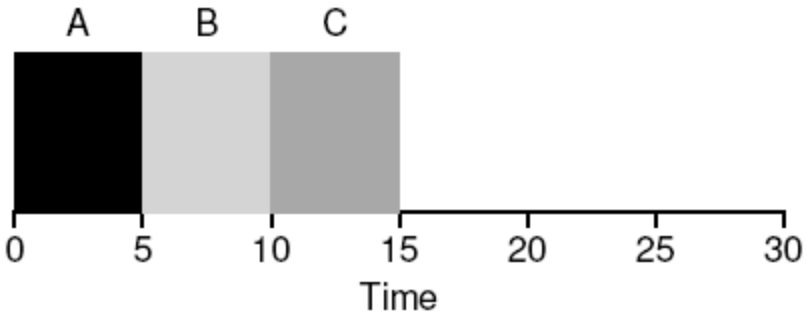
- A arrives at t = 0 and needs 100 secs, B and C arrive at t = 10, each needs 10 secs
- Using STCF, what is $T_{response}$ = ?
  - $T_{response}$ = (0+0+10)/3 = 3.33 –> not so good
- How can we build a scheduler that is sensitive to response time?

Round Robin (RR)

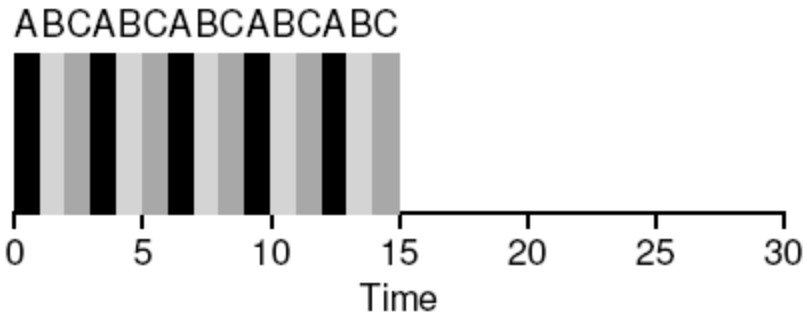
- Instead of running jobs to completion, RR
  - runs a job for a time slice
  - then switches to the next job in the run queue
  - repeatedly does so until all jobs are finished
- RR is considered ‘fair’, as it evenly divides the CPU among active processes on a small time scale
- Note that the length of a time slice must be a multiple of the timer-interrupt period
  - If the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms

Comparison of Response Time

- A, B, and C arrive at the same time, and each needs to run for 5 secs
- SJF average response time = 5



- RR average response time = 1



Time Slice

- The shorter it is, the better the performance of RR under the response time metric
- If the time slice is too short, the cost of context switching will dominate performance
- Design Choice:
  - Making it long enough to amortize the cost of context switching
  - Not too long that the system is no longer responsive

Amortization

- The length of the time slice is critical for RR performance
- Overhead = a fixed cost to some operation
- Incurring overhead less often, the total cost to the system is reduced
- For example
  - Time slice is set to 10 ms, and the context-switch cost is 1 ms
  - 10% of time is spent context switching (overhead of 10%)
- Amortize the cost by increasing the time slice to 100 ms and less than 1% of time is spent context switching (overhead of 1 %)

Turnaround Time vs. Response Time

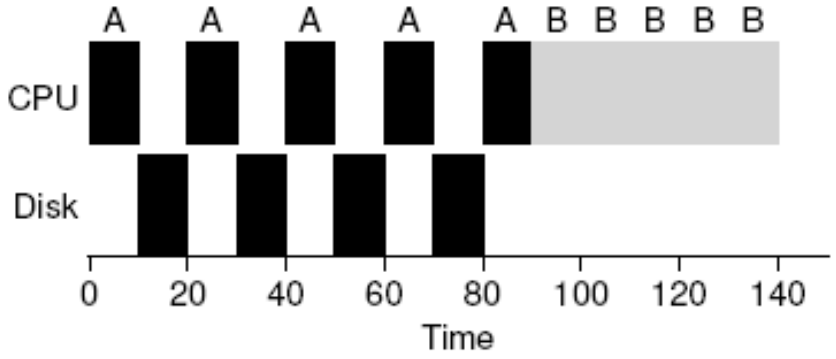
- Two families of algorithms
  - Runs the shortest job remaining and optimizes turnaround time (SJF and STCF) -> bad response time
  - Alternates between all jobs and optimizes response time (RR) -> bad turnaround time
- Trade-off is thus common in system design
- Next: we will consider
  - Jobs with I/O
  - Jobs with unknown execution time

Incorporating I/O

- When a job initiates an I/O request, it will not be needing CPU
  - It is blocked waiting for I/O completion
  - The scheduler schedules another job on the CPU
- When the I/O completes:
  - An interrupt is raised
  - The OS moves the process back to the ready state

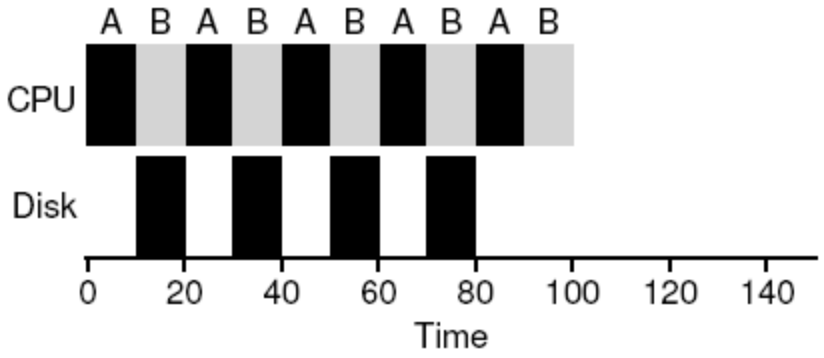
An Example With I/O Requests (1)

- Jobs A and B each needs 50 ms of CPU time
  - A runs for 10 ms and then issues an I/O request (assume I/O takes 10 ms)
  - B uses the CPU for 50 ms and performs no I/O
- Run A first, then B after, will result in a poor performance



An Example With I/O Requests (2)

- A common approach is to treat each 10-ms sub-job of A as an independent job
- When the system starts, its choice is whether to schedule a 10-ms A or a 50-ms B
- By treating each CPU burst as a job, the scheduler can make sure processes that are “interactive” get run frequently



The Question

- The OS usually knows very little about the length of each job
- Thus, how can we build an approach that behaves like SJF/STCF without such a priori knowledge?
- Answer: The Multi-Level Feedback Queue

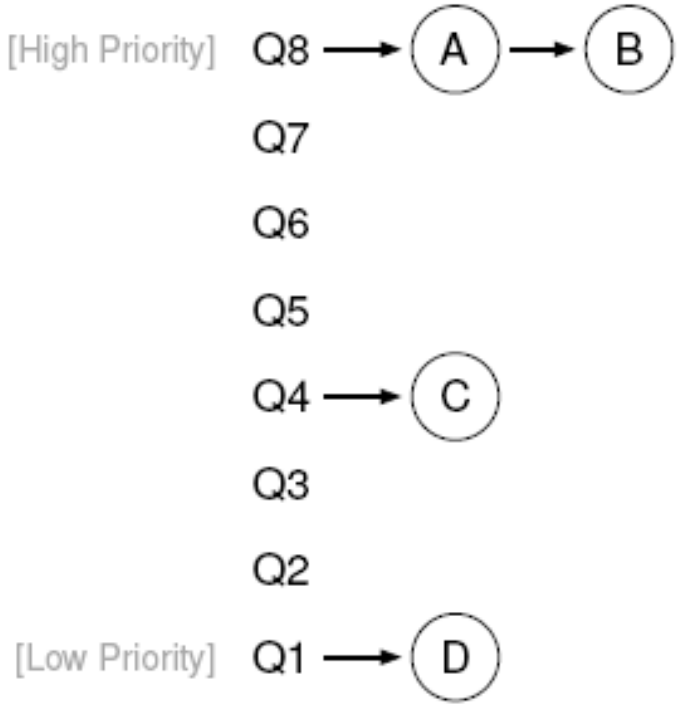
The Multi-Level Feedback Queue (MLFQ)

- Two goals
  - Optimize turnaround time
  - Minimize response time to allow a system to be responsive to interactive users
- MLFQ: Basic concept
  - Implement a number of distinct queues of different priority levels
  - At any given time, a job that is ready to run is on a single queue
  - MLFQ uses priorities to decide which queue should be selected
  - RR is used among jobs in the same queue
- The key to MLFQ scheduling lies in how the scheduler sets queue priorities

MLFQ Rules

- MLFQ varies the priority of a job based on its observed behavior
  - If a job repeatedly waits for input from the keyboard, the priority is high (behaves like an interactive process)
  - If a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority
  - MLFQ tries to learn about processes as they run, and use the history of the job to predict its future behavior
- We arrive at the 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

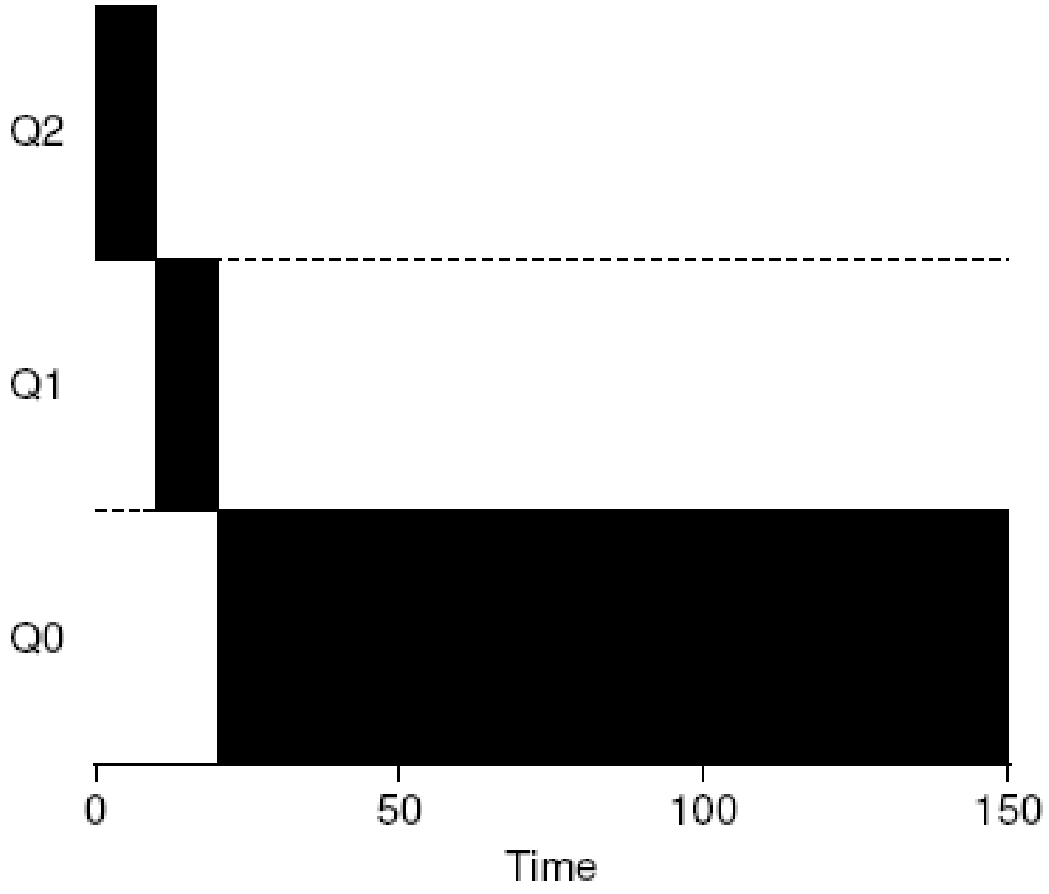


- From the rules, the scheduler would just alternate time slices between A and B because they are the highest priority jobs in the system

How to Change Priority

- ***Rule 3:*** When a job enters the system, it is placed at the highest priority (the topmost queue)
- ***Rule 4a:*** If a job uses up an entire time slice while running, its priority is reduced (i.e. it moves 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

One Long Running Job

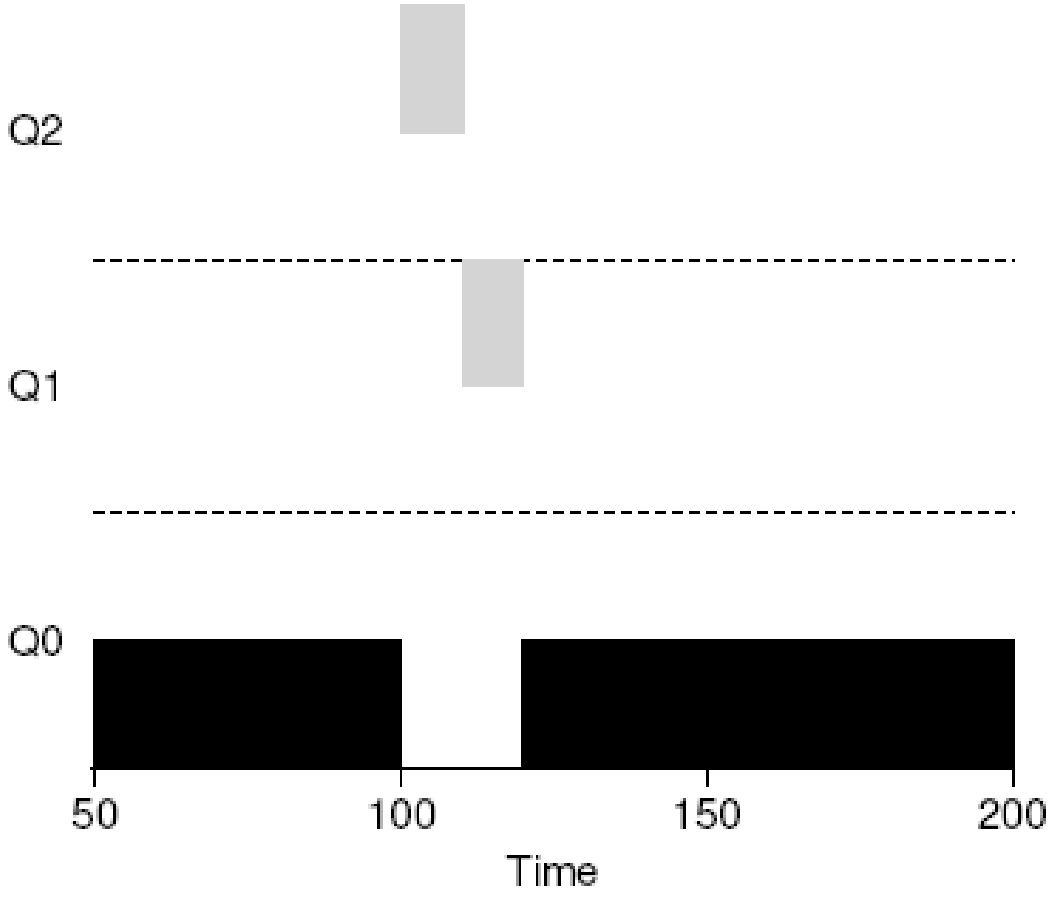


Start in Q2

Reduce to Q1

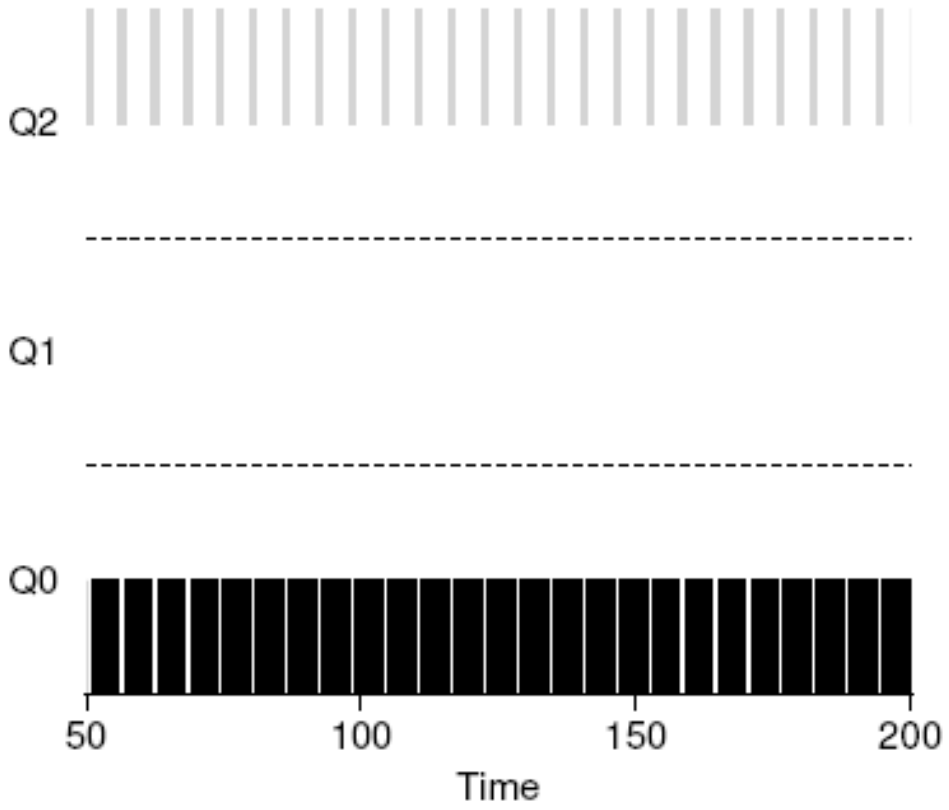
Then to Q0

Two Jobs



Grey: a short interactive job arrived at time T=100
Black: a long compute intensive job

A Mixed I/O-Intensive and CPU-Intensive Workload



Grey: an interactive job
Black: a batch job

Problems with the Current MLFQ

- Starvation
  - If too many interactive jobs, long-running jobs may never receive any CPU time
- A program can trick the scheduler to give it more than fair share of the resource
- A program may change its behavior over time
  - CPU-bound may transition to a phase of interactivity
  - With our current approach, such a job would not be treated like the other interactive jobs in the system

The Priority Boost

- Periodically boost the priority of all the jobs in system
- ***Rule 5***: After some time period S, move all the jobs in the system to the topmost queue
  - This rule fixes the starvation problem
  - If a CPU-bound job has become interactive, the scheduler can give it the priority boost
- What should S be set to?
  - If too high, long-running jobs could starve
  - If too low, interactive jobs may not get a proper share of the CPU

Better Accounting

- To guarantee the fairness, Rule #4 should be re-written
- ***Rule 4:*** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue)

How to Parameterize such a Scheduler

- How many queues should there be?
- How big should the time slice be per queue?
- How often should priority be boosted in order to avoid starvation and account for changes in behavior?
  - Some experience with workloads and subsequent tuning of the scheduler will lead to a satisfactory balance
  - Use math formula to adjust and/or manually done by a system admin

The Final Rules

- ***Rule 1:*** If Priority(A) > Priority(B), A runs (B doesn't)
- ***Rule 2:*** If Priority(A) = Priority(B), A & B run in RR
- ***Rule 3:*** When a job enters the system, it is placed at the highest priority (the topmost queue)
- ***Rule 4:*** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e. it moves down one queue)
- ***Rule 5:*** After some time period S, move all the jobs in the system to the topmost queue

MLFQ is used in

- BSD UNIX derivatives
- Solaris
- Windows NT and subsequent Windows operating systems

Proportional-Share Scheduler

- Instead of optimizing for turnaround or response time, just guarantee that each job obtains a certain percentage of CPU time
- Lottery scheduling
  - Hold a lottery to determine which process should go next
  - Processes that should run more often should be given more chances to win the lottery
  - Uses randomness – randomized approaches are often a robust and simple way to schedule things
    - Avoids corner cases
    - Lightweight (little state to track things)
    - Fast

Tickets Represent Your Share

- Introduce the concept of tickets
  - represent the shares that processes should get
- Imagine processes A and B
  - If A has 75 tickets and B has 25,
  - A is to receive 75% of the CPU and B 25%
- Lottery scheduling achieves this probabilistically (not deterministically) by holding a lottery often (every time slice)

Holding a Lottery

- The scheduler must know how many total tickets there are, e.g. 100
- The scheduler then picks a winning ticket, e.g. between 0 to 99
- The scheduler then loads the state of that winning process and runs it
  - e.g. If A hold tickets 0-74, B 75-99
  - Draw: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 49
  - Schedule: A B A A B A A A A A A B A B A A A A A A

Ticket Mechanisms: Ticket Currency

- *"Currency"* allows a user with a set of tickets to allocate tickets among their own jobs
- The system then automatically converts the currency into the correct global value
- e.g. A and B have each been given 100 tickets
  - A is running two jobs, A1 and A2, and gives them each 500 tickets (out of 1000 total) in User A's own currency
  - B is running only 1 job and gives it 10 tickets (out of 10 total)
  - The system will convert
    - 500 tickets from A1's and A2's to 50 each in the global currency
    - 10 tickets from B1 to 100 tickets in the global currency
  - The lottery will be held over the global ticket currency (200 total) to determine which job runs

Ticket Mechanisms: Ticket Transfer

- *"Transfers"* allow a process to temporarily hand off its tickets to another process
- Useful in client-server computing
  - a client process sends a message to a server asking it to do some work on the client's behalf
  - To speed up the work, the client can pass the tickets to the server in order to process its own request
  - When finished, the server transfers the tickets back

Ticket Mechanisms: Ticket Inflation

- *"Inflation"* allows a process to temporarily raise or drop its ticket value
- Can be applied in an environment where a group of processes trust one another
- If any one process knows it needs more CPU time, it can boost its ticket value without communicating with any other processes

Implementation

- A good random number generator to pick the winning ticket
- A simple data structure to track the processes, and the total number of tickets

Assume a process linked-list

head -> (A | 100) -> (B | 50) -> (C | 250) -> null

```
counter = 0;
winner = random(totaltickets); // get winner
list_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// current is the winner: schedule it...
```

Assign Tickets to Jobs

- How the system behaves is strongly dependent on how tickets are allocated
- One approach is to assume that the users know best
  - Each user is handed some number of tickets
  - A user allocates tickets to any jobs as desired
- Fancy algorithms can also be used

Summary

- ***Lottery Scheduling*** uses randomness in a clever way to achieve proportional share
- However, it leaves open the hard problem of ticket assignment
- General-purpose schedulers such as the MLFQ are thus used in many more systems