# CS310 Operating Systems

## Lecture 26 Scheduling – Multilevel Feedback Queue, Lottery Scheduling

Ravi Mittal

IIT Goa

# Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, University of California, Berkeley
  - Book: Operating System: Three Easy Pieces

# Reading

- CS162, Operating System and Systems Programming, University of California, Berkeley
- Book: Operating System: Three Easy Pieces

# Previous lectures ..

# First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also First In First Out (FIFO) or Run until done
  - In early systems, FCFS meant one program scheduled until done (including I/O)
  - Now, means keep CPU until thread blocks
- Simple Algorithm, Easy to implement (+)
- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
  - *Convoy effect:* short process stuck behind long process (-)

# Shortest Job First (SJF)

- Non-preemptive
- Run whatever job has least amount of computation to do
- Provably optimal
- Need to know run times in advance

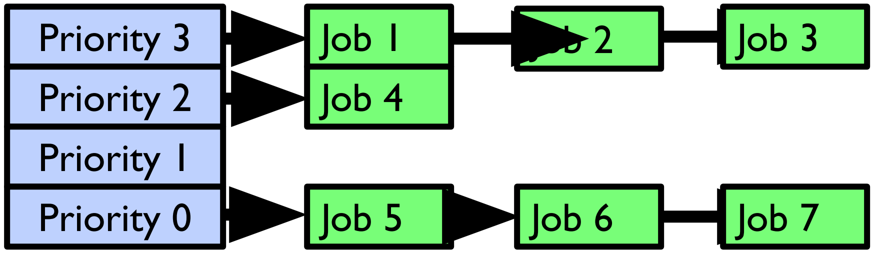
# Shortest Remaining Time First (SRTF)

- Preemptive version of SJF
- If job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
- Sometimes called Shortest Remaining Time to Completion First (SRTCF)
- Both SJF and SRTF:
  - These can be applied to whole program or current CPU burst
    - Idea is to get short jobs out of the system
    - Big effect on short jobs, only small effect on long ones
    - Result is better average response time

# Round Robin (RR) Scheduling

- Uses Preemption!
- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds
- After quantum expires, the process is preempted and added to the end of the ready queue
- n processes in ready queue and time quantum is q
    - Each process gets 1/n of the CPU time
    - In chunks of at most q time units
    - No process waits more than (n-1)q time units

# Multilevel Queue Scheduling – Strict Priority




- Execution Plan
  - Always execute highest-priority runnable jobs to completion
  - Each queue can be processed in RR with some time-quantum
  - A priority is assigned statically to each process, and a process remains in the same queue for the duration of the run time
- Problems
  - Starvation
    - Lower priority jobs don't get to run because higher priority jobs
  - Deadlock: Priority Inversion
    - Happens when low priority task holds a lock needed by high-priority task

# Today we will study

- Multilevel Feedback Queue
- Changing landscape of scheduling
- Proportionate share scheduling
- Lottery Scheduling

# Multilevel Feedback Queue

# Dealing with Starvation

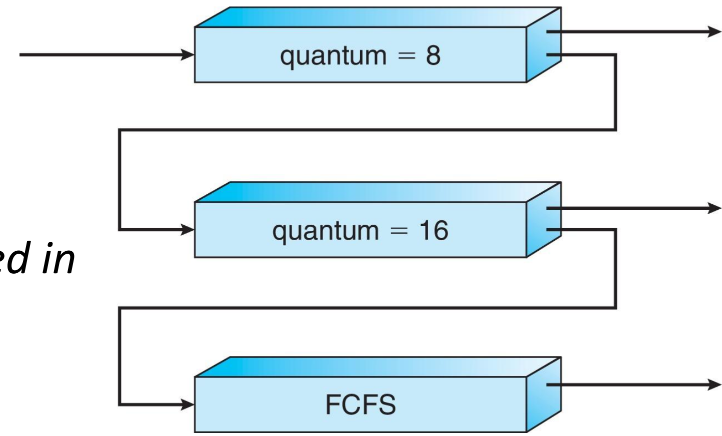
- Strict priority scheduling may lead to situation where low priority processes may starve
- One solution is
  - Don't assign priorities that are static – fixed for lifetime of the process
  - But, change priority of a process – dynamically
  - Scheduler can decrease the priority of the current running process
    - Penalizing it for taking too much CPU time
  - Alternatively, Scheduler keeps track of low priority processes that are not getting a chance to run
    - Increase their priority

# Multilevel Feedback Queue

- Multilevel queues with no strict priority
- Scheduler is allowed to change priority of a process
  - Dynamic priority
- A multilevel feedback queue uses two basic rules:
  - A new process gets placed in the highest priority queue
  - If a process does not finish its quantum (due to I/O)
    - Then it will stay at the same priority level (round robin)
    - Else it moves to the next lower priority level
- A process with long CPU bursts will use its entire time slice
  - Get preempted and get placed in a lower-priority queue
  - A highly interactive process will not use up its quantum and will remain at a high priority level

# Example of Multilevel Feedback Queue

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
- Scheduling
  - *A new process enters queue $Q_0$ which is served in RR*
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue $Q_1$
  - *At $Q_1$ job is again served in RR and receives 16 additional milliseconds*
    - If it still does not complete, it is preempted and moved to queue $Q_2$




- Starvation problem still exists
  - When new processes are frequently created or there are too many interactive processes □ CPU bound processes are never scheduled

# Cause for Starvation: Priorities?

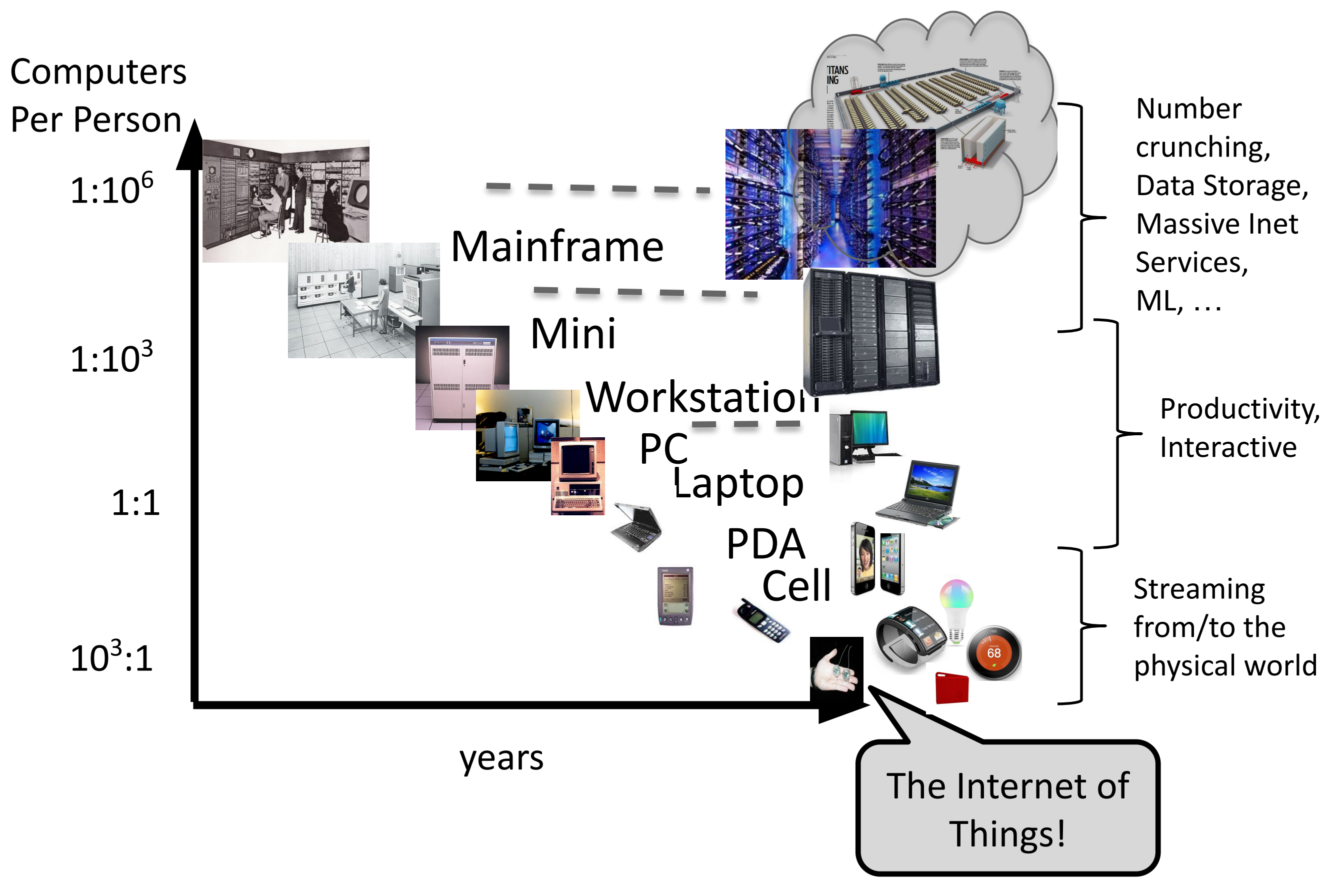
- The policies we've studied so far:
  - Always prefer to give the CPU to a prioritized job
  - Non-prioritized jobs may never get to run

- But priorities were a means, not an end
- Our end goal was to serve a mix of CPU-bound, I/O bound, and Interactive jobs effectively on common hardware
  - Give the I/O bound ones enough CPU to issue their next file operation and wait (on those slow discs)
  - Give the interactive ones enough CPU to respond to an input and wait (on those slow humans)

# Changing landscape ..

# Changing Landscape…

Bell's Law: New computer class every 10 years



# Changing Landscape of Scheduling

- Priority-based scheduling rooted in "time-sharing"
  - Allocating precious, limited resources across a diverse workload
    - CPU bound, vs interactive, vs I/O bound
- 80's brought about personal computers, workstations, and servers on networks
  - Different machines of different types for different purposes
  - Shift to fairness and avoiding extremes (starvation)
- 90's emergence of the web, rise of internet-based services, the data-center-is-the-computer
  - Server consolidation, massive clustered services, huge flash crowds
  - It's about predictability, $95^{th}$ percentile performance guarantees

**Does prioritizing some jobs *necessarily* starve those that aren't prioritized?**

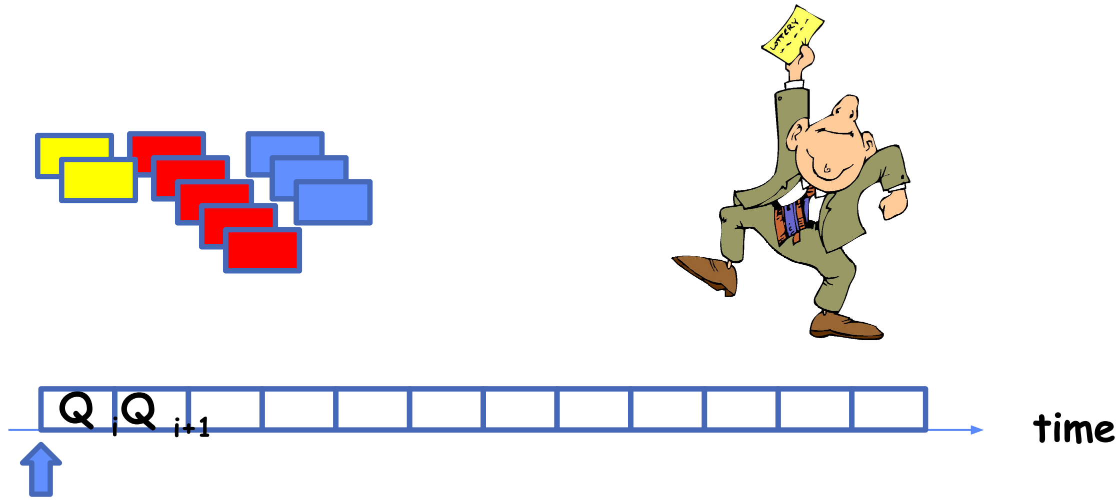
# Key Idea: Proportional-Share Scheduling

- The policies we've studied so far:
  - Always prefer to give the CPU to a prioritized job
  - Non-prioritized jobs may never get to run

- Instead, we can share the CPU *proportionally*
  - Give each job a share of the CPU according to its priority
  - Low-priority jobs get to run less often
  - But all jobs can at least make progress (no starvation)

# Lottery Scheduling (Fair Share Scheduling)

# Lottery Scheduling




- Given a set of jobs (the mix), provide each with a share of a resource
  - e.g., 50% of the CPU for Job A, 30% for Job B, and 20% for Job C
- Idea: Give out tickets according to the proportion each should receive
- Every quantum: draw one at random, schedule that job (thread) to run

# Lottery Scheduling


- Proportional share scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets? Users can define policy
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

**Use Randomness**

**Requires a good random number generator**

# Lottery Scheduling Example (Cont.)

- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket (total 11 tickets)

| # short jobs/ # long jobs | % of CPU each short jobs gets | % of CPU each long jobs gets |
|---|---|---|
| 1/1 | 91% | 9% |
| 0/2 | N/A | 50% |
| 2/0 | 50% | N/A |
| 10/1 | 9.9% | 0.99% |
| 1/10 | 50% | 5% |

- How to do ticket distribution when processes come and go and get blocked?
- Not a useful algorithm for general-purpose scheduling
- Useful for environments with long-running processes that may need to be allocated shares of CPUs
  - Example: running multiple virtual machines on a server

# Lottery Fairness

- Two jobs competing against each other
- Both have the same number of tickets say 100
- Both have the same run time R
- Ideally each job must finish roughly at the same time
  - Due to randomness, sometimes one job finishes before the other
- Fairness Metric: F
  - Time the first job competes / time the second job completes
  - Ex: R = 10; First job completes at time 10 and second job completes at time 20;  F= 10/20 = 0.5
  - When both finish at nearly the same time, F = 1

# A simulation

- R varies from 1 to 1000 over 30 trials

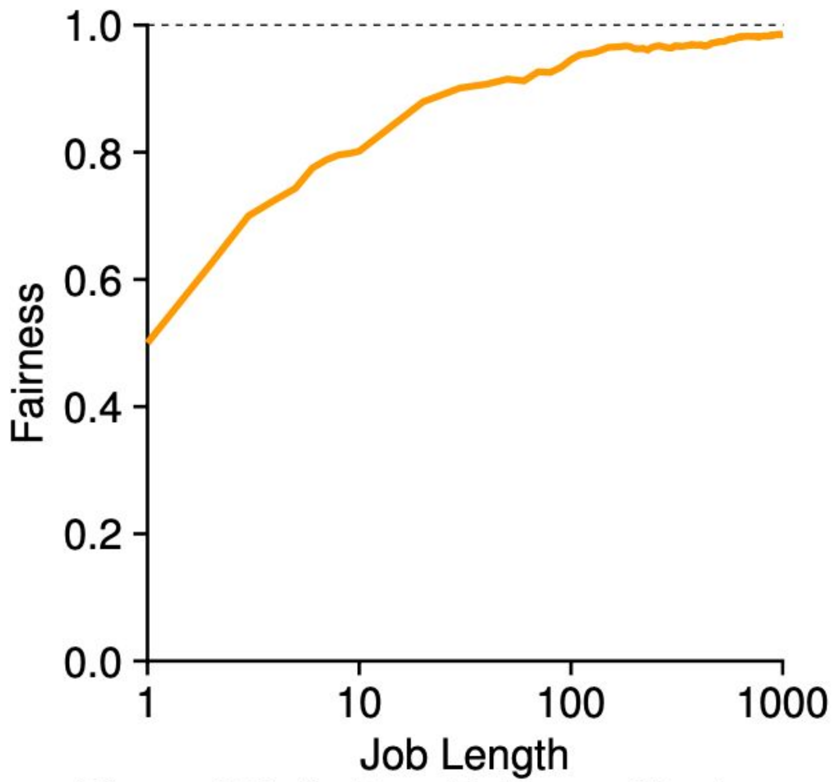



Figure 9.2: **Lottery Fairness Study**

# How to Assign Tickets?

- How many to each user?
- User knows the best
  - Each user is handed over a certain number of ticekts
  - User can allocate them to his/her jobs
- Ticket Assignment problem remains open.

# Lecture Summary

- Multilevel Feedback Queue
  - It has multiple levels of queues
  - It uses feedback to determine priority of each job
    - Based on how jobs behave over time
  - It is not based on concept of prior knowledge of the nature of the job … but observes the execution of a job and prioritize accordingly
- We have studied the concept of proportional – share scheduling
  - Lottery Scheduling
- Lottery scheduling uses randomness in a clever way to achieve proportional share