

CS 310 Operating Systems

Lecture 25 Scheduling – Round Robin, Priority Scheduling, Multilevel Queue Scheduling,

Ravi Mittal
IIT Goa

In this lecture we will study

- Last lecture
- Scheduling in Interactive Systems
- Round Robin Scheduling
- Priority Scheduling
- Strict Priority Scheduling Multilevel Queue
- Adaptive Scheduling - Introduction

Acknowledgements !

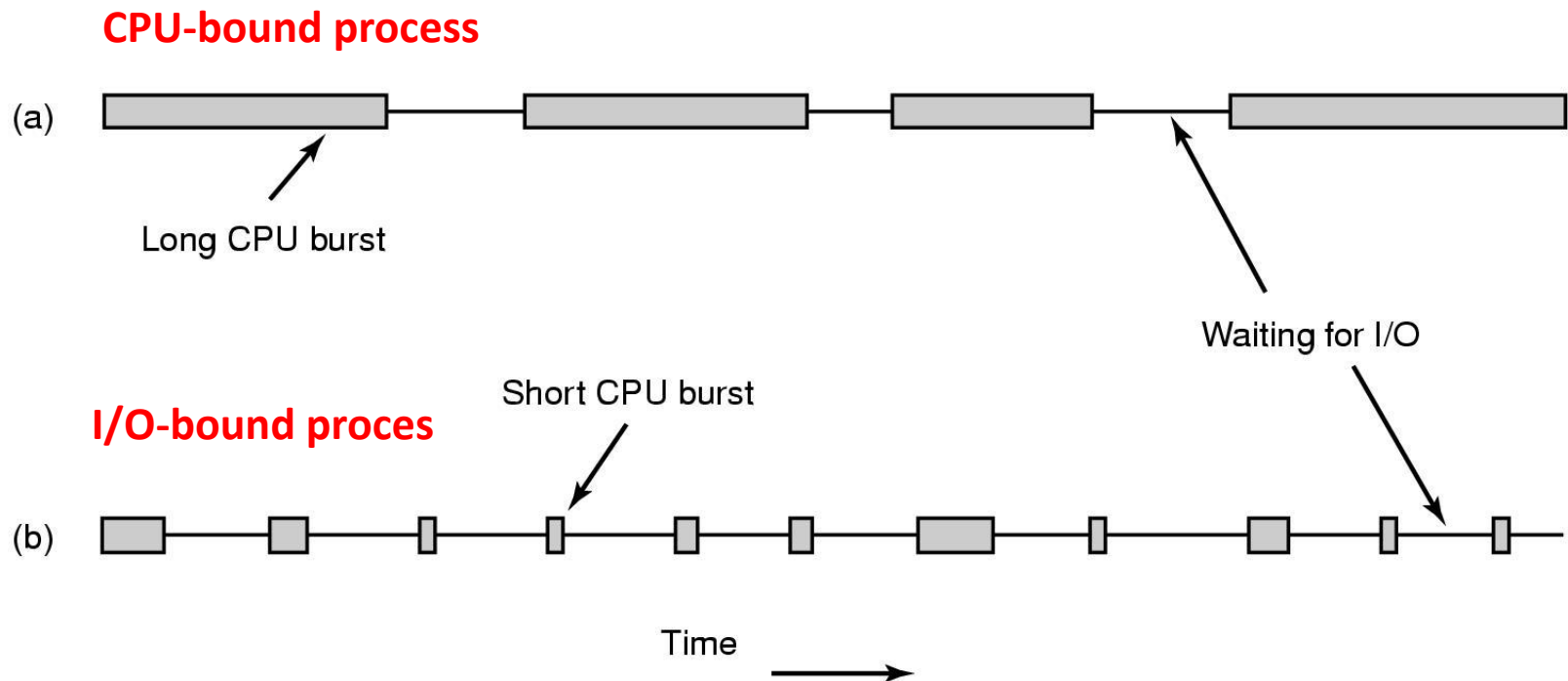
- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
 - CS162, Operating System and Systems Programming, University of California, Berkeley
 - Book: Modern Operating Systems, Andrew Tenenbaum, and Herbert Bos, 4th Edition, Pearson

Reading

- Book: Modern Operating Systems, Andrew Tenenbaum, and Herbert Bos, 4th Edition, Pearson
 - Chapter 2
- CS162, Operating System and Systems Programming, University of California, Berkeley

Last Class

CPU Bursts



- As processors become faster, processes tend to become more I/O bound
 - Why?
 - CPU is becoming faster than the I/O

Non-preemptive vs Preemptive scheduling

- Non-preemptive Scheduling Algorithm

- Picks up a process to run
- Let the process run until it blocks (for I/O or waiting for another process) or voluntarily releases the CPU
- A process may run for hours; it will not be forcibly suspended
 - No scheduling decisions are made during clock interrupts

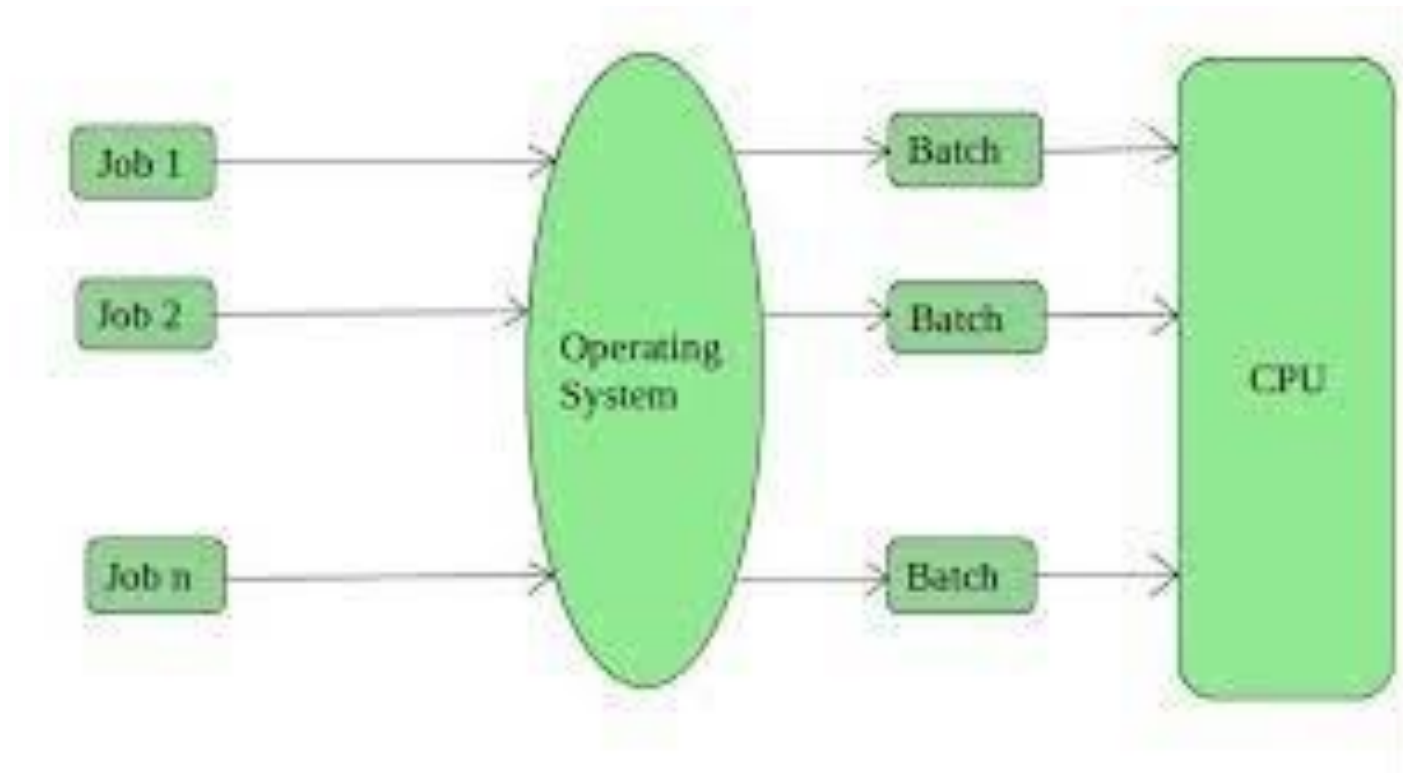
- Preemptive Scheduling

- Picks up a process to run
- lets the process run for a maximum of some fixed time
- At the end of time period, timer interrupt occurs
- In Kernel mode, scheduler picks up another ready process to run

Scheduling Algorithms - types

- Different environment require different scheduling algorithms
 - Different applications have different goals □ appropriate scheduling algorithms
- Categories of Scheduling Algorithms
 - Batch
 - Interactive
 - Real time (deadlines)

Batch System



Scheduling Policy Goals/Criteria

- Minimize Response Time
 - Time between issuing a command and getting result
- Maximize Throughput
 - Maximize operations (or jobs) per second
- Minimize Turnaround time
 - Average elapsed time – primarily for batch system
- Fairness
 - Share CPU among users in some equitable way

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

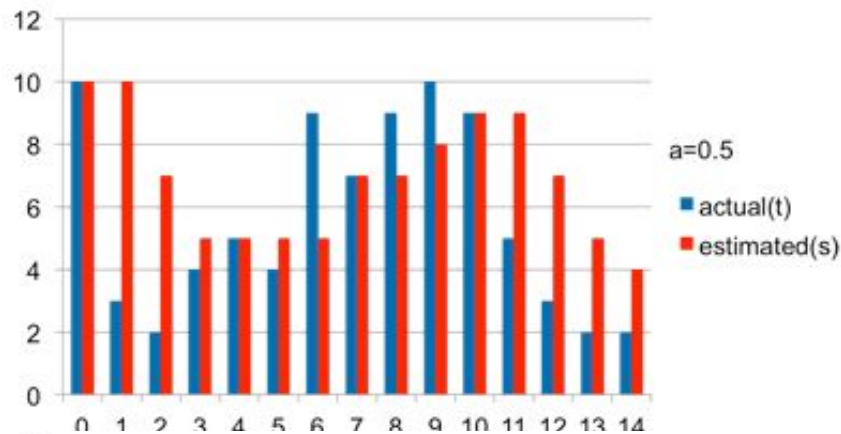
Estimating future CPU Burst time

- The SRTF algorithm can be applied to whole program or current CPU burst
- Sorting based on future burst time?
 - How do we know it ?
- Solution:
 - Predict future burst based on the past history
 - Use an estimator function on previous bursts:
Let t_{n-1} , t_{n-2} , t_{n-3} , etc. be previous CPU burst lengths.
Estimate next burst $T_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \dots)$
 - For example: Exponential Averaging
 - $T_n = \alpha t_{n-1} + (1-\alpha)T_{n-1}$ with $(0 < \alpha \leq 1)$, where
 - T_n : predicted size of the n^{th} CPU burst
 - t_{n-1} : the measured time of the $(n-1)^{th}$ burst
 - α : a weighing factor

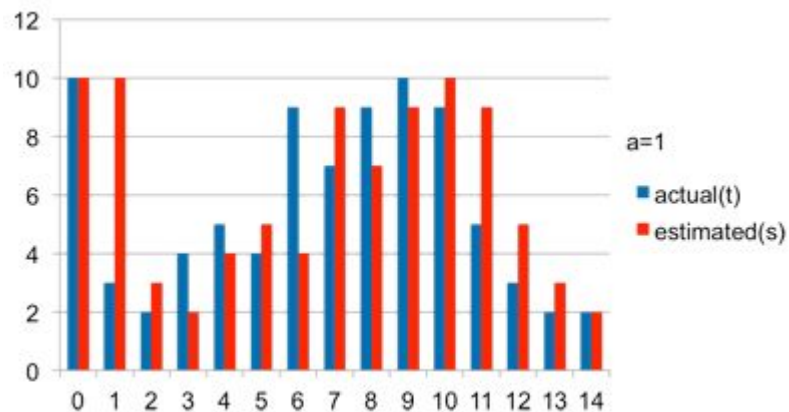
Estimating future CPU Burst time

- Weighing factor α can be adjusted based on how much to weigh past history versus last observation
- If $\alpha = 1$ then only the last observation of the CPU burst period counts
- If $\alpha = \frac{1}{2}$ then the last observation has as much weight as the historical weight

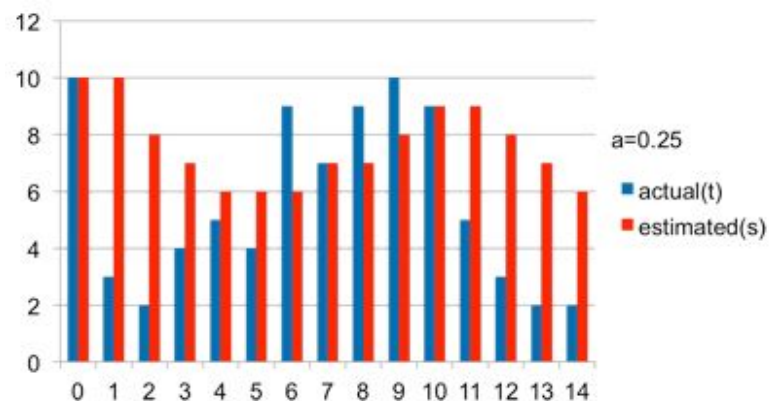
Exponential Average with $\alpha = 0.5, 1, 0.25$



Exponential Average ($\alpha = 0.5$)



Exponential Average ($\alpha = 1$)



Exponential Average ($\alpha = 0.25$)

Advantages and Disadvantages of SRTF

- Advantages

- This scheduling is optimal in that it always produces the **lowest average response time**
- Processes with short CPU bursts are given priority and hence run quickly

- Disadvantages

- Long-burst (CPU-intensive) processes are hurt with a long average waiting time
- In fact, if short-burst processes are always available to run, the long-burst ones may never get scheduled
 - **Starvation**
- The effectiveness of meeting the scheduling criteria relies on our ability to estimate the length of the next CPU burst

Useful metrics

- **Waiting time for process P :** time before P got scheduled
- **Average waiting time:** Average of all processes' wait time.
- **Completion time (response time):** Waiting time + Run time.
- **Average completion time (response time):** Average of all processes' completion time

Scheduling in Interactive Systems

Scheduling in Interactive Systems

- Round Robin
- Priority

Round Robin



"MY TURN! MY TURN!"

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

The magic number

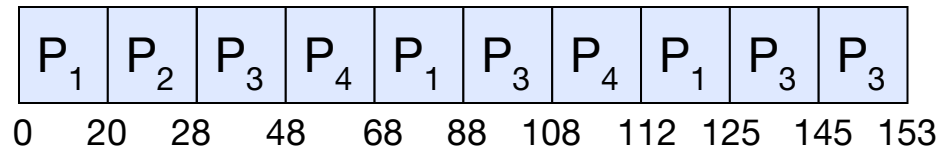
- What should q be?
 - q large \Rightarrow FCFS
 - q small \Rightarrow Interleaved
- q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum $q = 20$

- Example:

	<u>Process</u>	<u>Burst Time</u>
P_1		53
P_2		8
P_3		68
P_4		24

- The Gantt chart is:



- Waiting time for $P_1 = 0 + (68-20) + (112-88) = 72$
 $P_2 = (20-0) = 20$
 $P_3 = (28-0) + (88-48) + (125-108) + 0 = 85$
 $P_4 = (48-0) + (108-68) = 88$
- Average waiting time = $(72+20+85+88)/4 = 66\frac{1}{4}$
- Average completion time = $(125+28+153+112)/4 = 104\frac{1}{2}$

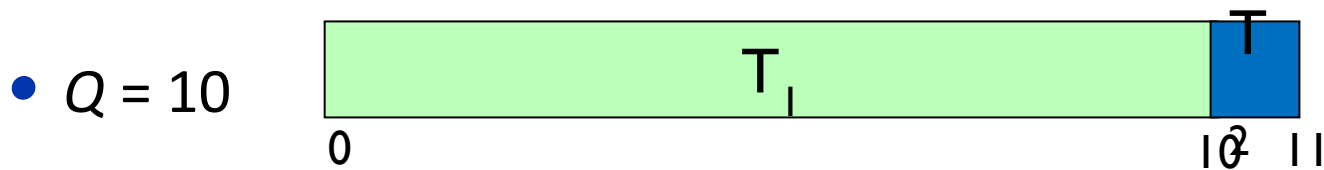
Round-Robin Quantum

- Assume that context switch overhead is 0
 - What happens when we *decrease* q ?
1. Avg. response time always **decreases** or **stays the same**
 2. Avg. response time always **increases** or **stays the same**
 3. Avg. response time can **increase, decrease, or stays the same**

Note: Response time: Time between issuing a command and getting result

Decrease in Response Time

- T_1 : Burst Length 10
- T_2 : Burst Length 1



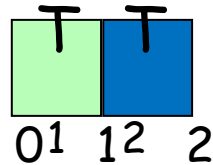
- Average Response Time = $(10 + 11)/2 = 10.5$



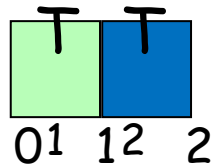
- Average Response Time = $(6 + 11)/2 = 8.5$

Same Response Time

- T1: Burst Length 1
- T2: Burst Length 1
- $Q = 10$



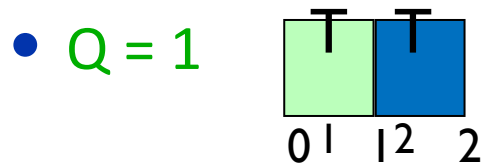
- Average Response Time = $(1 + 2)/2 = 1.5$
- $Q = 1$



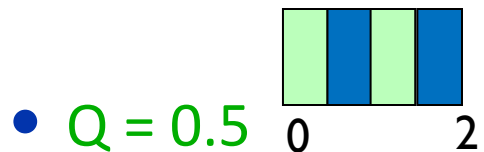
- Average Response Time = $(1 + 2)/2 = 1.5$

Increase in Response Time

- T1: Burst Length 1
- T2: Burst Length 1



- Average Response Time = $(1 + 2)/2 = 1.5$

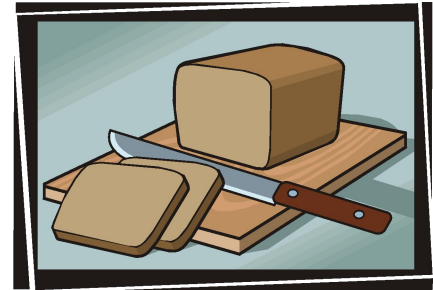


- Average Response Time = $(1.5 + 2)/2 = 1.75$

Round-Robin Scheduling: Discussion

- How do you choose time slice?

- What if too big?
 - Response time suffers
- What if time slice too small?
 - Throughput suffers!



- Actual choices of timeslice:

- Initially, UNIX timeslice one second:
 - Worked ok when UNIX was used by one or two people
- Need to balance short-job performance and long-job throughput:
 - Typical time slice today is between 10ms – 100ms
 - Typical context-switching overhead is 0.1ms – 1ms
 - Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

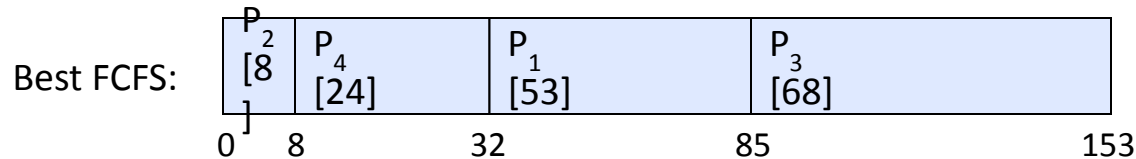
- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs start at the same time

- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
...
9	900	999
10	1000	1000

- Both RR and FCFS finish at the same time
- Average completion(response) time is much worse under RR!
 - Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
 - Total time for RR longer even for zero-cost switch!

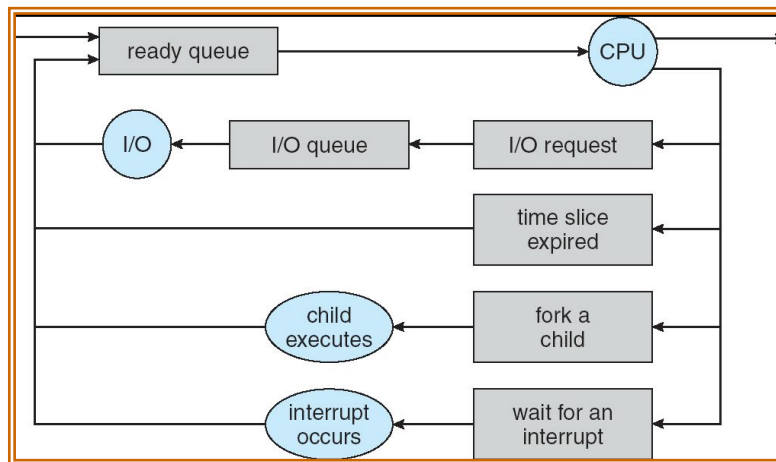
Earlier Example with Different Time Quantum



	Quantum	P ₁	P ₂	P ₃	P ₄	Average
Wait Time	Best FCFS	32	0	85	8	31½
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61½
	Q = 8	80	8	85	56	57½
	Q = 10	82	10	85	68	61½
	Q = 20	72	20	85	88	66½
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

How to Implement RR in the Kernel?

- FIFO Queue, as in FCFS
- But preempt job after quantum expires, and send it to the back of the queue
 - How? Timer interrupt!



Priority Scheduling

These people represent **waiting threads**.
They aren't running on any CPU core.

The bouncer represents a **semaphore**.
He won't allow threads to proceed
until instructed to do so.

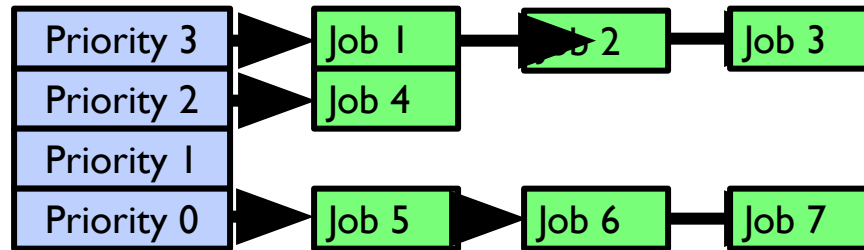


**HIGH
PRIORITY**

Priority Scheduling

- Run jobs according to their priority
- In RR or other Scheduling there is one queue of jobs
 - Scheduler selects the highest priority process
 - It takes $O(n)$ time
- Solution
 - Separate queue for each distinct priority
 - Schedule the process in the highest priority queue

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

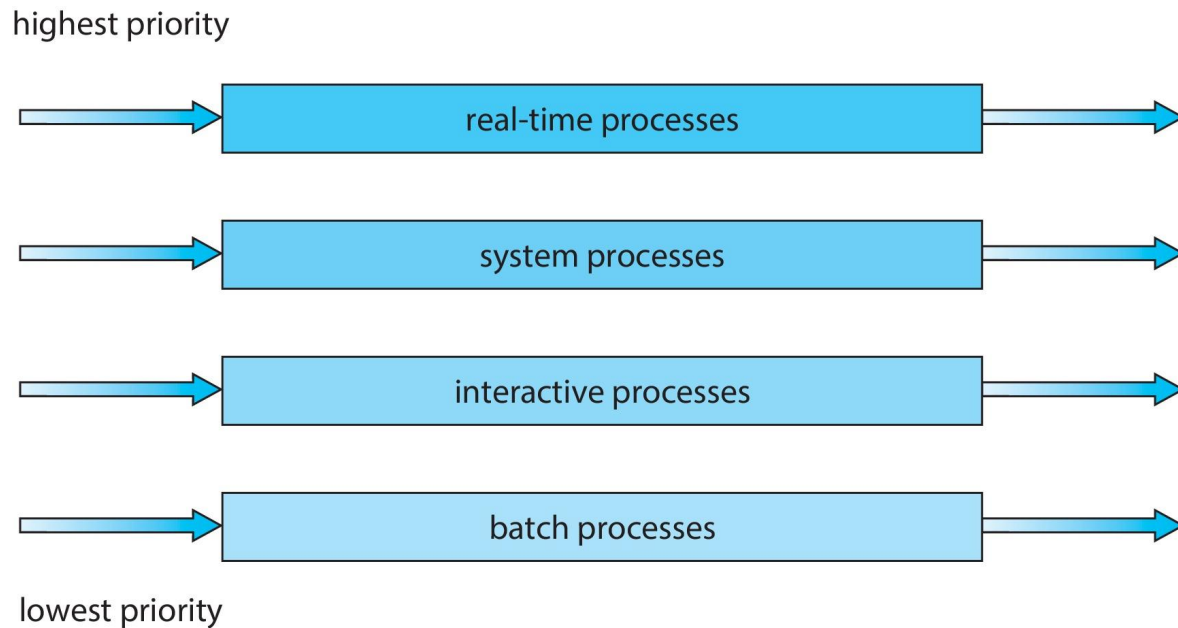
Multi level queue: Strict Priority Scheduling:

Fairness Issues

- Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
 - Long running low priority jobs may never get CPU
 - When you shut down machine, you may find 10-year-old job still waiting to run ??
- Must give long-running jobs (with low priority) a fraction of the CPU even when there are higher priority jobs to run

Multilevel Queue Scheduling (for different process types)

- Prioritization based upon process type



Adaptive Scheduling



"I purchased this great book on time management, but with my schedule I don't have the time to read it."

Adaptive Scheduling

- How can we adapt the scheduling algorithm based on threads' past behavior?
- Two steps:
 1. Based on past observations, predict what threads will do in the future
 2. Make scheduling decisions based on those predictions
- Now, let's look at the first step. How can we predict future behavior from past behavior?

Predicting Future Behavior

- Consider Round-Robin Scheduling
- If process exhausts quantum, has to be preempted
 - Consuming all of the CPU time it can: “CPU-Bound”
 - Likely to remain CPU-Bound
- If process blocks on I/O before quantum exhausted
 - Short CPU bursts, just to initiate I/O: “I/O-Bound”
 - Often interactive tasks
 - Likely to remain I/O-Bound and/or Interactive

How to implement Fairness?

- Could give each queue some fraction of the CPU?
- Could increase priority of jobs that don't get service
 - What rate should you increase priorities?
 - And, as system gets overloaded, no job gets CPU time, so everyone increases in priority
 - ⇒ Interactive jobs suffer

⇒ **Multilevel Feedback Queue**

Lecture Summary

- Interactive systems use scheduling that reduce **average response time**
- **Round Robin** is the oldest, simplest, fairest, and widely used scheduling algorithm
- In round robin algorithm, proper choice of **time quantum** is very important
 - If the time quantum is too short, there will too many context switches □ lower CPU utilization
 - If the time quantum is too big, there will be high values of wait time and response time □ poor response to short interactive requests
- Priority scheduling
 - Multilevel Queue Scheduling – Strict Priority
- **Adaptive Scheduling**

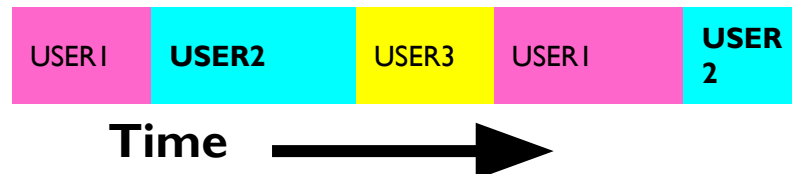
Backup

Types of Resource

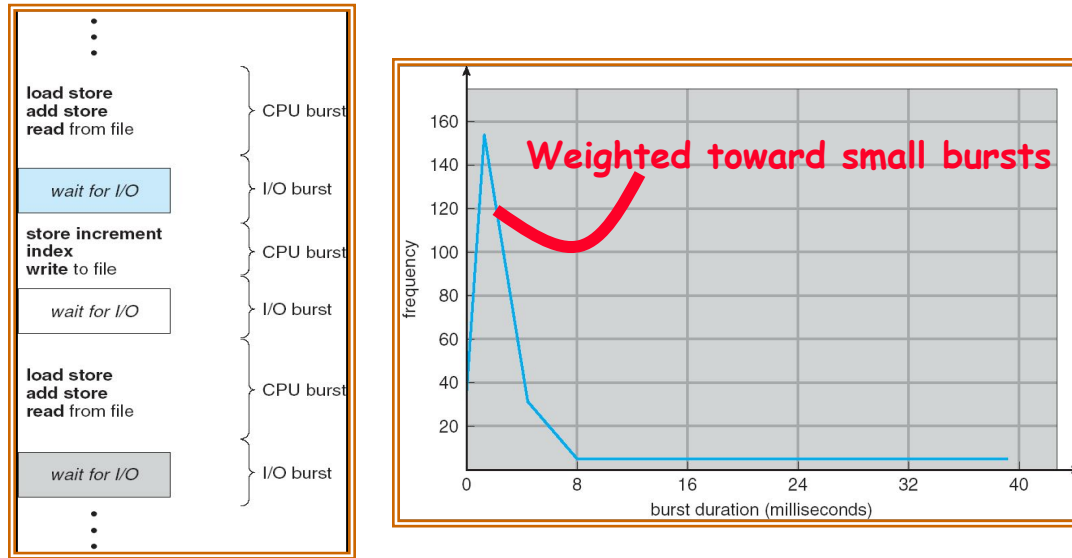
- **Preemptible**
 - OS can take resource away, use it for something else, and give it back later
 - E.g., CPU
- **Non-preemptible**
 - Once given resource, it can't be reused until voluntarily relinquished
 - E.g., disk space
- Given set of resources and set of requests for the resources, types of resource determines how OS manages it

Scheduling Assumptions

- Many implicit assumptions for CPU scheduling:
 - One program per user
 - One thread per program
 - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
 - For instance: is “fair” about fairness among users or programs?
 - If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system



Assumption: CPU Bursts



- Execution model: programs alternate between bursts of CPU and I/O
 - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
 - Each scheduling decision is about which job to give to the CPU for use in next CPU burst
 - If a process comes back after I/O wait, it counts as a fresh CPU burst
 - With time-slicing, thread may be forced to give up CPU before finishing current CPU burst