CMPT 300
Operating System I
CPU Scheduling – Chapter 5

Dr. Hazra Imran

Summer 2022

Admin notes

- Quiz 3 grades available on canvas

A2

Clicker

A non-preemptive algorithm makes a scheduling decision whenever ________________.

(1) a process changes from the blocked state to the ready state

(2) a process requests an unavailable resource

(3) a process terminates

A. only (1)

B. (2) or (3)

C. (1) (2) or (3)

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to continue executing that program

- **Dispatch latency:** time it takes for the dispatcher to stop one process and start another running

premptive algo.

Scheduling Criteria a.k.a. Metrics

- A variety of metrics are possible.
- Many metrics involve the concept of completion

Scheduling Algorithms : Measurements

- **CPU utilization**: how busy are we keeping the CPU?

1 t_0 t_4 P_1 P_2 P_3 P_4

- **Throughput**: how many processes are completed in a given unit of time P_1 P_2

2 P_1 10 unit 30

- **Turnaround time**: how long to finish a given process?
  - 3 This is wall-clock time: includes waiting on I/O, kernel overhead, …

- **Waiting time**: total time a process spends in ready state
  - 4 i.e. the process could run, but it doesn't have an available CPU

5

- **Response time**: how quickly the process begins producing output

- Scheduling algorithms can optimize for different measures

Possibilities for Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First- Come First-Served (FCFS) Scheduling

- Process ready-queue is a simple FIFO
  - New processes are added to the end of the FIFO
  - Process at the front of the FIFO gets the CPU next
  - A process holds the CPU until it blocks, yields, or terminates
  - When it yields or is unblocked, it goes to the end of the FIFO

non preemptive

- FCFS scheduling is non-preemptive or preemptive?

time sharing
system

high responsiveness

FCFS

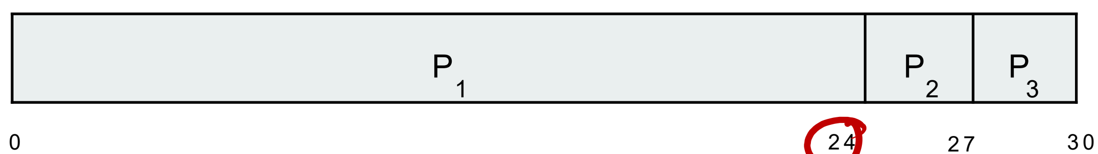
- No preemption of running processes
- Arrival times of processes are known
- Processors are assigned to processes on a first come first served basis
- Average waiting time can become quite large

First- Come First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order at time zero:
P_1 , P_2 , P_3

The Gantt Chart for the schedule is:



$P_1 = 0$
$P_2 = 24$
$P_3 = 27$

- Waiting time for each: ?
- Average waiting time: ?

$(0+24+27)/3 = 17$

First- Come First-Served (FCFS) Scheduling

Suppose that the processes arrive in the order: P_2 , P_3 , P_1

3 3 – 24

The Gantt chart for the schedule is:

17 ⇝ 3

$P_1 = 6$ $P_2 = 0$ $P_3 = 3$

- Waiting time for each: ?
- Average waiting time: ?

$(6+0+3)/3 = 3$

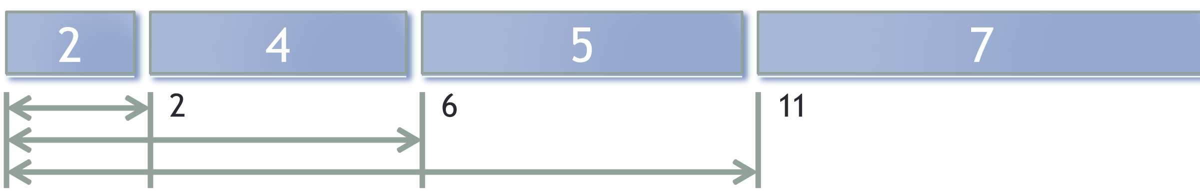
Convoy effect

↓

short process behind long process

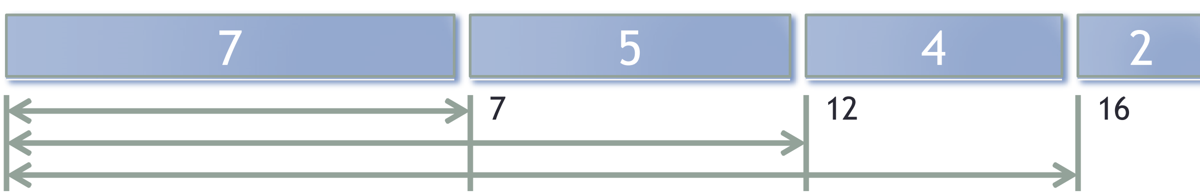
Shortest-Job-First Scheduling

- **Shortest-job-first** (SJF) scheduling orders processes based on how long their next CPU burst is expected to be
- Minimizes the average waiting time of processes
- Example: 4 processes with varying CPU-burst times:
  - 2 units, 4 units, 5 units, 7 units
- Gantt Chart of shortest-job-first ordering:



Wait times: 0, 2, 6, 11
Average wait time: 4.75

  - Longest job first (for comparison):



Wait times: 0, 7, 12, 16
Average wait time: 8.75

Example of Shortest Job First (SJF)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

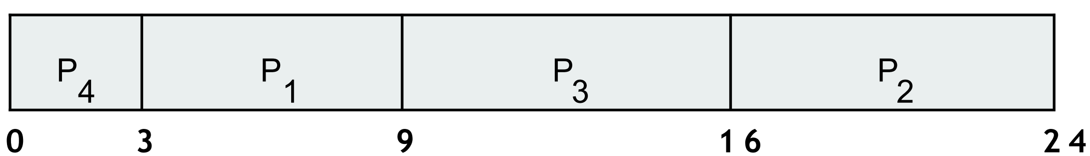
- SJF scheduling chart

- Average waiting time = ?

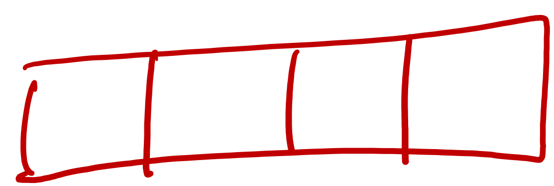
Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7



Shortest Job First (SJF) Scheduling

- Biggest challenge with SJF scheduling: Predicting the length of processes' next CPU burst!
- Usually the next CPU burst length is predicted using historical data
- Common: use **exponential average** of previous bursts

Determining Length of Next CPU Burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$

- Commonly, α set to ½

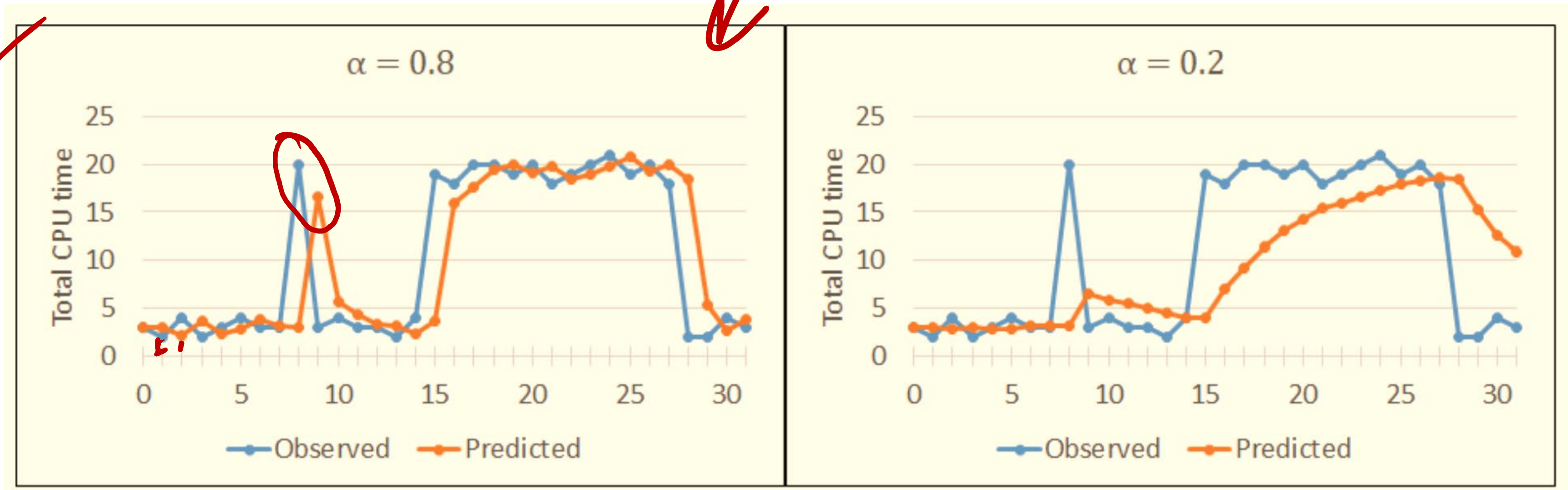
If $a = 0$, then $\tau_{n+1} = \tau n$ and recent history has no effect (current conditions are assumed to be transient).

If $a = 1$, then $\tau_{n+1} = t_n$ and only the most recent CPU burst matters (history is assumed to be old and irrelevant).

Example Burst Length Predictions

Tn

- Predicting total CPU time with α = 0.8 and α = 0.2.

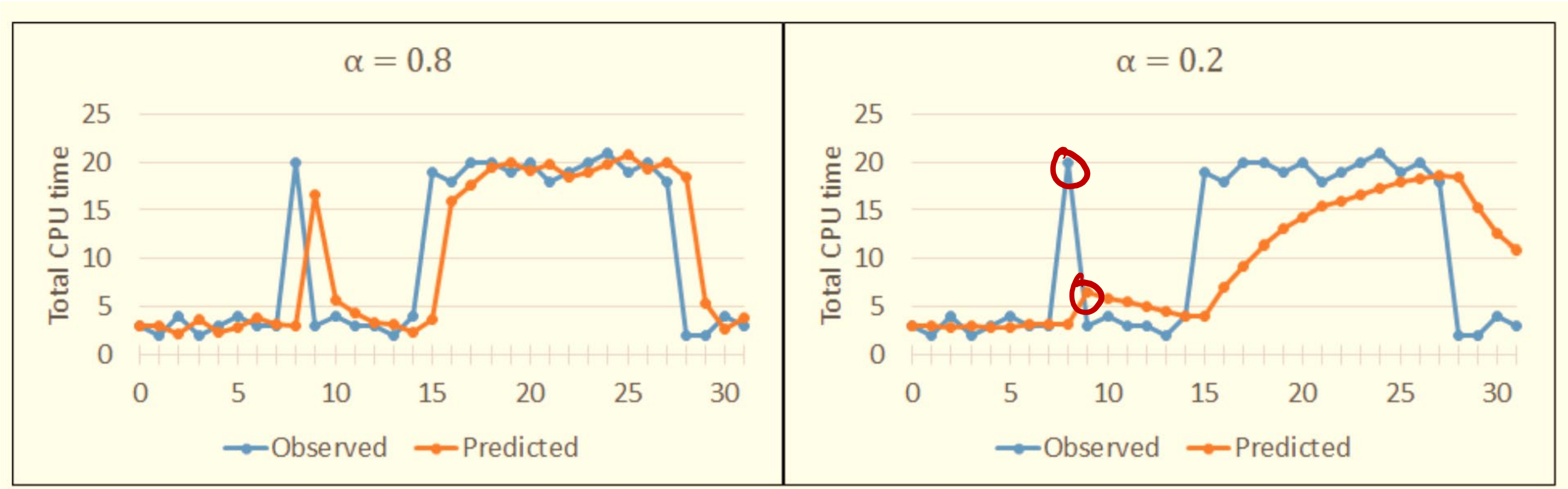


- outliers will have erratic behavior

+ very adaptble to the changes

Example Burst Length Predictions

- Predicting total CPU time with α = 0.8 and α = 0.2.



Clicker

What should be the value of α so that each prediction is based equally on the last observation and last prediction?

(A) 0.5

(B) 1

(C) No idea

Shortest-Job-First Scheduling

- Shortest-job-first scheduling can be preemptive or non- preemptive

- If preemptive, called **shortest-remaining-time-first** scheduling
  - If a new job is added to the ready queue with a shorter time, it preempts the current job on the processor

- Shortest-job-first scheduling can have **starvation** issues
  - Some ready processes may <u>never</u> receive the CPU!

- Scenario: Ready queue contains short jobs and long jobs

Example of Shortest-Remaining-Time-First

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

SRTF

- *Preemptive* SJF Gantt Chart?

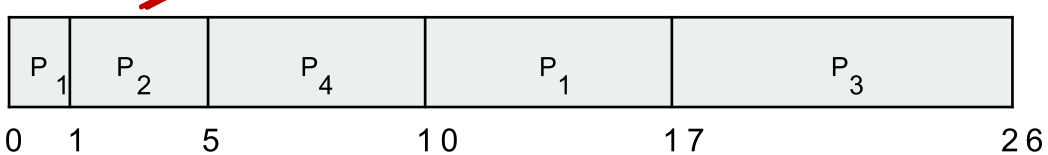
Example of Shortest-Remaining-Time-First

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart

$P_1 P_2$



- Average waiting time = ? msec

Example of Shortest-Remaining-Time-First

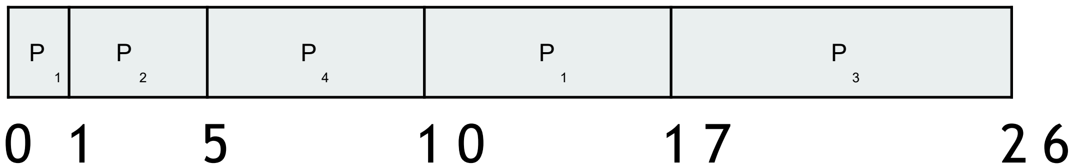
- *Preemptive* SJF Gantt Chart

P_1	P_2	P_4	P_1	P_3	
0	1	5	10	17	26

	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (CT - AT)	Waiting time (TT - BT)
P1	0	8	17	17	9
P2	1	4	5	4	0
P3	2	9	26	24	15
P4	3	5	10	7	2

Example of Shortest-Remaining-Time-First

- *Preemptive* SJF Gantt Chart



	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (CT - AT)	Waiting time (TT - BT)
P1	0	8	17	17	9
P2	1	4	5	4	0
P3	2	9	26	24	15
P4	3	5	10	7	2

Average waiting time =
9+0+15+2 = 26/4 = 6.5 ms

Priority Scheduling

- Shortest-job-first is an example of **priority scheduling**
  - In SJF, the shortest job has the highest priority

5

1

- Can also assign processes fixed priorities

int

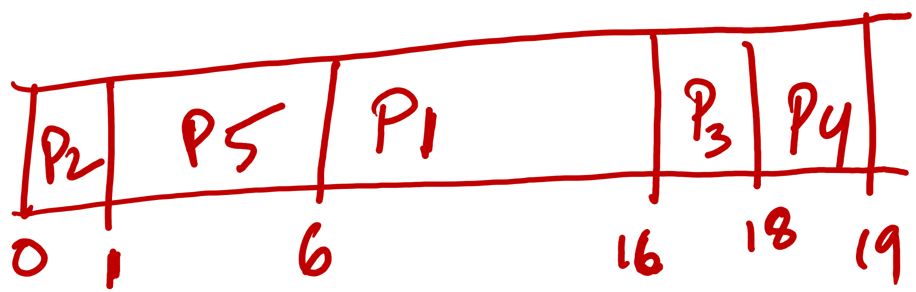
- Process priority is usually represented as a number
  - Varies whether higher or lower numbers correspond to high priority
- Priority scheduling can be **preemptive** or **non-preemptive**

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

highest

- Gantt Chart ?



- Average waiting time ?

~ 8.2

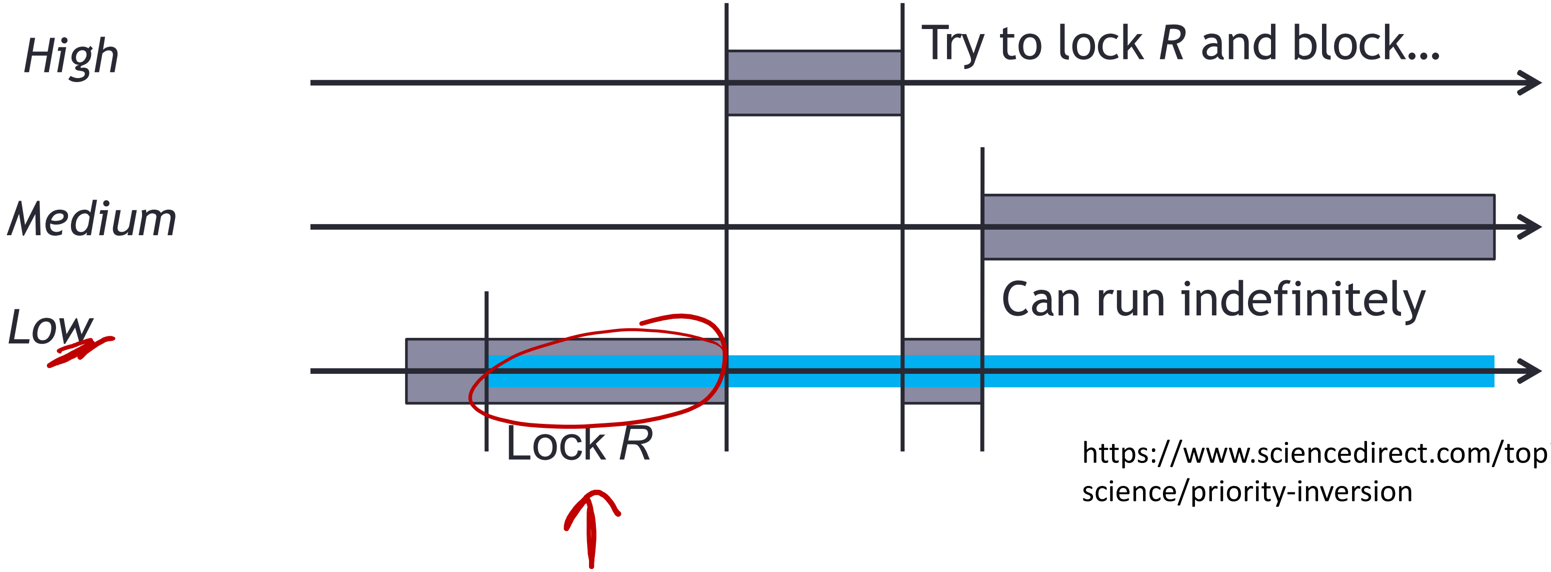
Priority Scheduling

- Priority scheduling is vulnerable to starvation
  aging
  - If high-priority processes are always able to run, lower-priority ready processes will never receive the CPU

- Priority scheduling can also suffer from **priority inversion**
  - Higher-priority processes are supposed to preempt lower-priority process…
  - Sometimes, in the context of resource locking, a lower-priority process can preempt a higher-priority process

Priority Inversion

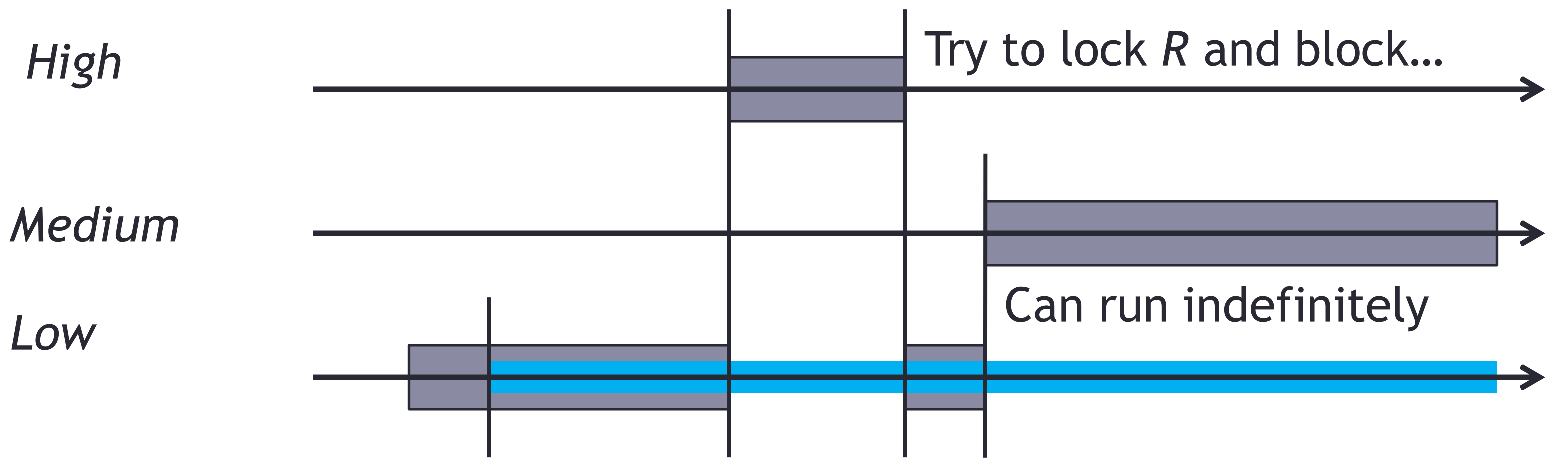
- Low-priority process *L* starts running, and locks shared resource *R*.
- High-priority process *H* starts running, preempting *L*. (But *L* still holds resource *R*.)
- *H* needs resource *R*, and attempts to lock it. *H* blocks; *L* resumes.
- Medium-priority process *M* starts running, preempting *L*. *M* doesn't need *R*, and it continues to run as long as it likes.



https://www.sciencedirect.com/topics/computer-science/priority-inversion

Priority Inversion

- Because *L* is preempted by *M*, it can never finish and release *R* so that *H* can resume its execution.
- Because high-priority processes often carry out system- critical tasks, frequently has very serious consequences



Priority Inversion: Solutions

- Several solutions to priority inversion issue
- **Random boosting** (Microsoft Windows)
- **Priority ceiling** protocols
  - Every lockable resource is assigned a priority ceiling: the highest priority of any process allowed to lock it
- **Priority inheritance** (aka **priority donation**) protocols
  - If a high-priority process *H* is blocked waiting for a resource held by a low-priority process *L*, *H* temporarily donates its priority to *L*
  - A process' priority is the maximum of its own priority, and the priorities of all processes it is currently blocking

Priority Donation

- Priority donation has its own issues
- Frequently, blocked processes can form a chain