BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

CPU Scheduling Algorithms

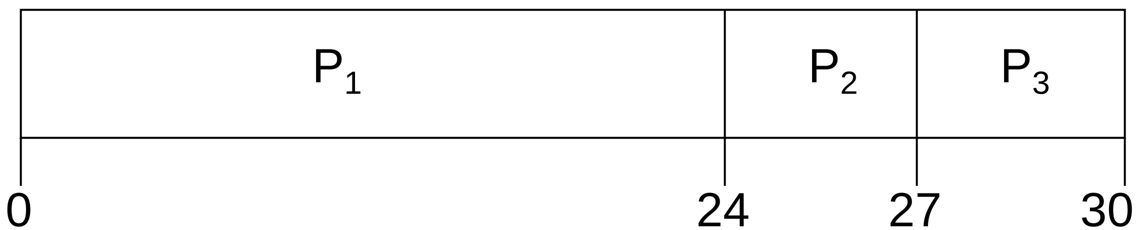
<u>Notice:</u> The slides for this lecture have been largely based on those from the course text *Operating Systems Concepts, 9th ed.*, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source. Revised by X.M. from notes by Perrone.

Scheduling Algorithms

First-Come, First-Served (FCFS)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The **Gantt Chart** for the schedule is:



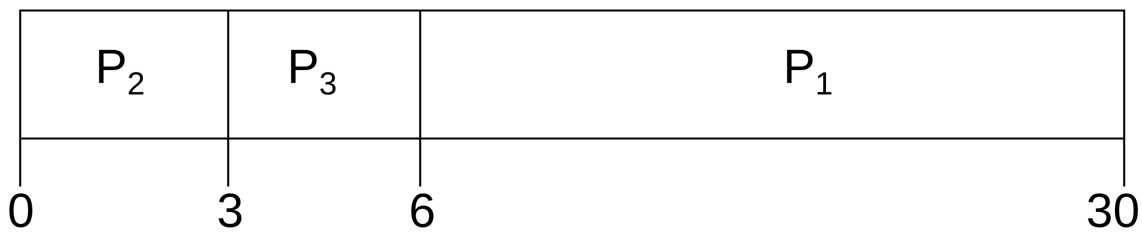
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Issues with FCFS

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case.
- *Convoy effect:* all process are stuck waiting until a long process terminates.

Shortest-Job-First (SJF)

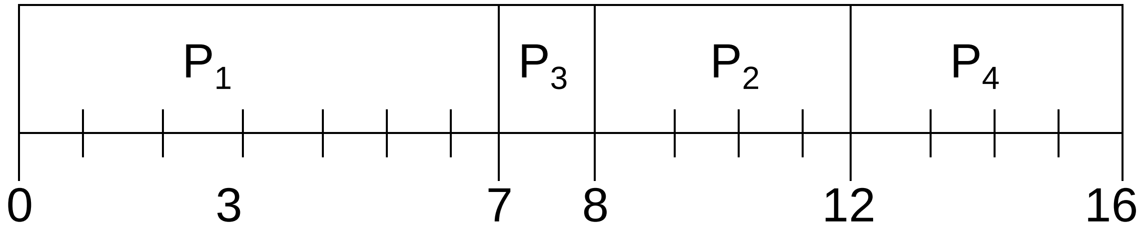
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - **Nonpreemptive** – once CPU given to a process it cannot be preempted until completing its CPU burst.
  - **Preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is **optimal** – gives minimum average waiting time for a given set of processes.

Question: Is this practical? How can one determine the length of a CPU-burst?

Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (non-preemptive)

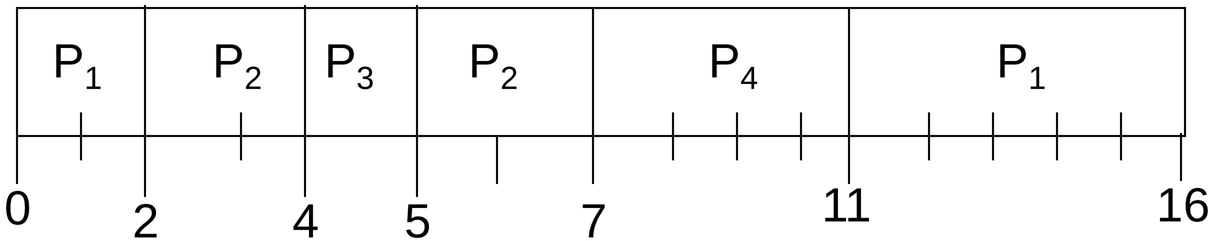


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

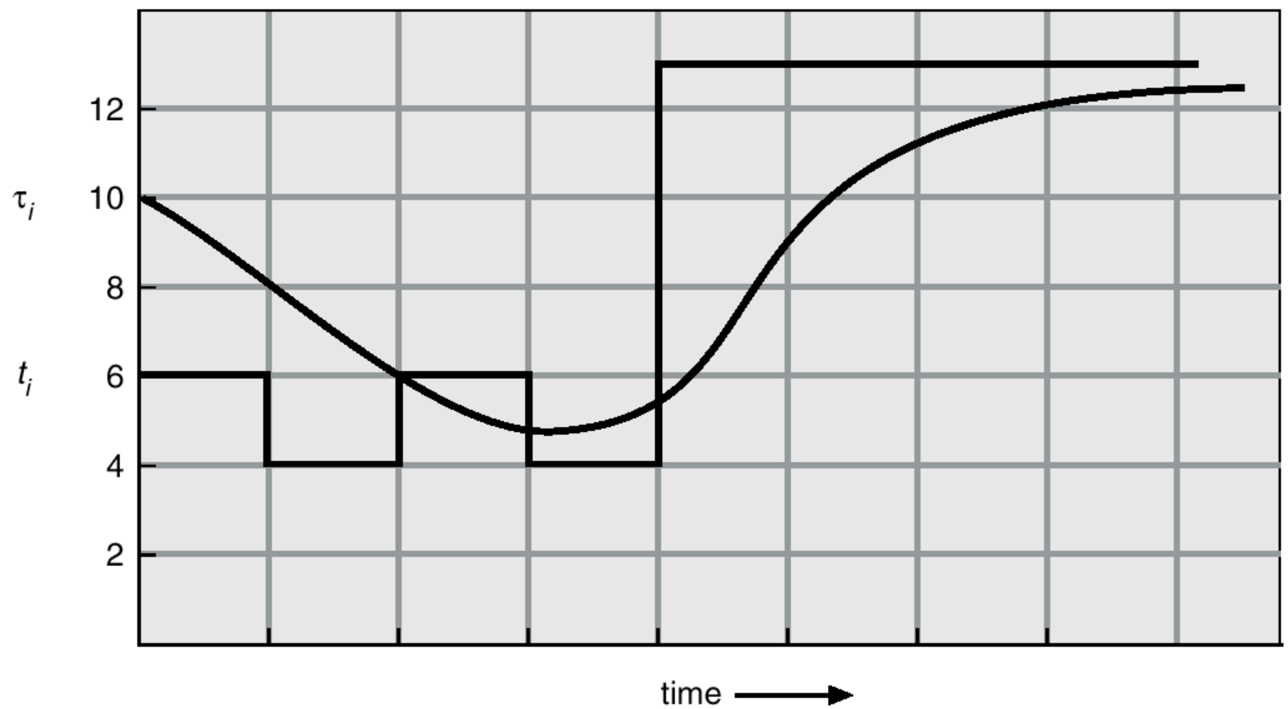
Determining Length of Next CPU-Burst

- We can only *estimate* the length.
- This can be done by using the length of previous CPU bursts, using exponential averaging:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

1. $t_n =$ actual lenght of n^{th} CPU burst
2. $\tau_n =$ predicted value for the CPU burst at time n
3. $0 \le \alpha \le 1$
4. The effect of the value of α?

Prediction of the Length of the Next CPU-Burst



CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

The graph is shown when α is 0.5

Class Exercise

- Given the actual CPU bursts are 6, 4, 6, 4, 13, 13, 13, and the initial estimate of τ is 10 as in previous slide, show the first three predictions when α takes the value of
  - 0.2
  - 0.7
- When α is 0.2, estimates are 9.2, 8.16, 7.73
- When α is 0.7, estimates are 7.2, 4.96, 5.69

Priority Scheduling

- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (typically, smallest integer $\equiv$ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU-burst time.
- Problem: **Starvation** – low priority processes may never execute.
- Solution: **Aging** – as time progresses increase the priority of the process.

Process Priority in Linux

- Priority scheduling is commonly used in production OSes such as Linux
- In Linux, the priority values range from -20 (most favorite) to 20 (least favorite)
- Try `ps al` command on a Linux terminal
- We can run a CPU intensive job and use the `nice` command to set its priority, or `renice` command to change its priority.

```
[xmeng@polaris practice]$ ./a.out &
[xmeng@polaris lectures]$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  5886 12939 11780  20   0 117528  2284 n_tty_  Ss+  pts/1      0:01 -bin/tcsh
0  5886 15993 11782  20   0 108128  1000 -      R+   pts/0      0:00 ps l
0  5886 15994 12939  20   0   3920   340 hrtime S    pts/1      0:00 ./a.out

[xmeng@polaris lectures]$ renice 10 15994
15994: old priority 0, new priority 10

[xmeng@polaris lectures]$ ps l
…
0  5886 15994 12939  30  10   3920   340 hrtime SN   pts/1      0:00 ./a.out
…
```

Round Robin (RR)

- Each process gets a small unit of CPU time (time ***quantum***), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.

RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

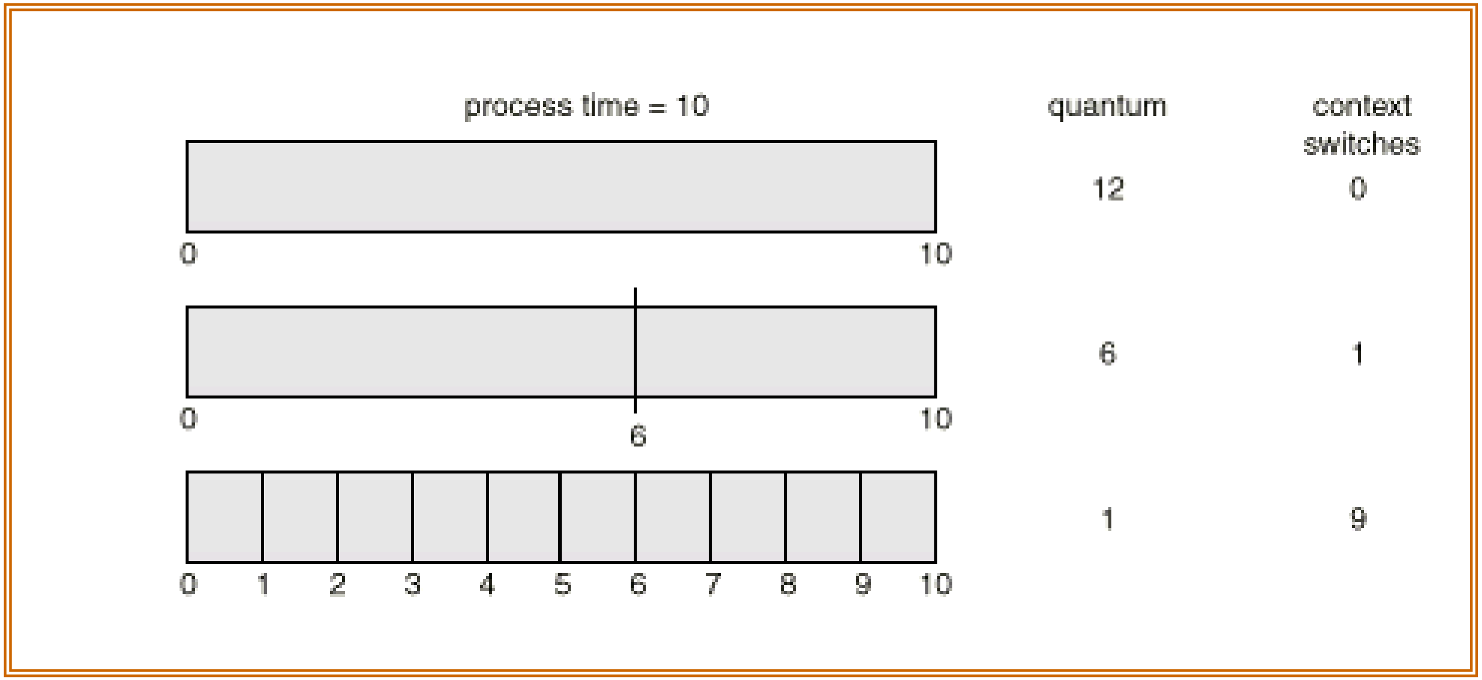
- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

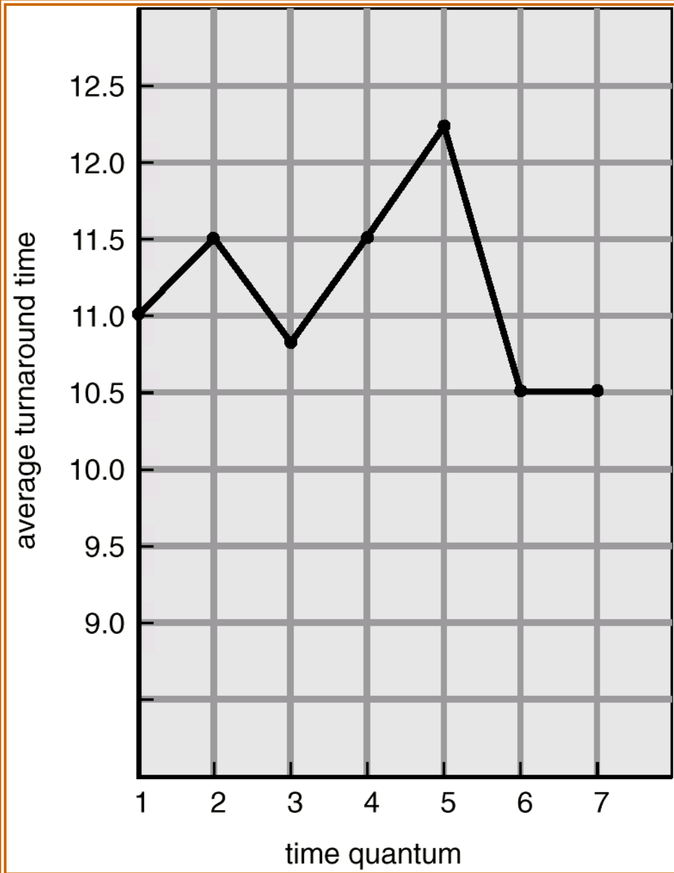
- Typically, higher average turnaround than SJF, but better *response.*

Time Quantum and Context Switch Time



Question: What influences the choice of value for the quantum?

Turnaround Time Varies with the Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Performance of RR

- Effects of the quantum length *q*:
  - *q* large ⇒ FIFO.
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high.
  - If *q* is extremely small, and we ignore the context switch cost, the result is processor sharing.

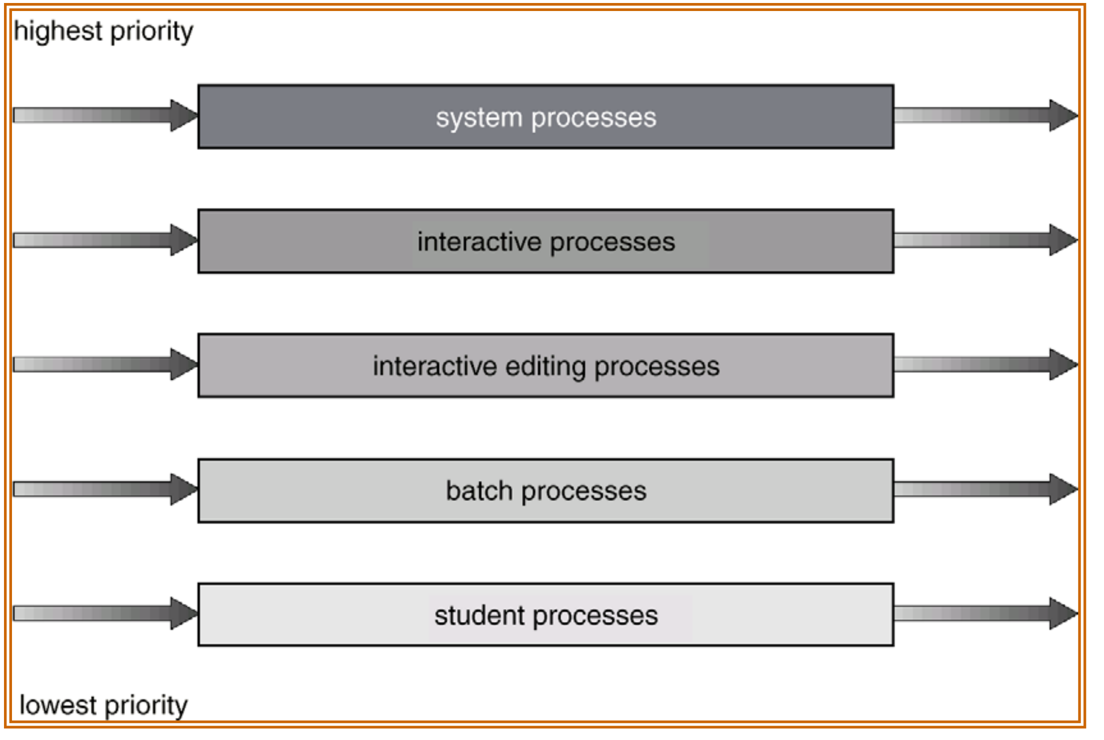
Multilevel Queue

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)

- Each queue has its own scheduling algorithm.
  - foreground: RR
  - background: FCFS

- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR.
  - 20% to background in FCFS .

Multilevel Queue Scheduling



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.

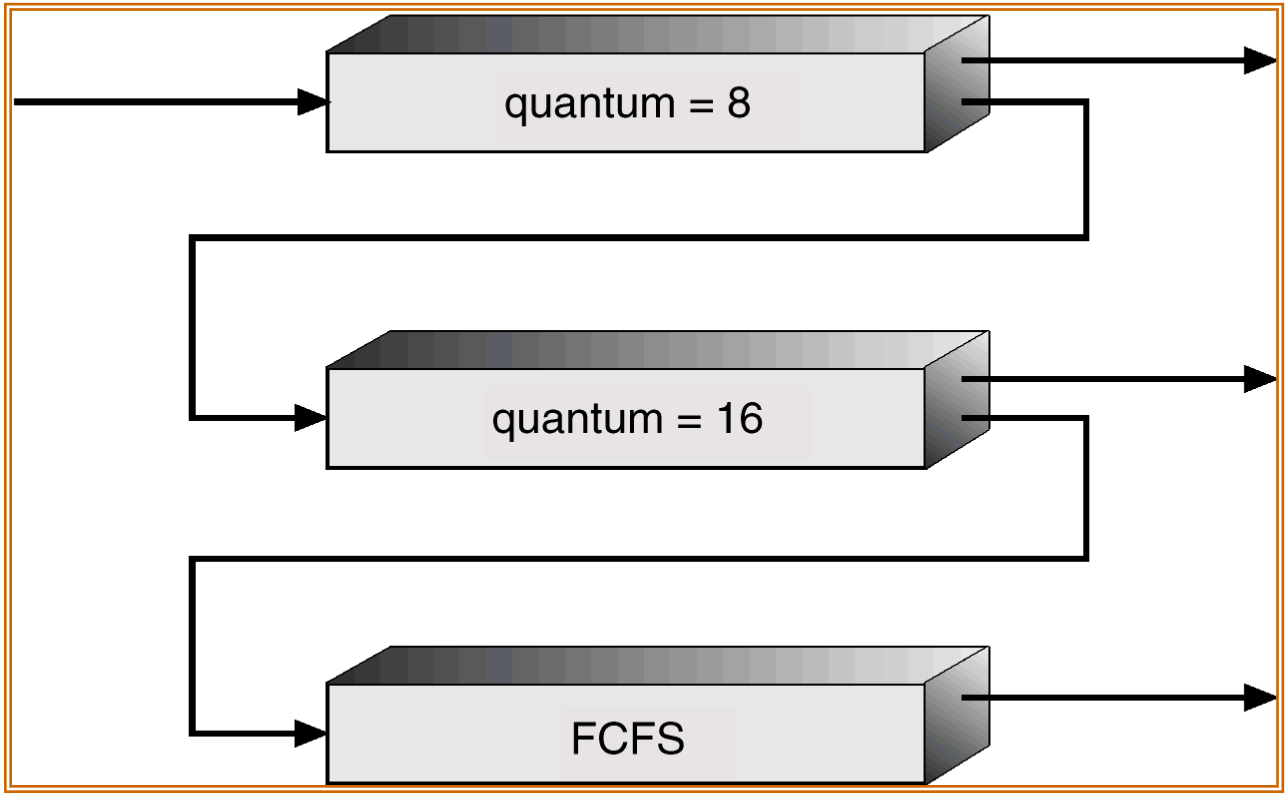
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues,
  - scheduling algorithms for each queue,
  - method used to determine when to upgrade a process,
  - method used to determine when to demote a process,
  - method used to determine which queue a process will enter when that process needs service.

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds (most favorite queue)
  - Q_1 – time quantum 16 milliseconds
  - Q_2 – FCFS (least favorite queue)

- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Linux Scheduling

- Linux maintains 140 run queues, one for each priority level
- Two sets of queues,
  - queues 0-99 for real time processes
  - queues 100-139 for regular processes
- The priority values can be set by the system call nice (2)
  - only the super users can decrement the nice values
- Different priority gives different CPU quanta

https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf

Determine Quantum Values

- Calculate quantum
  - $q = (140 - SP) \times 20$ if $SP < 120$
  - $q = (140 - SP) \times 5$ if $SP \geq 120$
  - where SP is the static priority
- Higher priority process get longer quanta
- Basic idea: important processes should run longer
- Other mechanisms used for quick interactive response

https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf

Typical Quantum Values

	Static Pri	Nice	Quantum
Highest static	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low static	130	+10	50 ms
Lowest static	139	+20	5 ms

https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf

POSIX Thread Scheduling

- A process could contain multiple threads or a single thread.
- POSIX thread scheduling defines *scope* of thread scheduling.
  - PTHREAD_SCOPE_SYSTEM : a thread contends for CPU as if it were a process
  - PTHREAD_SCOPE_PROCESS : all threads in a process are grouped together to contend for CPU

Scope Examples

PTHREAD_SCOPE_SYSTEM : If there is one process P1 with 10 threads with scope PTHREAD_SCOPE_SYSTEM and a single threaded process P2, P2 will get one time slice out of 11 and every thread in P1 will get one time slice out of 11.

PTHREAD_SCOPE_PROCESS : If there is a process with 4 PTHREAD_SCOPE_PROCESS threads and 4 PTHREAD_SCOPE_SYSTEM threads, then each of the PTHREAD_SCOPE_SYSTEM threads will get a fifth of the CPU and the other 4 PTHREAD_SCOPE_PROCESS threads will share the remaining fifth of the CPU. The amount of CPU time for the four PTHREAD_SCOPE_PROCESS threads is determined by thread scheduling policy and priority.

http://www.icir.org/gregor/tools/pthread-scheduling.html

Other Scheduling Parameters

- Other scheduling parameters can be set or examined (get) using the pthread library calls *pthread_getschedparam()* and *pthread_setschedparam()*.
- The priority and scheduling policy are meaningful only within the threads that are in the same scope.