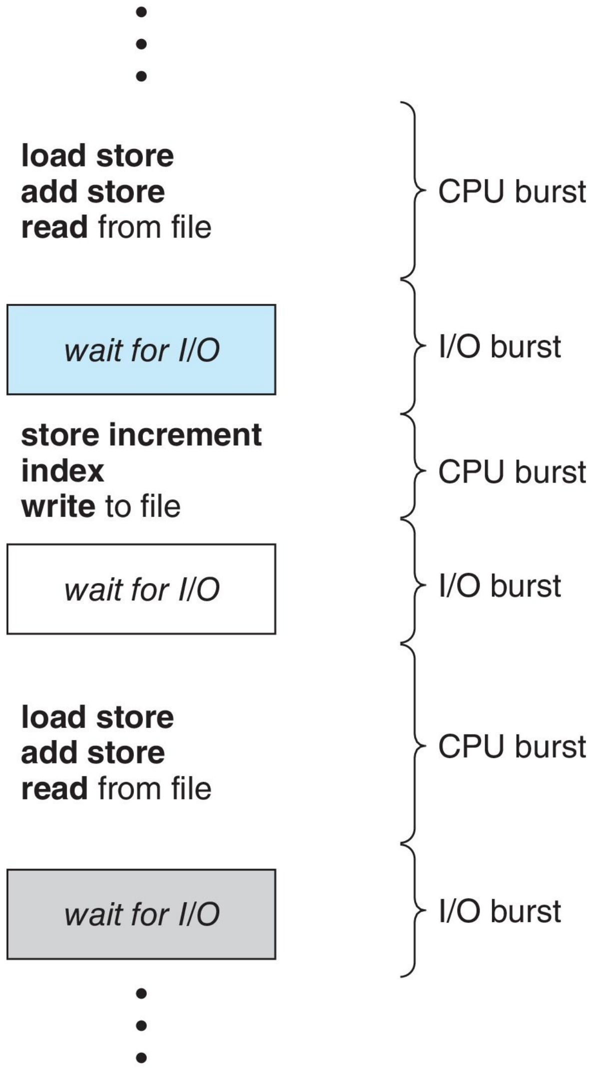
05. CPU Scheduling

CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

CPU-I/O Burst Cycle

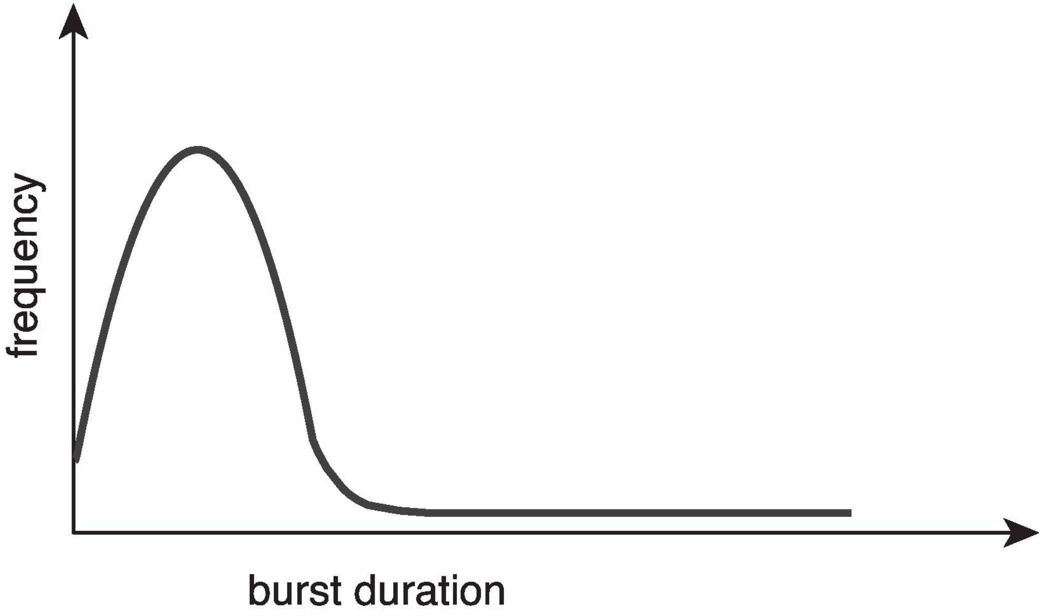
- Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst



Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts



CPU Scheduler

- The CPU scheduler selects from among the processes in ready queue, and allocates a CPU core to one of them
- CPU scheduling decisions may take place when a process:
    1. Switches from running to waiting state
    2. Switches from running to ready state
    3. Switches from waiting to ready
    4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling
    - A new process (if one exists in the ready queue) must be selected for execution
- For situations 2 and 3, however, there is a choice

Preemptive and Non-preemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **non-preemptive**
  - Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state
- Otherwise, it is **preemptive**
  - Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms
  - Preemptive scheduling can result in **race conditions** when data are shared among several processes
    - We will learn this later on

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – the number of processes that complete their execution per time unit
- Turnaround time – the amount of time to execute a particular process
- Waiting time – the amount of time a process has been waiting in the ready queue
- Response time – the amount of time it takes from when a request was submitted until the first response is produced

Scheduling Algorithm Goals

- Scheduling algorithms can have many different goals
  - Maximize CPU utilization
  - Maximize throughput
  - Minimize turnaround time
  - Minimize waiting time
  - Minimize response time
- Batch systems
  - Strive for job throughput, turnaround time (supercomputers)
- Interactive systems
  - Strive to minimize response time for interactive jobs (PC)

First-Come, First-Served (FCFS) Scheduling

- Run processes in the order that they arrive
- Example: Suppose that the processes arrive in the order: P1, P2, P3
  - Waiting time for P1 = 0; P2 = 24; P3 = 27
    - Average waiting time: (0 + 24 + 27) / 3 = 17
  - Throughput: 3 processes / 30 sec = 0.1 process/sec
  - Turnaround Time: P1 = 24; P2 = 27; P3 = 30
    - Average turnaround time: (24 + 27 + 30) / 3 = 27

Process	Burst Time
P_1	24
P_2	3
P_3	3

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order: P2, P3, P1
  - Waiting time for P1 = 6; P2 = 0; P3 = 3
    - Average waiting time: (6 + 0 + 3) / 3 = 3
  - Throughput: 3 processes / 30 sec = 0.1 process/sec
  - Turnaround Time: P1 = 30; P2 = 3; P3 = 6
    - Average turnaround time: (30 + 3 + 6) / 3 = 13
      - Much less than 27

FCFS Convoy Effect

- Short process behind long process

https://www.mirror.co.uk/news/uk-news/supermarket-fastest-queues-revealed-chain-9725720

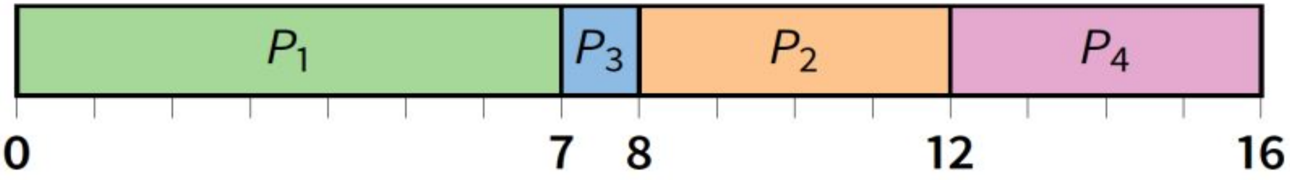
Shortest-Job-First (SJF) Scheduling

- Associate the length of its next CPU burst with each process
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal
  - It gives minimum average waiting time for a given set of processes
- Two schemes
  - Non-preemptive
    - Once CPU given to the process it cannot be preempted until completes its CPU burst
  - Preemptive
    - If a new process arrives with CPU burst length less than remaining time of current executing process, preempt
      - Known as the shortest-remaining-time-first (SRTF)

SJF/SRTF Examples

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

- Non-preemptive



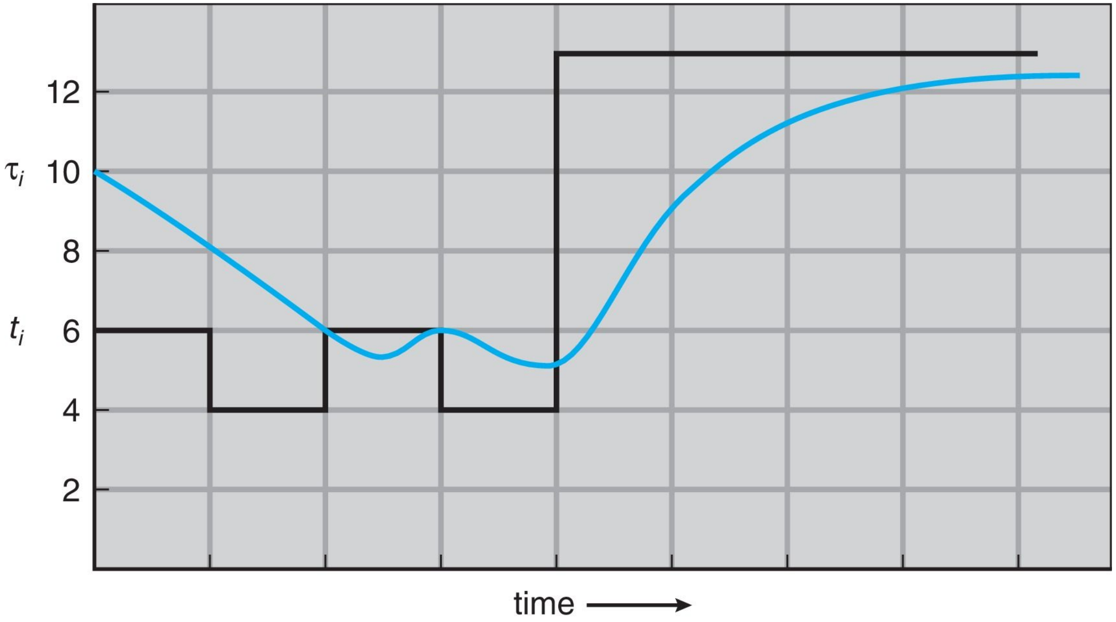
- Preemptive

Determining Length of Next CPU Burst

- The difficulty is knowing the length of the next CPU request
  - Could ask the user
  - Estimate
- We can estimate the next CPU burst using the length of previous CPU bursts
  - E.g., 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: $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

Prediction of the Length of the Next CPU Burst



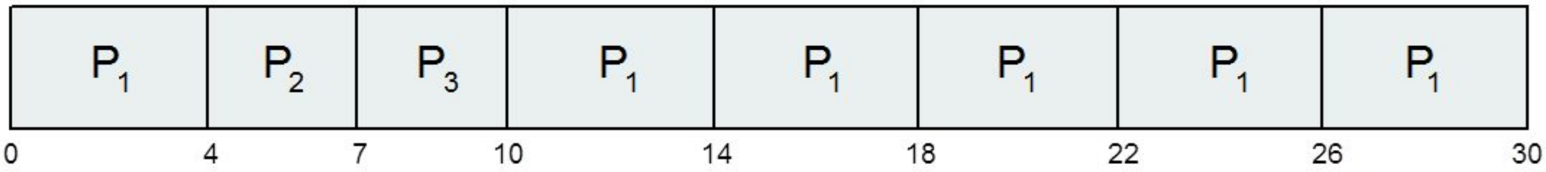
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Round Robin (RR)

- Each process gets a small unit of CPU time
  - Time quantum/slice q, usually 10-100 milliseconds
- After a quantum has elapsed, the process is preempted and added to the end of the ready queue
  - Timer interrupts every quantum to schedule next process
- If there are n processes in the ready queue and the time quantum is q
  - 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
- Performance
  - Large q → FIFO
  - Small q → q must be large with respect to context switch, otherwise overhead is too high

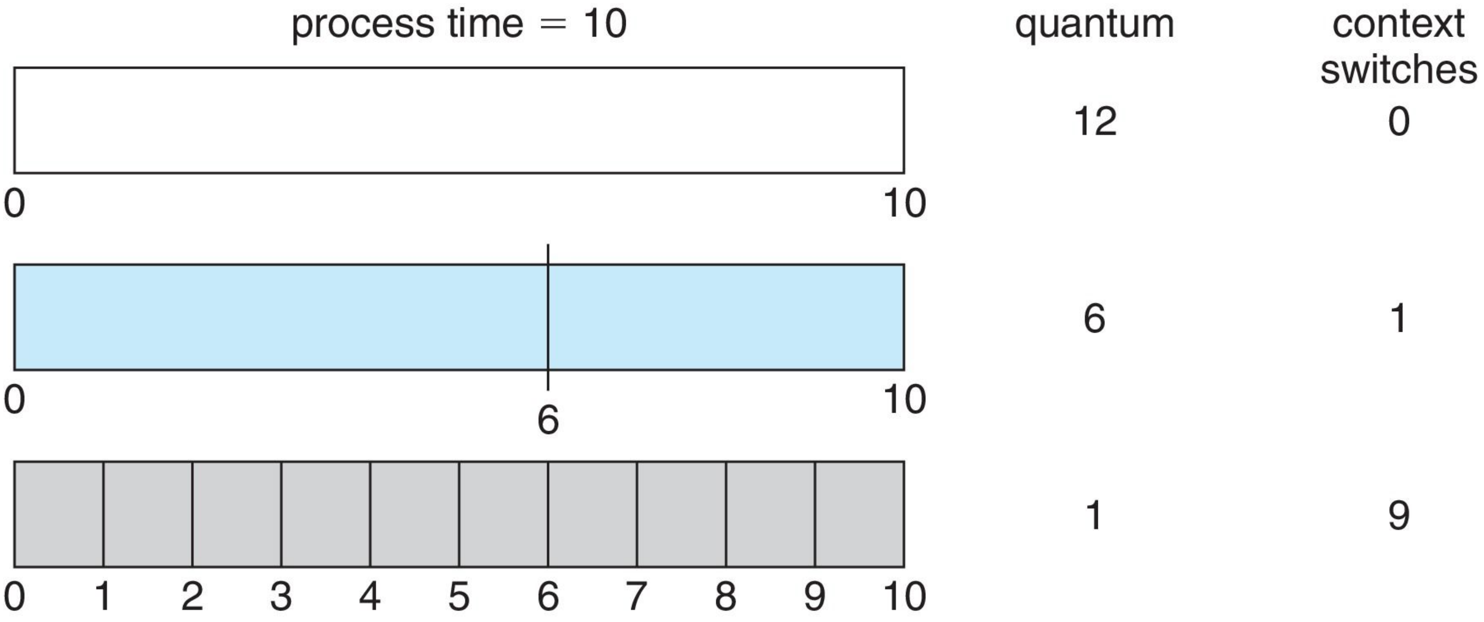
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

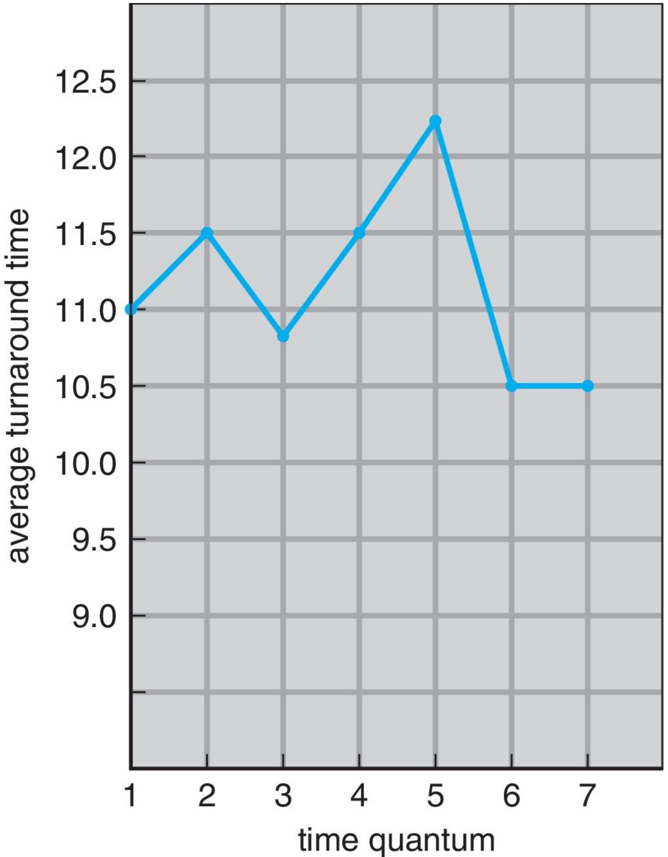


- Typically, higher average turnaround than SJF, but better response
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

Priority Scheduling

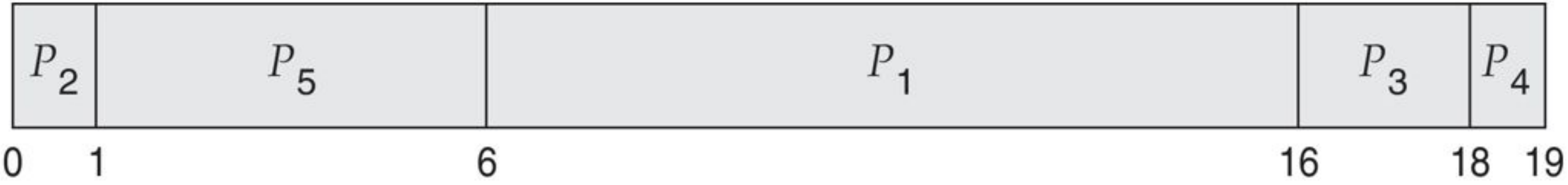
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smaller integer usually means higher priority)
  - Preemptive
  - Nonpreemptive
- SJF is actually a priority scheduling where priority is the inverse of predicted next CPU burst time

Starvation

- There is a problem of starvation
  - Low priority processes may never execute
- Solution: aging – as time progresses change the priority of processes
  - Increase priority as a function of waiting time
  - Decrease priority as a function of CPU consumption

Example of Priority Scheduling

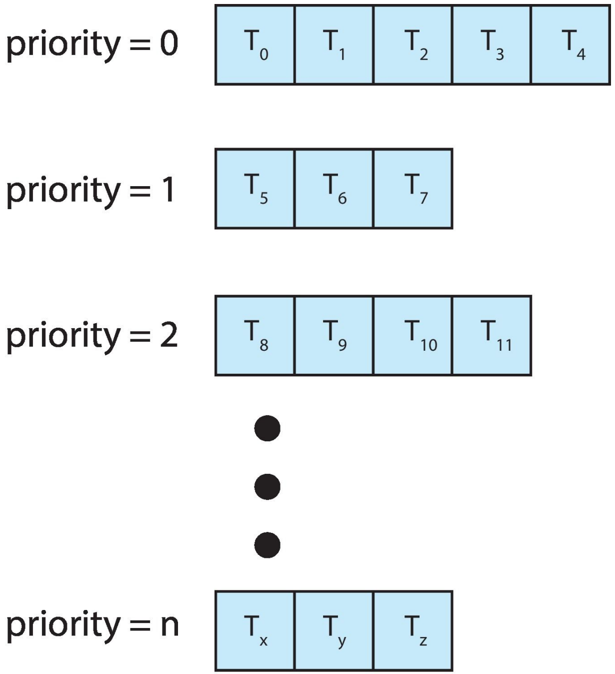
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



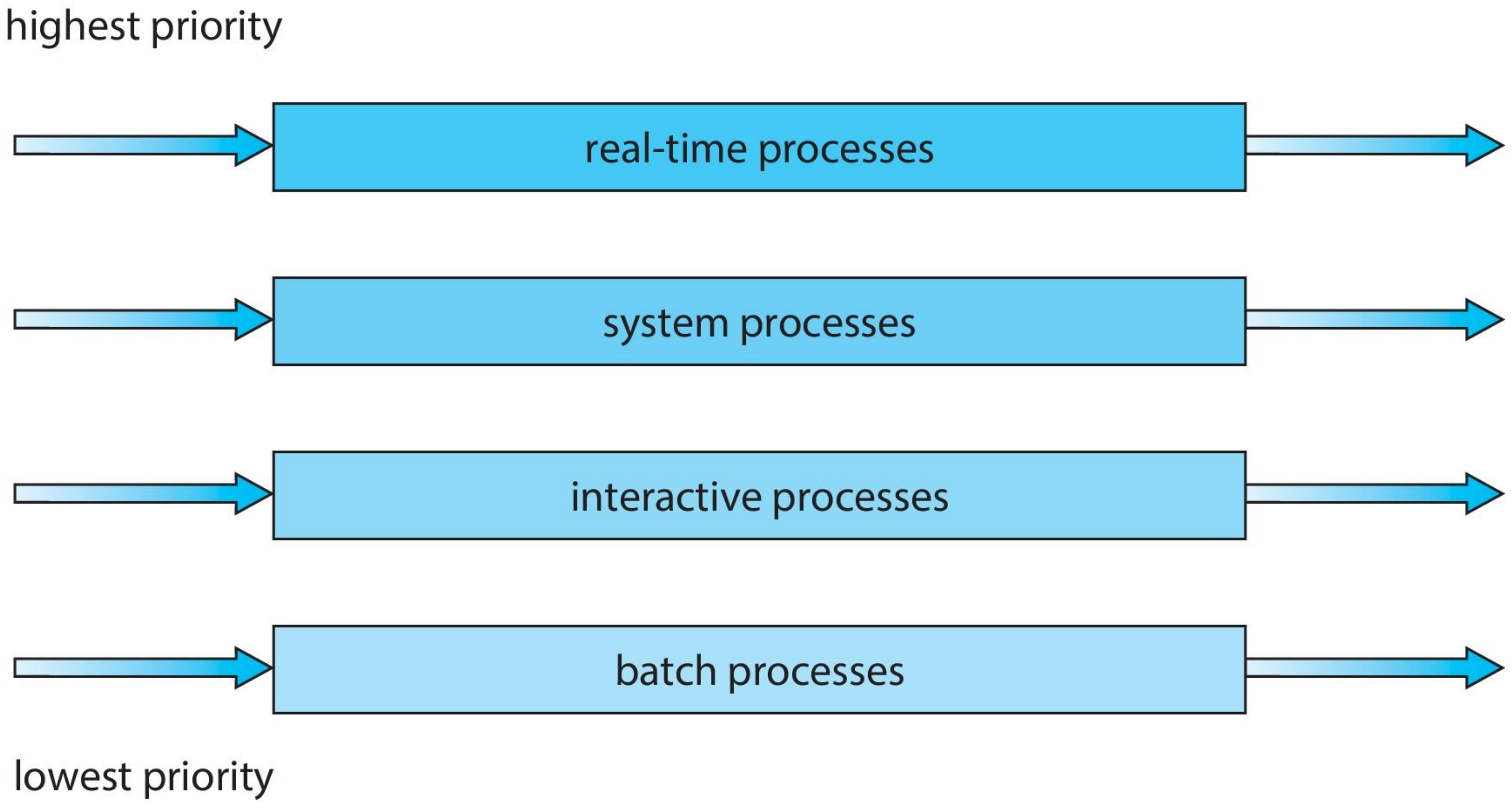
- Average waiting time = 8.2

Multilevel Queue

- With priority scheduling, we can have separate queues for each priority
  - Schedule the process in the highest-priority queue



Prioritization based on Process Type



Multilevel Feedback Queue

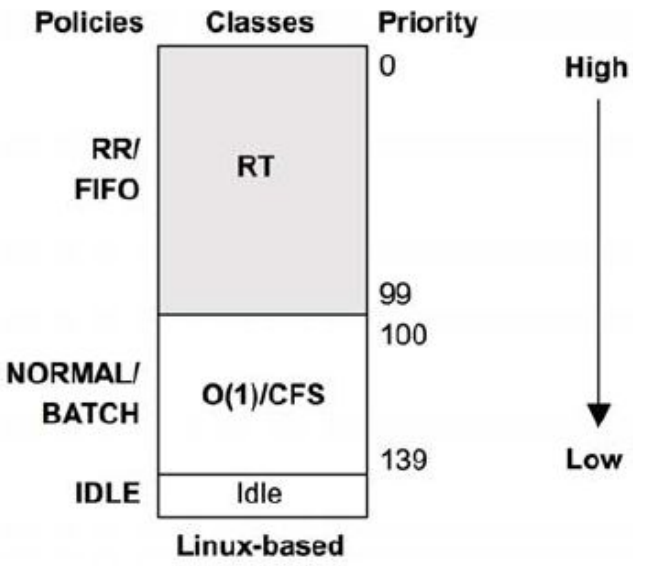
- A process can move between the various queues
  - Number of queues
  - Scheduling algorithms for each queue
  - Aging can be implemented using multilevel feedback queue
- Example: multilevel feedback queue in BSD
  - Every runnable process on one of 32 run queues
    - Kernel runs process on highest-priority non-empty queue
    - Round-robins among processes on same queue
  - Process priorities dynamically computed
    - Processes moved between queues to reflect priority changes

Brief History of Schedulers in Linux

- 1.2 kernel: circular queue for runnable tasks with RR policy
  - Simple and fast! But not scalable, no SMP (i.e., multiple processors)
- 2.2 kernel: introduced idea of scheduling classes
  - Real-time tasks, non-preemptible tasks, non-real-time tasks; support for SMP
- 2.4 kernel: O(N) scheduler (iterated over every process during scheduling)
  - Lacked scalability, weak for real-time systems, inefficient (O(N))
- 2.6 kernel: O(1) scheduler (constant time)
  - Don’t iterate over every task; two run queues for every priority level: active and expired
  - More scalable, but heuristics for determining whether a task is interactive because large and difficult to reason about
- 2.6.23 kernel: Completely Fair Scheduler (CFS)
  - Current default scheduler in Linux

Scheduler Classes

- Linux has different algorithms for scheduling different types of processes
  - Called scheduler classes
- Each class implements a different but “pluggable” algorithm for scheduling
  - Within a class, you can set the policy
- RT class: SCHED_DEADLINE, SCHED_FIFO, SCHED_RR
- CFS class: SCHED_NORMAL and SCHED_BATH
  - The “default” class is called SCHED_NORMAL; this is the CFS



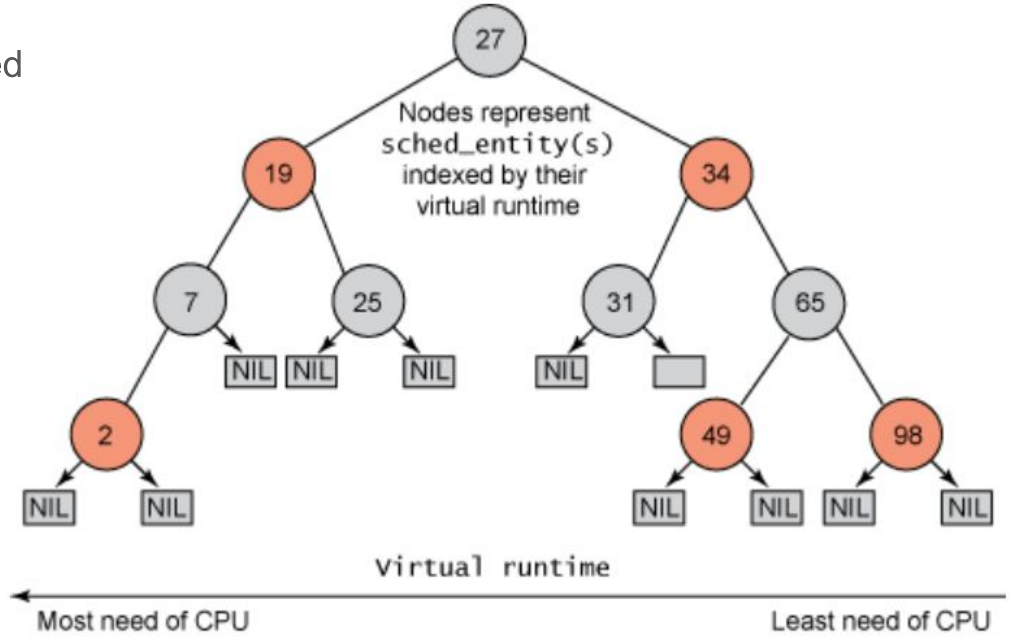
“Systems Performance: Enterprise and the Cloud” by Brendan Gregg

CFS

- Simple concept: model process scheduling as if the system had an ideal, perfectly multitasking processor
  - Each process gets 1/n of the processor’s time (n is number of runnable processes)
- Instead of a “fixed” timeslice, CFS calculates how long each process should run as a function of the total number of runnable processes
  - Use the nice value to weight this proportion of processor a process receives
  - If all nice values are equal: all processes get an equal proportion of processor time
- Uses a simple counting technique known as virtual runtime (vruntime)
  - Lower vruntime => a process hasn’t had its “fair share”

Virtual Runtime

- As a process runs, it accumulates vruntime
- When scheduler needs to pick a new process, it picks the process with the lowest vruntime
  - A red-black tree is used



How Much Time to Execute?

- How is the timeslice calculated?

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

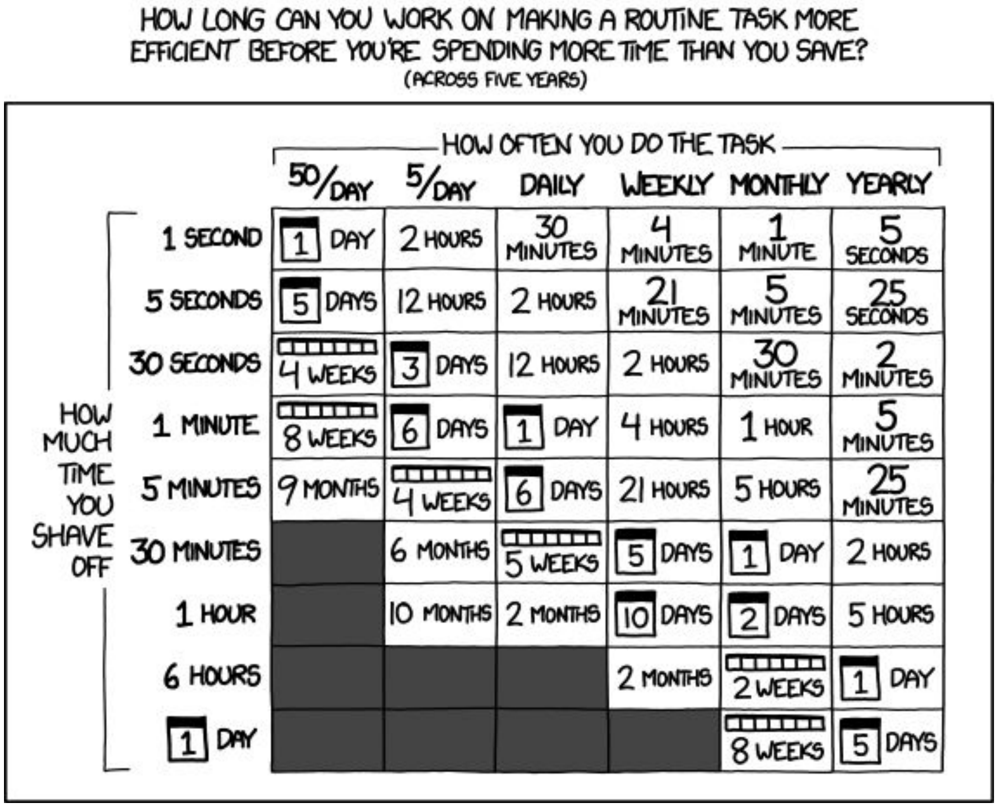
  - Negative implies higher priority (you are “less nice”)

Homework

- Read Chapter 4

Next Lecture

- We will look at threads



Credit: https://xkcd.com/1205/