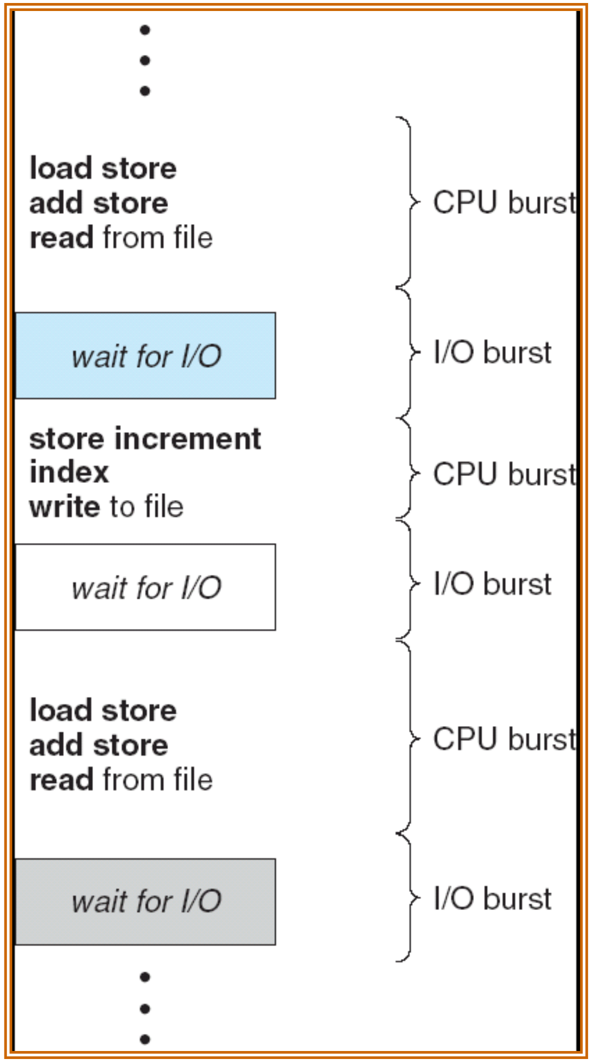
COSC 4600
Operating Systems Design

Spring 2019

Chapter 6: CPU Scheduling

Basic Concepts

- Goal
  - Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait



Scheduling Criteria

- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

CPU Scheduler

- Selects from among the processes in the ready queue, and allocates the CPU to one of them
- Non-preemptive Scheduling
- Preemptive scheduling

Non-preemptive Scheduling

- A running process is only suspended when it blocks or terminates
- Pros: CPU utilization?
  - high CPU utilization
  - Does not require special HW (like a timer)
- Cons: response time and fairness?
  - Poor performance for response time and fairness
  - Limited choice of scheduling algorithms

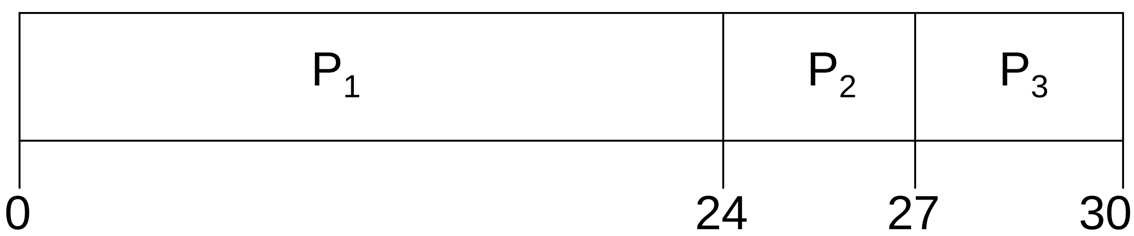
Preemptive Scheduling

- A Running process can be taken off the CPU for any (or no) reason. Suspended by the scheduler.
- Pros: response time and fairness?
  - No limitations on the choice of scheduling algorithms
  - Increased Fairness and response time
- Cons: CPU utilization? why?
  - Addition overheads (frequent context switching)
  - deceased CPU utilization

First-Come, First-Served (FCFS) non-preemptive scheduling

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



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

FCFS non-preemptive scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- 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

FCFS non-preemptive scheduling

- Pros: implementation? overhead?
  - Simple to understand and implement
  - Very fast selection for scheduling
- Cons: time-sharing OS?
  - Avg. waiting time varies
  - Not suitable for time-sharing OS

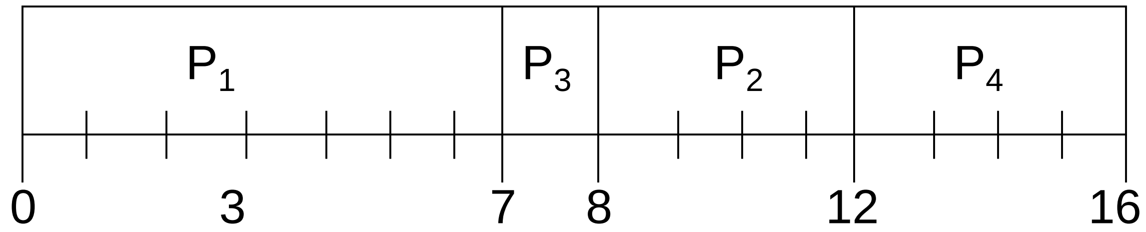
Shortest-Job-First (SJF) Scheduling

- 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 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.
- SJF gives minimum ___ for a given set of processes
  - average waiting time

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

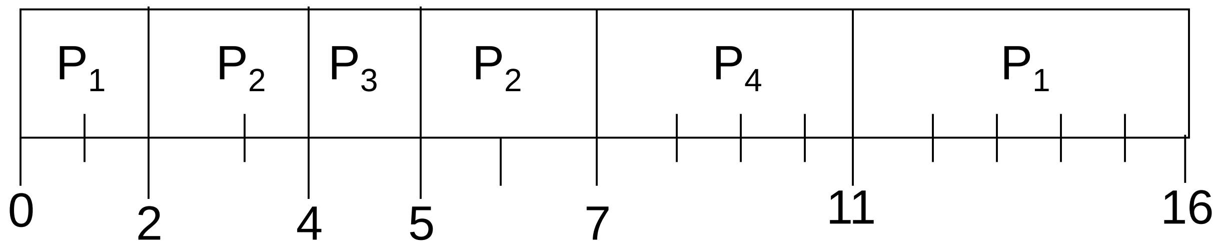


- Average waiting time = (0 + 3 + 6 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



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

Determining Length of Next CPU Burst

- Very difficult to estimate the length
- Estimate 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 : $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n .$

SJF Scheduling

- Pros:
  - Yields minimum avg. waiting time for a given set of processes
- Cons:
  - Estimation?
    - Difficult to estimate CPU burst time
  - If there is always new shorter job comes, what will happen?
    - Starvation (indefinite blocking)

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ 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

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.
- Performance
  - *q* large
    - Becomes FIFO
  - *q* small
    - *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

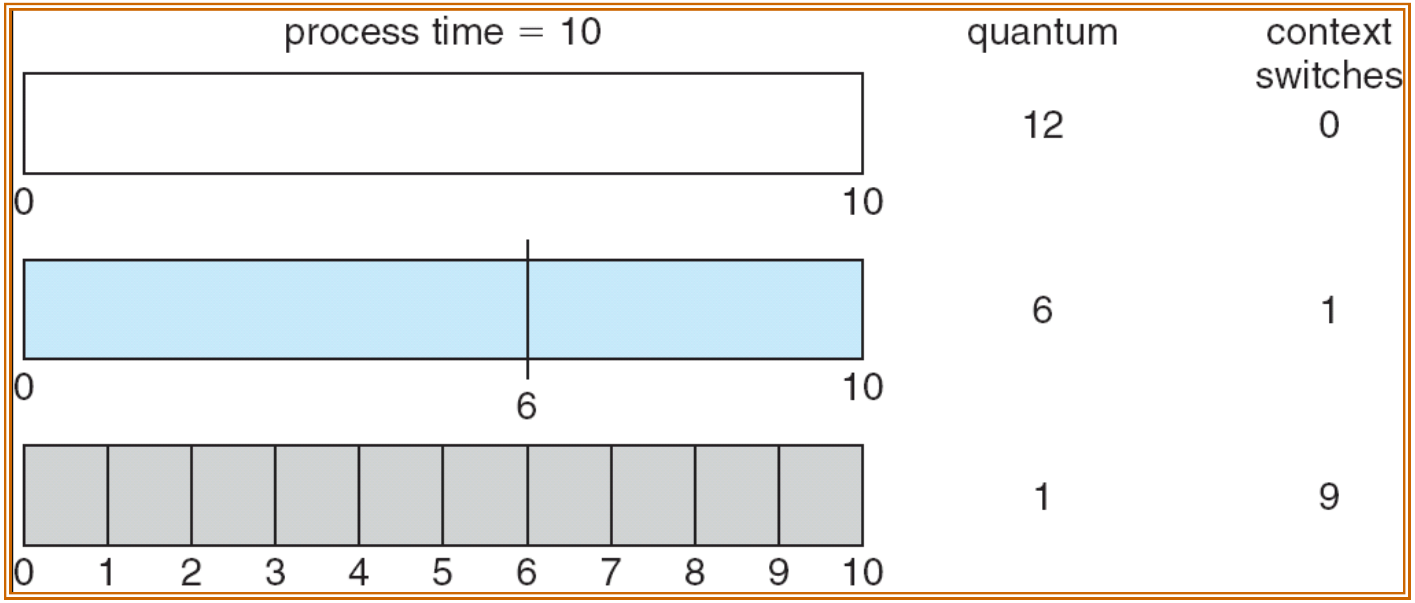
- The Gantt chart is:

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*

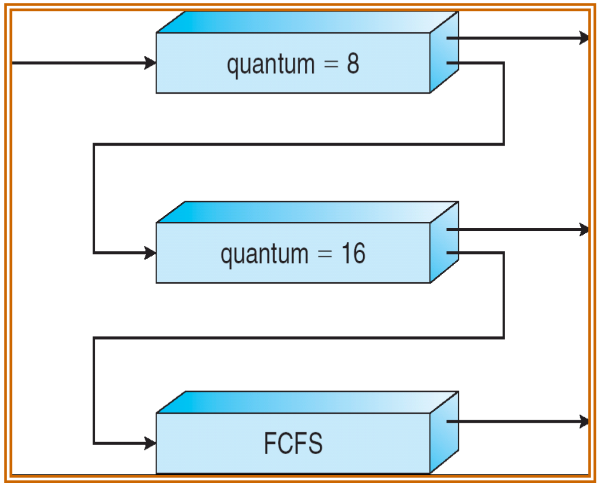
Time Quantum and Context Switch Time



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
  - Serve all from foreground then from background.
    - Possibility of starvation. Solution?
    - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes;
      - e.g. 80% to foreground in RR; 20% to background in FCFS

Multilevel Feedback Queue



- A process can move between the various queues with considering aging.
- Example
  - Three queues:
    - Q_0 – RR with time quantum 8 milliseconds
    - Q_1 – RR time quantum 16 milliseconds
    - Q_2 – FCFS
  - Scheduling
    - A new job enters queue Q_0. 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, the job is again served and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multi-Processor Scheduling

- Very complex
- Depends on the underlying HW structure
- Heterogeneous processors: Difficult!
    - Process may have code compiled for only 1 CPU.

Homogeneous processors

- Homogeneous processors with separated ready queues
  - load balancing problem
- Homogeneous processors with shared ready queue
  - Symmetric: Processors are self-scheduling
    - Need to be synchronized to access the shared ready queue to prevent multiple processors trying to load the same process in the queue.
  - Asymmetric: There is one master processor who distributes next processes to the other processors.
    - Simpler than symmetric approach

Real Time Systems

- An absolute deadline for process execution must be met.
- Take the additional parameter of deadline into account in scheduling

	P1	P2	P3	P4	P5
deadline	10:00	10:05	9:00	11:00	11:10

- Assign CPU to the process that is in the greatest danger of missing it's deadline.

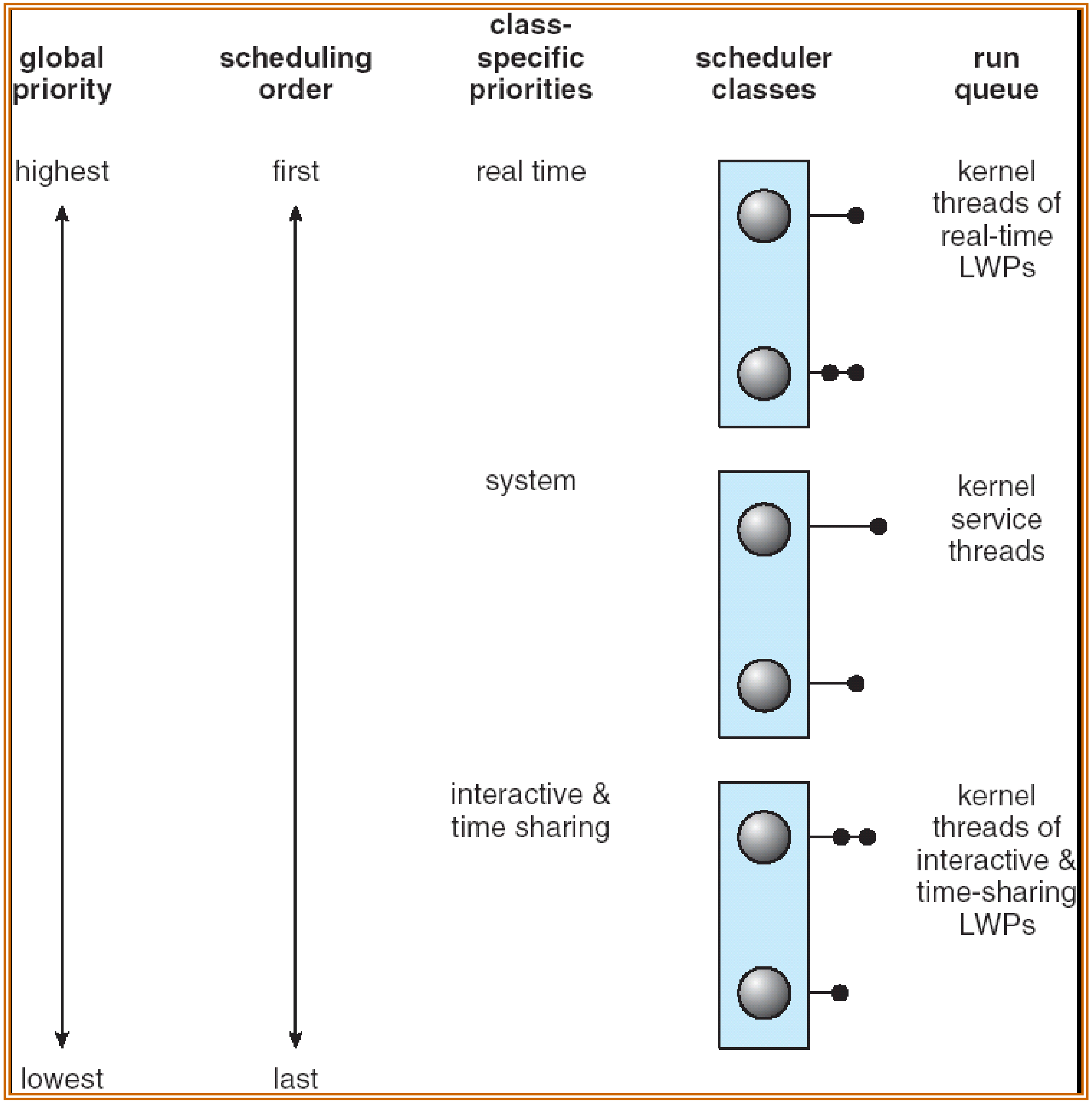
Choosing an Algorithm

- There is no “best” algorithm, just better suited.
- Need to choose the goals that are more important to the environment.

Operating System Examples

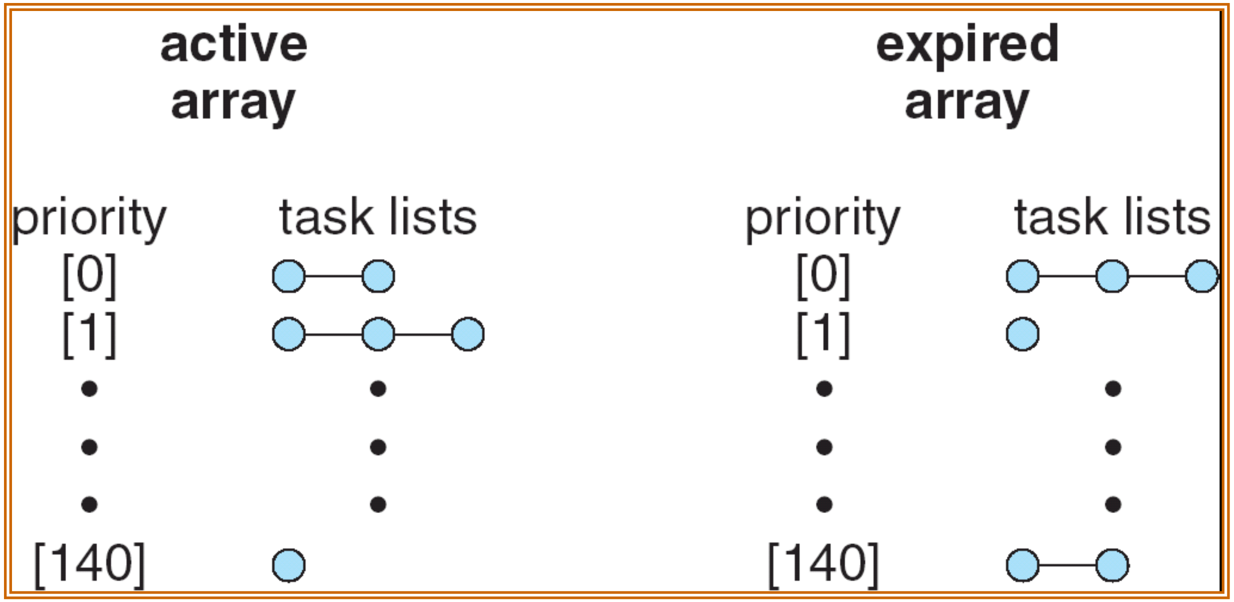
- Solaris scheduling
- Linux scheduling

Solaris 2 Scheduling



Linux Scheduling

- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history



Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP

- Global Scheduling – How the kernel decides which kernel thread to run next

Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based scheduling algorithm

- FIFO queue is used if there are multiple threads with the same priority

Time-Slicing

Since the JVM doesn't ensure time-slicing, the yield() method may be used:

```
while (true) {
    // perform CPU-intensive task
    . . .
    Thread.yield();
}
```

This yields control to another thread of higher or equal priority.

Thread Priorities

Priority	Comment
Thread.MIN_PRIORITY	Minimum Thread Priority
Thread.MAX_PRIORITY	Maximum Thread Priority
Thread.NORM_PRIORITY	Default Thread Priority

Priorities May Be Set Using setPriority() method:

setPriority(Thread.NORM_PRIORITY + 2);

Summary

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Operating Systems Examples
- Thread Scheduling
- Java Thread Scheduling