Chapter 5

CPU scheduling

Contents

Objectives

- To introduce CPU scheduling, which is basis for multiprogrammed operating system
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

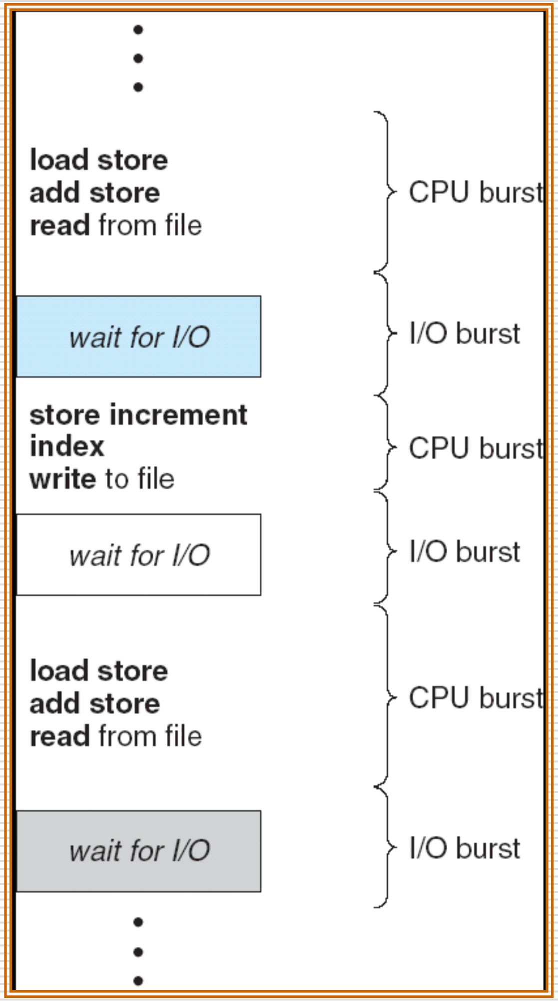
CPU Scheduling

- Problems
  - What principle
    - Algorithms
  - When to schedule
  - HoW to assign
    - Context switch

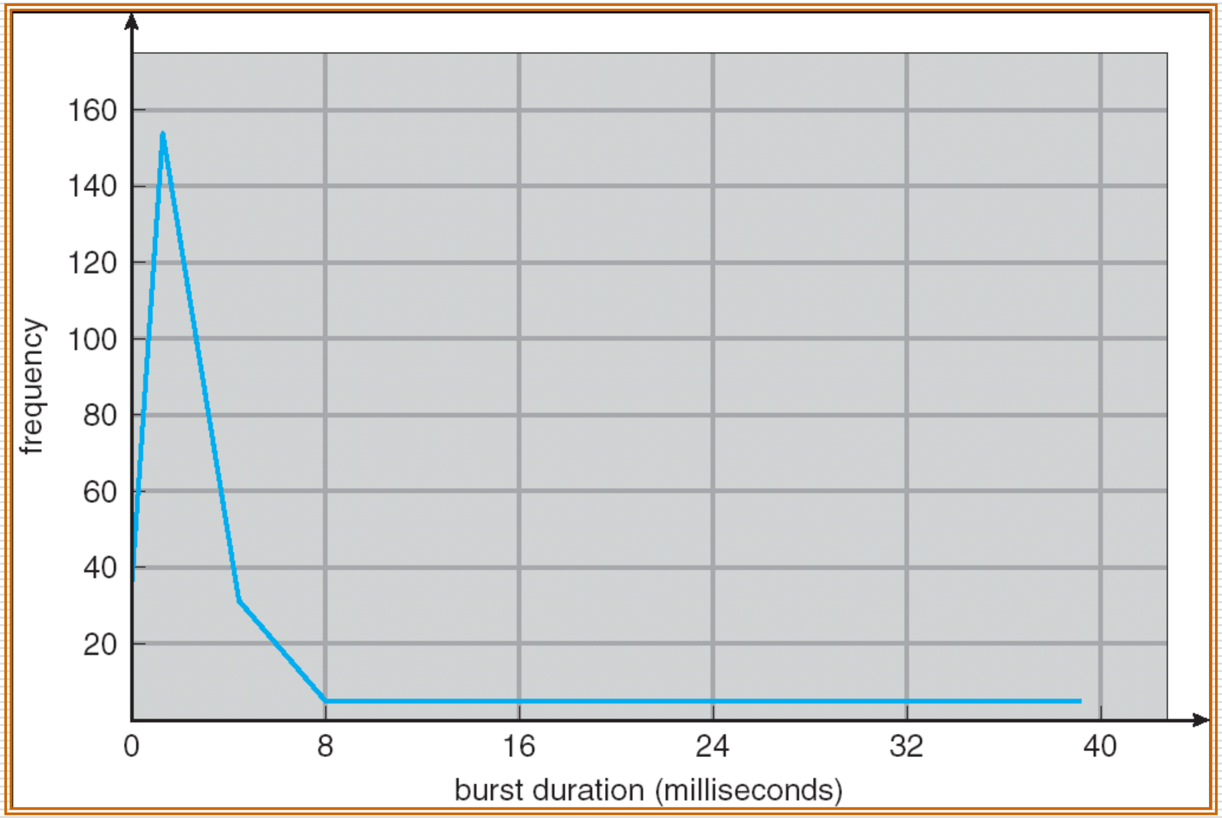
Basic Concepts

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

Alternating Sequence of CPU And I/O Bursts

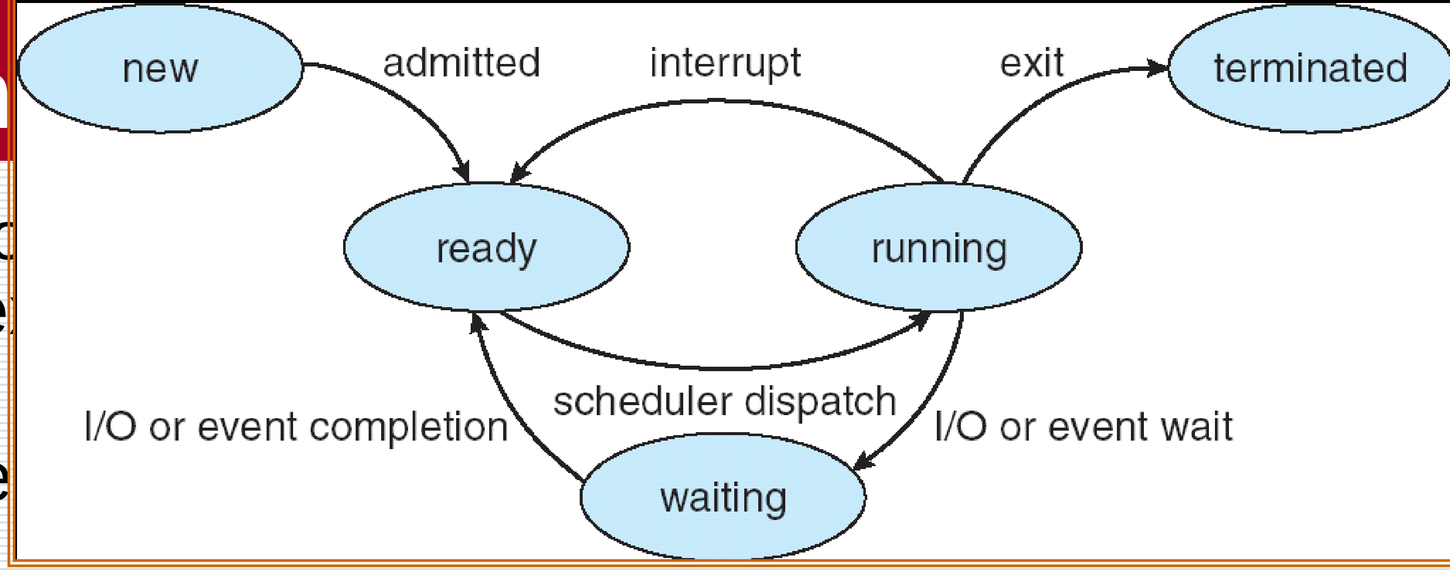


Histogram of CPU-burst Times



CPU Sch

- Selects fr
  ready to e
  them



- CPU sche
  process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready state
  4. Terminates
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*
- For preemptive scheduling, more details should be considered, e.g. the shared data.

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 restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- 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)

Optimization Criteria

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

Scheduling algorithms

- First Come, First Served, FCFS
- Shortest-Job-First, SJF
- Priority Scheduling
- Round Robin, RR
- multilevel queue-scheduling
- multilevel feedback queue scheduling

Gantt Chart

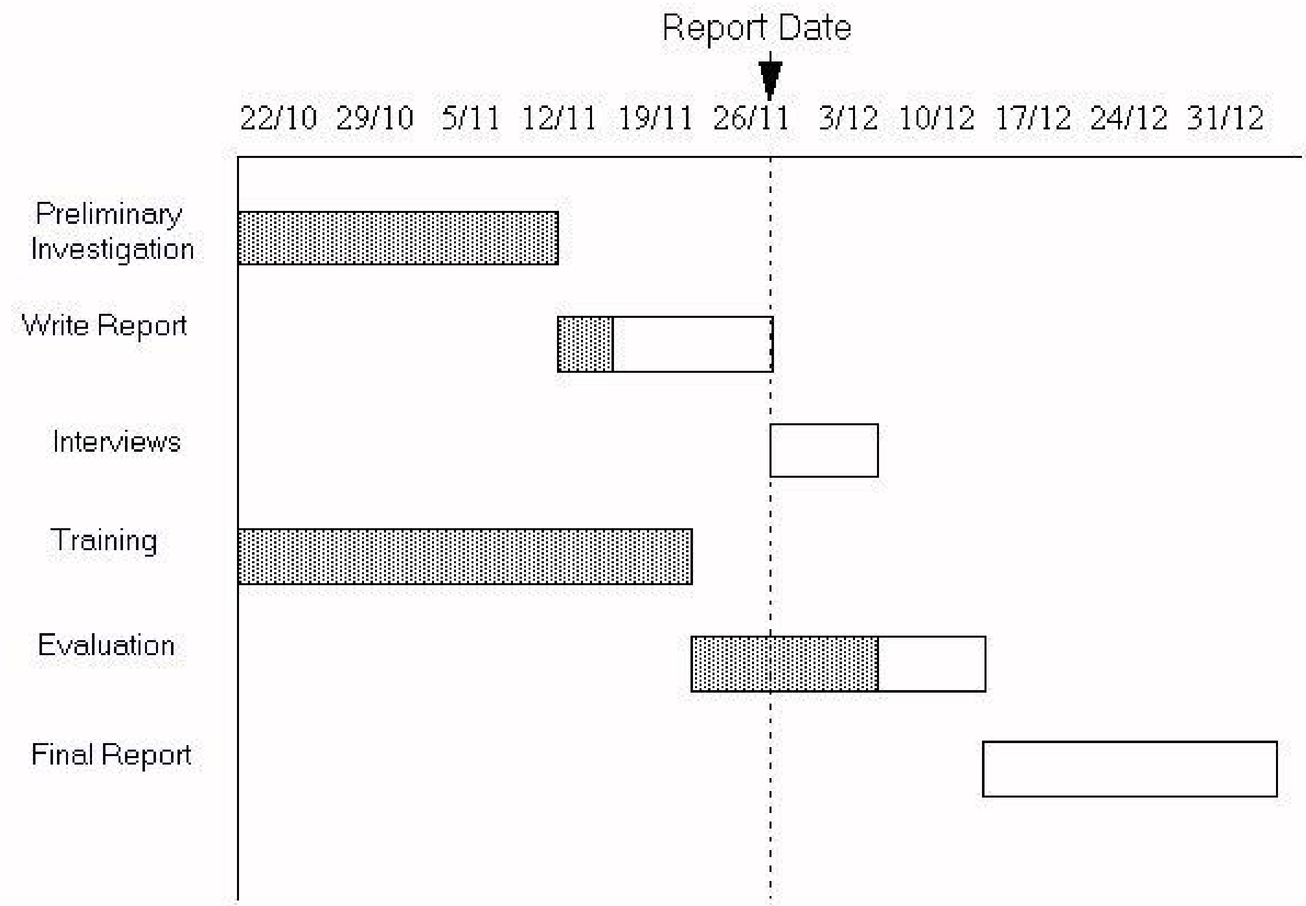


Figure 1: Gantt Chart

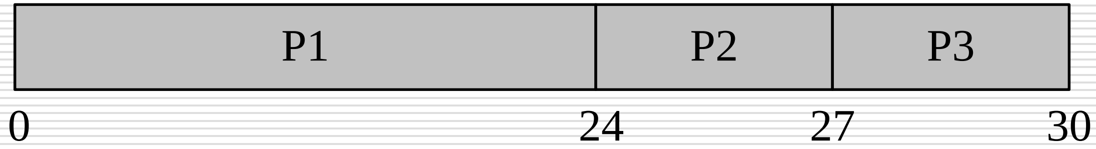
Average waiting time

- The time all the processes wait for CPU scheduling in ready queue.

First-Come, First-Served (FCFS) Scheduling

Process	Burst_Time
P1	24
P2	3
P3	3

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



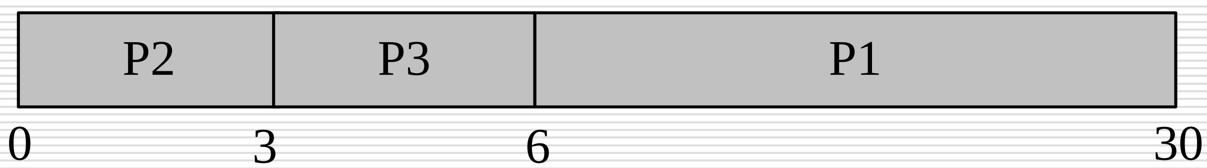
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

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* short process behind long process

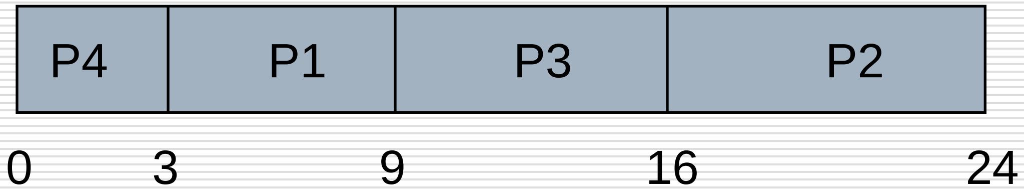
Characteristics

- Fair
- Short jobs wait for long time

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

process	burst-time
P1	6
P2	8
P3	7
P4	3



- SJF is optimal – gives minimum average waiting time for a given set of processes

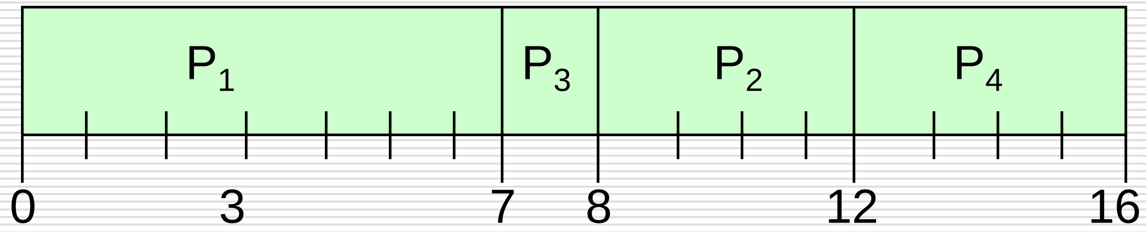
Shortest-Job-First (SJF) Scheduling

- 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. This scheme is known as the Shortest-Remaining-Time-First (SRTF)

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)

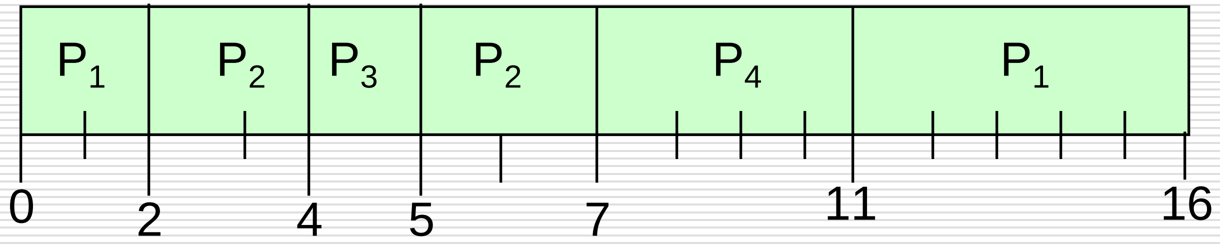


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

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

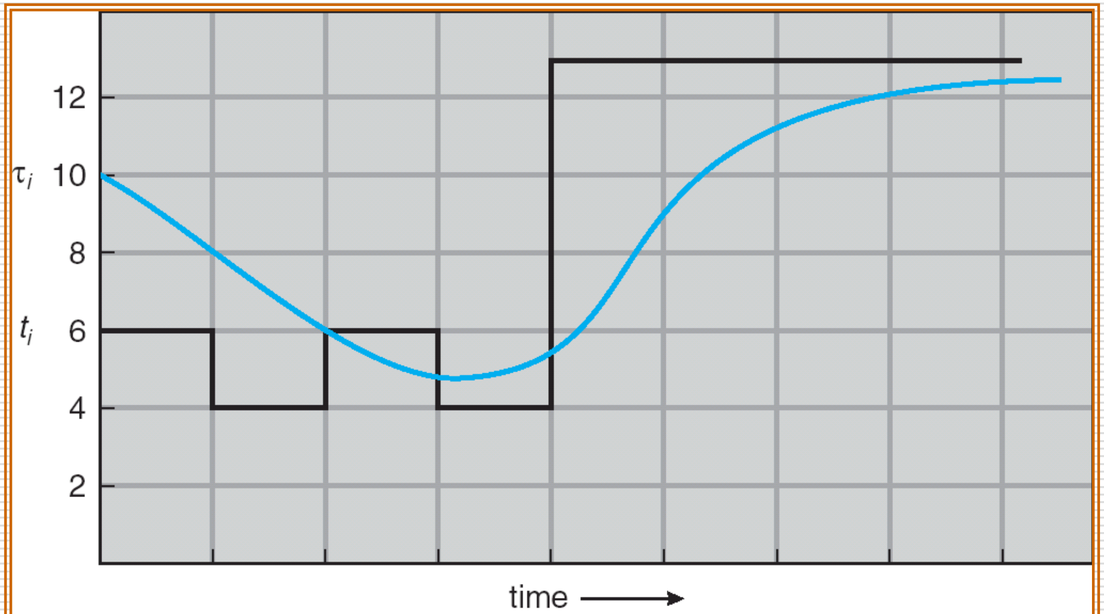
Determining Length of Next CPU Burst

- Can only estimate the length
- 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 :

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

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	. . .

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Shortest-Job-First (SJF) Scheduling

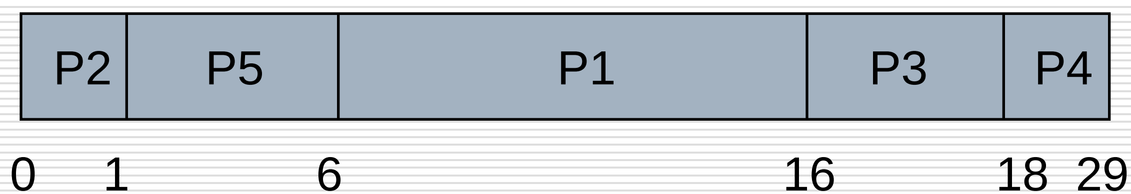
- Disadvantages
  - Not fair to long burst-time job
  - User can shorten the burst-time
  - Can not guarantee that the urgent job is processed as soon as possible
  - Starvation problem

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

Priority Scheduling

process	burst-time	priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



Priority Scheduling

- Problem
  - Infinite blocking (starvation)
  - Starvation – low priority processes may never execute
- Solution
  - Aging
    - Priority will be increased with time goes by

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.

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

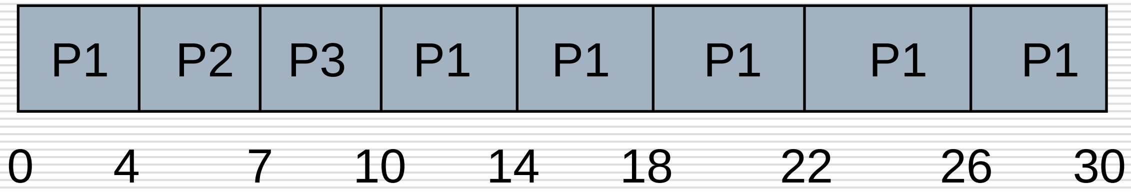
0 20 37 57 77 97 117 121 134 154 162

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

Round Robin

process	burst-time
P1	24
P2	3
P3	3

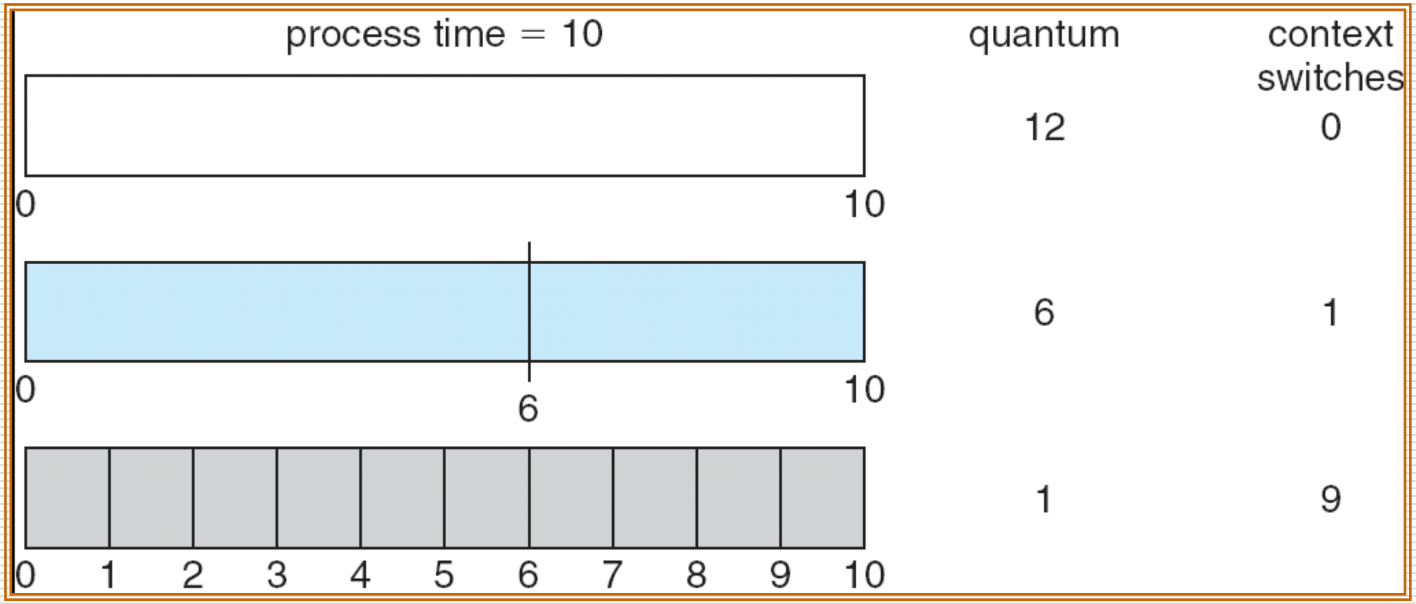
Quantum is 4ms



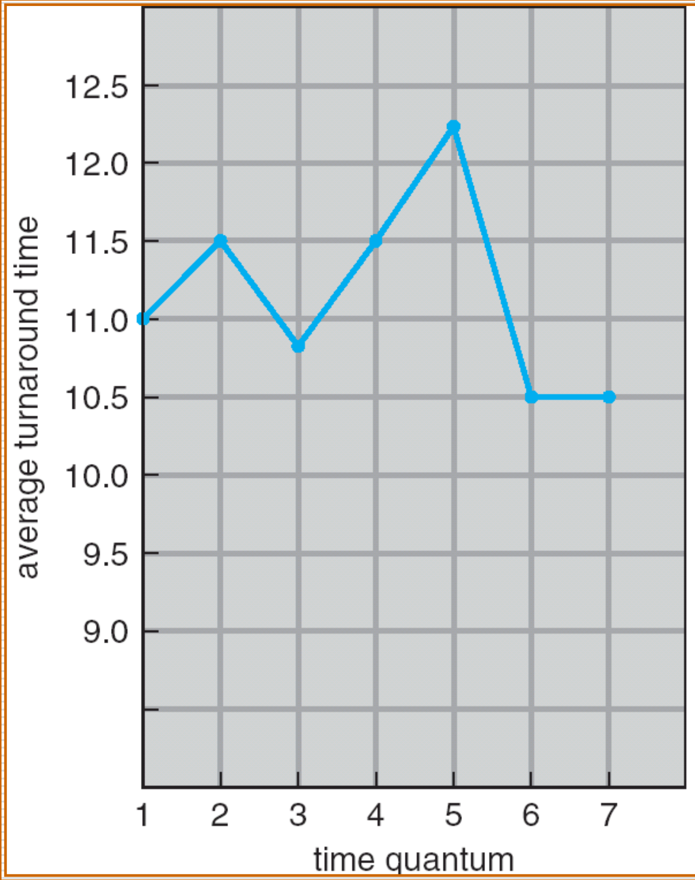
Round Robin

- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

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

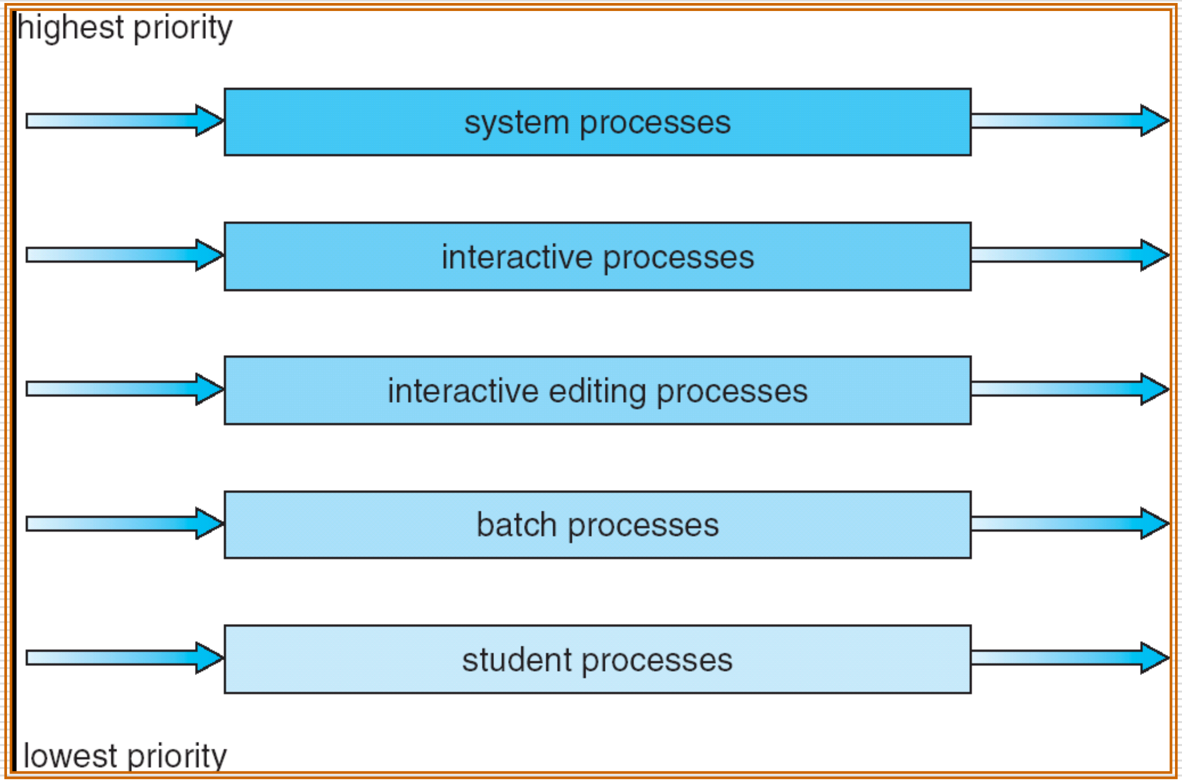
Round Robin

- Time quantum selection
  - Fixed
  - changeable
- Factors affect time quantum
  - System response time
  - Processes that are ready
  - Capability of CPU

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



Not flexible!

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

Multilevel Feedback Queue

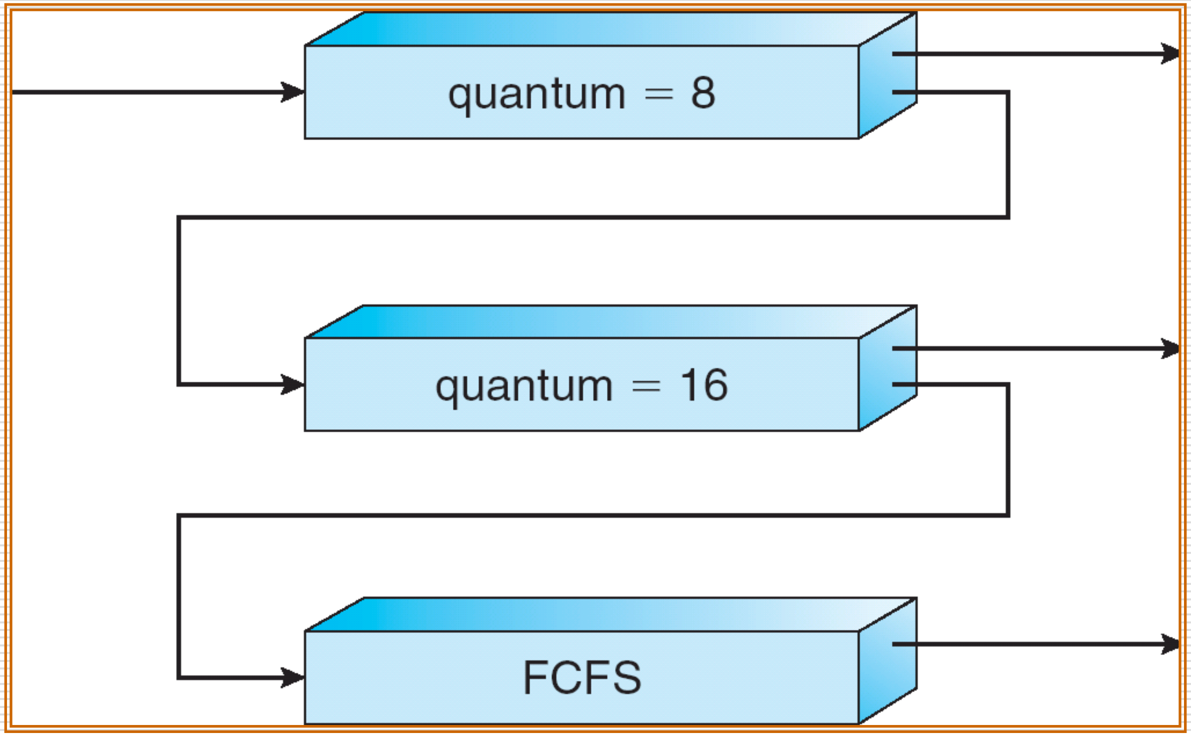
Process can move among different queues

- system sets many queues
  - Different time quantum is assigned to different queues, queue with higher priority has shorter time quantum.
  - Different queue can use FCFS scheduling
- New process is assigned to the first level queue
- When the first level queue is empty, then the processes in the second queue is scheduled, and so on.
- When the time quantum is used up, then the job is moved to the next queue.

Example of Multilevel Feedback Queue

- 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 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



Multilevel Feedback Queues

- High resource utility rate
- Fast response time
- Complicated implementation

Multiple-Processor Scheduling

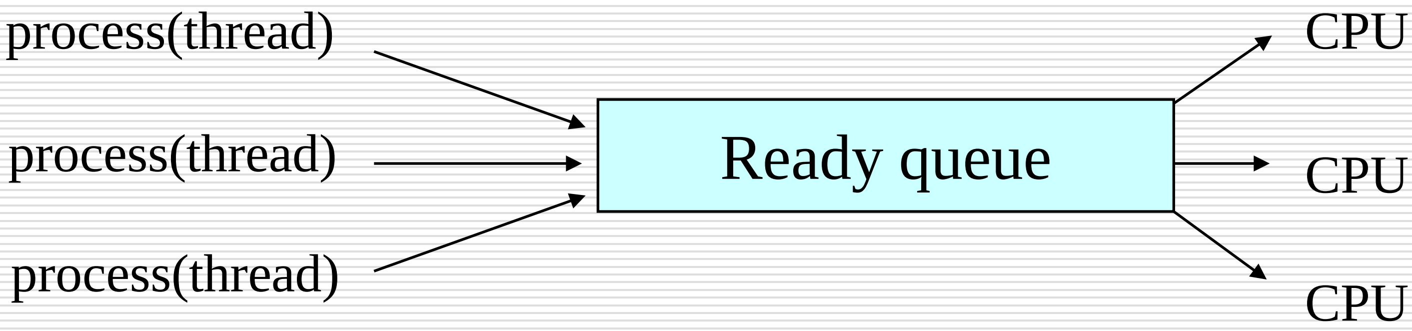
- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
- *Load sharing*
  - separate ready queue for each processor,
    - not really balanced;
  - common ready queue *Q* for all processors
    - each process accesses *Q* on its own,
    - master/slave assignment.
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing

self scheduling

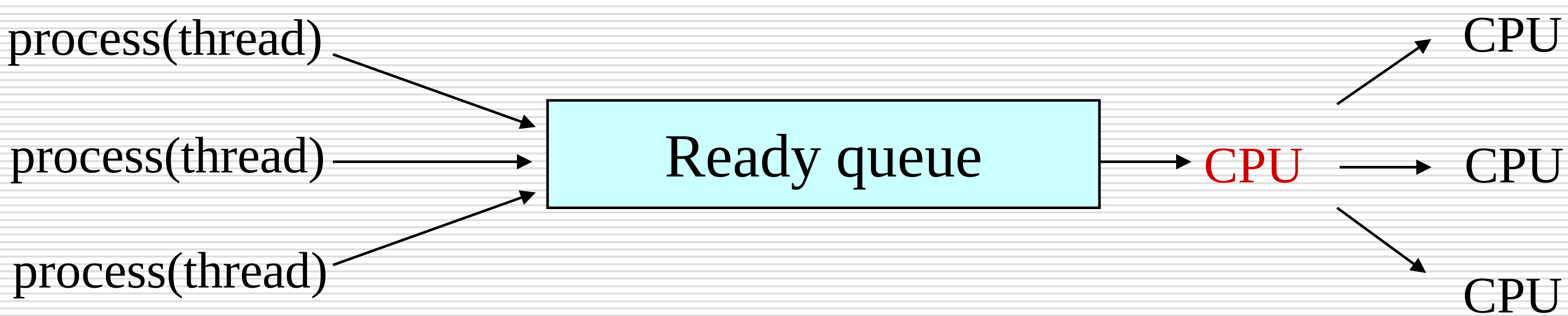
- balanced scheduling
- Only one ready queue
  - No process to assign task
  - Balanced

  - Bottleneck problem with more CPU
  - Thread can be assigned to different processors
  - Can not guarantee that all processes in same group can be scheduled simultaneously

self scheduling



Asymmetric scheduling



Processor affinity

- A process has an affinity for the processor on which it is currently running.
- Soft affinity
- Hard affinity

Load balancing

- It attempts to keep the workload evenly distributed across all processors in an SMP system.
- Push migration
- Pull migration

Symmetric Multithreading

- Hyperthreading Technology

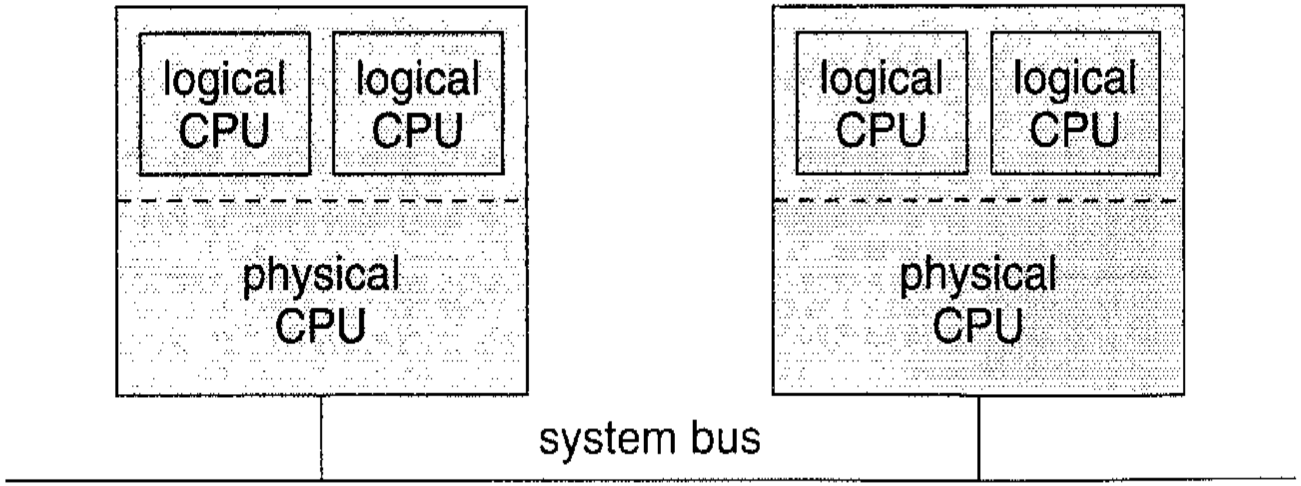


Figure 5.8 A typical SMT architecture

Real-Time Scheduling

- Satisfy every task’s time constraint
- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones

Thread Scheduling

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

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

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
        int i;
       pthread t tid[NUM THREADS];
       pthread attr t attr;
       /* get the default attributes */
       pthread attr init(&attr);
       /* set the scheduling algorithm to PROCESS or SYSTEM */
       pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
       /* set the scheduling policy - FIFO, RT, or OTHER */
       pthread attr setschedpolicy(&attr, SCHED OTHER);
       /* create the threads */
       for (i = 0; i < NUM THREADS; i++)
              pthread create(&tid[i],&attr,runner,NULL);
```

Pthread Scheduling API

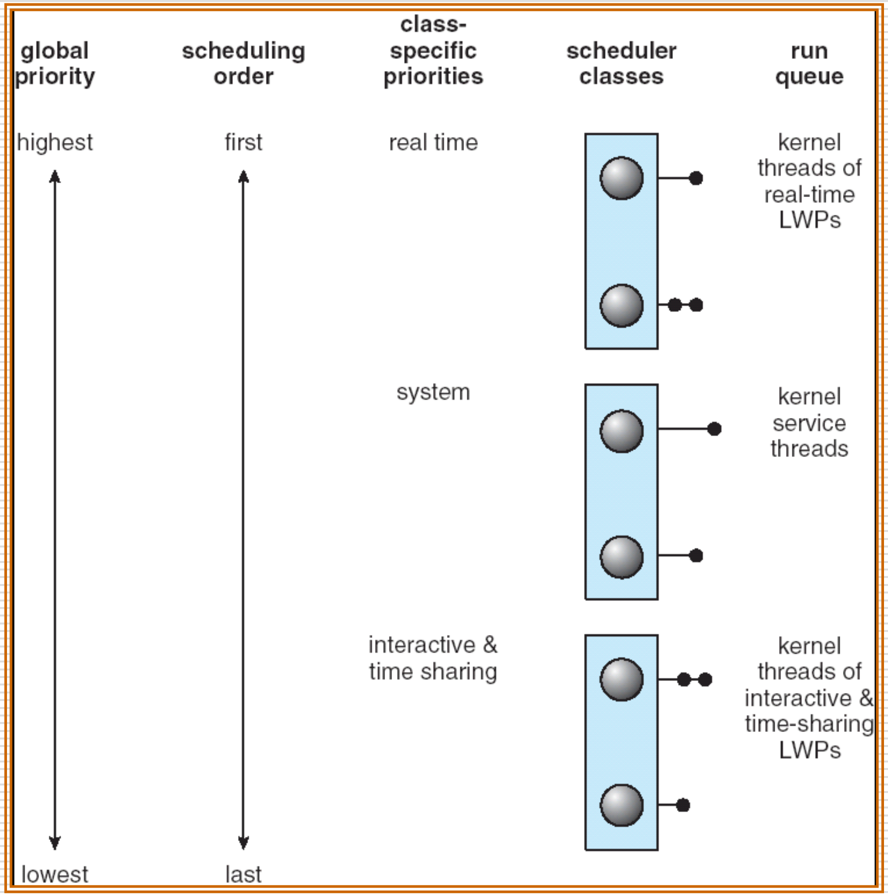
```
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
        pthread join(tid[i], NULL);
}
 /* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```

Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris 2 Scheduling

- Solaris uses priority-based thread scheduling
- It defines four classes of scheduling:
  - Real-time
  - System
  - Time sharing
  - Interactive



Solaris Dispatch Table

- Priority-a higher number indicates a higher priority
- Time quantum-the time quantum for the associated priority
- The quantum expired-the new priority of a thread that has used its entire time quantum without blocking.
- Return from sleep-the priority of a thread that is returning from sleeping.
- Solaris 9
  - Fixed priority
  - Fair share

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Windows XP Priorities

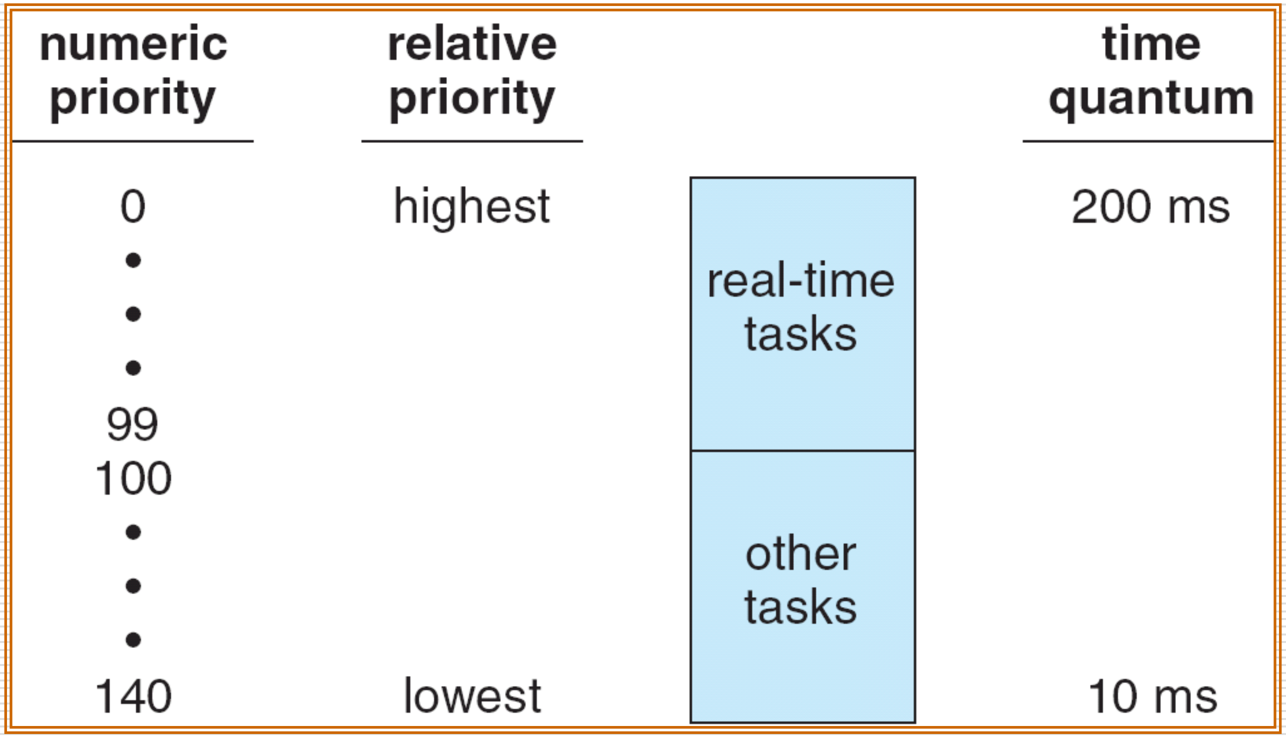
- Dispatcher
- XP uses a priority-based, preemptive scheduling algorithm
- Variable class 1-15
- Real-time class 16-31
- Once being selected to run, a thread will run until it is preempted by a higher-priority thread, until its terminates, until its time quantum ends, or until it calls a blocking system call.

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

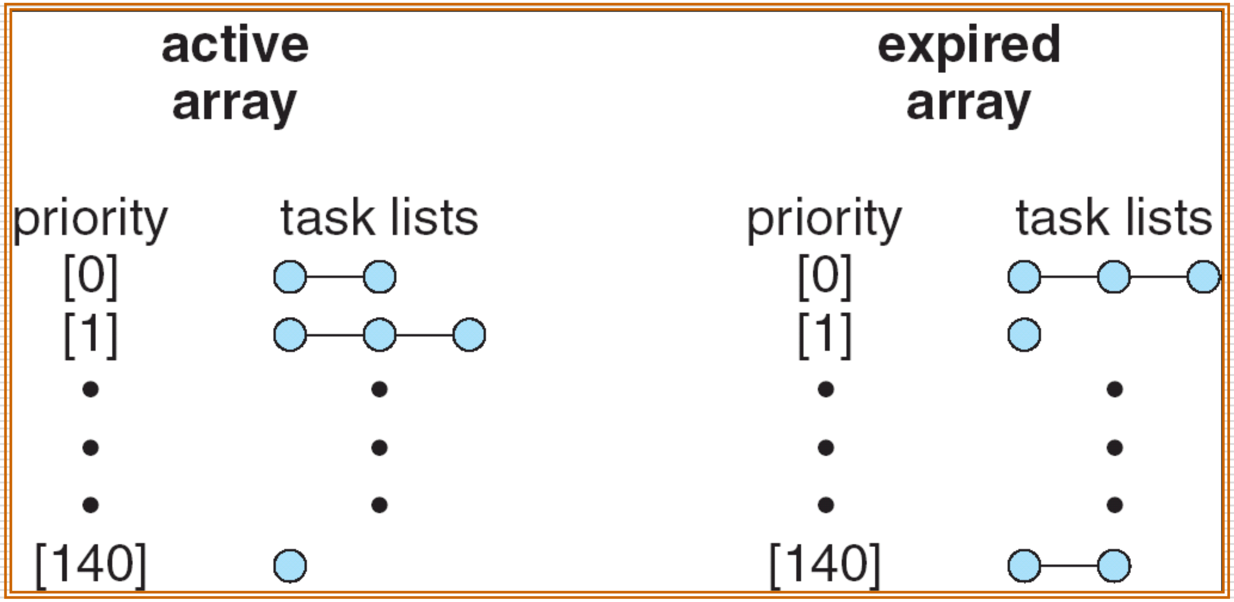
Linux Scheduling

- Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: 1—99; 100—140;
- Two algorithms: time-sharing and real-time
- 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
- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

The Relationship Between Priorities and Time-slice length



List of Tasks Indexed According to Prorities

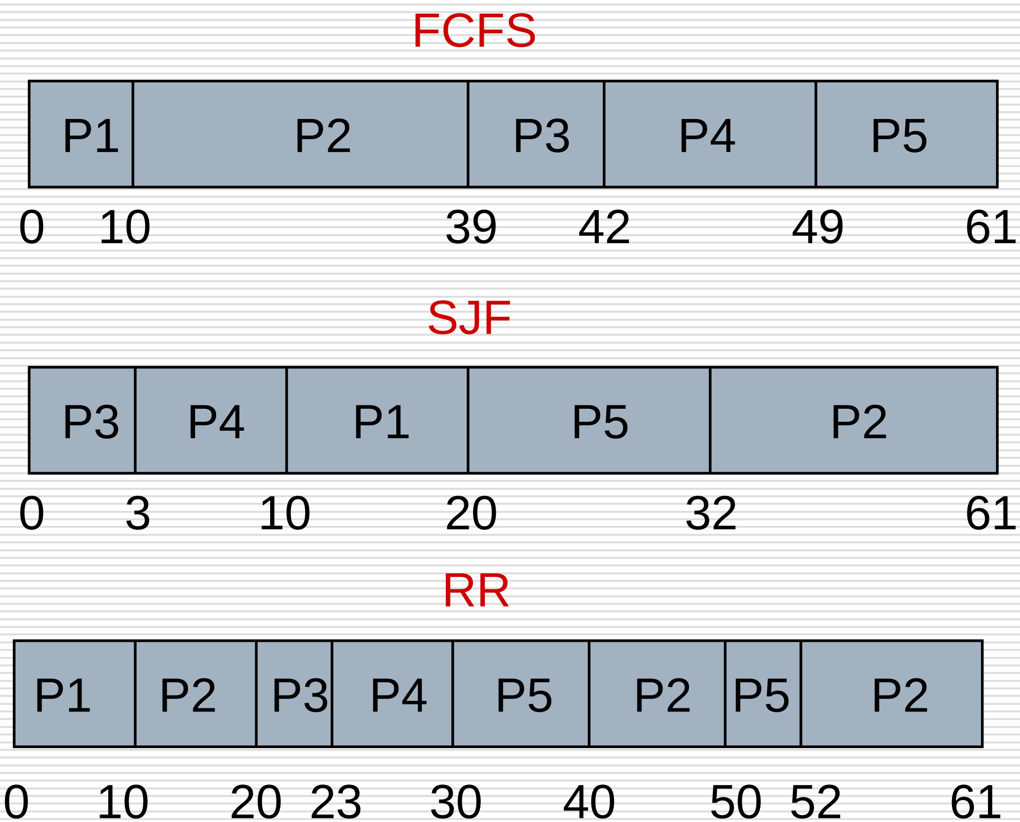


Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queuing models
- Simulations
- Implementation

Deterministic modeling

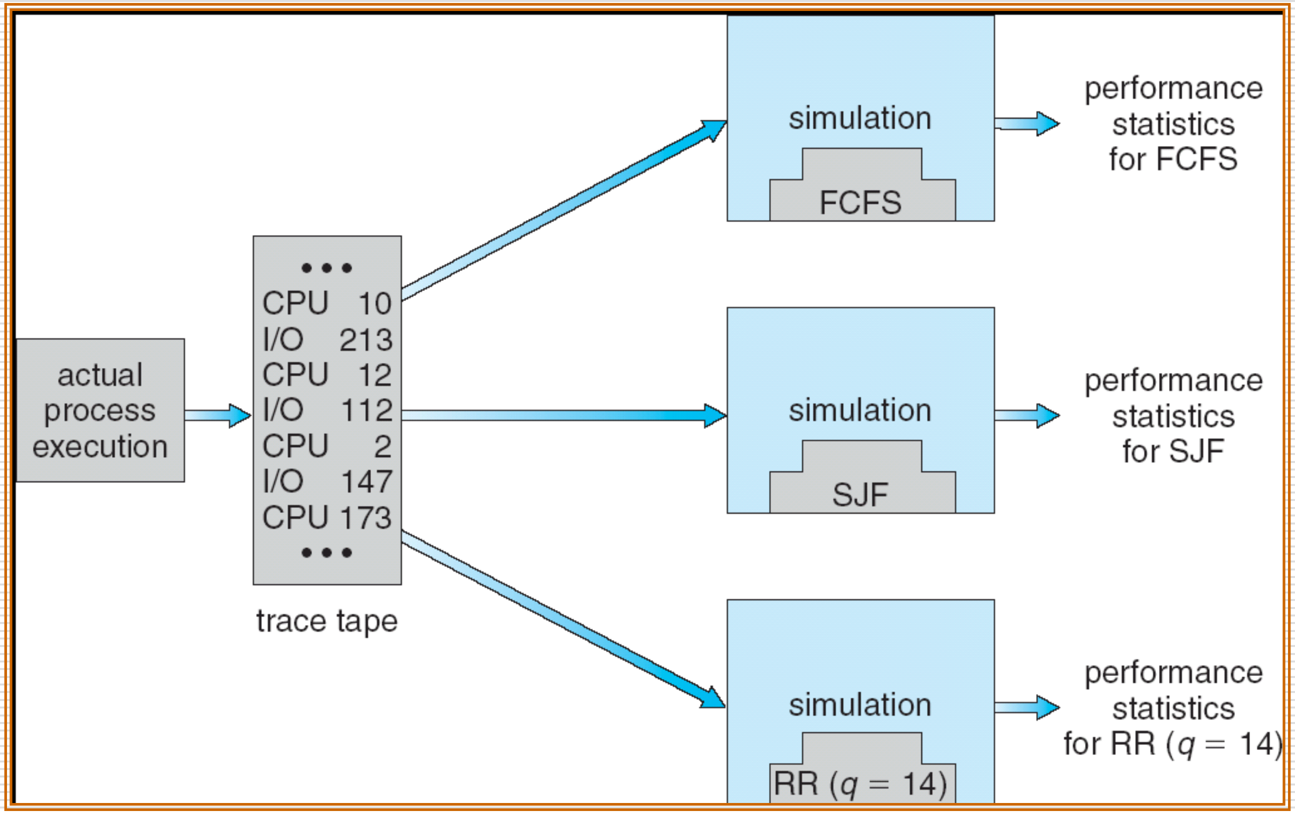
process	burst-time
P1	10
P2	29
P3	3
P4	7
P5	12



Queueing models

- Processes vary from day to day, so there is no static set of processes to fuse for deterministic modeling
- What can be determined is the distribution of CPU and I/O burst time, arrival rate and service
- n is average queue length
- W is average waiting time
- λ is average arrival rate
- $n = \lambda \times W$ (Little's formula)

Simulations



- The expense is incurred not only in coding the algorithm and modifying the operating system to support it but also in the reaction of the users to a constantly changing OS

Implementation

- The only completely accurate way to evaluate a scheduling algorithm is to code it up.

Assignment

- 5.4, 5.5, 5.9

End of Chapter 5

Any Question?