COP 4610 — Chapter 5
CPU Scheduling

Dr. Ming Zhao

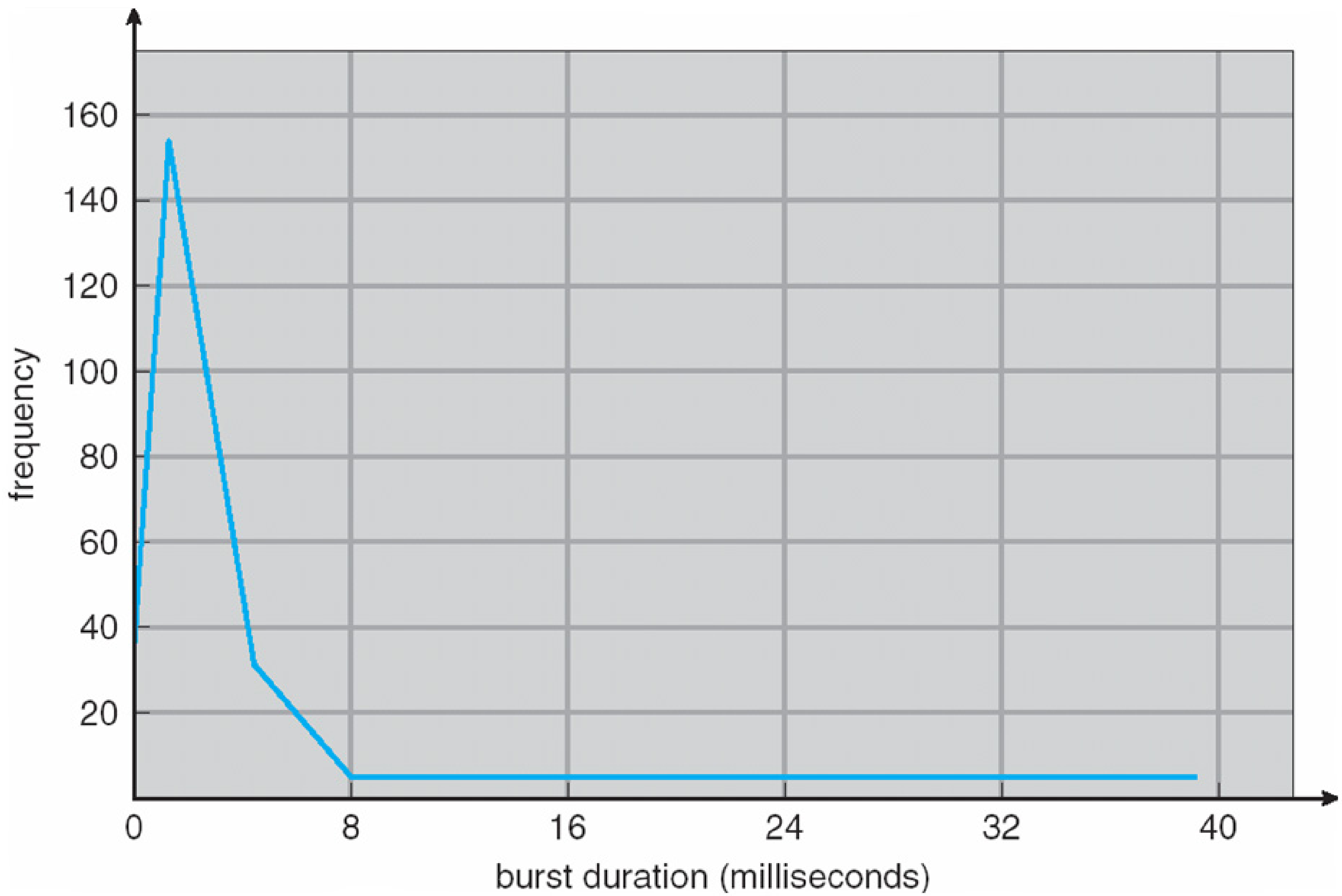
Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Basic Concepts

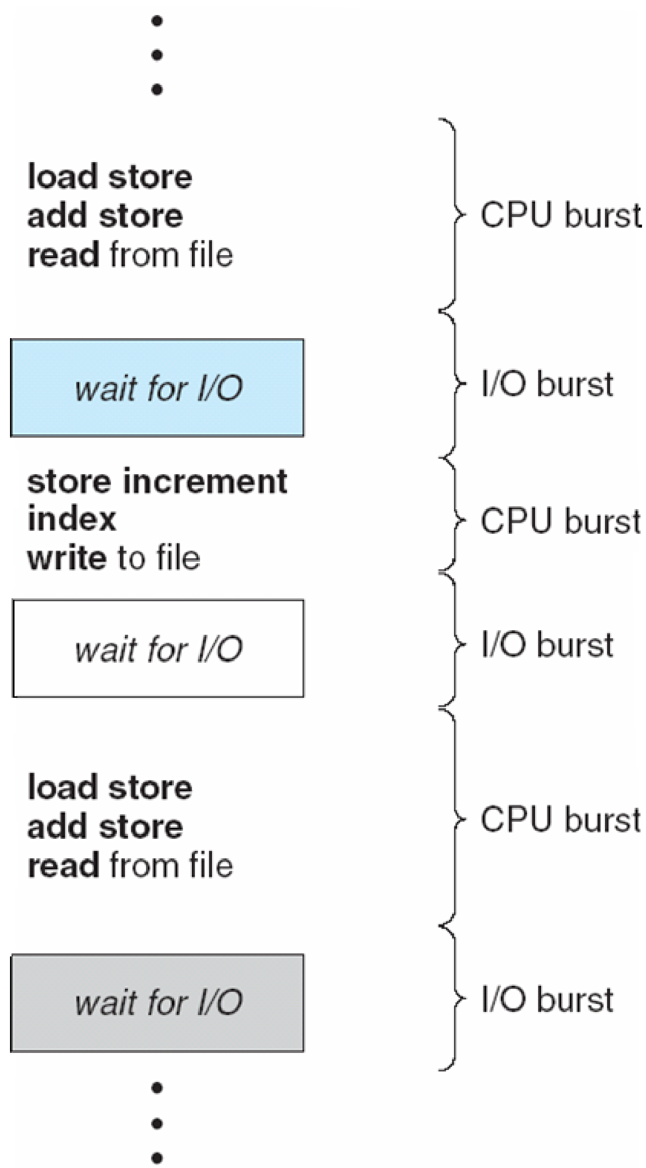
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle
  - Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution

Histogram of CPU-burst Times



- Exponential or hyperexponential distribution
  - A large number of short CPU bursts and a small number of long CPU bursts

Alternating Sequence of CPU and I/O Bursts



CPU Scheduler

- Short-term scheduler
  - Selects from among the processes in memory that are ready to execute and allocates the CPU to one of them
- 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
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

- The module that gives control of CPU to the process selected by the short-term scheduler
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- Dispatch latency
  - Time for the dispatcher to stop one process and start another running
  - Invoked during every process switch; Should be as fast as possible

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

Scheduling Algorithm Optimization Criteria

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

- In most cases, optimize the average measure
  - Sometimes, optimize the minimum or maximum values
  - Sometimes, optimize the variance

First-Come, First-Served Scheduling

- The process that requests the CPU first is allocated the CPU first
  - Can be easily managed with a FIFO queue
  - Simple to implement
- But the average waiting time is often quite long

FCFS Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

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

FCFS Scheduling

- Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- 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 processes wait for one long process to get off CPU
  - Low CPU and device utilization

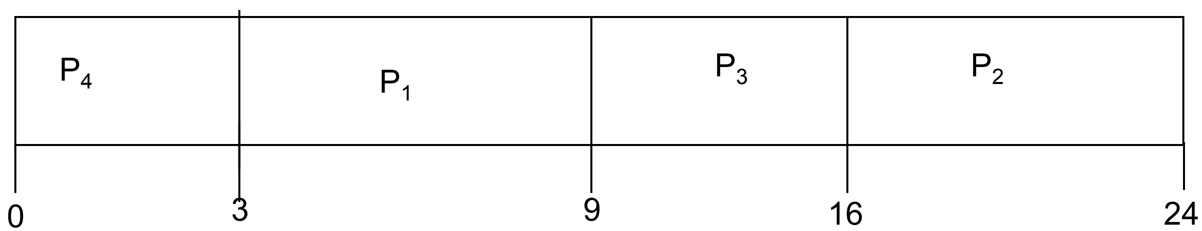
Shortest-Job-First (SJF) Scheduling

- Schedule the process with the shortest next CPU burst
- SJF is optimal — gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

Example of SJF

Proces	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

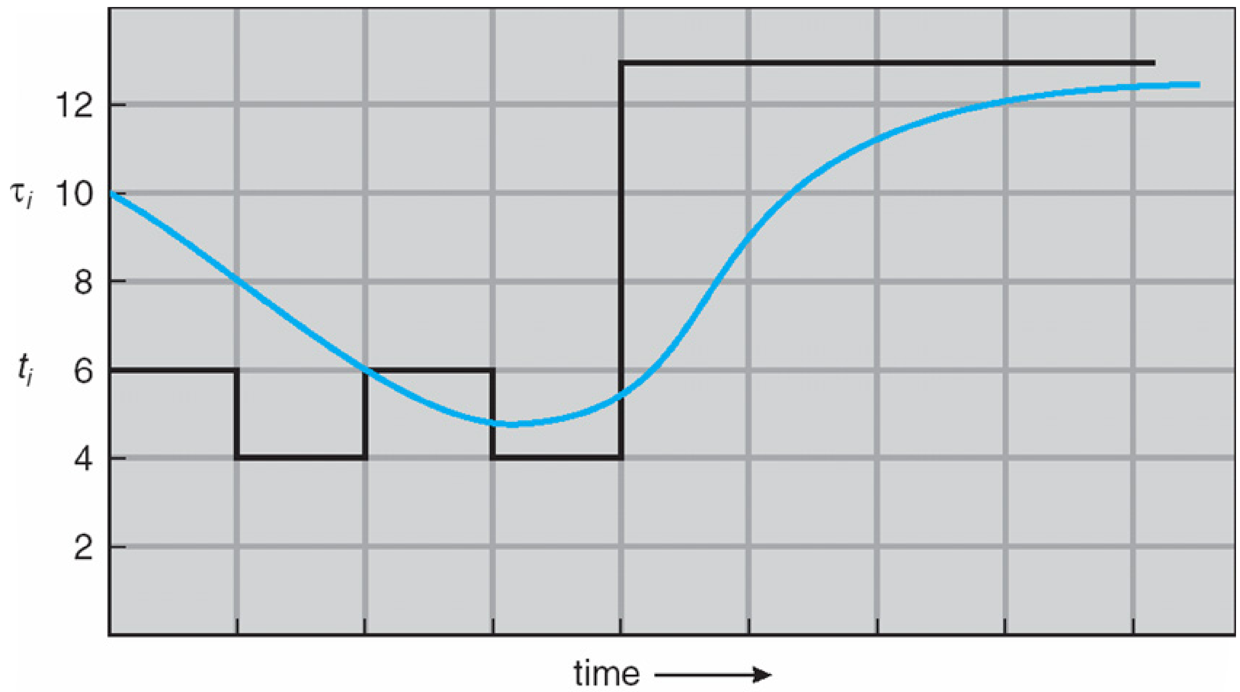
Determining Length of Next CPU Burst

- 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. $\alpha, 0 \leq \alpha \leq 1$
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

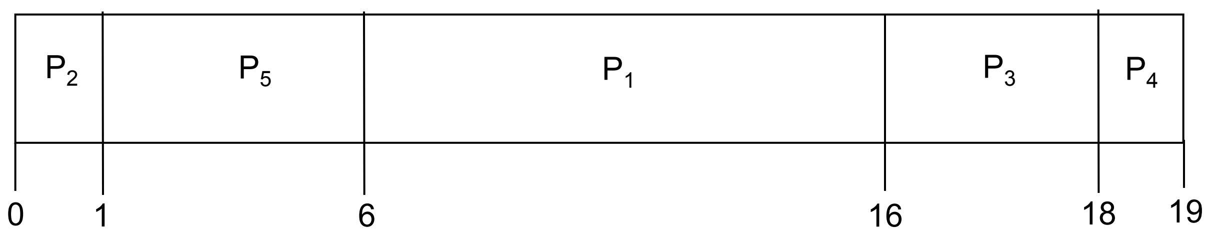
Priority Scheduling

- Allocate CPU to the process with the highest priority
- 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

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

- Priority scheduling chart



- Average waiting time = 8.2

Round-Robin Scheduling

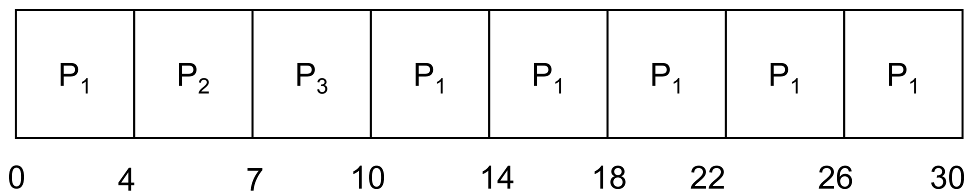
- Specifically designed for time-sharing
  - Similar to FCFS but with preemption
  - Each process gets a small unit of CPU time (*time quantum*)
    - Usually 10-100 milliseconds
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.

- Performance depends heavily on the size of time quantum (q)
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ overhead from context switch can be very high
    - q must be large with respect to context switch time

Example of RR with Time Quantum = 4

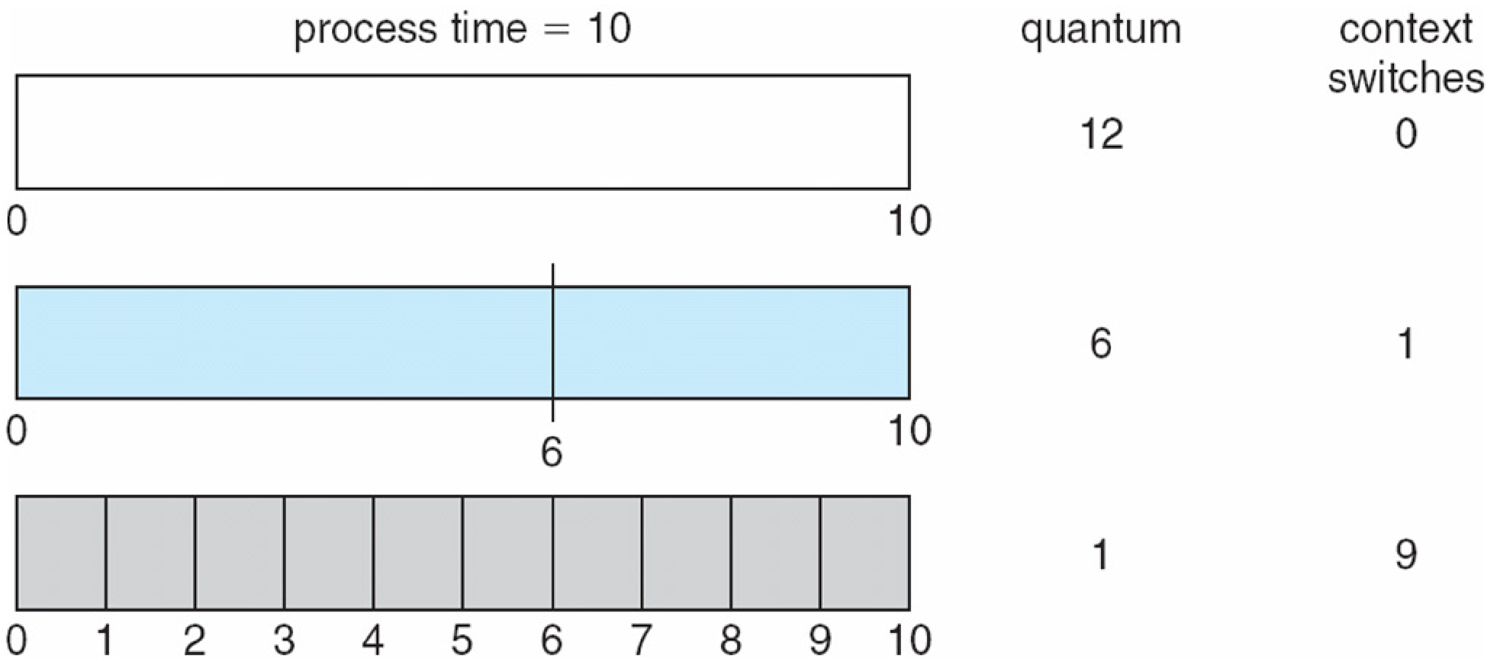
Process	Burst Time
P_1	24
P_2	3
P_3	3

- RR scheduling chart



- Average waiting time = 5.66

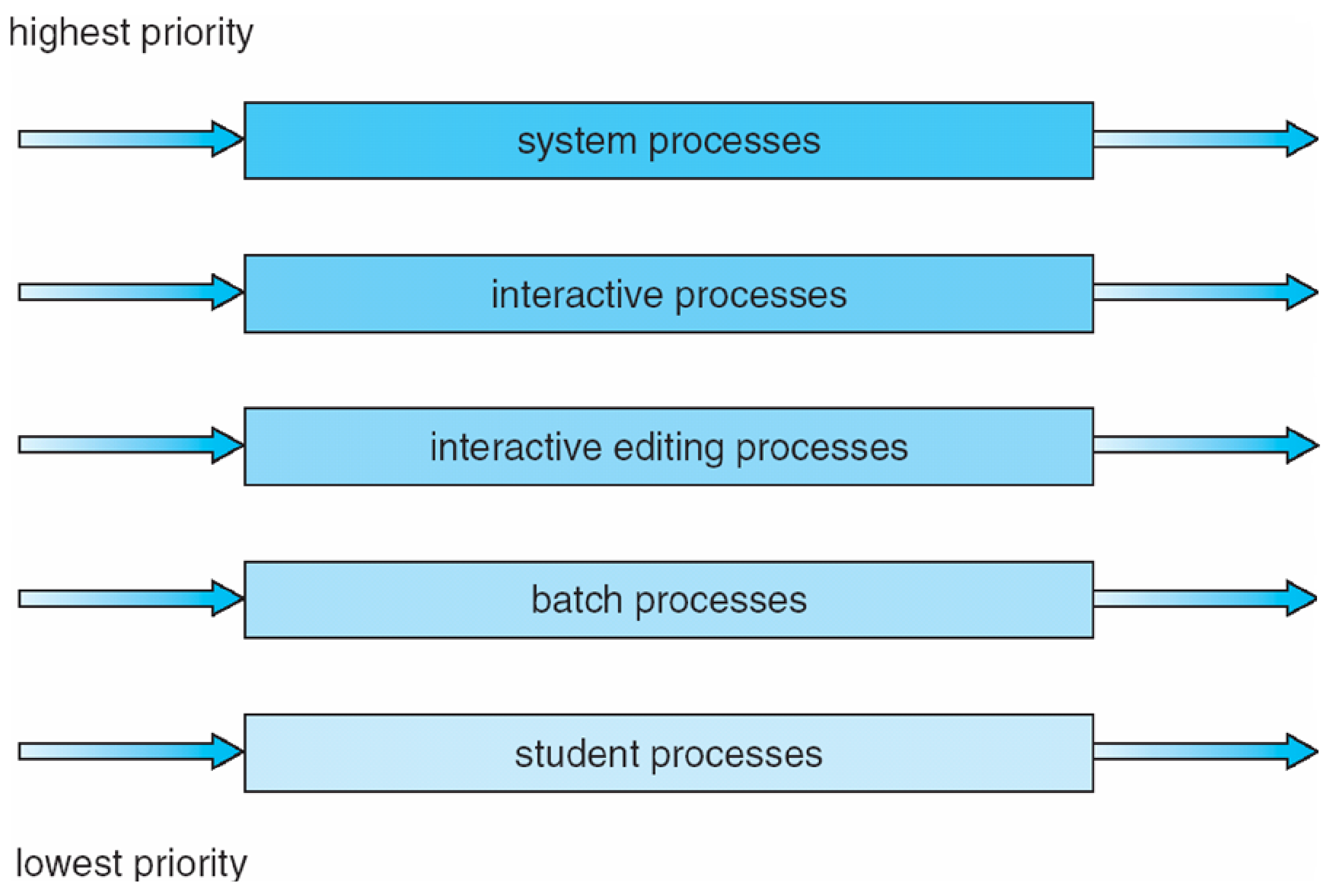
Time Quantum and Context Switch Time



Multilevel Queue Scheduling

- Designed for processes with different scheduling needs
  - Foreground (interactive) processes
  - Background (batch) processes
- Partition the ready queue into separate queues
  - Each queue has its own scheduling algorithm
    - Foreground – RR
    - Background – FCFS
- Example of scheduling of five queues

Multilevel Queue Scheduling



Multilevel Queue Scheduling

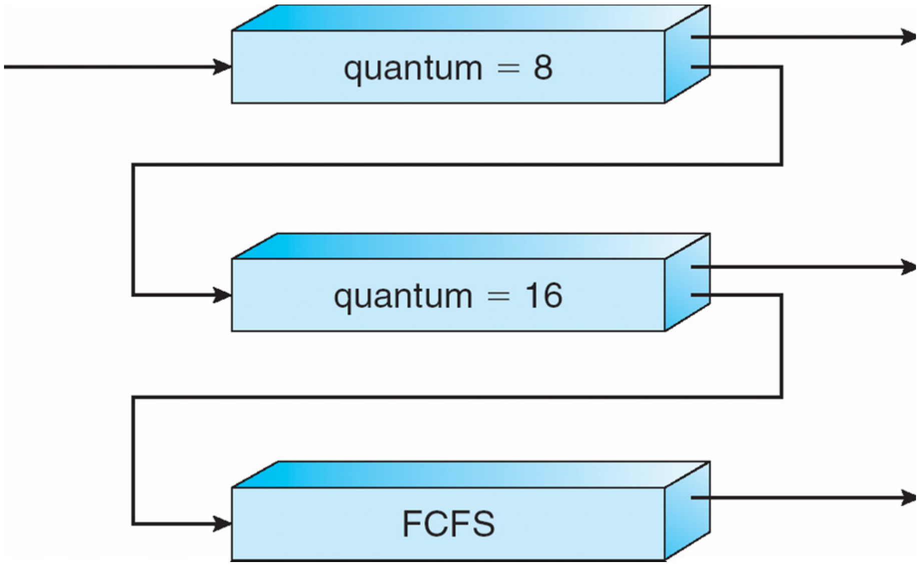
- Scheduling must also be done among the queues
  - Fixed priority scheduling
    - Foreground queue has absolute priority over background queue
  - Time slice among the queues
    - Each queue gets a certain portion of the CPU time which it can schedule amongst its processes
    - E.g., 80% to foreground in RR, 20% to background in FCFS

Multilevel Feedback Queue Scheduling

- A process can move between the various queues
  - Separate processes according to the characteristics of their CPU bursts
    - If a process uses too much CPU, move it to a lower-priority queue
  - Can implement aging to prevent starvation
    - If a process has waited too long, move it to a higher-priority queue

- Example:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

Multilevel Feedback Queue Scheduling



- This algorithm gives highest priority to any process with a CPU burst of 8ms or less
- Processes with a CPU burst between 8ms and 24ms are also served quickly but with lower priority
- Long processes automatically sink to the bottom queue and served with left over CPU cycles

Thread Scheduling

- Distinction between user-level and kernel-level threads
  - User-level threads are scheduled by thread library
  - Kernel-level threads are scheduled by OS
- In many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
- OS schedules kernel thread onto physical CPU
  - Known as **system-contention scope (SCS)** since scheduling competition is among all threads in system
  - In one-to-one model, thread scheduling uses only SCS

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - Specify contention scope
    - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
    - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
  - Set/get the contention scope
    - pthread_attr_setscope(pthread_attr_t *attr, int scope)
    - pthread_attr_getscope(pthread_attr_t *attr, int *scope)

Pthread Scheduling

- API also allows specifying the scheduling policy
  - Scheduling policy
    - SCHED_FIFO
    - SCHED_RR
    - SCHED_OTHER
  - Set/get the scheduling policy
    - pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)
    - pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)

Pthread Scheduling Example

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

Multiple-Processor Scheduling

- Scheduling is more complex with multiple CPUs
  - Here we focus on homogeneous multi-processors
- Asymmetric multiprocessing
  - One processor handles all scheduling, I/O, and other system activities; Other processors execute only user code
  - Reduce the need for data sharing because only one processor accesses the system data structures
- Symmetric multiprocessing (SMP)
  - Each processor is self-scheduling
  - All processes may be in a common ready queue or each processor may have its own private ready queue

Load Balancing

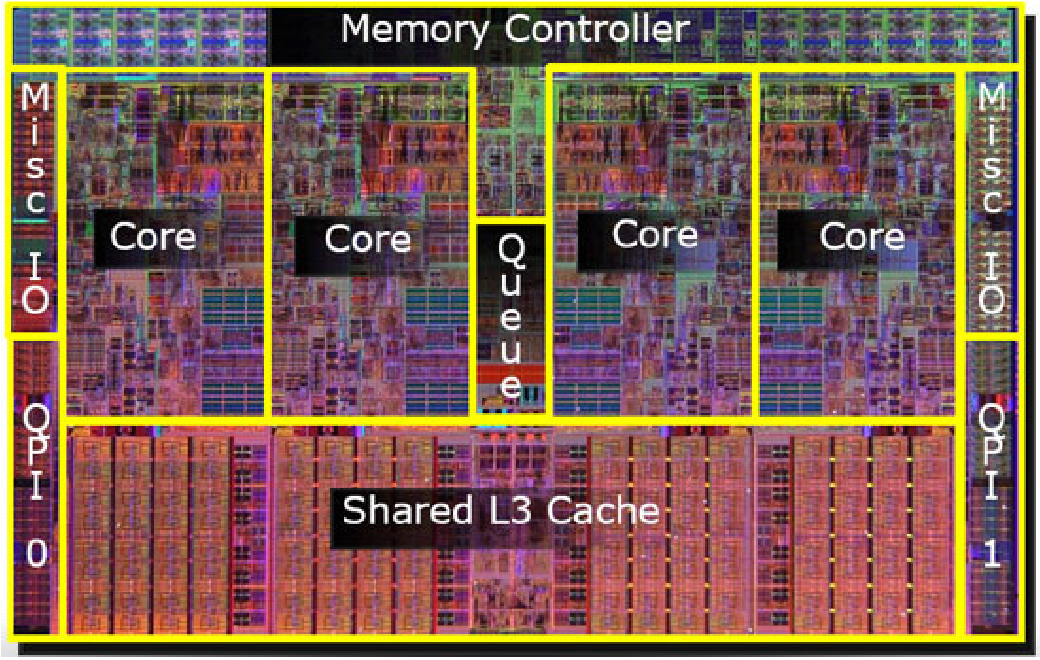
- Keep the workload evenly distributed across all processors
  - Fully utilize the multi-processor resources
  - Only necessary on systems with per-processor queue
- Push migration
  - Load on each processor is periodically checked and processes are moved from overloaded processors to idle or less-busy ones
- Pull migration
  - An idle processor pull a waiting task from a busy processor
- E.g., Linux runs its load-balancing algorithm every 200ms or whenever the run queue of a processor is empty

Multi-core Processors

- Recent trend to place multiple processor cores on same physical chip
  - Faster and consume less power compared to traditional single-core multiprocessors
- Multiple threads per core also growing
  - Take advantage of memory stall in one thread to make progress on another thread
  - From an OS perspective, each hardware thread appears as a logical processor
- Multithreaded multi-core processors
  - E.g., Intel Core i7-860 has 4 cores per chip, 2 threads per core

Intel Core i7

Processor Number	i7-860
# of Cores	4
# of Threads	8
Clock Speed	2.8 GHz
Max Turbo Frequency	3.46 GHz
Intel® Smart Cache	8 MB
Bus/Core Ratio	21
DMI	2.5 GT/s
Instruction Set	64-bit
Instruction Set Extensions	SSE4.2
Embedded Options Available	Yes
Supplemental SKU	No
Lithography	45 nm



Multithreaded Multi-core CPU Scheduling

- Two level scheduling
  - At the first level, OS chooses which software thread to run on each hardware thread (logical processor)
  - At the second level, each core decides which hardware thread to run

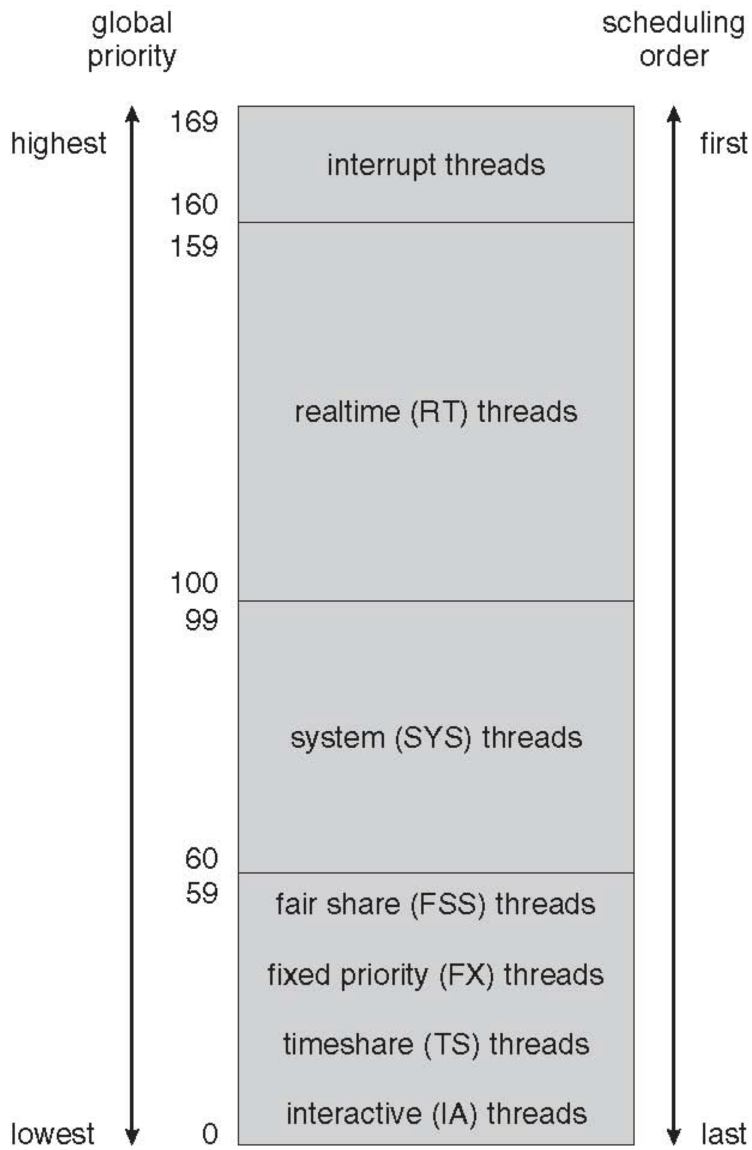
Solaris Scheduling

- Priority-based thread scheduling where each thread belongs to one of six classes
  - Time sharing (default), interactive, real time, system, fair share, fixed priority
  - Time-sharing class uses multilevel feedback queue scheduling
    - An inverse relationship between priorities and time slices
  - Interactive class uses the same scheduling policy but gives windowing applications higher priority
  - Real-time class has the highest priority

Solaris Dispatch Table

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

Solaris Scheduling



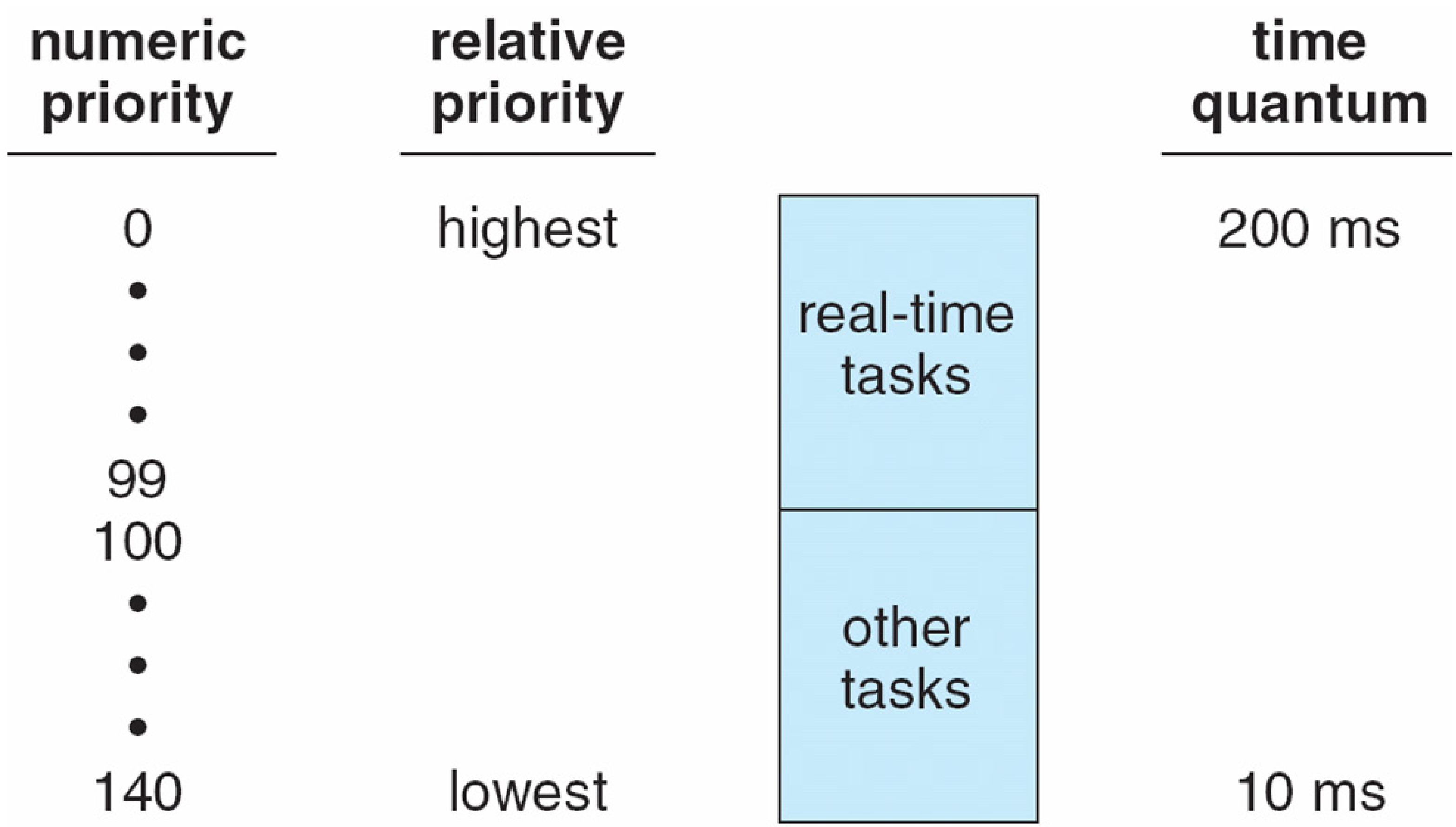
Windows XP Scheduling

- Priority-based, preemptive thread scheduling
  - 32-level priority with two classes
    - Variable class: priority 1-15
    - Real-time class: 16-31
  - When a thread in variable class runs out its time quantum
    - It is interrupted and its priority is lowered
  - When a thread in variable class is released from a wait
    - Its priority is boosted
  - In addition, a foreground process also receives priority boost

Linux Scheduling

- Priority-based, preemptive scheduling
  - Two priority ranges
    - Real-time : 0-99
    - Time-sharing: 100-140 (nice value)
  - Real-time tasks are assigned static priorities
  - Time-sharing tasks are assigned dynamic priorities
    - Nice values plus or minus 5, determined by their interactivity
    - Interactivity is determined by how long it has been waiting for I/O
    - Scheduler favors interactive tasks which typically have longer sleeping time

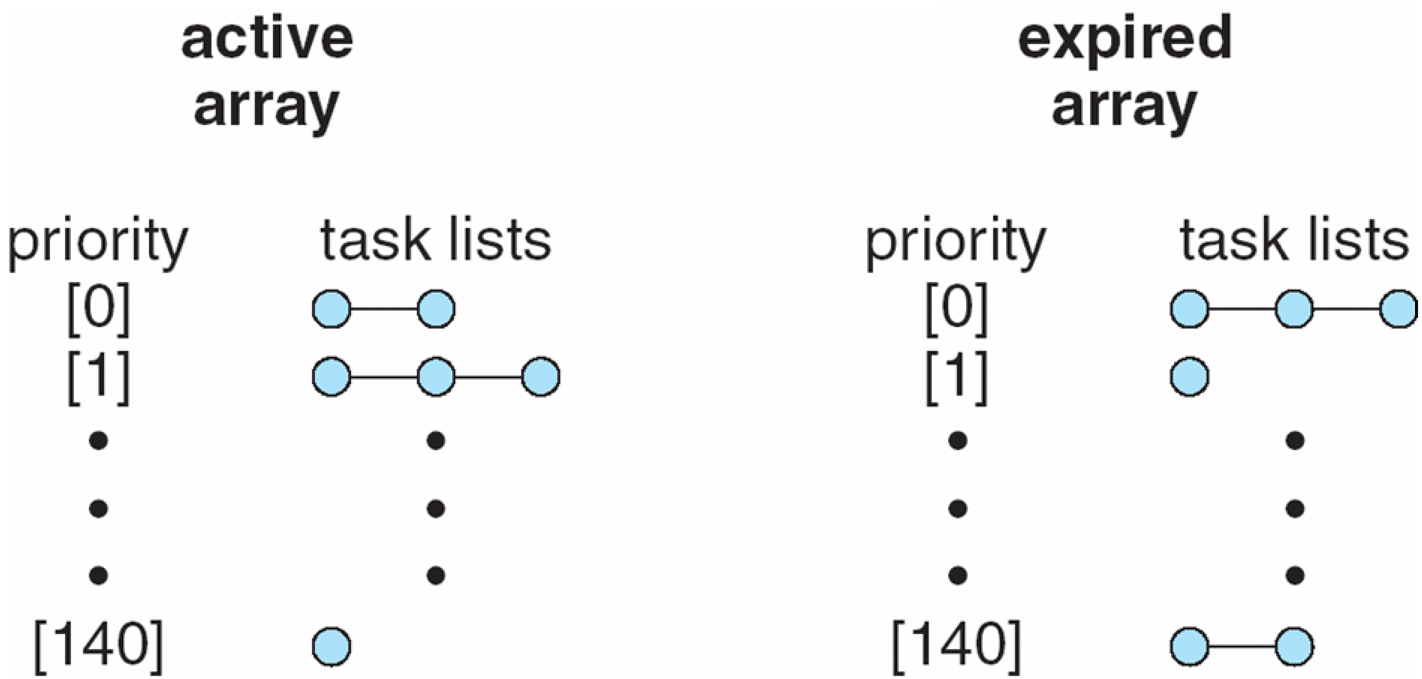
Priorities and Time-slice Length



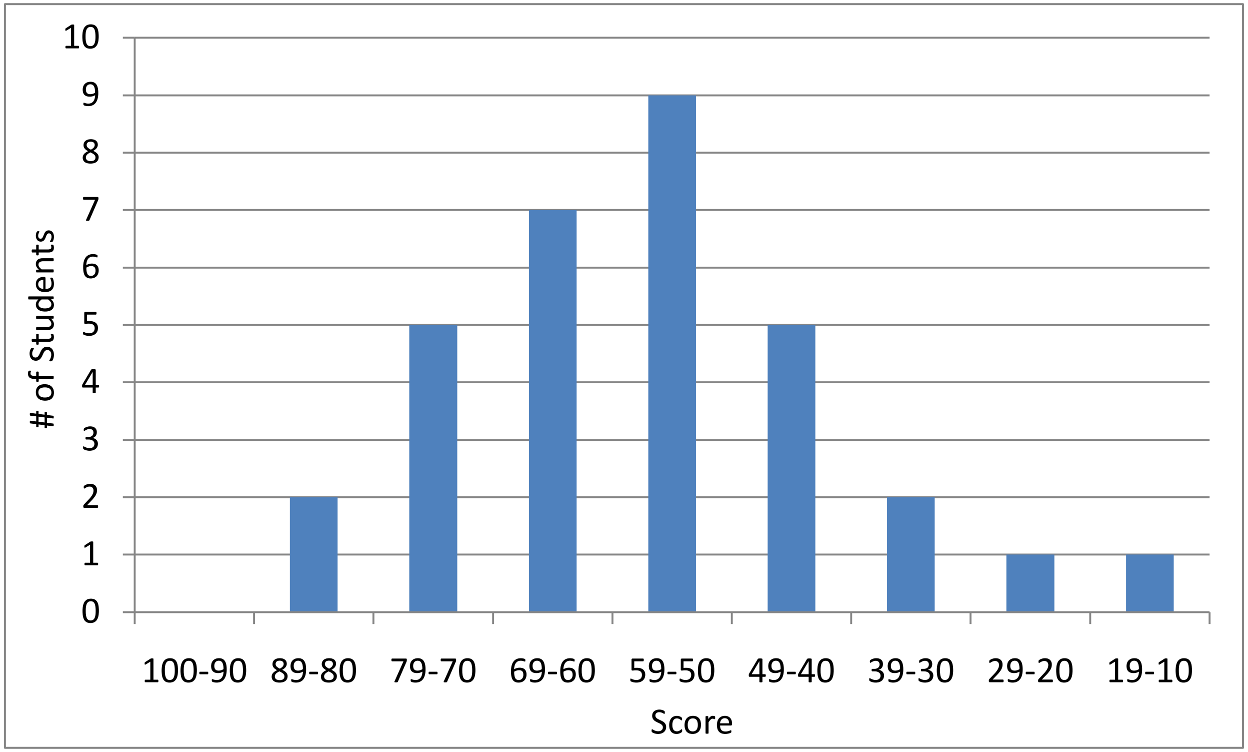
Linux Scheduling

- Kernel maintains a list of all runnable tasks in runqueue
  - Each processor maintains its own runqueue and schedules itself independently
  - Each runqueue contains two priority arrays
    - Active: all tasks with time remaining in their time slices
    - Expired: all tasks with their time slices expired
  - Each priority array contains a list of tasks indexed by priority
    - Scheduler executes the task with the highest priority in active array
    - When all tasks have exhausted their time slices, the two arrays are exchanged

List of Tasks Indexed According to Priorities



Quiz 1 Results



Avg	56
Min	16
Max	83
Median	54