VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SWE3001-Operating Systems

Prepared By

Dr. L. Mary Shamala

Assistant Professor

SCOPE/VIT

Module 4: CPU Scheduling

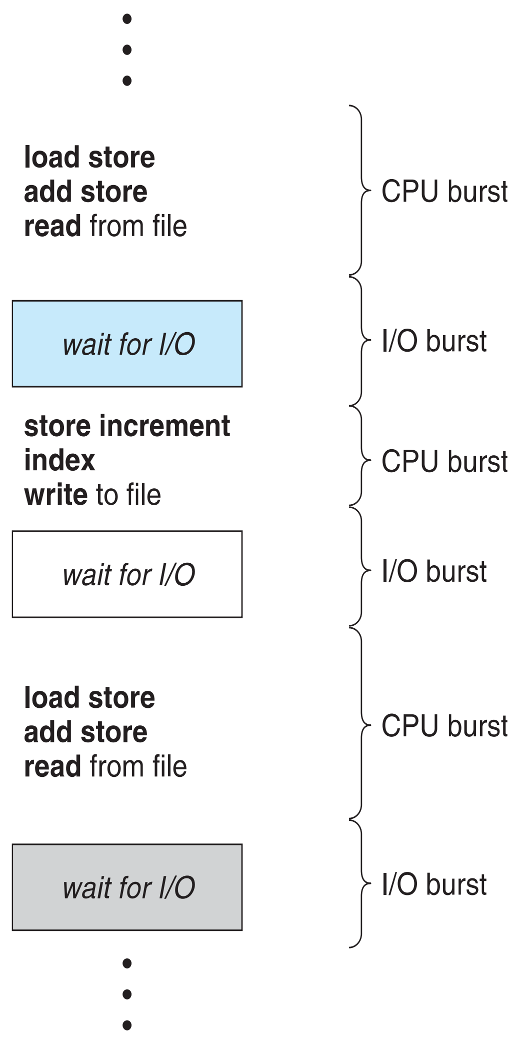
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling

Objectives

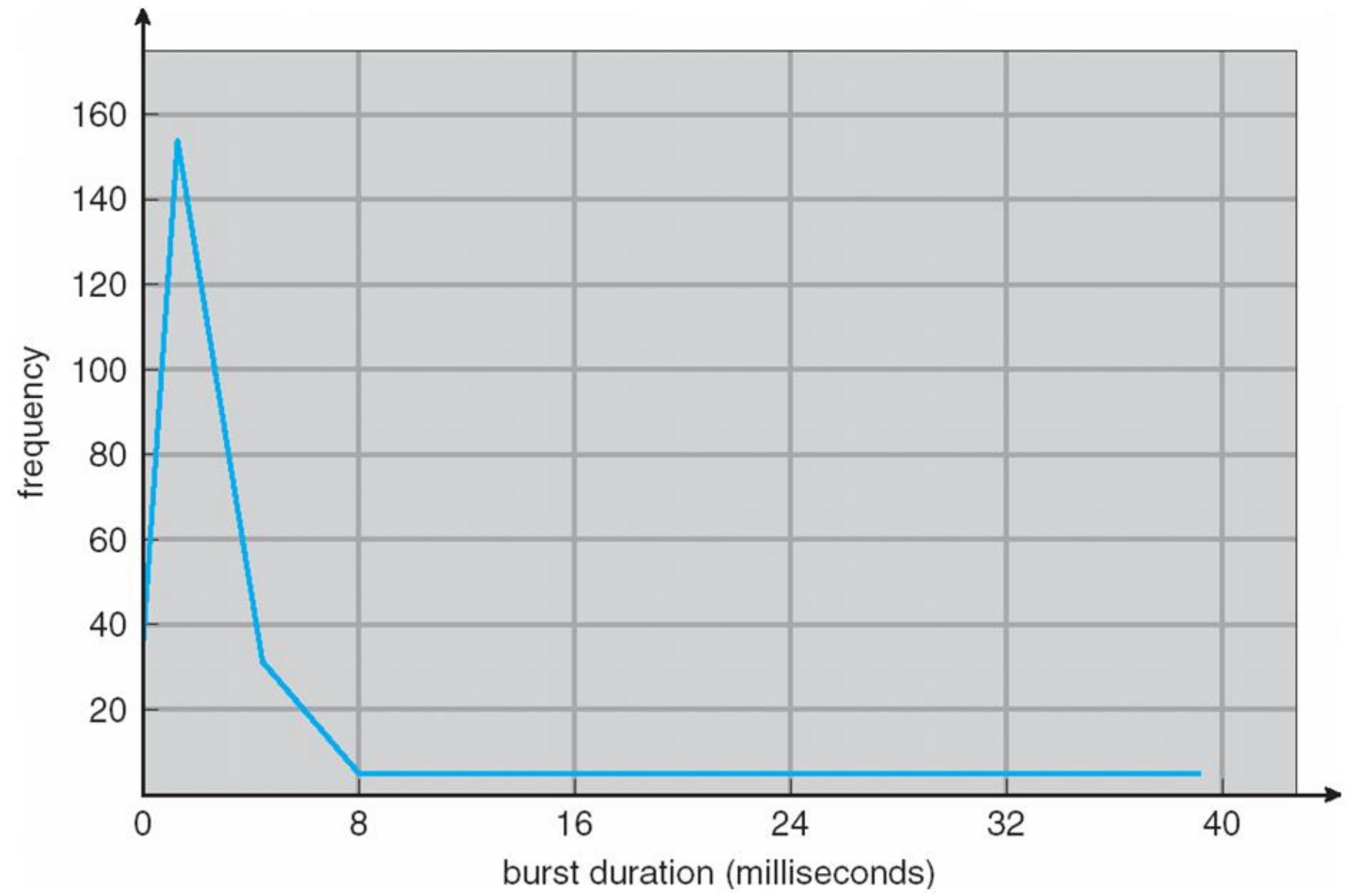
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts: CPU– I/O Burst Cycle

- Maximum CPU utilization obtained with multiprogramming
- Almost all computer resources are scheduled before use.
- The CPU is one of the primary computer resources and its scheduling is central to OS design.
- 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**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed
- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
- The ready queue is not necessarily a first-in, first-out (FIFO) queue.
- A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- The records in the queues are generally process control blocks (PCBs) of the processes

Preemptive Vs Non-preemptive Scheduling

- 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
- Scheduling under 1 and 4 is **nonpreemptive**→once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state
- All other scheduling is **preemptive**
  - Special hardware is needed
  - Results in race conditions when data are shared among several processes.
  - Affects the design of the operating system kernel
  - Consider interrupts occurring during crucial OS activities

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
- The dispatcher should be as fast as possible
- **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)
- It is desirable to:
  - maximize CPU utilization and throughput
  - minimize turnaround time, waiting time, and response time.

Scheduling Algorithm Optimization Criteria

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

Scheduling Objectives

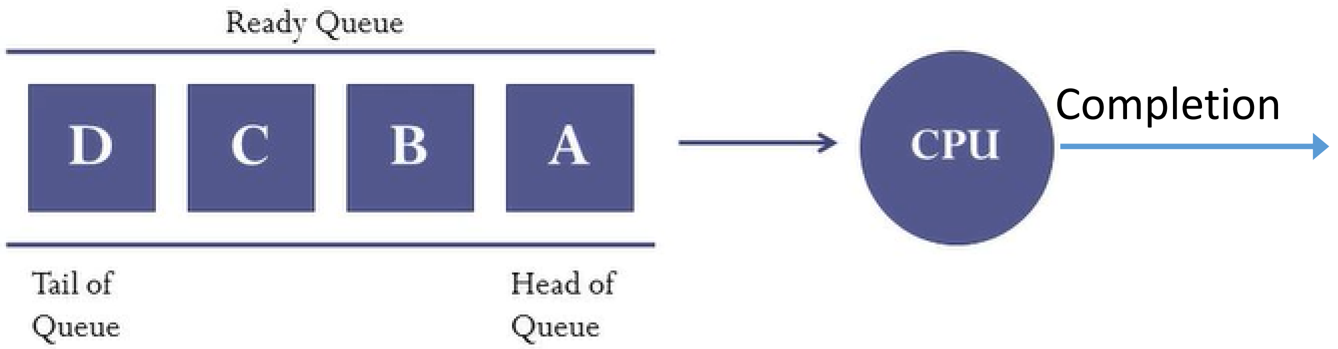
- A scheduling discipline should:
  - Be fair
  - Maximize throughput
  - Be predictable
  - Balance resource use
  - Provide tolerable response time
  - Avoid Indefinite postponement
  - Enforce priorities
  - Give preference to processes holding key resources
  - Degrade gracefully under heavy loads
  - Minimize overhead

Scheduling Algorithms

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-First (SJF) Scheduling
3. Priority Scheduling
4. Round Robin Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

First- Come, First-Served (FCFS) Scheduling

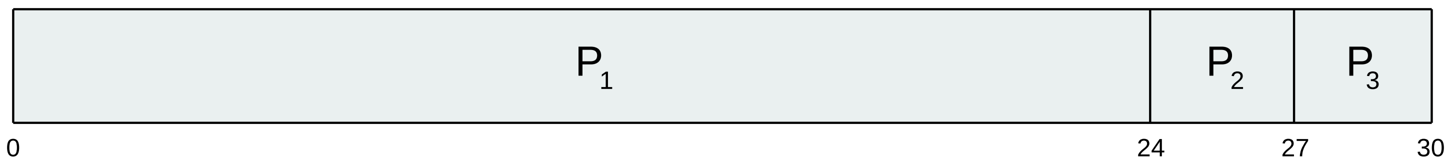
- Simplest scheduling algorithm
- The process that requests the CPU first is allocated the CPU first.
- FCFS policy is easily managed with a FIFO queue
- FCFS scheduling algorithm is non preemptive



First- Come, First-Served (FCFS) 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
  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 ms**

Waiting Time = Turn Around Time - Burst Time

Turn Around Time = Completion Time - Arrival Time

FCFS Scheduling (Cont.)

- The average waiting time under the FCFS policy is often quite long.
- Consider the performance of FCFS scheduling in a dynamic situation
  - Consider one CPU-bound and many I/O-bound processes
  - **Convoy effect** - short process behind long process to get off the CPU
  - Results in lower CPU and device utilization
- FCFS scheduling algorithm is nonpreemptive.
  - Troublesome for time-sharing systems

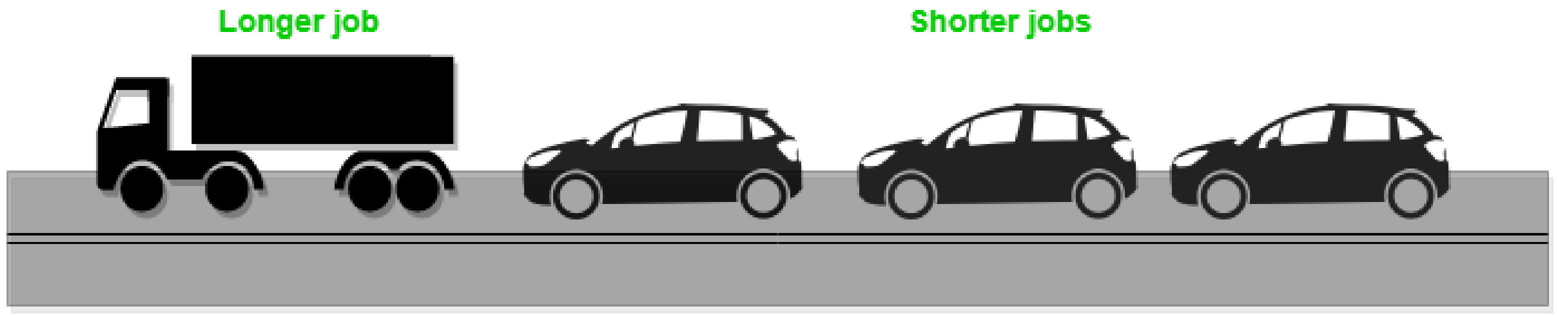


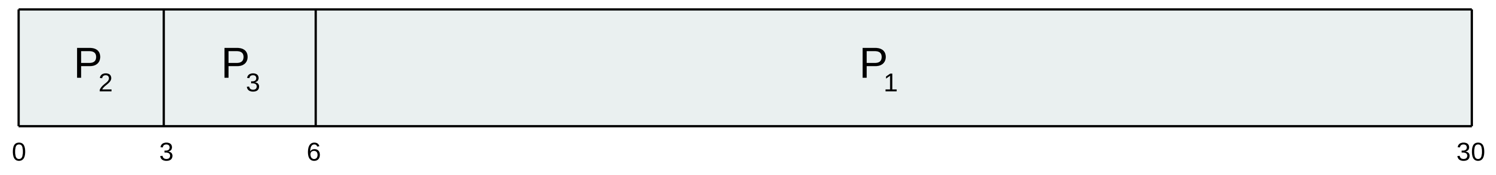
Figure - The Convey Effect, Visualized

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 ms**
- Much better than previous case

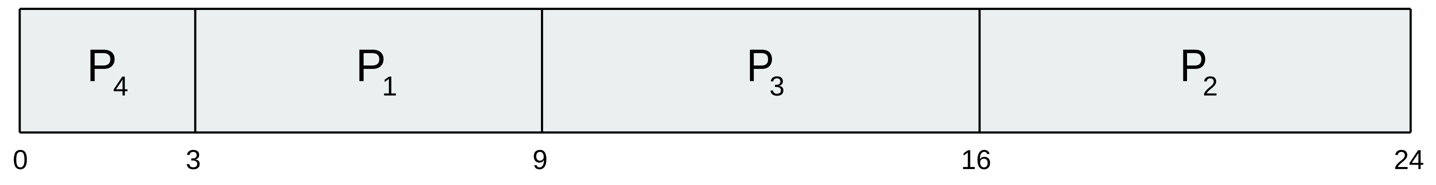
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
- Use these lengths to schedule the process with the shortest time
- If the next CPU bursts of two processes are the same, FCFS scheduling breaks the tie
- Scheduling depends on the length of the next CPU burst of a process
- SJF is optimal – gives minimum average waiting time for a given set of processes
- SJF scheduling is used frequently in long-term scheduling

Example of SJF

Process	Burst Time(ms)
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7ms
- The SJF scheduling algorithm is optimal, as it gives the minimum average waiting time for a given set of processes

Shortest-Job-First (SJF) Scheduling…

- The difficulty with the SJF algorithm is knowing the length of the next CPU request
- It cannot be implemented at the level of short-term CPU scheduling.
- Instead, approximate SJF scheduling
- To determine the length of the next CPU burst
  - Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

Determining Length of Next CPU Burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

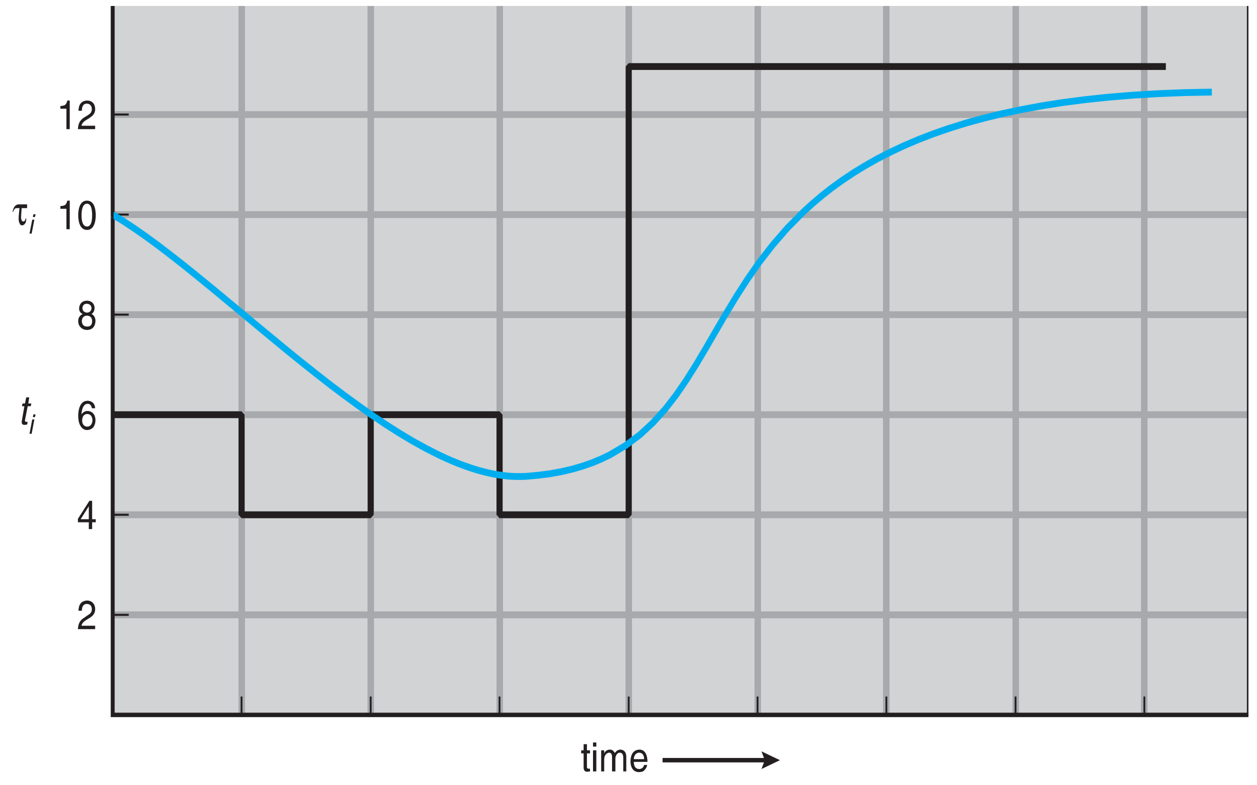
1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

- α controls the relative weight of recent and past history

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- Commonly, α is set to ½ , so recent history and past history are equally weighted

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 …

Shortest-Job-First (SJF) Scheduling…

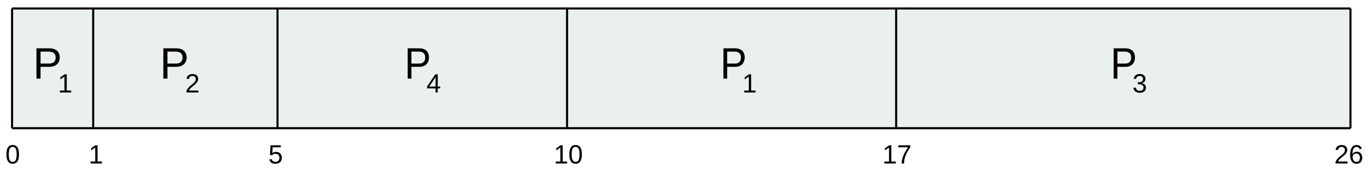
- The SJF algorithm can be either preemptive or non-preemptive.
- A preemptive SJF algorithm will preempt the currently executing process, whereas a non preemptive SJF algorithm will allow the currently running process to finish its CPU burst
- The preemptive version is called **shortest-remaining-time-first Scheduling**

Example of Shortest-Remaining-Time-First

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = **6.5 msec**

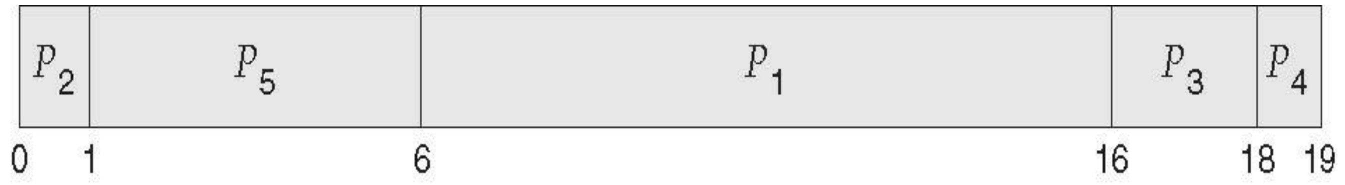
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
- Equal-priority processes are scheduled in FCFS order.
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

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 Gantt Chart



- Average waiting time = 8.2 msec

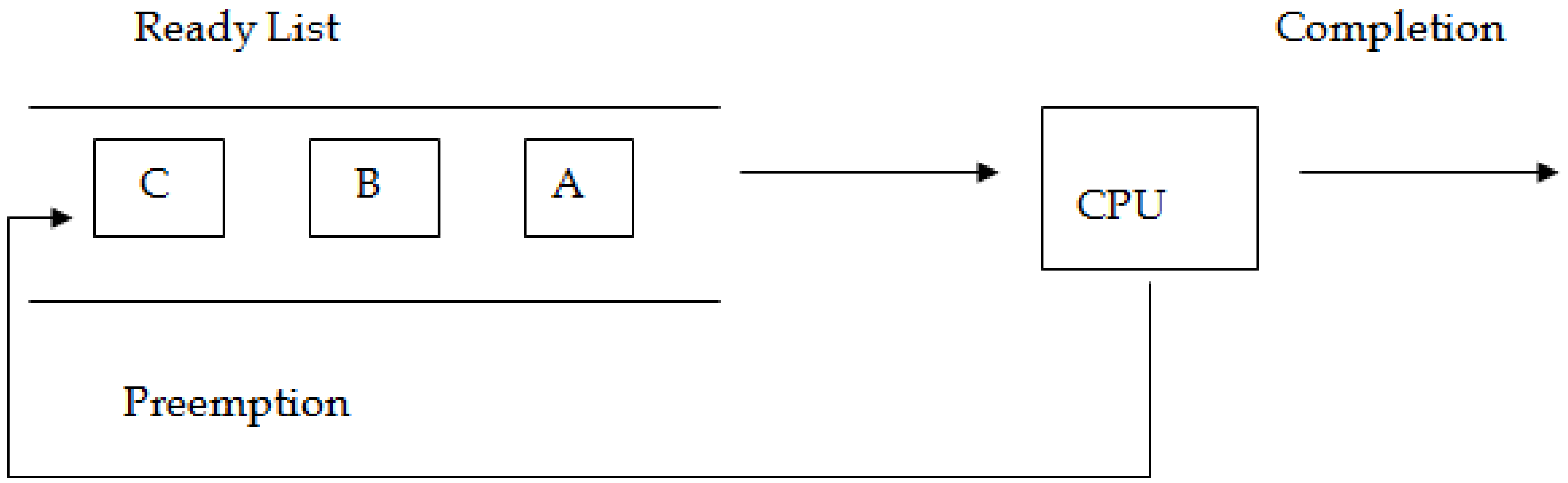
Priority Scheduling…

- Priorities can be defined either internally or externally
- Priority scheduling can be either preemptive or non-preemptive
  - The priority of new process is compared with the priority of the currently running process.
  - The preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
  - Non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue
- Problem ≡ **Indefinite blocking** or **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Round Robin (RR) Scheduling

- **Round-robin (RR) scheduling** algorithm is designed especially for time sharing systems
- A small unit of time, called a **time quantum** or **time slice**, is defined.
- A time quantum is generally from 10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue.
- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue
- Timer interrupts every quantum to schedule next process

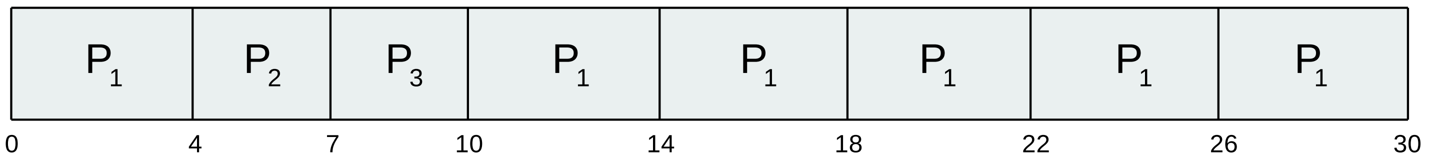
Round Robin (RR) Scheduling…



Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

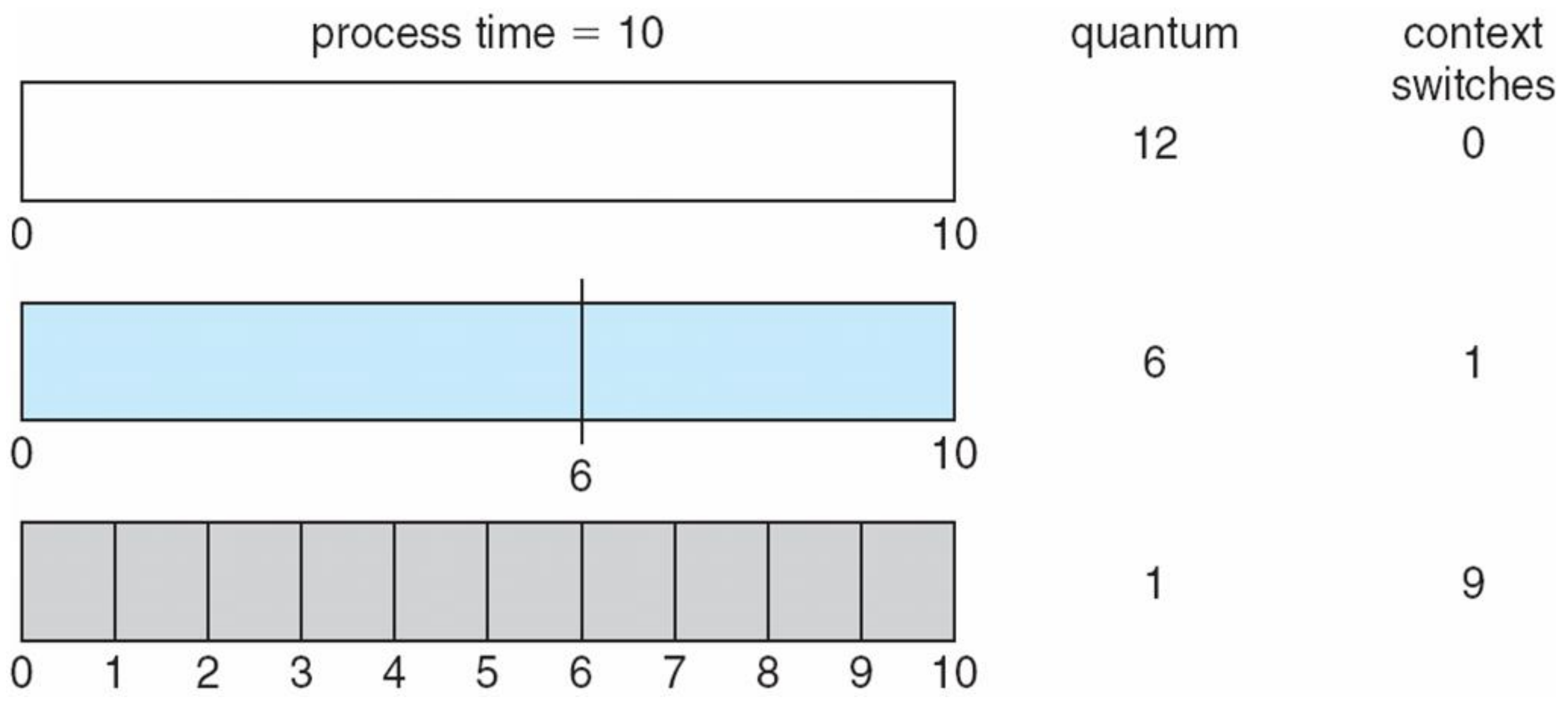


- Average waiting time is 17/3 = **5.66 ms**
- Typically, higher average turnaround than SJF, but better ***response time***

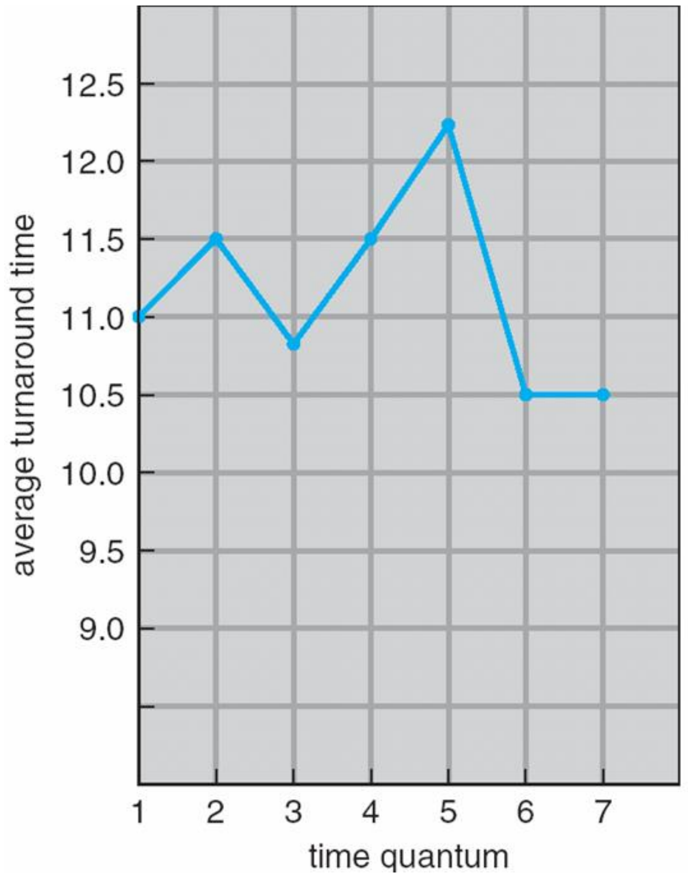
Round Robin (RR)

- Typically, higher average turnaround than SJF, but better ***response time***
- The RR scheduling algorithm is preemptive
- 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.
- The performance of RR algorithm depends heavily on the size of the time quantum.
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high
  - q is usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



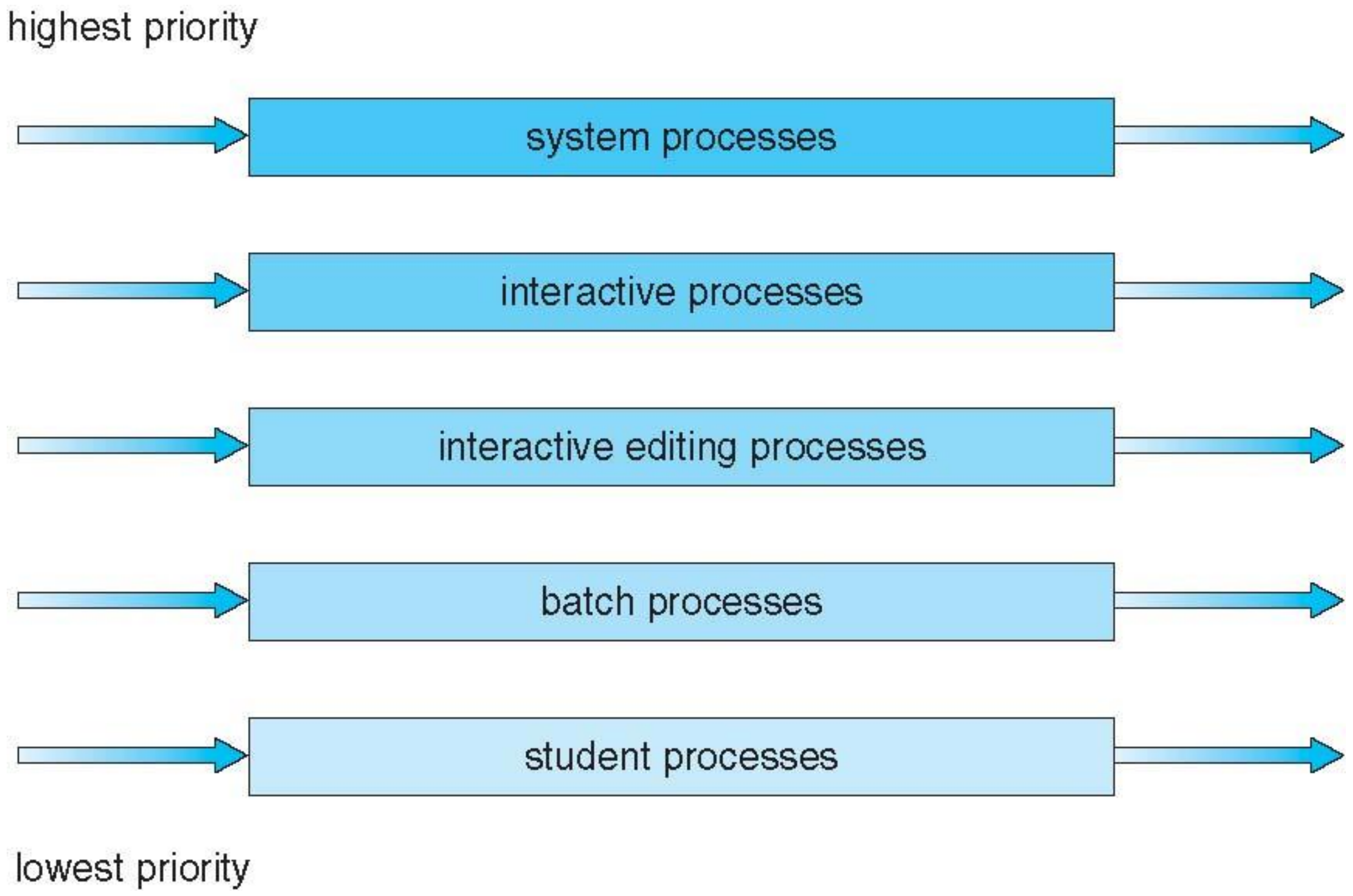
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **Foreground** (interactive)
  - **Background** (batch)
- Processes are permanently assigned to a queue
- 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 Queue Scheduling…

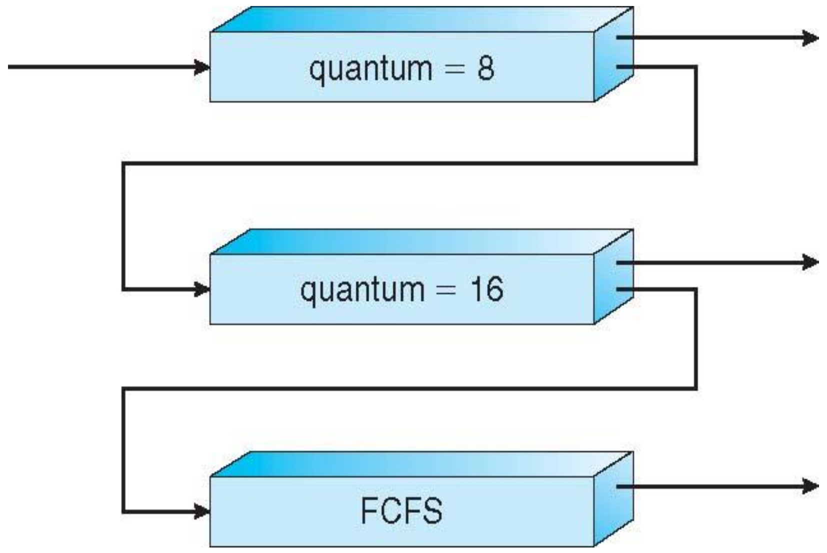
- Each queue has absolute priority over lower-priority queues.
- Another possibility is to time-slice among the queues.
  - Each queue gets a certain portion of the CPU time, which it can then schedule among its various processes
- Processes are permanently assigned to a queue when they enter the system.
- Processes do not move from one queue to the other
- Advantage: low scheduling overhead
- Limitation: It is inflexible.

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- Leaves I/O-bound and interactive processes in the higher-priority queues.
- A process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

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 Queue…

- 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 scheduler is the most general CPU-scheduling algorithm.
- It can be configured to match a specific system under design
- It is also the most complex algorithm

Practice Problems

- Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

a) What is the average turnaround time for these processes with the FCFS scheduling algorithm

b) What is the average turnaround time for these processes with the SJF scheduling algorithm?

Practice Problems…

- Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

The processes are assumed to have arrived in the order P1 , P2 , P3 , P4 , P5 , all at time 0.

a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b) What is the turnaround time of each process for each of the scheduling algorithms in part a?

c) What is the waiting time of each process for each of these scheduling algorithms? Which of the algorithms results in the minimum average waiting time (over all processes)?

Practice Problems…

- The following processes are being scheduled using a preemptive, round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an idle task (which consumes no CPU resources and is identified as Pidle). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

Thread	Priority	Burst	Arrival
P1	40	20	0
P2	30	25	25
P3	30	25	30
P4	35	15	60
P5	5	10	100
P6	10	10	105

a) Show the scheduling order of the processes using a Gantt chart.

b) What is the turnaround time for each process?

c) What is the waiting time for each process?

d) What is the CPU utilization rate?

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- User-level threads are managed by a thread library, and the kernel is unaware of them.
- To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread.
- Contention Scope
  - **Process-contention scope (PCS)**
  - **System-contention scope (SCS)**

Thread Scheduling

- Many-to-one and many-to-many models, thread library schedules user-level threads to run on an available LWP
  - Known as **process-contention scope** (**PCS**) since competition for the CPU takes place among threads belonging to the same process
  - Require the operating system to schedule the kernel thread onto a physical CPU
  - Typically done via priority set by programmer
  - PCS is preemptive
  - There is no guarantee of time slicing among threads of equal priority
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition for CPU takes place among all threads in system
  - Systems using the one-to-one model such as Windows, Linux, and Solaris, schedule threads using only SCS

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - **PTHREAD_SCOPE_PROCESS** schedules threads using PCS scheduling
  - **PTHREAD_SCOPE_SYSTEM** schedules threads using SCS scheduling
- On systems implementing the many-to-many model, the PTHREAD SCOPE PROCESS policy schedules user-level threads onto available LWPs.
- The PTHREAD SCOPE SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems(maps threads using the one-to-one policy).
- The Pthread IPC provides two functions for getting and settingthe contention scope policy:
  - **pthread attr setscope(pthread attr t *attr, int scope)**
  - **pthread attr getscope(pthread attr t *attr, int *scope)**
- Only certain contention scope values are allowed in some systems – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API…

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* 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);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

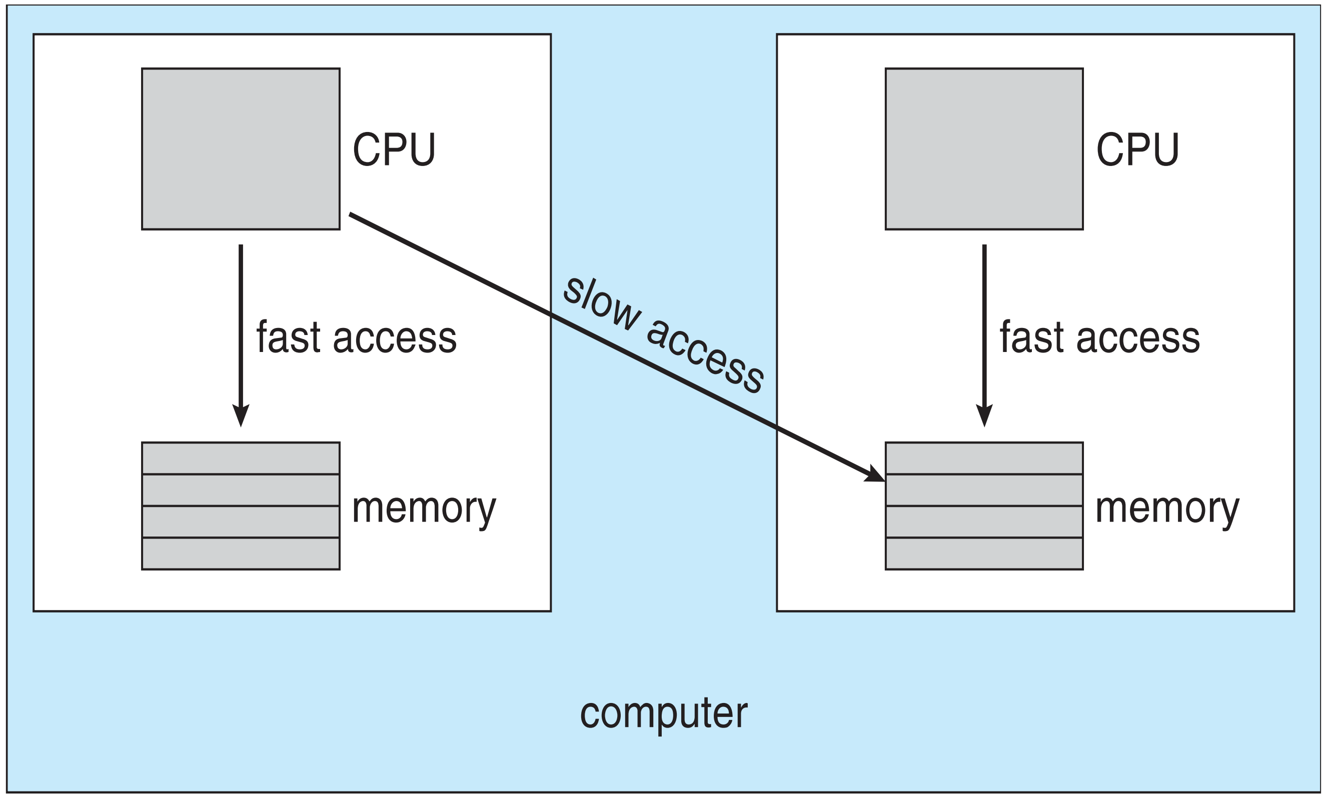
Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- Approaches to Multiple-Processor Scheduling
  1. **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
  2. **Symmetric multiprocessing** (**SMP**) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
     - Common in modern OS: Windows, Linux, and Mac OS X

Issues in SMP systems- Processor Affinity

- Most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor.
- **Processor affinity** – process has affinity for processor on which it is currently running
- Processor affinity takes several forms:
  - **soft affinity-** OS will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.
  - **Hard affinity-** Allows a process to specify a subset of processors on which it may run
  - Many systems provide both soft and hard affinity; example: Linux
  - Variations including **processor sets**

NUMA and CPU Scheduling



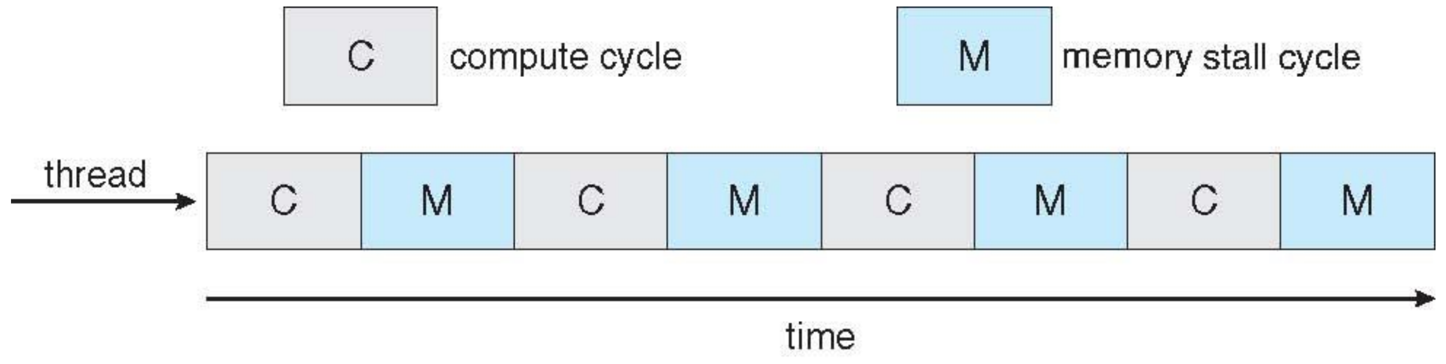
Note that memory-placement algorithms can also consider affinity

Issues in SMP systems– Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed across all processors in an SMP system.
- Load balancing is necessary only on systems where each processor has its own private queue of eligible processes to execute.
- Two general approaches to load balancing
  1. **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  2. **Pull migration** – idle processors pulls waiting task from busy processor
- Push and pull migration are implemented in parallel on load-balancing systems.
- Load balancing often counteracts the benefits of processor affinity

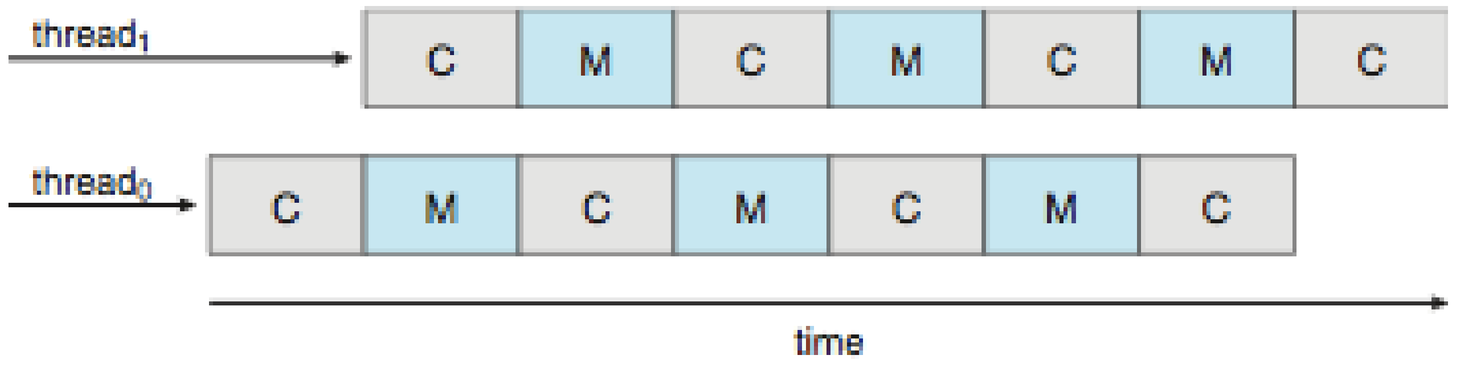
Issues in SMP systems: Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- SMP systems that use multicore processors are faster and consumes less power
- Multicore processors may complicate scheduling issues
- **Memory stall:** when a processor accesses memory, it spends a significant amount of time waiting for the data to become available.



Multithreaded Multicore System

- Solution: Multithreaded processor cores (Multiple hardware threads per core)
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- A dual-threaded processor core



  - Each hardware thread appears as a logical processor that is available to run a software thread
  - Thus, in a dual-threaded, dual-core system, four logical processors are presented to the operating system.

Multithreaded Multicore System

- Two ways to multithread a processing core:
  - Coarse- grained multithreading
    - Thread executes on a processor until a long-latency event such as a memory stall occurs
    - Cost of switching between threads is high
  - Fine-grained(Interleaved) multithreading
    - Switches between threads at a much finer level of granularity
    - Cost of switching between threads is small
- Multicore processor requires two different levels of scheduling
  - The operating system chooses which software thread to run on each hardware thread (logical processor)
  - A second level of scheduling specifies how each core decides which hardware thread to run - RR, dynamic urgency value…

Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it.
- The CPU is allocated to the selected process by the dispatcher.
- First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.
- Shortest- job-first (SJF) scheduling is optimal, providing the shortest average waiting time.
- Round-robin (RR) scheduling is more appropriate for a time-shared system.
- Multilevel queue algorithms allow different algorithms to be used for different classes of processes.
- Multilevel feedback queues allow processes to move from one queue to another.
- Multiple processors allow each processor to schedule itself independently.
- Issues related to multiprocessor scheduling include processor affinity, load balancing, and multicore processing

Review Questions

- Explain the difference between preemptive and non-preemptive scheduling.
- What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?
- Distinguish between PCS and SCS scheduling
- Why is it important for the scheduler to distinguish I/O -bound programs from CPU-bound programs?
- Which of the following scheduling algorithms could result in starvation?
  a. First-come, first-served
  b. Shortest job first
  c. Round robin
  d. Priority