MODULE 2: MULTI-THREADED PROGRAMMING

2.1 Multi-Threaded Programming

• A thread is a basic unit of CPU utilization.
• It consists of
→ thread ID
→ PC
→ register-set and
→ stack.
• It shares with other threads belonging to the same process its code-section & data-section.
• A traditional (or heavy weight) process has a single thread of control.
• If a process has multiple threads of control, it can perform more than one task at a time. Such a process is called **multi-threaded process** (Figure 2.1).

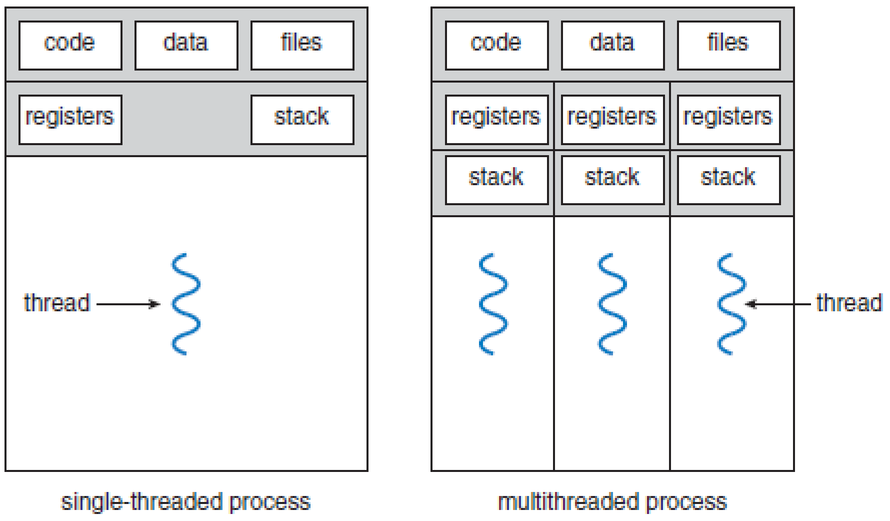


Figure 2.1 Single-threaded and multithreaded processes

2.1.1 Motivation

1) The software-packages that run on modern PCs are multithreaded.
An application is implemented as a separate process with several threads of control.
For ex: A word processor may have
→ first thread for displaying graphics
→ second thread for responding to keystrokes and
→ third thread for performing grammar checking.
2) In some situations, a single application may be required to perform several similar tasks.
For ex: A web-server may create a separate thread for each client request.
This allows the server to service several concurrent requests.
3) RPC servers are multithreaded.
When a server receives a message, it services the message using a separate thread.
This allows the server to service several concurrent requests.
4) Most OS kernels are multithreaded;
Several threads operate in kernel, and each thread performs a specific task, such as
→ managing devices or
→ interrupt handling.

2.1.2 Benefits

1) Responsiveness

• A program may be allowed to continue running even if part of it is blocked.
Thus, increasing responsiveness to the user.

2) Resource Sharing

• By default, threads share the memory (and resources) of the process to which they belong.
Thus, an application is allowed to have several different threads of activity within the same address-space.

3) Economy

• Allocating memory and resources for process-creation is costly.
Thus, it is more economical to create and context-switch threads.

4) Utilization of Multiprocessor Architectures

• In a multiprocessor architecture, threads may be running in parallel on different processors.
Thus, parallelism will be increased.

2.2 Multi-Threading Models

• Support for threads may be provided at either
  1) The user level, for **user threads** or
  2) By the kernel, for **kernel threads**.

• User-threads are supported above the kernel and are managed without kernel support.
  Kernel-threads are supported and managed directly by the OS.

• Three ways of establishing relationship between user-threads & kernel-threads:
  1) Many-to-one model
  2) One-to-one model and
  3) Many-to-many model.

2.2.1 Many-to-One Model

• Many user-level threads are mapped to one kernel thread (Figure 2.2).
• Advantage:
  1) Thread management is done by the thread library in user space, so it is efficient.
• Disadvantages:
  1) The entire process will block if a thread makes a blocking system-call.
  2) Multiple threads are unable to run in parallel on multiprocessors.
• For example:
  → Solaris green threads
  → GNU portable threads.

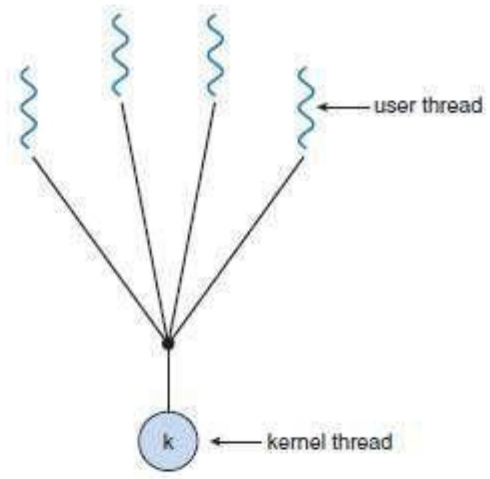


Figure 2.2 Many-to-one model

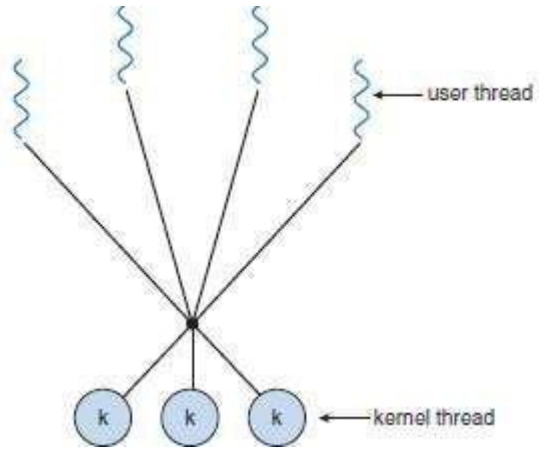


Figure 2.3 One-to-one model

2.2.2 One-to-One Model

• Each user thread is mapped to a kernel thread (Figure 2.3).
• Advantages:
  1) It provides more concurrency by allowing another thread to run when a thread makes a blocking system-call.
  2) Multiple threads can run in parallel on multiprocessors.
• Disadvantage:
  1) Creating a user thread requires creating the corresponding kernel thread.
• For example:
  → Windows NT/XP/2000
  → Linux

2.2.3 Many-to-Many Model

- Many user-level threads are multiplexed to a smaller number of kernel threads (Figure 2.4).
- Advantages:
    1) Developers can create as many user threads as necessary
    2) The kernel threads can run in parallel on a multiprocessor.
    3) When a thread performs a blocking system-call, kernel can schedule another thread for execution.

Two Level Model

- A variation on the many-to-many model is the two level-model (Figure 2.5).
- Similar to M:M, except that it allows a user thread to be bound to kernel thread.
- For example:
    → HP-UX
    → Tru64 UNIX

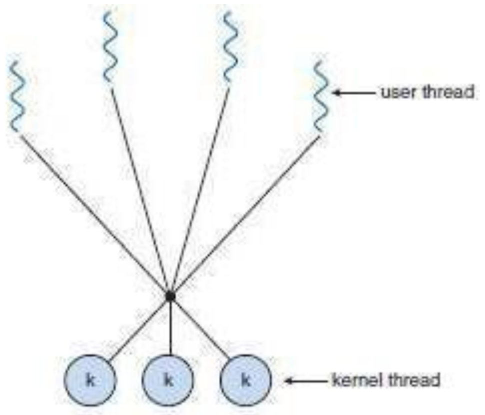


Figure 2.4 Many-to-many model

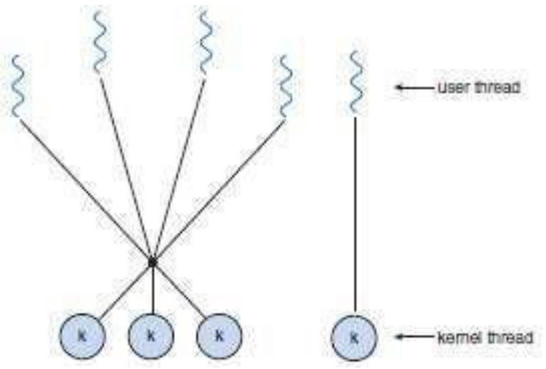


Figure 2.5 Two-level model

2.3 Thread Libraries

- It provides the programmer with an API for the creation and management of threads.
- Two ways of implementation:

  1) First Approach

  ➢ Provides a library entirely in user space with no kernel support.

  ➢ All code and data structures for the library exist in the user space.

  2) Second Approach

  ➢ Implements a kernel-level library supported directly by the OS.

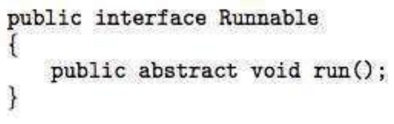
  ➢ Code and data structures for the library exist in kernel space.
- Three main thread libraries: 1) POSIX Pthreads
  2) Win32 and
  3) Java.

2.3.1 Pthreads

- This is a POSIX standard API for thread creation and synchronization.
- This is a specification for thread-behavior, not an implementation.
- OS designers may implement the specification in any way they wish.
- Commonly used in: UNIX and Solaris.

2.3.2 Java Threads

- Threads are the basic model of program-execution in
  → Java program and
  → Java language.
- The API provides a rich set of features for the creation and management of threads.
- All Java programs comprise at least a single thread of control.
- Two techniques for creating threads:
  1) Create a new class that is derived from the Thread class and override its run() method.
  2) Define a class that implements the Runnable interface. The Runnable interface is defined as follows:



```
public interface Runnable
{
    public abstract void run();
}
```

2.4 Threading Issues

2.4.1 fork() and exec() System-calls

• fork() is used to create a separate, duplicate process.
• If one thread in a program calls fork(), then
  1) Some systems duplicates all threads and
  2) Other systems duplicate only the thread that invoked the fork().
• If a thread invokes the exec(), the program specified in the parameter to exec() will replace the entire process including all threads.

2.4.2 Thread Cancellation

• This is the task of terminating a thread before it has completed.
• Target thread is the thread that is to be canceled
• Thread cancellation occurs in two different cases:
  1) **Asynchronous cancellation**: One thread immediately terminates the target thread.
  2) **Deferred cancellation**: The target thread periodically checks whether it should be terminated.

2.4.3 Signal Handling

• In UNIX, a signal is used to notify a process that a particular event has occurred.
• All signals follow this pattern:
  1) A signal is generated by the occurrence of a certain event.
  2) A generated signal is delivered to a process.
  3) Once delivered, the signal must be handled.
• A signal handler is used to process signals.
• A signal may be received either synchronously or asynchronously, depending on the source.
  1) Synchronous signals
  ➢ Delivered to the same process that performed the operation causing the signal.
  ➢ E.g. illegal memory access and division by 0.
  2) Asynchronous signals
  ➢ Generated by an event external to a running process.
  ➢ E.g. user terminating a process with specific keystrokes <ctrl><c>.
• Every signal can be handled by one of two possible handlers:
  1) A Default Signal Handler
  ➢ Run by the kernel when handling the signal.
  2) A User-defined Signal Handler
  ➢ Overrides the default signal handler.
• In **single-threaded programs**, delivering signals is simple.
  In **multithreaded programs**, delivering signals is more complex. Then, the following options exist:
  1) Deliver the signal to the thread to which the signal applies.
  2) Deliver the signal to every thread in the process.
  3) Deliver the signal to certain threads in the process.
  4) Assign a specific thread to receive all signals for the process.

2.4.4 Thread Pools

• The basic idea is to
  → create a no. of threads at process-startup and
  → place the threads into a pool (where they sit and wait for work).
• Procedure:
  1) When a server receives a request, it awakens a thread from the pool.
  2) If any thread is available, the request is passed to it for service.
  3) Once the service is completed, the thread returns to the pool.
• Advantages:
  1) Servicing a request with an existing thread is usually faster than waiting to create a thread.
  2) The pool limits the no. of threads that exist at any one point.
• No. of threads in the pool can be based on factors such as
  → no. of CPUs
  → amount of memory and
  → expected no. of concurrent client-requests.

MODULE 2 (CONT.): PROCESS SCHEDULING

2.5 Basic Concepts

• In a single-processor system,
  → only one process may run at a time.
  → other processes must wait until the CPU is rescheduled.
• Objective of multiprogramming:
  → to have some process running at all times, in order to maximize CPU utilization.

2.5.1 CPU-I/0 Burst Cycle

• Process execution consists of a cycle of
  → CPU execution and
  → I/O wait (Figure 2.6 & 2.7).
• Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…
• Finally, a CPU burst ends with a request to terminate execution.
• An I/O-bound program typically has many short CPU bursts.
  A CPU-bound program might have a few long CPU bursts.

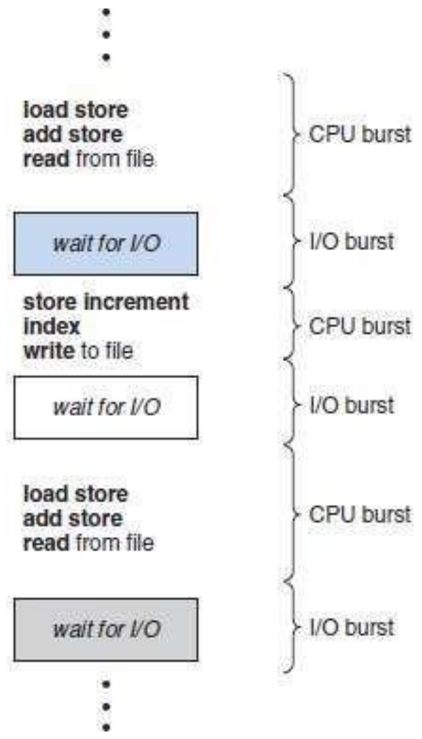


Figure 2.6 Alternating sequence of CPU and I/O bursts

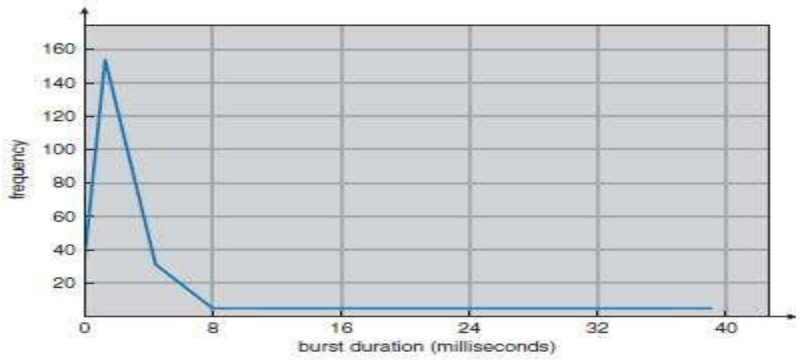


Figure 2.7 Histogram of CPU-burst durations

2.5.2 CPU Scheduler

• This scheduler
  → selects a waiting-process from the ready-queue and
  → allocates CPU to the waiting-process.

• The ready-queue could be a FIFO, priority queue, tree and list.

• The records in the queues are generally process control blocks (PCBs) of the processes.

2.5.3 CPU Scheduling

• Four situations under which CPU scheduling decisions take place:
  1) When a process switches from the running state to the waiting state.
     For ex; I/O request.
  2) When a process switches from the running state to the ready state.
     For ex: when an interrupt occurs.
  3) When a process switches from the waiting state to the ready state.
     For ex: completion of I/O.
  4) When a process terminates.

• Scheduling under 1 and 4 is non-preemptive.
  Scheduling under 2 and 3 is preemptive.

Non Preemptive Scheduling

• 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.

Preemptive Scheduling

• This is driven by the idea of prioritized computation.

• Processes that are runnable may be temporarily suspended

• Disadvantages:
  1) Incurs a cost associated with access to shared-data.
  2) Affects the design of the OS kernel.

2.5.4 Dispatcher

• It gives control of the CPU to the process selected by the short-term scheduler.

• The function involves:
  1) Switching context
  2) Switching to user mode &
  3) Jumping to the proper location in the user program to restart that program.

• It should be as fast as possible, since it is invoked during every process switch.

• **Dispatch latency** means the time taken by the dispatcher to
  → stop one process and
  → start another running.

2.6 Scheduling Criteria

• Different CPU-scheduling algorithms
→ have different properties and
→ may favor one class of processes over another.

• Criteria to compare CPU-scheduling algorithms:

1) CPU Utilization
➢ We must keep the CPU as busy as possible.
➢ In a real system, it ranges from 40% to 90%.

2) Throughput
➢ Number of processes completed per time unit.
➢ For long processes, throughput may be 1 process per hour;
For short transactions, throughput might be 10 processes per second.

3) Turnaround Time
➢ The interval from the time of submission of a process to the time of completion.
➢ Turnaround time is the sum of the periods spent
→ waiting to get into memory
→ waiting in the ready-queue
→ executing on the CPU and
→ doing I/O.

4) Waiting Time
➢ The amount of time that a process spends waiting in the ready-queue.

5) Response Time
➢ The time from the submission of a request until the first response is produced.
➢ The time is generally limited by the speed of the output device.

• We want
→ to maximize CPU utilization and throughput and
→ to minimize turnaround time, waiting time, and response time.

2.7 Scheduling Algorithms

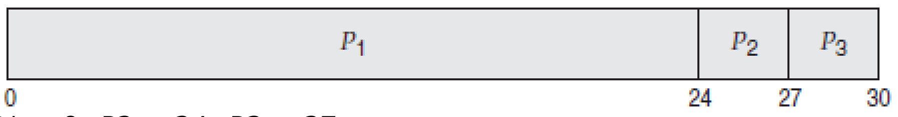
• CPU scheduling deals with the problem of deciding which of the processes in the ready-queue is to be allocated the CPU.
• Following are some scheduling algorithms:
  1) FCFS scheduling (First Come First Served)
  2) Round Robin scheduling
  3) SJF scheduling (Shortest Job First)
  4) SRT scheduling
  5) Priority scheduling
  6) Multilevel Queue scheduling and
  7) Multilevel Feedback Queue scheduling

2.7.1 FCFS Scheduling

• The process that requests the CPU first is allocated the CPU first.
• The implementation is easily done using a FIFO queue.
• Procedure:
  1) When a process enters the ready-queue, its PCB is linked onto the tail of the queue.
  2) When the CPU is free, the CPU is allocated to the process at the queue's head.
  3) The running process is then removed from the queue.
• Advantage:
  1) Code is simple to write & understand.
• Disadvantages:
  1) **Convoy effect:** All other processes wait for one big process to get off the CPU.
  2) Non-preemptive (a process keeps the CPU until it releases it).
  3) Not good for time-sharing systems.
  4) The average waiting time is generally not minimal.
• Example: Suppose that the processes arrive in the order P1, P2, P3.

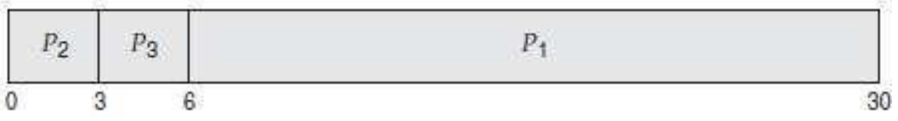
Process	Burst Time
P_1	24
P_2	3
P_3	3

• The Gantt Chart for the schedule is as follows:



• Waiting time for P1 = 0; P2 = 24; P3 = 27
  Average waiting time: (0 + 24 + 27)/3 = 17

• Suppose that the processes arrive in the order P2, P3, P1.
• The Gantt chart for the schedule is as follows:



• Waiting time for P1 = 6;P2 = 0; P3 = 3
  Average waiting time: (6 + 0 + 3)/3 = 3

2.7.2 SJF Scheduling

• The CPU is assigned to the process that has the smallest next CPU burst.
• If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.
• For long-term scheduling in a batch system, we can use the process time limit specified by the user, as the ‗length‛
• SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst
• Advantage:

1) The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.

• Disadvantage:

1) Determining the length of the next CPU burst.

• SJF algorithm may be either 1) non-preemptive or
2) preemptive.

1) Non preemptive SJF

➢ The current process is allowed to finish its CPU burst.

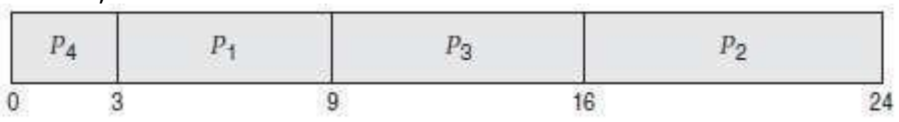
2) Preemptive SJF

➢ If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted.
➢ It is also known as **SRTF** scheduling (Shortest-Remaining-Time-First).

• Example (for non-preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

• For non-preemptive SJF, the Gantt Chart is as follows:

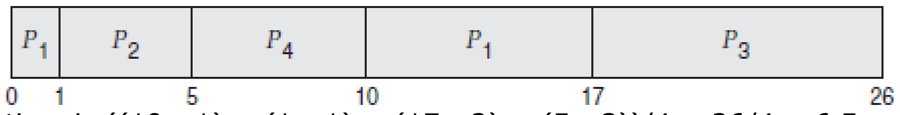


• Waiting time for P1 = 3; P2 = 16; P3 = 9; P4=0
Average waiting time: (3 + 16 + 9 + 0)/4 = 7

• Example (preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

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

• For preemptive SJF, the Gantt Chart is as follows:



• The average waiting time is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5.

2.7.3 Priority Scheduling

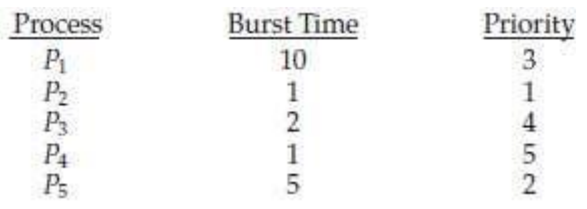
• A priority is associated with each process.
• The CPU is allocated to the process with the highest priority.
• Equal-priority processes are scheduled in FCFS order.
• Priorities can be defined either internally or externally.

1) **Internally-defined** priorities.
➢ Use some measurable quantity to compute the priority of a process.
➢ For example: time limits, memory requirements, no. of open files.
2) **Externally-defined** priorities.
➢ Set by criteria that are external to the OS
➢ For example:
→ importance of the process
→ political factors

• Priority scheduling can be either preemptive or nonpreemptive.

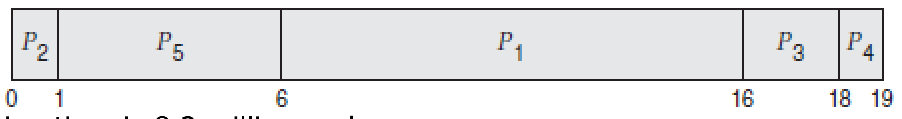
1) Preemptive
➢ The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process.
2) Non Preemptive
➢ The new process is put at the head of the ready-queue

• Advantage:
1) Higher priority processes can be executed first.
• Disadvantage:
1) Indefinite blocking, where low-priority processes are left waiting indefinitely for CPU.
Solution: **Aging** is a technique of increasing priority of processes that wait in system for a long time.
• Example: Consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., P5, with the length of the CPU-burst time given in milliseconds.



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

• The Gantt chart for the schedule is as follows:



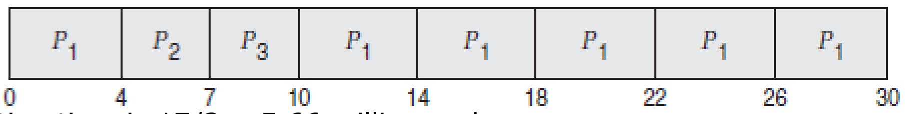
• The average waiting time is 8.2 milliseconds.

2.7.4 Round Robin Scheduling

• Designed especially for timesharing systems.
• It is similar to FCFS scheduling, but with preemption.
• A small unit of time is called a **time quantum** (or time slice).
• Time quantum is ranges from 10 to 100 ms.
• The ready-queue is treated as a **circular queue**.
• The CPU scheduler
→ goes around the ready-queue and
→ allocates the CPU to each process for a time interval of up to 1 time quantum.
• To implement:
The ready-queue is kept as a FIFO queue of processes
• CPU scheduler
1) Picks the first process from the ready-queue.
2) Sets a timer to interrupt after 1 time quantum and
3) Dispatches the process.
• One of two things will then happen.
1) The process may have a CPU burst of less than 1 time quantum.
In this case, the process itself will release the CPU voluntarily.
2) If the CPU burst of the currently running process is longer than 1 time quantum,
the timer will go off and will cause an interrupt to the OS.
The process will be put at the tail of the ready-queue.
• Advantage:
1) Higher average turnaround than SJF.
• Disadvantage:
1) Better response time than SJF.
• Example: Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	24
P_2	3
P_3	3

• The Gantt chart for the schedule is as follows:



• The average waiting time is 17/3 = 5.66 milliseconds.
• The RR scheduling algorithm is preemptive.
No process is allocated the CPU for more than 1 time quantum in a row. If a process' CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready-queue..
• The performance of algorithm depends heavily on the size of the time quantum (Figure 2.8 & 2.9).
1) If time quantum=very large, RR policy is the same as the FCFS policy.
2) If time quantum=very small, RR approach appears to the users as though each of n processes has its own processor running at l/n the speed of the real processor.
• In software, we need to consider the effect of context switching on the performance of RR scheduling
1) Larger the time quantum for a specific process time, less time is spend on context switching.
2) The smaller the time quantum, more overhead is added for the purpose of context-switching.

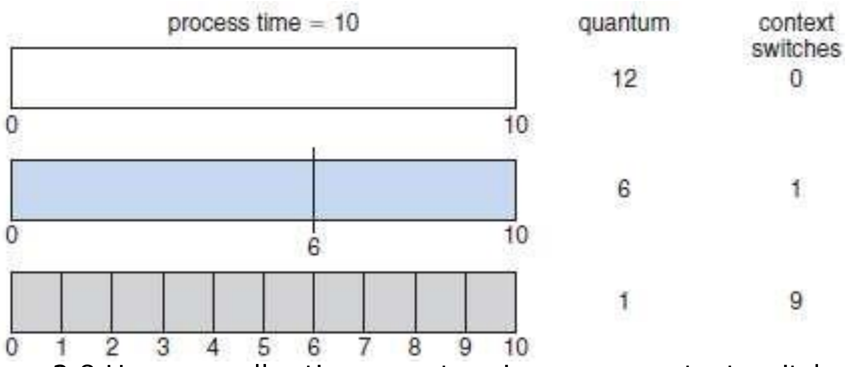
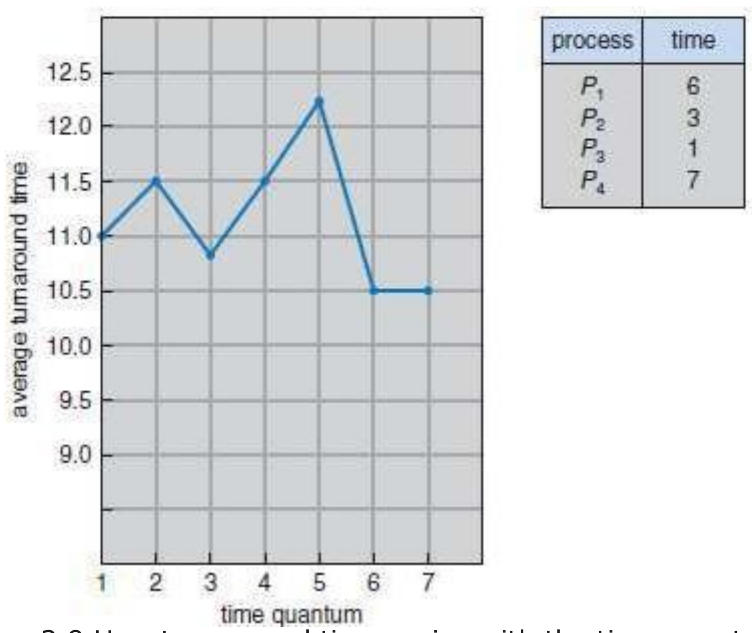


Figure 2.8 How a smaller time quantum increases context switches



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 2.9 How turnaround time varies with the time quantum

2.7.5 Multilevel Queue Scheduling

• Useful for situations in which processes are easily classified into different groups.
• For example, a common division is made between
→ foreground (or interactive) processes and
→ background (or batch) processes.
• The ready-queue is partitioned into several separate queues (Figure 2.10).
• The processes are permanently assigned to one queue based on some property like
→ memory size
→ process priority or
→ process type.
• Each queue has its own scheduling algorithm.
For example, separate queues might be used for foreground and background processes.

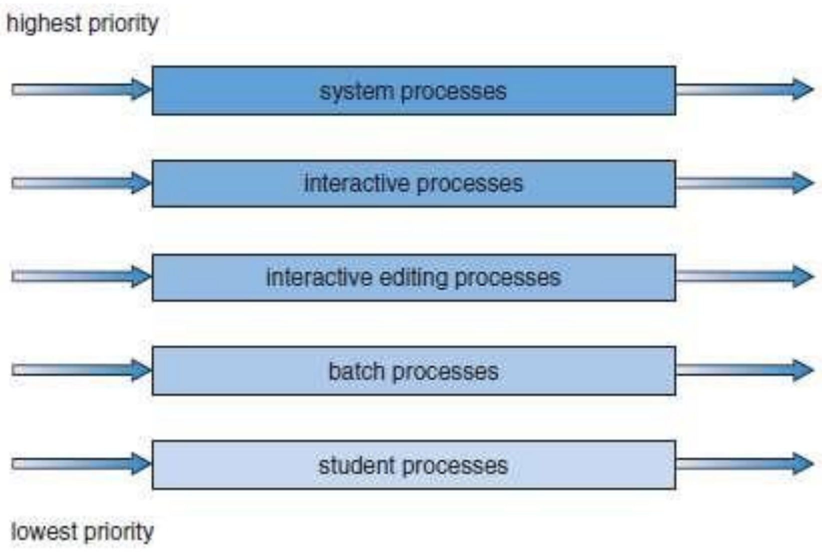


Figure 2.10 Multilevel queue scheduling

• There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
For example, the foreground queue may have absolute priority over the background queue.
• **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

2.7.6 Multilevel Feedback Queue Scheduling

- A process may move between queues (Figure 2.11).
- The basic idea:

  Separate processes according to the features of their CPU bursts. For example

  1) If a process uses too much CPU time, it will be moved to a lower-priority queue.

  ¤ This scheme leaves I/O-bound and interactive processes in the higher-priority queues.

  2) If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue

  ¤ This form of aging prevents starvation.

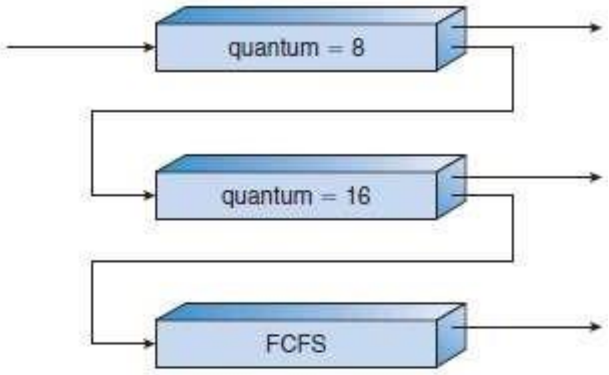


Figure 2.11 Multilevel feedback queues.

- In general, a multilevel feedback queue scheduler is defined by the following parameters:
  1) The number of queues.
  2) The scheduling algorithm for each queue.
  3) The method used to determine when to upgrade a process to a higher priority queue.
  4) The method used to determine when to demote a process to a lower priority queue.
  5) The method used to determine which queue a process will enter when that process needs service.

2.8 Multiple Processor Scheduling

• If multiple CPUs are available, the scheduling problem becomes more complex.
• Two approaches:

1) Asymmetric Multiprocessing

➢ The basic idea is:

i) A master server is a single processor responsible for all scheduling decisions, I/O processing and other system activities.
ii) The other processors execute only user code.

➢ Advantage:

i) This is simple because only one processor accesses the system data structures, reducing the need for data sharing.

2) Symmetric Multiprocessing

➢ The basic idea is:

i) Each processor is self-scheduling.
ii) To do scheduling, the scheduler for each processor

i. Examines the ready-queue and
ii. Selects a process to execute.

➢ Restriction: We must ensure that two processors do not choose the same process and that processes are not lost from the queue.

2.8.1 Processor Affinity

• In SMP systems,

1) Migration of processes from one processor to another are avoided and
2) Instead processes are kept running on same processor. This is known as processor affinity.

• Two forms:

1) Soft Affinity

➢ When an OS try to keep a process on one processor because of policy, but cannot guarantee it will happen.
➢ It is possible for a process to migrate between processors.

2) Hard Affinity

➢ When an OS have the ability to allow a process to specify that it is not to migrate to other processors. Eg: Solaris OS

2.8.2 Load Balancing

• This attempts to keep the workload evenly distributed across all processors in an SMP system.
• Two approaches:

1) Push Migration

➢ A specific task periodically checks the load on each processor and if it finds an imbalance, it evenly distributes the load to idle processors.

2) Pull Migration

➢ An idle processor pulls a waiting task from a busy processor.

2.8.3 Symmetric Multithreading

• The basic idea:

1) Create multiple logical processors on the same physical processor.
2) Present a view of several logical processors to the OS.

• Each logical processor has its own architecture state, which includes general-purpose and machine-state registers.
• Each logical processor is responsible for its own interrupt handling.
• SMT is a feature provided in hardware, not software.

2.9 Thread Scheduling

• On OSs, it is kernel-level threads but not processes that are being scheduled by the OS.
• 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 be mapped to an associated kernel-level thread.

2.9.1 Contention Scope

• Two approaches:

1) Process-Contention scope

➢ On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP.
➢ Competition for the CPU takes place among threads belonging to the same process.

2) System-Contention scope

➢ The process of deciding which kernel thread to schedule on the CPU.
➢ Competition for the CPU takes place among all threads in the system.
➢ Systems using the one-to-one model schedule threads using only SCS.

2.9.2 Pthread Scheduling

• Pthread API that allows specifying either PCS or SCS during thread creation.
• Pthreads identifies the following contention scope values:

1) PTHREAD_SCOPEJPROCESS schedules threads using PCS scheduling.
2) PTHREAD-SCOPE_SYSTEM schedules threads using SCS scheduling.

• Pthread IPC provides following two functions for getting and setting the contention scope policy:

1) pthread_attr_setscope(pthread_attr_t *attr, int scope)
2) pthread_attr_getscope(pthread_attr_t *attr, int *scope)

Exercise Problems

1) Consider the following set of processes, with length of the CPU burst time given in milliseconds:

Process	Arrival Time	Burst Time	Priority
P1	0	10	3
P2	0	1	1
P3	3	2	3
P4	5	1	4
P5	10	5	2

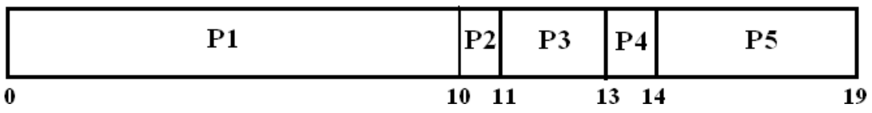
(i) Draw four Gantt charts illustrating the execution of these processing using FCFS, SJF, a non preemptive priority and RR (Quantum=2) scheduling.
(ii) What is the turn around time of each process for each scheduling algorithm in (i).
(iii) What is waiting time of each process in (i)

Solution:

$$\text{Average turn around time} = \frac{\text{Sum of waiting time of individual process}}{\text{Number of processes}}$$

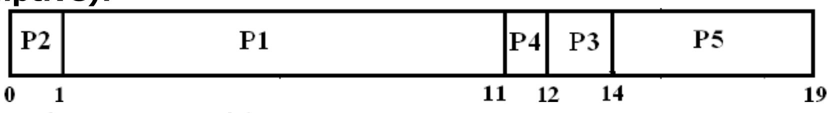
$$\text{Average waiting time} = \frac{\text{Sum of turn around time of individual process}}{\text{Number of processes}}$$

(i) FCFS:



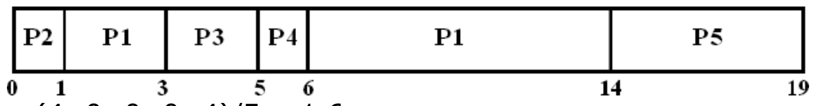
Average waiting time = (0+10+8+8+4)/5 = 6
Average turnaround time = (10+11+13+14+19)/5 = 13.4

(ii) SJF (non-preemptive):



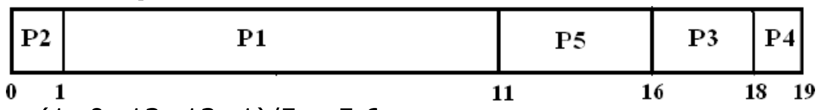
Average waiting time = (1+0+9+6+4)/5 = 4
Average turnaround time = (11+1+14+12+19)/5 = 11.4

SJF (preemptive):



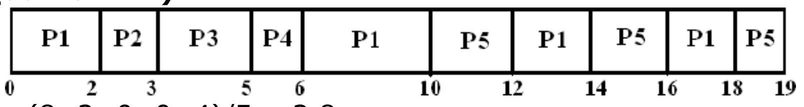
Average waiting time = (4+0+0+0+4)/5 = 1.6
Average turnaround time = (14+1+5+6+19)/5 = 9

(iii) Non-preemptive, Priority:



Average waiting time = (1+0+13+13+1)/5 = 5.6
Average turnaround time = (11+1+18+19+16)/5 = 13

(iv) Round Robin (Quantum=2):



Average waiting time = (8+2+0+0+4)/5 = 2.8
Average turnaround time = (18+3+5+6+19)/5 = 10.2

2) Consider the following set of process with arrival time:

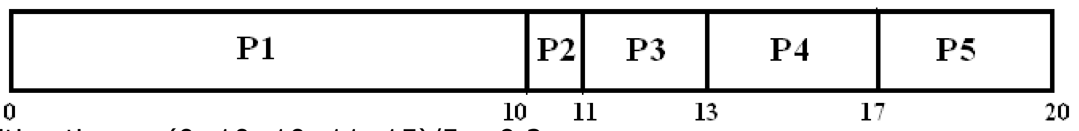
i) Draw grant chart using FCFS, SJF preemptive and non preemptive scheduling.

ii) Calculate the average waiting and turnaround time for each process of the scheduling algorithm.

Process	Arrival Time	Burst Time
P1	0	10
P2	0	1
P3	1	2
P4	2	4
P5	2	3

Solution:

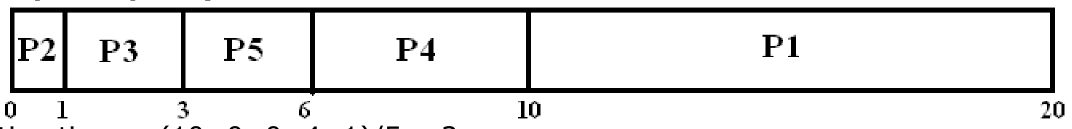
(i) FCFS:



Average waiting time = (0+10+10+11+15)/5 = 9.2

Average turnaround time = (10+11+13+17+20)/5 = 14.2

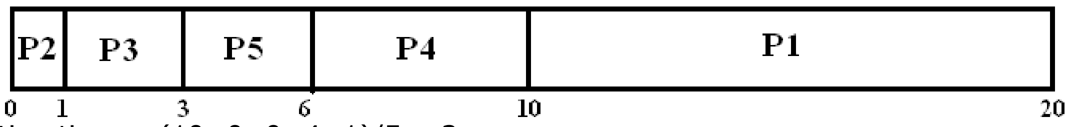
(ii) SJF (non-preemptive):



Average waiting time = (10+0+0+4+1)/5 = 3

Average turnaround time = (20+1+3+10+6)/5 = 8

(iii) SJF (preemptive):



Average waiting time = (10+0+0+4+1)/5 = 3

Average turnaround time = (20+1+3+10+6)/5 = 8

3) Consider following set of processes with CPU burst time (in msec)

Process	Arrival Time	Burst Time
P0	0	6
P1	1	3
P2	2	1
P3	3	4

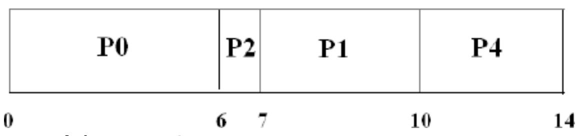
i) Draw Gantt chart illustrating the execution of above processes using SRTF and non preemptive SJF
ii) Find the turnaround time for each process for SRTF and SJF. Hence show that SRTF is faster than SJF.

Solution:

$$\text{Average turn around time} = \frac{\text{Sum of waiting time of individual process}}{\text{Number of processes}}$$

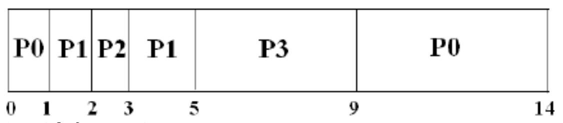
$$\text{Average waiting time} = \frac{\text{Sum of turn around time of individual process}}{\text{Number of processes}}$$

(i) Non-preemptive SJF:



Average waiting time = (0+6+4+7)/4 = 4.25
Average turnaround time = (6+10+7+14)/4 = 9.25

(ii) SRTF (preemptive SJF):



Average waiting time = (8+1+0+2)/4 = 2.75
Average turnaround time = (14+5+3+9)/4 = 7.75

Conclusion:
Since average turnaround time of SRTF(7.75) is less than SJF(9.25), SRTF is faster than SJF.

4) Following is the snapshot of a cpu

Process	Arrival Time	Burst Time
P1	0	10
P2	1	29
P3	2	03
P4	3	07

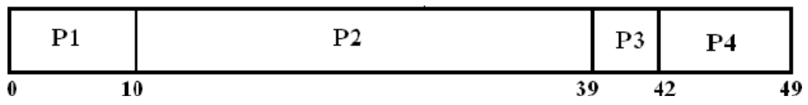
Draw Gantt charts and calculate the waiting and turnaround time using FCFS, SJF and RR with time quantum 10 scheduling algorithms.

Solution:

$$\text{Average turn around time} = \frac{\text{Sum of waiting time of individual process}}{\text{Number of processes}}$$

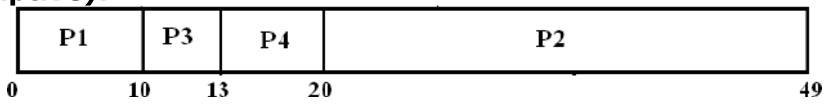
$$\text{Average waiting time} = \frac{\text{Sum of turn around time of individual process}}{\text{Number of processes}}$$

(i) FCFS:



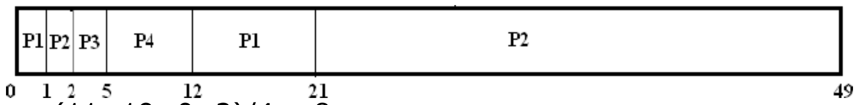
Average waiting time = (0+9+37+39)/4 = 21.25
Average turnaround time = (10+39+42+49)/4= 35

(ii) SJF (non-preemptive):



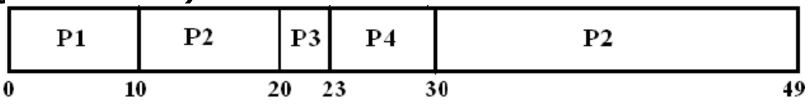
Average waiting time = (0+19+8+10)/4 = 9.25
Average turnaround time = (10+49+13+20)/4 =
19

SJF (preemptive):



Average waiting time = (11+19+0+2)/4 = 8
Average turnaround time = (21+49+5+12)/4 = 21.75

(iii) Round Robin (Quantum=10):



Average waiting time = (0+19+18+20)/4 = 14.25
Average turnaround time = (10+49+23+30)/4 =
28

Wisdom is knowing. Skill is know how to do it. Virtue is doing it.

5) Consider the following set of process:

Process	Arrival Time	Burst Time
P1	0	5
P2	1	1
P3	2	4

Compute average turn around time and average waiting time using

i) FCFS

ii) Preemptive SJF and

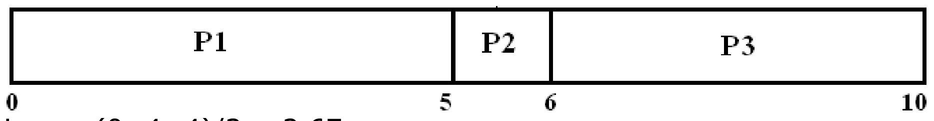
iii) RR (quantum-4).

Solution:

$$\text{Average turn around time} = \frac{\text{Sum of waiting time of individual process}}{\text{Number of processes}}$$

$$\text{Average waiting time} = \frac{\text{Sum of turn around time of individual process}}{\text{Number of processes}}$$

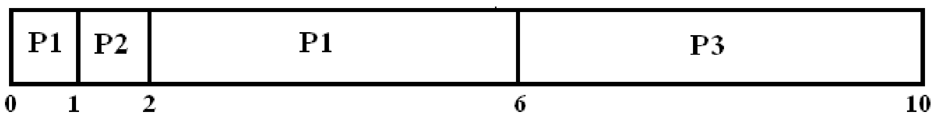
(i) FCFS:



Average waiting time = (0+4+4)/3 = 2.67

Average turnaround time = (5+6+10)/3 = 6.67

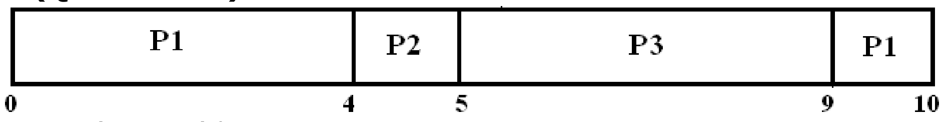
(ii) SJF (preemptive):



Average waiting time = (1+0+4)/3 = 1.67

Average turnaround time = (6+2+10)/3 = 6

(iii) Round Robin (Quantum=4):



Average waiting time = (5+3+3)/3 = 3.34

Average turnaround time = (10+5+9)/3 = 8

MODULE 2 (CONT.): PROCESS SYNCHRONIZATION

2.10 Synchronization

• Co-operating process is one that can affect or be affected by other processes.
• Co-operating processes may either
→ share a logical address-space (i.e. code & data) or
→ share data through files or
→ messages through threads.
• Concurrent-access to shared-data may result in data-inconsistency.
• To maintain data-consistency:
The orderly execution of co-operating processes is necessary.
• Suppose that we wanted to provide a solution to **producer-consumer problem** that fills all buffers.
We can do so by having an variable counter that keeps track of the no. of full buffers.
Initially, counter=0.
➢ counter is incremented by the producer after it produces a new buffer.
➢ counter is decremented by the consumer after it consumes a buffer.
• **Shared-data:**

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer Process:

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Consumer Process:

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

• A situation where several processes access & manipulate same data concurrently and the outcome of the execution depends on particular order in which the access takes place, is called a **race condition**.
• Example:

counter++ could be implemented as:

$register_1 = \texttt{counter}$
$register_1 = register_1 + 1$
$\texttt{counter} = register_1$

counter— may be implemented as:

$register_2 = \texttt{counter}$
$register_2 = register_2 - 1$
$\texttt{counter} = register_2$

• Consider this execution interleaving with counter = 5 initially:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

• The value of counter may be either 4 or 6, where the correct result should be 5. This is an example for race condition.
• To prevent race conditions, concurrent-processes must be synchronized.

2.11 Critical-Section Problem

• **Critical-section** is a segment-of-code in which a process may be
→ changing common variables
→ updating a table or
→ writing a file.

• Each process has a critical-section in which the shared-data is accessed.

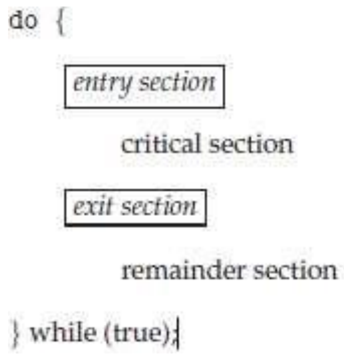
• General structure of a typical process has following (Figure 2.12):

1) Entry-section
➢ Requests permission to enter the critical-section.

2) Critical-section
➢ Mutually exclusive in time i.e. no other process can execute in its critical-section.

3) Exit-section
➢ Follows the critical-section.

4) Remainder-section



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Figure 2.12 General structure of a typical process

• Problem statement:
-Ensure that when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section‖ .

• A solution to the problem must satisfy the following 3 requirements:

1) Mutual Exclusion
➢ Only one process can be in its critical-section.

2) Progress
➢ Only processes that are not in their remainder-section can enter their critical-section, and the selection of a process cannot be postponed indefinitely.

3) Bounded Waiting
➢ There must be a bound on the number of times that other processes are allowed to enter their critical-sections after a process has made a request to enter its critical-section and before the request is granted.

• Two approaches used to handle critical-sections:

1) Preemptive Kernels
➢ Allows a process to be preempted while it is running in kernel-mode.

2) Non-preemptive Kernels
➢ Does not allow a process running in kernel-mode to be preempted.

2.12 Peterson's Solution

• This is a classic **software-based solution** to the critical-section problem.
• This is limited to 2 processes.
• The 2 processes alternate execution between
→ critical-sections and
→ remainder-sections.
• The 2 processes share 2 variables (Figure 2.13):

```
int turn;
boolean flag[2];
```

where turn = indicates whose turn it is to enter its critical-section.
(i.e., if turn==i, then process Pi is allowed to execute in its critical-section).
flag = used to indicate if a process is ready to enter its critical-section.
(i.e. if flag[i]=true, then Pi is ready to enter its critical-section).

```
do {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 2.13 The structure of process Pi in Peterson�+s solution

• To enter the critical-section,
→ firstly process Pi sets flag[i] to be true and
→ then sets turn to the value j.
• If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
• The final value of turn determines which of the 2 processes is allowed to enter its critical-section first.
• To prove that this solution is correct, we show that:
1) Mutual-exclusion is preserved.
2) The progress requirement is satisfied.
3) The bounded-waiting requirement is met.

2.13 Synchronization Hardware

2.13.1 Hardware based Solution for Critical-section Problem

• A lock is a simple tool used to solve the critical-section problem.
• Race conditions are prevented by following restriction (Figure 2.14).
—A process must acquire a lock before entering a critical-section.
The process releases the lock when it exits the critical-section‖ .

```
do {
     acquire lock
          critical section
     release lock
          remainder section
} while (TRUE);
```

Figure 2.14: Solution to the critical-section problem using locks

2.13.2 Hardware instructions for solving critical-section problem

• Modern systems provide special hardware instructions
→ to test & modify the content of a word atomically or
→ to swap the contents of 2 words atomically.
• Atomic-operation means an operation that completes in its entirety without interruption.

2.13.2.1 TestAndSet()

• This instruction is executed atomically (Figure 2.15).
• If two TestAndSet() are executed simultaneously (on different CPU), they will be executed sequentially in some arbitrary order.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 2.15 The definition of the test and set() instruction

2.13.2.2 TestAndSet with Mutual Exclusion

• If the machine supports the TestAndSet(), we can implement mutual-exclusion by declaring a boolean variable lock, initialized to false (Figure 2.16).

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 2.16 Mutual-exclusion implementation with test and set()

2.13.2.3 Swap()

- This instruction is executed atomically (Figure 2.17).
- If the machine supports the Swap(), then mutual-exclusion can be provided as follows:
    1) A global boolean variable lock is declared and is initialized to false.
    2) In addition, each process has a local Boolean variable key (Figure 2.18).

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

Figure 2.17 The definition of swap() instruction

```
do {
   key = TRUE;
   while  (key == TRUE)
      Swap(&lock,&key);

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 2.18 Mutual-exclusion implementation with the swap() instruction

2.13.2.4 Bounded waiting Mutual Exclusion with TestAndSet()

- Common data structures are

```
boolean waiting[n];
boolean lock;
```

- These data structures are initialized to false (Figure 2.19).

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

Figure 2.19 Bounded-waiting mutual-exclusion with TestandSet()

2.15 Semaphores

• A semaphore is a synchronization-tool.
• It used to control access to shared-variables so that only one process may at any point in time change the value of the shared-variable.
• A semaphore(S) is an integer-variable that is accessed only through 2 atomic-operations:
  1) wait() and
  2) signal().
• wait() is termed P ("to test").
  signal() is termed V ("to increment").

• **Definition of wait():**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

Definition of signal():

```
signal(S) {
    S++;
}
```

• When one process modifies the semaphore-value, no other process can simultaneously modify that same semaphore-value.
• Also, in the case of wait(S), following 2 operations must be executed without interruption:
  1) Testing of S(S<=0) and
  2) Modification of S (S--)

2.14.1 Semaphore Usage

Counting Semaphore

• The value of a semaphore can range over an unrestricted domain

Binary Semaphore

• The value of a semaphore can range only between 0 and 1.

• On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual-exclusion.

1) Solution for Critical-section Problem using Binary Semaphores

• Binary semaphores can be used to solve the critical-section problem for multiple processes.

• The ‗n‘ processes share a semaphore mutex initialized to 1 (Figure 2.20).

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

Figure 2.20 Mutual-exclusion implementation with semaphores

2) Use of Counting Semaphores

• Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.

• The semaphore is initialized to the number of resources available.

• Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).

• When a process releases a resource, it performs a signal() operation (incrementing the count).

• When the count for the semaphore goes to 0, all resources are being used.

• After that, processes that wish to use a resource will block until the count becomes greater than 0.

3) Solving Synchronization Problems

• Semaphores can also be used to solve synchronization problems.

• For example, consider 2 concurrently running-processes:

P1 with a statement S1 and
P2 with a statement S2.

• Suppose we require that S2 be executed only after S1 has completed.

• We can implement this scheme readily

→ by letting P1 and P2 share a common semaphore synch initialized to 0, and

→ by inserting the following statements in process P1

```
S1;
signal(synch);
```

and the following statements in process P2

```
wait(synch);
S2;
```

• Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

2.14.2 Semaphore Implementation

• Main disadvantage of semaphore: Busy waiting.

• **Busy waiting**: While a process is in its critical-section, any other process that tries to enter its critical-section must loop continuously in the entry code.

• Busy waiting wastes CPU cycles that some other process might be able to use productively.

• This type of semaphore is also called a **spinlock** (because the process "spins" while waiting for the lock).

• To overcome busy waiting, we can modify the definition of the wait() and signal() as follows:

1) When a process executes the wait() and finds that the semaphore-value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.

2) A process that is blocked (waiting on a semaphore S) should be restarted when some other process executes a signal(). The process is restarted by a wakeup().

• We assume 2 simple operations:

1) **block()** suspends the process that invokes it.

2) **wakeup(P)** resumes the execution of a blocked process P.

• We define a semaphore as follows:

```
typedef struct {
     int value;
     struct process *list;
} semaphore;
```

• **Definition of wait():**

```
wait(semaphore *S) {
          S->value--;
          if (S->value < 0) {
               add this process to S->list;
               block();
          }
}
```

Definition of signal():

```
signal(semaphore *S) {
          S->value++;
          if (S->value <= 0) {
               remove a process P from S->list;
               wakeup(P);
          }
}
```

• This (critical-section) problem can be solved in two ways:

1) In a **uni-processor** environment

¤ Inhibit interrupts when the wait and signal operations execute.

¤ Only current process executes, until interrupts are re-enabled & the scheduler regains control.

2) In a **multiprocessor** environment

¤ Inhibiting interrupts doesn't work.

¤ Use the hardware / software solutions described above.

2.14.3 Deadlocks & Starvation

• Deadlock occurs when 2 or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

• The event in question is the execution of a signal() operation.

• To illustrate this, consider 2 processes, Po and P1, each accessing 2 semaphores, S and Q. Let S and Q be initialized to 1.

```
    P0              P1

 wait(S);        wait(Q);
 wait(Q);        wait(S);
    .               .
    .               .
    .               .
 signal(S);      signal(Q);
 signal(Q);      signal(S);
```

• Suppose that Po executes wait(S) and then P1 executes wait(Q).

When Po executes wait(Q), it must wait until P1 executes signal(Q).

Similarly, when P1 executes wait(S), it must wait until Po executes signal(S).

Since these signal() operations cannot be executed, Po & P1 are deadlocked.

• Starvation (indefinite blocking) is another problem related to deadlocks.

• **Starvation** is a situation in which processes wait indefinitely within the semaphore.

• Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

2.15 Classic Problems of Synchronization

1) Bounded-Buffer Problem
2) Readers and Writers Problem
3) Dining-Philosophers Problem

2.15.1 The Bounded-Buffer Problem

• The bounded-buffer problem is related to the producer consumer problem.
• There is a pool of n buffers, each capable of holding one item.
• **Shared-data**

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

where,
¤ mutex provides mutual-exclusion for accesses to the buffer-pool.
¤ empty counts the number of empty buffers.
¤ full counts the number of full buffers.

• The symmetry between the producer and the consumer.
¤ The producer produces full buffers for the consumer.
¤ The consumer produces empty buffers for the producer.

• **Producer Process:**

```
do {
     . . .
   /* produce an item in next_produced */
     . . .
   wait(empty);
   wait(mutex);
     . . .
   /* add next_produced to the buffer */
     . . .
   signal(mutex);
   signal(full);
} while (true);
```

Consumer Process:

```
do {
   wait(full);
   wait(mutex);
     . . .
   /* remove an item from buffer to next_consumed */
     . . .
   signal(mutex);
   signal(empty);
     . . .
   /* consume the item in next_consumed */
     . . .
} while (true);
```

2.15.2 The Readers-Writers Problem

• A data set is shared among a number of concurrent processes.
• **Readers** are processes which want to only read the database (DB).
Writers are processes which want to update (i.e. to read & write) the DB.
• Problem:
➢ Obviously, if 2 readers can access the shared-DB simultaneously without any problems.
➢ However, if a writer & other process (either a reader or a writer) access the shared-DB simultaneously, problems may arise.
Solution:
➢ The writers must have exclusive access to the shared-DB while writing to the DB.
• **Shared-data**

```
semaphore mutex, wrt;
int readcount;
```

where,
¤ mutex is used to ensure mutual-exclusion when the variable readcount is updated.
¤ wrt is common to both reader and writer processes.
wrt is used as a mutual-exclusion semaphore for the writers.
wrt is also used by the first/last reader that enters/exits the critical-section.
¤ readcount counts no. of processes currently reading the object.

Initialization
mutex = 1, wrt = 1, readcount = 0

Writer Process:

```
do {
  wait(rw_mutex);
     . . .
  /* writing is performed */
     . . .
  signal(rw_mutex);
} while (true);
```

Reader Process:

```
do {
  wait(mutex);
  read_count++;
  if (read_count == 1)
     wait(rw_mutex);
  signal(mutex);
     . . .
  /* reading is performed */
     . . .
  wait(mutex);
  read_count--;
  if (read_count == 0)
     signal(rw_mutex);
  signal(mutex);
} while (true);
```

• The readers-writers problem and its solutions are used to provide **reader-writer locks** on some systems.
• The mode of lock needs to be specified:
1) read mode
➢ When a process wishes to read shared-data, it requests the lock in read mode.
2) write mode
➢ When a process wishes to modify shared-data, it requests the lock in write mode.
• Multiple processes are permitted to concurrently acquire a lock in read mode,
but only one process may acquire the lock for writing.
• These locks are most useful in the following situations:
1) In applications where it is easy to identify
→ which processes only read shared-data and
→ which threads only write shared-data.
2) In applications that have more readers than writers.

2.15.3 The Dining-Philosophers Problem

• Problem statement:
   - There are 5 philosophers with 5 chopsticks (semaphores).
   - A philosopher is either eating (with two chopsticks) or thinking.
   - The philosophers share a circular table (Figure 2.21).
   - The table has
      → a bowl of rice in the center and
      → 5 single chopsticks.
   - From time to time, a philosopher gets hungry and tries to pick up the 2 chopsticks that are closest to her.
   - A philosopher may pick up only one chopstick at a time.
   - Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
   - When hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
   - When she is finished eating, she puts down both of her chopsticks and starts thinking again.

• Problem objective:
  To allocate several resources among several processes in a deadlock-free & starvation-free manner.

• Solution:
   - Represent each chopstick with a semaphore (Figure 2.22).
   - A philosopher tries to grab a chopstick by executing a wait() on the semaphore.
   - The philosopher releases her chopsticks by executing the signal() on the semaphores.
   - This solution guarantees that no two neighbors are eating simultaneously.
   - **Shared-data**
      semaphore chopstick[5];

   Initialization
      chopstick[5]={1,1,1,1,1}.

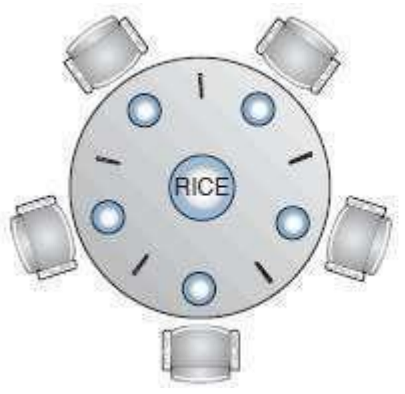
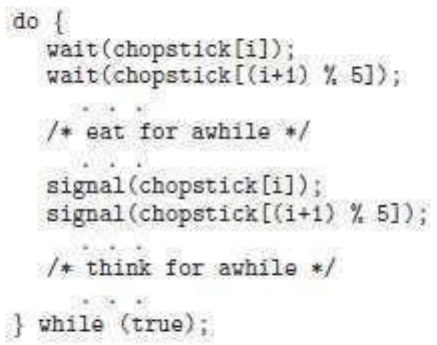


Figure 2.21 Situation of dining philosophers



```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Figure 2.22 The structure of philosopher

• Disadvantage:
   1) Deadlock may occur if all 5 philosophers become hungry simultaneously and grab their left chopstick. When each philosopher tries to grab her right chopstick, she will be delayed forever.

• Three possible remedies to the deadlock problem:
  1) Allow **at most 4** philosophers to be sitting simultaneously at the table.
  2) Allow a philosopher to pick up her chopsticks **only if both chopsticks are available**.
  3) Use an **asymmetric solution**; i.e. an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

2.16 Monitors

- **Monitor** is a high-level synchronization construct.
- It provides a convenient and effective mechanism for process synchronization.

Need for Monitors

- When programmers use semaphores incorrectly, following types of errors may occur:

    1) Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore ―mutex‖ are executed, resulting in the following execution:

```
signal(mutex);
   ...
   critical section
   ...
wait(mutex);
```

    ➢ In this situation, several processes may be executing in their critical-sections simultaneously, violating the mutual-exclusion requirement.

    2) Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
   ...
   critical section
   ...
wait(mutex);
```

    ➢ In this case, a deadlock will occur.

    3) Suppose that a process omits the wait(mutex), or the signal(mutex), or both.

    ➢ In this case, either mutual-exclusion is violated or a deadlock will occur.

2.16.1 Monitors Usage

• A **monitor type** presents a set of programmer-defined operations that are provided to ensure mutual-exclusion within the monitor.

• It also contains (Figure 2.23):

→ declaration of variables

→ bodies of procedures(or functions).

• A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal-parameters.

Similarly, the local-variables of a monitor can be accessed by only the local-procedures.

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }
            .
            .
            .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Figure 2.23 Syntax of a monitor

• Only one process at a time is active within the monitor (Figure 2.24).

• To allow a process to wait within the monitor, a condition variable must be declared, as

```
condition x, y;
```

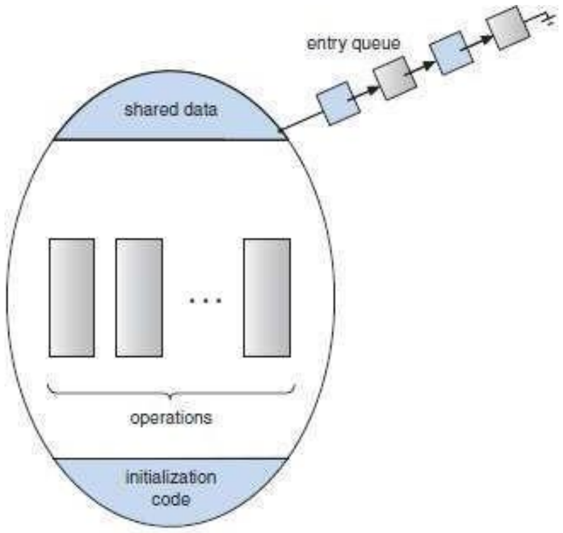
• Condition variable can only be used with the following 2 operations (Figure 2.25):

1) x.signal()

➢ This operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

2) x.wait()

➢ The process invoking this operation is suspended until another process invokes x.signal().



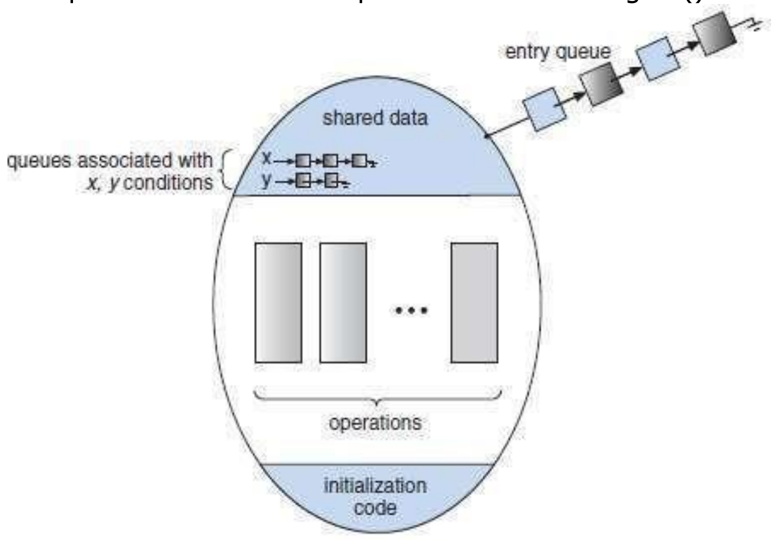


Figure 2.24 Schematic view of a monitor Figure 2.25 Monitor with condition variables

• Suppose when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x.
• Both processes can conceptually continue with their execution. Two possibilities exist:

1) Signal and wait

➢ P either waits until Q leaves the monitor or waits for another condition.

2) Signal and continue

➢ Q either waits until P leaves the monitor or waits for another condition.

2.16.2 Dining-Philosophers Solution Using Monitors

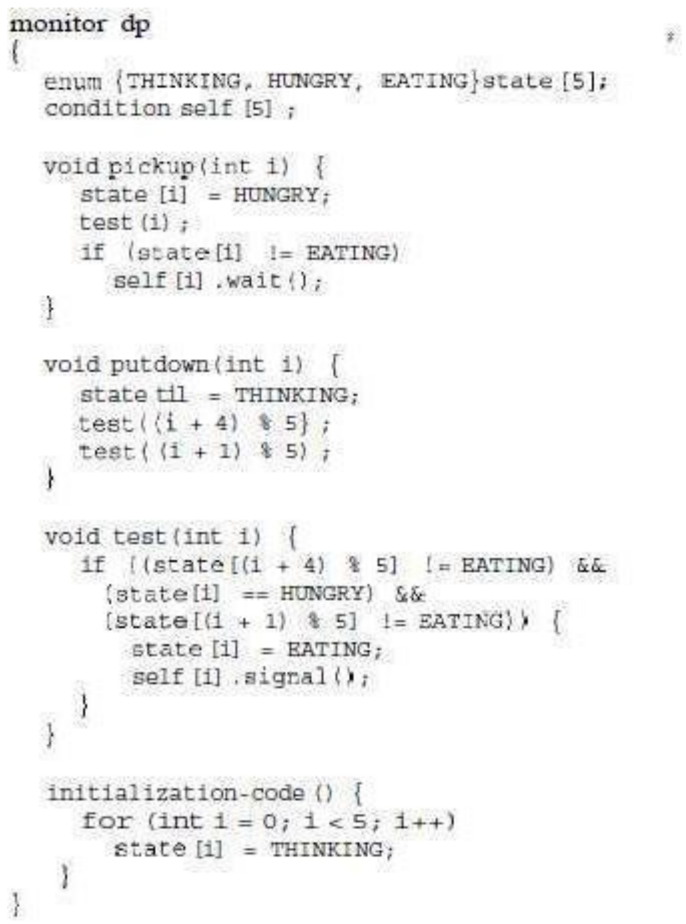
• The restriction is

➢ A philosopher may pick up her chopsticks only if both of them are available.

• Description of the solution:

1) The distribution of the chopsticks is controlled by the monitor dp (Figure 2.26).
2) Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process.
3) After the successful completion of the operation, the philosopher may eat.
4) Following this, the philosopher invokes the putdown() operation.
5) Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```



```
monitor dp
{
   enum {THINKING, HUNGRY, EATING}state [5];
   condition self [5] ;

   void pickup(int i)  {
      state [i]  = HUNGRY;
      test (i) ;
      if  (state[i]  != EATING)
        self [i] .wait();
   }

   void putdown(int i)  {
      state til  = THINKING;
      test((i + 4)  % 5} ;
      test( (i + 1)  % 5) ;
   }

   void test(int i)  {
      if  ((state[(i + 4)  % 5]  != EATING)  &&
        (state[i]  == HUNGRY)  &&
        (state[(i + 1)  % 5]  != EATING)) {
          state [i]  = EATING;
          self [i] .signal();
      }
   }

   initialization-code () {
      for (int i = 0; i < 5; i++)
        state [i]  = THINKING;
    }
}
```

Figure 2.26 A monitor solution to the dining-philosopher problem

2.16.3 Implementing a Monitor using Semaphores

• A process
  → must execute wait(mutex) before entering the monitor and
  → must execute signal(mutex) after leaving the monitor.

• Variables used:
```
semaphore mutex;        // (initially = 1)
semaphore next;         // (initially = 0)
int next-count = 0;
```
where
  ¤ mutex is provided for each monitor.
  ¤ next is used a signaling process to wait until the resumed process either leaves or waits
  ¤ next-count is used to count the number of processes suspended

• Each external procedure F is replaced by
```
wait(mutex);
   ...
  body of F
   ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

• Mutual-exclusion within a monitor is ensured.

• How condition variables are implemented ?
  For each condition variable x, we have:
```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

• **Definition of x.wait()**
```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

Definition of x.signal()
```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

2.16.4 Resuming Processes within a Monitor

• Problem:

If several processes are suspended, then how to determine which of the suspended processes should be resumed next?

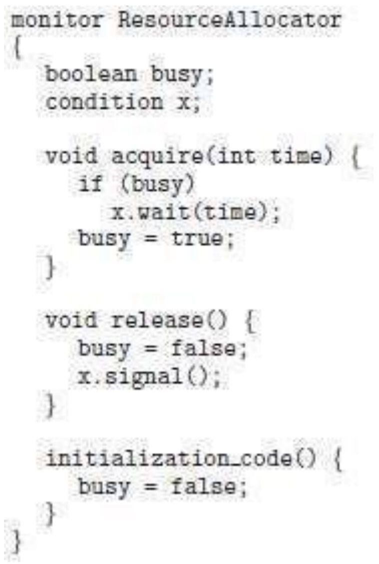
Solution-1: Use an FCFS ordering i.e. the process that has been waiting the longest is resumed first.

Solution-2: Use conditional–wait construct i.e. x.wait(c)

¤ c is a integer expression evaluated when the wait operation is executed (Figure 2.27).

¤ Value of c (a priority number) is then stored with the name of the process that is suspended.

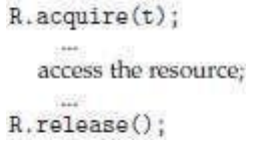
¤ When x.signal is executed, process with smallest associated priority number is resumed next.



```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

Figure 2.27 A monitor to allocate a single resource

• ResourceAllocator monitor controls the allocation of a single resource among competing processes.
• Each process, when requesting an allocation of the resource, specifies the maximum time it plans to use the resource.
• The monitor allocates the resource to the process that has the shortest time-allocation request.
• A process that needs to access the resource in question must observe the following sequence:



```
R.acquire(t);
   ...
   access the resource;
   ...
R.release();
```

where R is an instance of type ResourceAllocator.

• Following problems can occur:

➢ A process might access a resource without first gaining access permission to the resource.
➢ A process might never release a resource once it has been granted access to the resource.
➢ A process might attempt to release a resource that it never requested.
➢ A process might request the same resource twice.