COURSE NAME: OPERATING SYSTEMS

COURSE CODE:

SEMESTER: 6

MODULE: 2

NUMBER OF HOURS: 10

CONTENTS:

- **Multi-threaded Programming**:
  - Multithreading models
  - Thread Libraries
  - Threading issues.
- **Process Scheduling**:
  - Basic concepts
  - Scheduling Criteria;
  - Scheduling Algorithms;
  - Multiple-processor scheduling;
  - Thread scheduling.
- **Process Synchronization**:
  - Synchronization: The critical section problem;
  - Peterson's solution;
  - Synchronization hardware;
  - Semaphores;
  - Classical problems of synchronization;
  - Monitors.
- **Question Bank**:

WEB RESOURCES:

https://www.geeksforgeeks.org/operating-systems/

https://www.tutorialspoint.com/operating_system/index.htm

MODULE 2

MULTITHREADED 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 **multithreaded process**

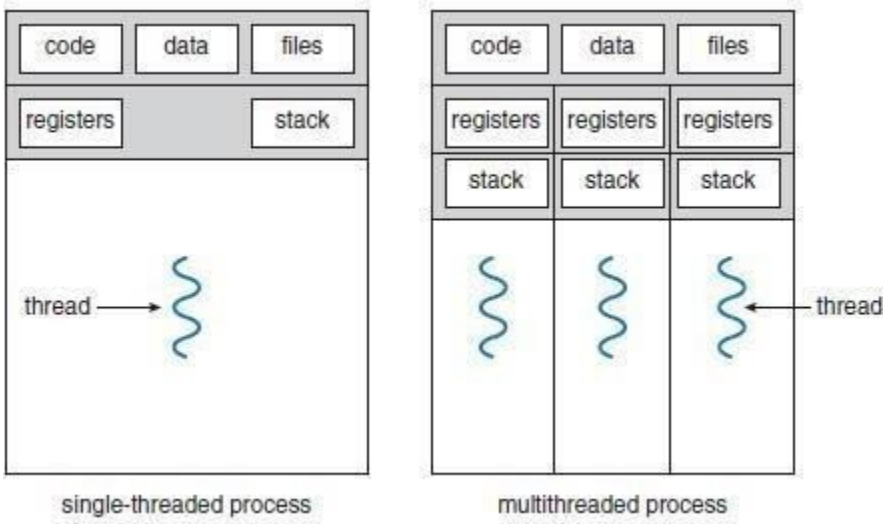
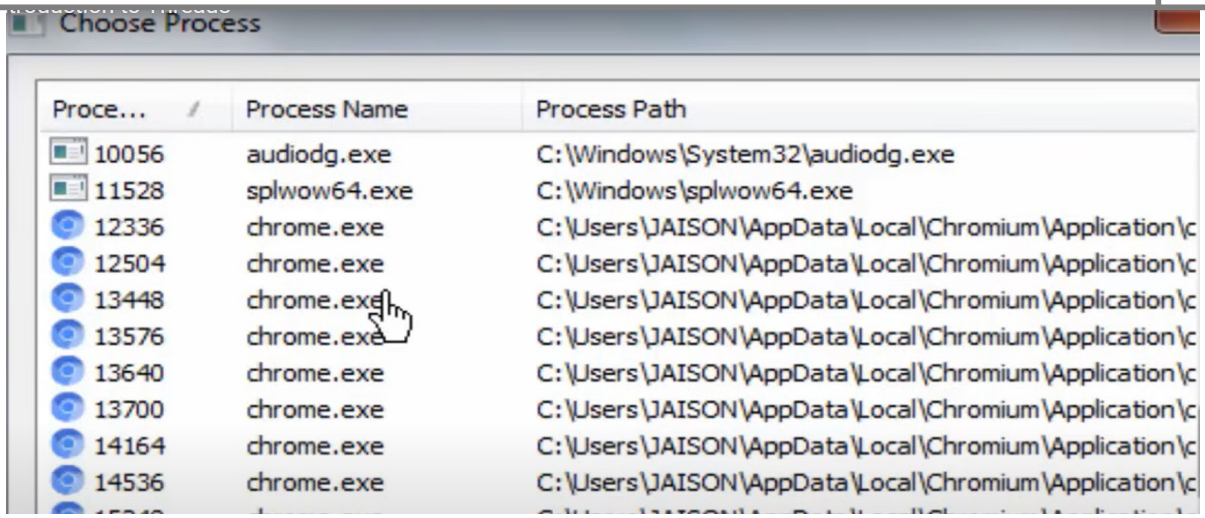
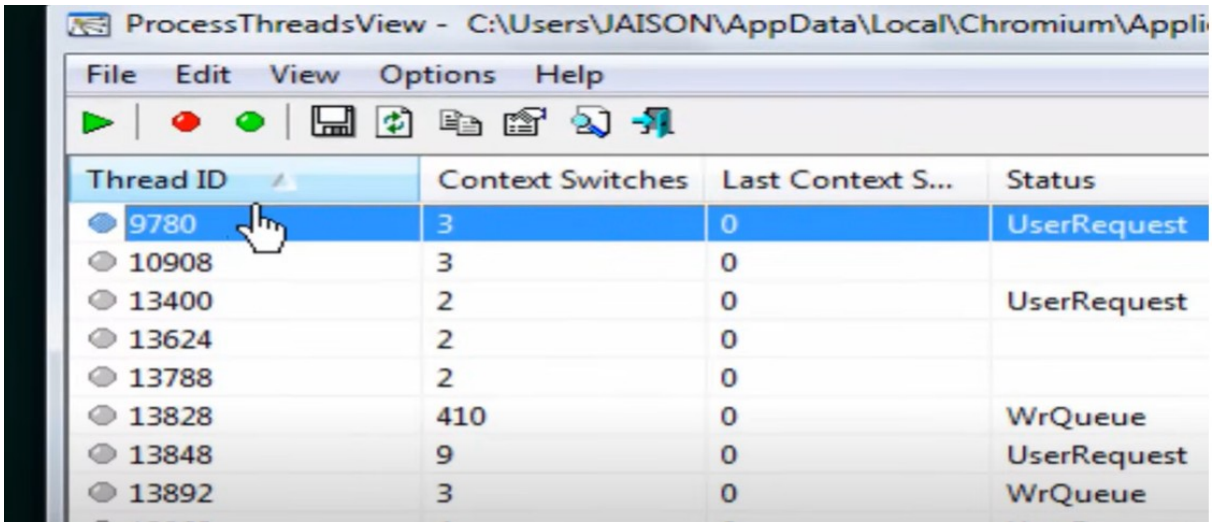


Fig: Single-threaded and multithreaded processes





Motivation for Multithreaded Programming

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
- 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 requests. This allows the server to service several concurrent requests.

3. Most OS kernels are multithreaded;

- Several threads operate in the kernel, each performing a specific task, such as managing devices or interrupt handling.

Benefits of Multithreaded Programming

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

- *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.
- *Economy* Allocating memory and resources for process creation is costly. Thus, it is more economical to create and context-switch threads.
- *Utilization of Multiprocessor Architectures* In a multiprocessor architecture, threads may run in parallel on different processors, increasing parallelism.

MULTITHREADING 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 a relationship between user threads & kernel threads:
  1. Many-to-one model
  2. One-to-one model and
  3. Many-to-many model.

Many-to-One Model

- Many user-level threads are mapped to one kernel thread.

Advantages:

- Thread management is done by the thread library in user space, so it is efficient.

Disadvantages:

- The entire process will block if a thread makes a blocking system call.
- Multiple threads are unable to run in parallel on multiprocessors.

- For example:
  - Solaris green threads
  - GNU portable threads.

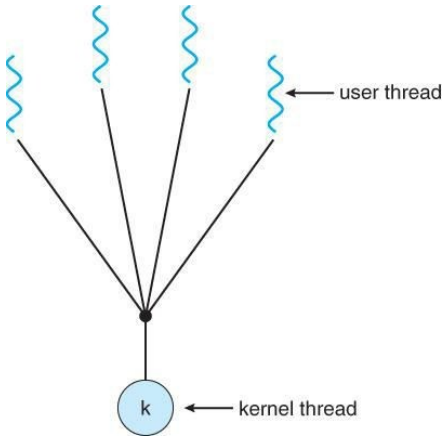


Fig: Many-to-one model

One-to-One Model

- Each user thread is mapped to a kernel thread.

Advantages:

- It provides more concurrency by allowing another thread to run when a thread makes a blocking system call.
- Multiple threads can run in parallel on multiprocessors.

Disadvantage:

- Creating a user thread requires creating the corresponding kernel thread.

- For example:
  - Windows NT/XP/2000, Linux

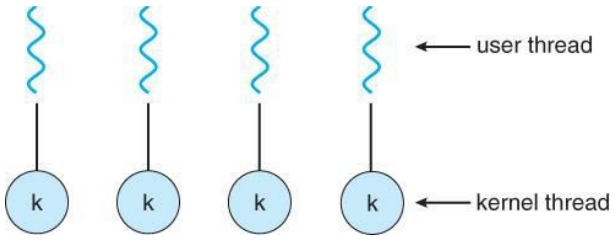


Fig: one-to-one model

Many-to-Many Model

- Many user-level threads are multiplexed to a smaller number of kernel threads.

Advantages:

- Developers can create as many user threads as necessary
- The kernel threads can run in parallel on a multiprocessor.
- 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
- Similar to M:N, except that it allows a user thread to be bound to kernelthread.
- for example:
  - HP-UX
  - Tru64 UNIX

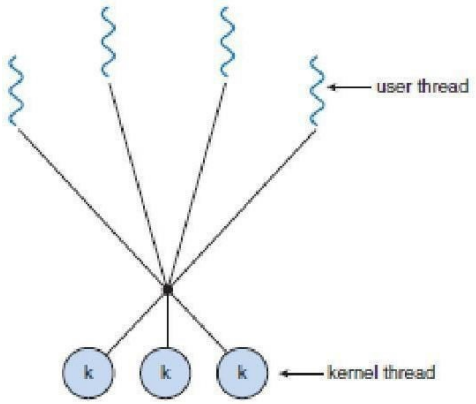


Fig: Many-to-many model

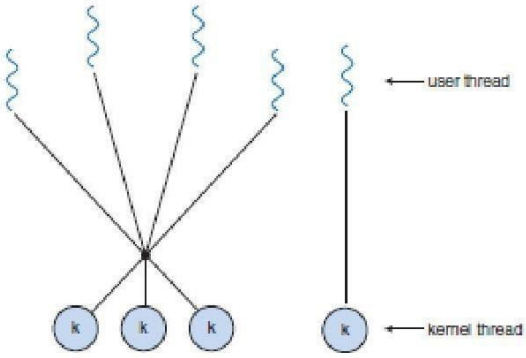


Fig: Two-level model

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. SecondApproach

  Provides a library entirely in user space with no kernel support. All code and data structures for the library exist in the user space.

Three main thread libraries:

1. POSIXP threads
2. Win32 and
3. Java.

Pthreads

- This is a POSIX standard API for thread creation andsynchronization.
- This is a specification for thread-behavior, not an implementation.
- OS designers may implement the specification in any way theywish.
- Commonly used in: UNIX andSolaris.

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

```
/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Win32 threads

- Implements the one-to-onemapping
- Each threadcontains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Java Threads

- Threads are the basic model of program-executionin
  - 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 ofcontrol.
- 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();
}
```

THREADING ISSUES

fork() and exec() System-calls

- fork() is used to create a separate, duplicateprocess.
- 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 forkO.
- If a thread invokes the exec(), the program specified in the parameter to exec() will replace the entire process including allthreads.

Thread Cancellation

- This is the task of terminating a thread before it hascompleted.
- Target thread is the thread that is to be cancelled
- Thread cancellation occurs in two differentcases:
    1. ***Asynchronous cancellation***: One thread immediately terminates the targetthread.
    2. ***Deferred cancellation***: The target thread periodically checks whether it should be terminated.

Signal Handling

- In UNIX, a signal is used to notify a process that a particular event hasoccurred.
- All signals follow thispattern:
    1. A signal is generated by the occurrence of a certainevent.
    2. A generated signal is delivered to aprocess.
    3. Once delivered, the signal must behandled.
- A signal handler is used to processsignals.
- A signal may be received either synchronously or asynchronously, depending on thesource.

    1. Synchronoussignals
    - Delivered to the same process that performed the operation causing the signal.
    - E.g. illegal memory access and division by 0.

    2. Asynchronoussignals
    - 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 possiblehandlers:

  1. A Default SignalHandler

  - Run by the kernel when handling the signal.

  2. A User-defined SignalHandler

  - Overrides the default signal handler.
- In ***single-threaded programs***, delivering signals is simple (since signals are always delivered to a process).
- 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 process
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.

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:
  - Servicing a request with an existing thread is usually faster than waiting to create a thread.
  - The pool limits the no. of threads that exist at any one point.
- No. of threads in the pool can be based on actors such as
  - no. of CPUs
  - amount of memory and
  - expected no. of concurrent client-requests.

THREAD SPECIFIC DATA

- Threads belonging to a process share the data of the process.
- this sharing of data provides one of the benefits of multithreadedprogramming.
- In some circumstances, each thread might need its own copy of certain data. We will call such data **thread-specific data**.
- For example, in a transaction-processing system, we might service each transaction in a separatethread.

- Furthermore, each transaction may be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specificdata.

SCHEDULER ACTIVATIONS

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to theapplication.
- Scheduler activations provide **upcalls**a communication mechanism from the kernel to the threadlibrary
- This communication allows an application to maintain the correct number kernel threads
- One scheme for communication between the user-thread library and the kernel is known as **scheduler activation.**

PROCESS SCHEDULING

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 ofmultiprogramming:
  - To have some process running at all times, in order to maximize CPU utilization.

CPU-I/0 Burst Cycle

- Process execution consists of a cycleof
  - CPU execution and
  - I/O wait
- 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 terminateexecution.
- An I/O-bound program typically has many short CPUbursts.
- A CPU-bound program might have a few long CPU bursts.

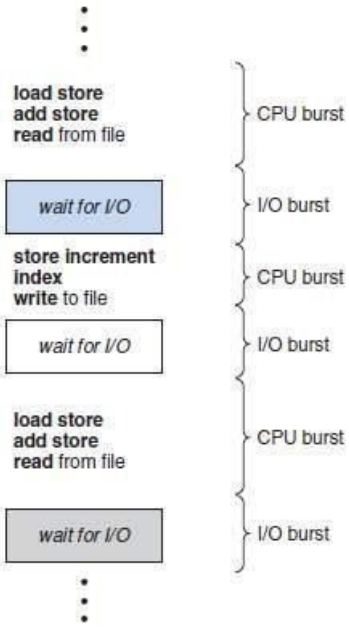


Fig Alternating sequence of CPU and I/O bursts

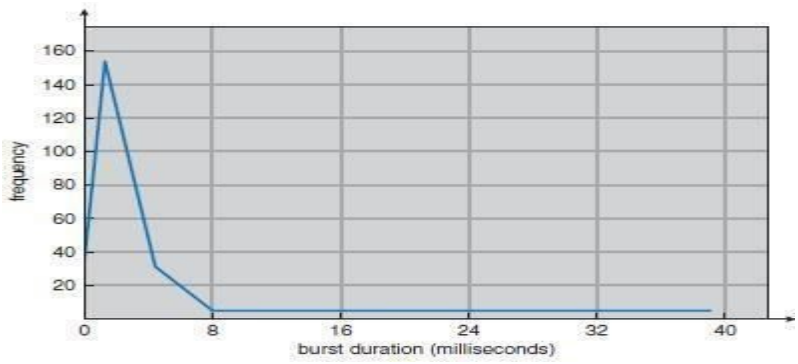


Fig: Histogram of CPU-burst durations

CPU Scheduler

- Thisscheduler
  - 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 andlist.
- The records in the queues are generally process control blocks (PCBs) of theprocesses.

CPU Scheduling

- Four situations under which CPU scheduling decisions takeplace:
  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 prioritizedcomputation.
- Processes that are runnable may be temporarilysuspended
- Disadvantages:
  1. Incurs a cost associated with access toshared-data.
  2. Affects the design of the OSkernel.

Dispatcher

- It gives control of the CPU to the process selected by the short-termscheduler.
- The functioninvolves:
    1. Switchingcontext
    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 dispatcherto
    - stop one process and
    - start another running.

SCHEDULING CRITERIA:

In choosing which algorithm to use in a particular situation, depends upon the properties of the various algorithms.Many criteria have been suggested for comparing CPU-scheduling algorithms. The criteria include the following:

1. **CPU utilization:** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
2. **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

3. **Turnaround time.** This is the important criterion which tells how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. 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/0.
4. **Waiting time**: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0, it affects only the amount of time that a process spends waiting in the ready queue.Waiting time is the sum of the periods spent waiting in the ready queue.
5. **Response time:**In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

SCHEDULING ALGORITHMS

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

FCFS Scheduling

- The process that requests the CPU first is allocated the CPUfirst.
- The implementation is easily done using a FIFOqueue.
- Procedure:
  1. When a process enters the ready-queue, its PCB is linked onto the tail of thequeue.
  2. When the CPU is free, the CPU is allocated to the process at the queue'shead.
  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 theCPU.
  2. Non-preemptive (a process keeps the CPU until it releasesit).
  3. Not good for time-sharingsystems.
  4. The average waiting time is generally notminimal.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Example: Suppose that the processes arrive in the order P1, P2,P3.
- The Gantt Chart for the schedule is asfollows:

- Waiting time for *P1* = 0; *P2* = 24; *P3* =27

  Average waiting time: (0 + 24 + 27)/3 = 17ms

- Suppose that the processes arrive in the order P2, P3,P1.

• The Gantt chart for the schedule is asfollows:

• Waiting time for P1 = 6;P2 = 0; P3 =3

• Average waiting time: (6 + 0 + 3)/3 = 3ms

SJF Scheduling

• The CPU is assigned to the process that has the smallest next CPUburst.

• If two processes have the same length CPU burst, FCFS scheduling is used to break thetie.

• 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 CPUburst

• 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 preemptiveSJF***

   The current process is allowed to finish its CPU burst.

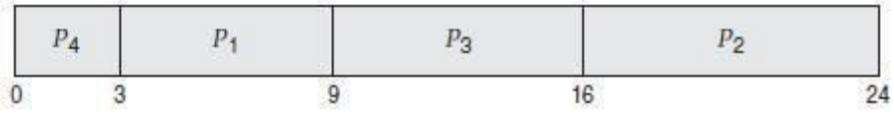
2. ***PreemptiveSJF***

   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 inmilliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

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



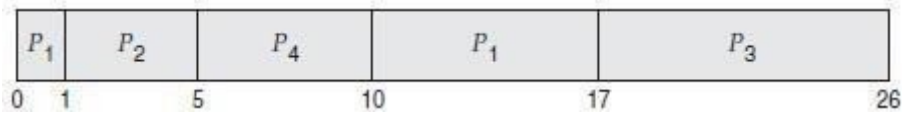
- Waiting time for P1 = 3; P2 = 16; P3 = 9; 4=0 Average waiting time: (3 + 16 + 9 + 0)/4= 7

preemptive SJF/SRTF: Consider the following set of processes, with the length

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

of the CPU- burst time given inmilliseconds.

- For preemptive SJF, the Gantt Chart is asfollows:



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

Priority Scheduling

- A priority is associated with eachprocess.
- The CPU is allocated to the process with the highestpriority.
- Equal-priority processes are scheduled in FCFSorder.
- Priorities can be defined either internally orexternally.
  1. ***Internally-defined*** priorities.
     - Use some measurable quantity to compute the priority of a process.
     - For example: time limits, memory requirements, no. f 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 non-preemptive. 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***

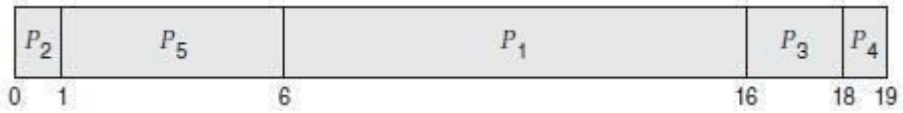
  The new process is put at the head of the ready-queue

- Advantage:
  - Higher priority processes can be executed first.
- Disadvantage:
  - 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 inmilliseconds.

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 asfollows:



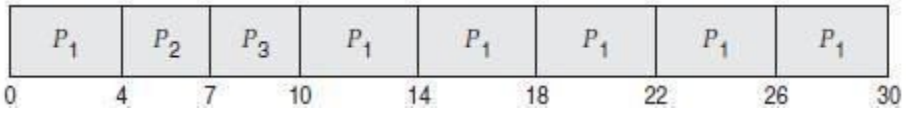
• The average waiting time is 8.2milliseconds.

Round Robin Scheduling

• Designed especially for timesharingsystems.
• It is similar to FCFS scheduling, but with preemption.
• A small unit of time is called a ***time quantum****(or timeslice).*
• Time quantum is ranges from 10 to 100ms.
• The ready-queue is treated as a **circularqueue**.
• The CPUscheduler
  - 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
• CPUscheduler
  1. Picks the first process from theready-queue.
  2. Sets a timer to interrupt after 1 time quantumand
  3. Dispatches theprocess.
• One of two things will thenhappen.
  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:
  - Higher average turnaround than SJF.
• Disadvantage:
  - 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 inmilliseconds.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart for the schedule is asfollows:



- The average waiting time is 17/3 = 5.66milliseconds.

- *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.
  1. If time quantum=very large, RR policy is the same as the FCFSpolicy.
  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.

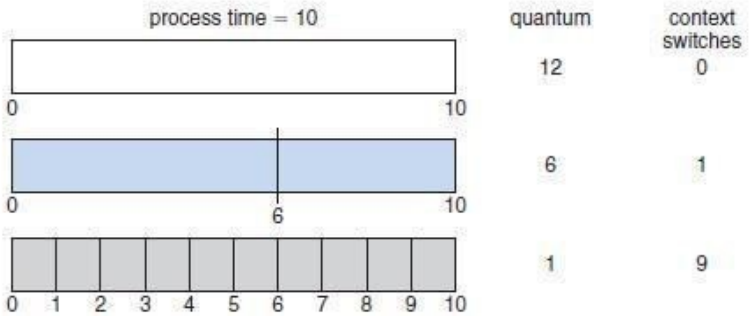


Fig: How a smaller time quantum increases context switches

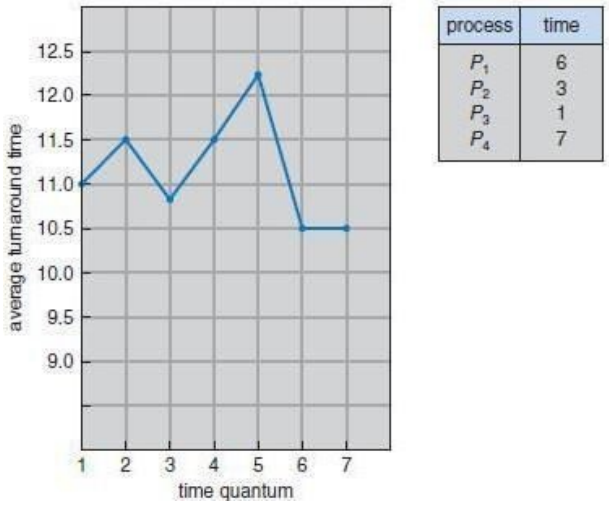


Fig: How turnaround time varies with the time quantum

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 (Figure2.19).
- 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.

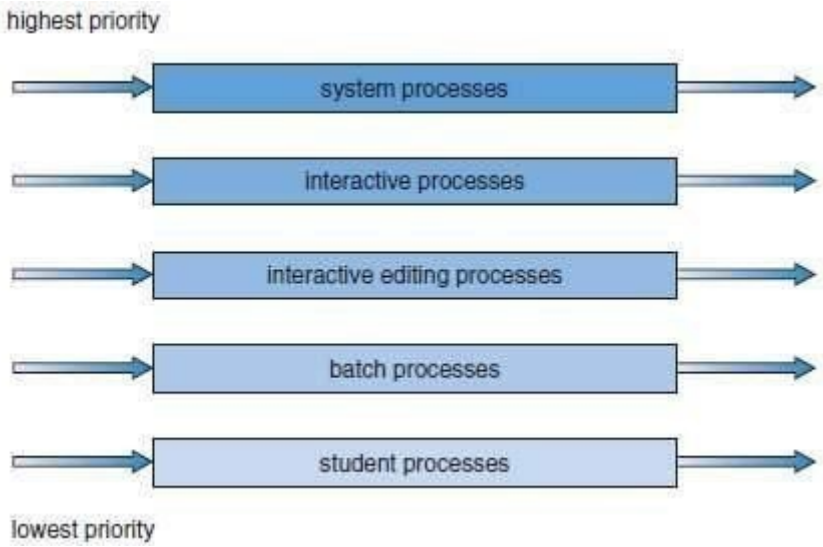


Fig Multilevel queue scheduling

- 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

Multilevel Feedback Queue Scheduling

- A process may move between queues
- The basic idea: Separate processes according to the features of their CPU bursts. Forexample
  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.

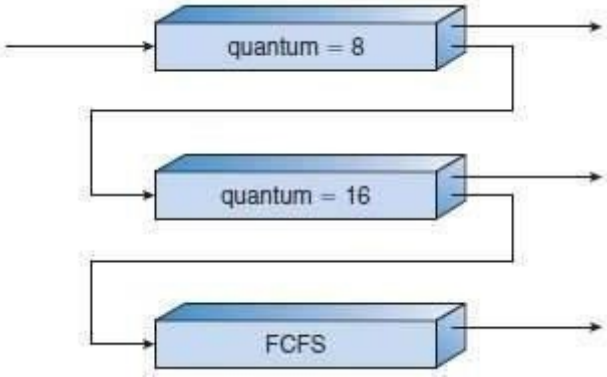


Figure 2.20 Multilevel feedback queues

In general, a multilevel feedback queue scheduler is defined by the followingparameters:

1. The number ofqueues.
2. The scheduling algorithm for eachqueue.
3. The method used to determine when to upgrade a process to a higher priorityqueue.
4. The method used to determine when to demote a process to a lower priorityqueue.
5. The method used to determine which queue a process will enter when that process needs service

MULTIPLE PROCESSOR SCHEDULING

- If multiple CPUs are available, the scheduling problem becomes morecomplex.
- Twoapproaches:

AsymmetricMultiprocessing The basic idea is:

- A master server is a single processor responsible for all scheduling decisions, I/O processing and other systemactivities.
- The other processors execute only user code.
- Advantage: This is simple because only one processor accesses the system data structures, reducing the need for data sharing.

Symmetric Multiprocessing

The basic idea is:

- Each processor is self-scheduling.
- To do scheduling, the scheduler for eachprocessor
- Examines the ready-queue and
- 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.

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. ***SoftAffinity***
     - 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

Load Balancing

- This attempts to keep the workload evenly distributed across all processors in an SMPsystem.
- Twoapproaches:
  1. ***PushMigration***

     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. ***PullMigration***

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

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, notsoftware.

THREAD SCHEDULING

- On OSs, it is kernel-level threads but not processes that are being scheduled by theOS.
- User-level threads are managed by a thread library, and the kernel is unaware ofthem.
- To run on a CPU, user-level threads must be mapped to an associated kernel-levelthread.

Contention Scope

- Twoapproaches:

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 sameprocess.

2. System-Contentionscope

- The process of deciding which kernel thread to schedule on theCPU.
- Competition for the CPU takes place among all threads in thesystem.
- Systems using the one-to-one model schedule threads using onlySCS.

Pthread Scheduling

- Pthread API that allows specifying either PCS or SCS during threadcreation.
- Pthreads identifies the following contention scopevalues:
  1. PTHREAD_SCOPEJPROCESS schedules threads using PCSscheduling.
  2. PTHREAD-SCOPE_SYSTEM schedules threads using SCSscheduling.
- Pthread IPC provides following two functions for getting and setting the contention scopepolicy:
  1. pthread_attr_setscope(pthread_attr_t *attr, intscope)
  2. pthread_attr_getscope(pthread_attr_t *attr, int*scop)

PROCESS SYNCHRONIZATION

- A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- Concurrent access to shared data may result in data inconsistency. To maintain data consistency, various mechanisms is required to ensure the orderly execution of cooperating processes that share a logical address space.

Producer- Consumer Problem

- A Producer process produces information that is consumed by consumer process.
- To allow producer and consumer process to run concurrently, A Bounded Buffer can be used where the items are filled in a buffer by the producer and emptied by the consumer.
- The original solution allowed at most **BUFFER_SIZE - 1** item in the buffer at the same time. To overcome this deficiency, an integer variable ***counter***, initialized to 0 isadded.
- ***counter*** is incremented every time when a new item is added to the buffer and is decremented every time when one item removed from thebuffer.

The code for the ***producer process*** can be modified as follows:

```
while (true) {

        /* produce an item and put in nextProduced*/ while
             (counter == BUFFER_SIZE)
                        ; // do nothing
                    buffer [in] = nextProduced;
                    in = (in + 1) % BUFFER_SIZE;
                    counter++;
}
```

The code for the ***consumer process*** can be modified as follows:

```
while (true){
                while (counter ==0)
                        ; // donothing
                    nextConsumed =buffer[out];
                     out = (out + 1) % BUFFER_SIZE;
                   counter--;
                      /* consume the item in nextConsumed */
        }
```

- **Race Condition**
  When the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.
- ***Illustration:***
  Suppose that the value of the variable ***counter*** is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently. The value of the variable counter may be 4, 5, or 6 but the only correct result is counter == 5, which is generated correctly if the producer and consumer execute separately.

The value of ***counter*** may be incorrect as shown below:

The statement counter++ could be implemented as

```
register1= counter
register1 = register1 + 1
counter =register1
```

The statement counter-- could be implemented as

```
register2 =counter
register2 = register2 – 1
count = register2
```

The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order. One such interleaving is

Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1=counter	{register1 = 5}
S1: producer execute register1 = register1+1	{register1 = 6}
S2: consumer execute register2=counter	{register2 = 5}
S3: consumer execute register2 = register2-1	{register2 = 4}
S4: producer execute counter=register1	{count =6}
S5: consumer execute counter=register2	{count =4}

- ***Note:*** It is arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter==6".
- *Definition* ***Race Condition***: A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **RaceCondition.**
- To guard against the race condition, ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, ***the processes are synchronized*** in some way.

The Critical Section Problems

- Consider a system consisting of n processes {Po, P1 , ... ,Pn-1}.
- Each process has a segment of code, called a **critical section** in which the process may be changing common variables, updating a table, writing a file, and soon
- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the sametime.
- The critical-section problem is to design a protocol that the processes can use to cooperate.

The general structure of a typical process Pi is shown in below figure.

- Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**. The remaining code is the **reminder section**.

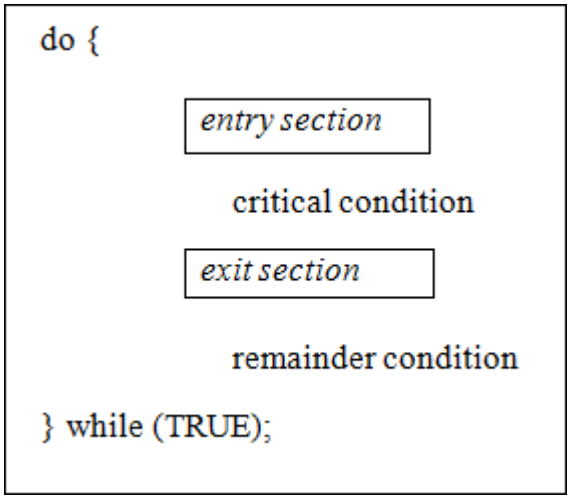


Figure: General structure of a typical process Pi

A solution to the critical-section problem must satisfy the following **three requirements**:

1. **Mutual exclusion:**If process Pi is executing in its critical section, then no other processes can be executing in their criticalsections.

2. **Progress:**If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponedindefinitely.

3. **Bounded waiting:**There exists a bound, or limit, 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 that request isgranted.

PETERSON'S SOLUTION

- This is a classic software-based solution to the critical-section problem. There are no guarantees that Peterson's solution will work correctly on modern computer architectures
- Peterson's solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_o and P_1 or Pi and P_j where j = 1-i

Peterson's solution requires the two processes to share two data items:

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

- **turn:** The variable turn indicates whose turn it is to enter its critical section. **Ex:** if turn == i, then process P_i is allowed to execute in its criticalsection
- **flag:** The flag array is used to indicate if a process is ready to enter its critical section. **Ex:** if flag [i] is true, this value indicates that P_i is ready to enter its critical section.

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j)
                ; // do nothing
         critical section
        flag[i] = FALSE;

                remainder section

    } while (TRUE);
```

Figure: The structure of process P_i in Peterson's solution

- To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can doso.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last, the other will occur but will be over written immediately.

- The eventual value of turn determines which of the two processes is allowed to enter its critical sectionfirst

To prove that solution is correct, then we need to show that

1. Mutual exclusion ispreserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

1. To prove Mutual exclusion

- Each pi enters its critical section only if either flag [j] == false or turn ==i.
- If both processes can be executing in their critical sections at the same time, then flag [0] == flag [1]==true.
- These two observations imply that Pi and Pj could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes (Pj) must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn==j").
- However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section, as a result, mutual exclusion ispreserved.

2. To prove Progress and Bounded-waiting

- A process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] ==true and turn=== j; this loop is the only onepossible.
- If Pj is not ready to enter the critical section, then flag [j] ==false, and Pi can enter its criticalsection.
- If Pj has set flag [j] = true and is also executing in its while statement, then either turn === i or turn ===j.
    - If turn == i, then Pi will enter the criticalsection.
    - If turn== j, then Pj will enter the criticalsection.
- However, once Pj exits its critical section, it will reset flag [j] = false, allowing Pi to enter its criticalsection.
- If Pj resets flag [j] to true, it must also set turn to i.
- Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (boundedwaiting).

SYNCHRONIZATIONHARDWARE

- The solution to the critical-section problem requires a simple tool-a**lock.**
- Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section and it releases the lock when it exits the critical section

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

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

- The critical-section problem could be solved simply in a uniprocessor environment if interrupts are prevented from occurring while a shared variable was being modified. In this manner, the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the sharedvariable.
- But this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

TestAndSet ( ) and Swap( ) instructions

- Many modern computer systems provide special hardware instructions that allowto **test** and **modify** the content of a word or to **swap** the contents of two words **atomically**, that is, as one uninterruptibleunit.
- Special instructions such as TestAndSet () and Swap() instructions are used to solve the critical-sectionproblem
- The TestAndSet () instruction can be defined as shown in Figure. The important characteristic of this instruction is that it is executedatomically.

Definition:

```
booleanTestAndSet (boolean *target)
 {
    booleanrv = *target;
    *target = TRUE;
    return rv:
 }
```

Figure: The definition of the TestAndSet () instruction.

- Thus, if two TestAndSet () instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then implementation of mutual exclusion can be done by declaring a Boolean variable lock, initialized to false.

```
do {
        while ( TestAndSet (&lock ))
              ;   // do nothing
            //    critical section
        lock =FALSE;
              //      remaindersection
    } while (TRUE);
```

Figure: Mutual-exclusion implementation with TestAndSet ()

- The Swap() instruction, operates on the contents of two words, it is defined as shown below

Definition:

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

Figure: The definition of the Swap ( ) instruction

- Swap() it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.
- A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process Pi is shown in below

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

              //    critical section
         lock =FALSE;
              //      remaindersection
    } while (TRUE);
```

Figure: Mutual-exclusion implementation with the Swap() instruction

- These algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded- waiting requirement.
- Below algorithm using the TestAndSet () instruction that satisfies all the critical- section requirements. The common data structures are

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

These data structures are initialized to false.

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&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:Bounded-waiting mutual exclusion with TestAndSet ()

1. To prove the mutual exclusionrequirement

- Note that process Pi can enter its critical section only if either waiting [i] == false or key ==false.
- The value of key can become false only if the TestAndSet( ) isexecuted.
- The first process to execute the TestAndSet( ) will find key== false; all others must wait.
- The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

2. To prove the progressrequirement

Note that, the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

3. To prove the bounded-waitingrequirement

- Note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n 1, 0, ... , i 1).
- It designates the first process in this ordering that is in the entry section (waiting[j] ==true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

SEMAPHORE

- A semaphore is a synchronization tool is used solve various synchronization problem and can be implementedefficiently.
- Semaphore do not require busywaiting.
- A semaphore S is an integer variable that, is accessed only through two standard atomic operations: wait () and signal (). The wait () operation was originally termed P and signal() was calledV.

Definition of wait ():

```
wait (S) {
        while S <= 0
        ; // no-op
    S--;
```

Definition of signal ():

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

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Binary semaphore

- The value of a binary semaphore can range only between 0 and1.
- Binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion. Binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, mutex, initialized to1

Each process Pi is organized as shown in below figure

```
do {
        wait (mutex);
                // Critical Section
          signal (mutex);
                // remainder section
} while (TRUE);
```

Figure: Mutual-exclusion implementation with semaphores

Counting semaphore

- The value of a counting semaphore can range over an unrestricteddomain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number ofinstances.
- 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. When a process releases a resource, it performs a signal()operation.
- 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.

Implementation

- The main disadvantage of the semaphore definition requires busywaiting.
- 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.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.

- 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 thelock.

Semaphore implementation with no busy waiting

- The definition of the wait() and signal() semaphore operations ismodified.
- When a process executes the wait () operation and finds that the semaphore value is not positive, it mustwait.
- However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process toexecute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup( ) operation, which changes the process from the waiting state to the ready state. The process is then placed in the readyqueue.

- To implement semaphores under this definition, we define a semaphore as a "C' struct:

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

- Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

- The wait() semaphore operation can now be defined as:

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

- The signal () semaphore operation can now be defined as

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

  - The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic systemcalls.
  - In this implementation semaphore values may be negative. If a semaphore value is negative, its magnitude is the number of processes waiting on thatsemaphore.

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal( ) operation. When such a state is reached, these processes are said to be deadlocked.
- To illustrate this, consider a system consisting of two processes, Po and P1, each accessing two semaphores, S and Q, set to the value 1

P_0	P_1
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 cam1ot be executed, Po and P1 are deadlocked.

- Another problem related to deadlocks is indefinite blocking or starvation: 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.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

- Bounded-BufferProblem
- Readers and WritersProblem
- Dining-PhilosophersProblem

Bounded-Buffer Problem

- N buffers, each can hold one item
- Semaphore **mutex**initialized to the value 1
- Semaphore **full** initialized to the value0
- Semaphore empty initialized to the value N.

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

The structure of the producer process:

```
while (true)
{
        wait (full);
        wait (mutex);
        // remove an item from buffer
        signal (mutex);
        signal (empty);
        // consume the removed item
}
```

The structure of the consumerprocess:

Readers-Writers Problem

- A data set is shared among a number of concurrentprocesses
- Readers – only read the data set; they do **not** perform anyupdates
- Writers – can both read andwrite.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the sametime.
  - SharedData
  - Dataset
  - Semaphore **mutex**initialized to 1.
  - Semaphore **wrt**initialized to1.
  - Integer **readcount**initialized to 0.

```
while (true)
{
        wait (wrt) ;
        // writing is performed
        signal (wrt) ;
}
```

The structure of a writerprocess

```
while (true)
{
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)
        // reading is performed
        wait (mutex) ;
        readcount - - ;
        if (readcount == 0)
                signal (wrt) ;
        signal (mutex) ;
}
```

The structure of a readerprocess

Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five singlechopsticks.

A philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinkingagain.

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-freemanner.

Solution: One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on thatsemaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher *i*is shown

```
while (true)
{
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
        // eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
        // think
}
```

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at thetable.

- Allowaphilosophertopickupherchopsticksonlyifbothchopsticksareavailable.
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her leftchopstick.

Problems with Semaphores

Correct use of semaphore operations:

- signal (mutex) …. wait (mutex) : Replace signal with wait andvice-versa
- wait (mutex) … wait(mutex)
- Omitting of wait (mutex) or signal (mutex) (orboth)

Monitor

- An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.
- A ***monitor type***is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.
- The monitor construct ensures that only one process at a time is active within the monitor.

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

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Syntax of the monitor

- To have a powerful Synchronization schemes a *condition* construct is added to the Monitor. So synchronization scheme can be defined with one or more variables of type *condition* Two operations on a conditionvariable:

  Condition x, y

- The only operations that can be invoked on a condition variable are wait() and signal(). The operation

  x.wait () – a process that invokes the operation is suspended.

  x.signal () – resumes one of processes (if any) that invoked x.wait ()

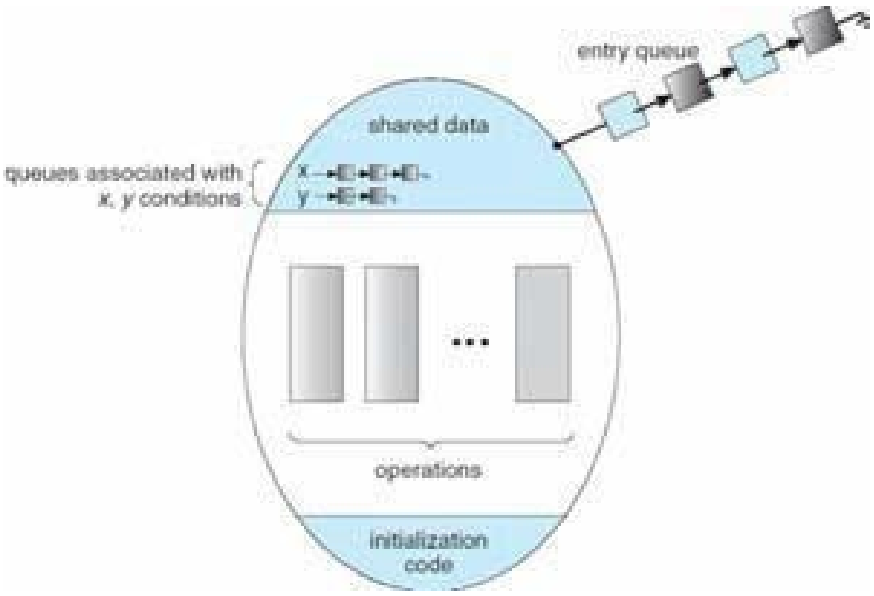


Fig: Monitor with Condition Variables

Solution to Dining Philosophers

Each philosopher I invokes the operations **pickup()** and **putdown()** in the following

```
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[i] = THINKING;
        // test left and right neighbors
        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;
}
}
```

- For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus ,each external function F is replaced by

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

- For each condition x, we introduce a semaphore x sem and an integer variable x count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

- The operation x.signal() can be implementedas

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

Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then to determine which of the suspended processes should be resumed next, one simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. For this purpose, the **conditional-wait** construct can be used. This construct has theform

x.wait(c);

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumednext.

```
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;
   }
}
```

- The Resource Allocator monitor shown in the above Figure, which controls the allocation of a single resource among competingprocesses.
- 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.

- The monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the 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 neverrequested.
- A process might request the same resource twice (without first releasing the resource).

QUESTION BANK

1. What is a thread? What is TCB?
2. Write a note on multithreading models.
3. What is thread cancellation?
4. What is signal handling?
5. Explain The various Threading issues
6. What do you mean by
   a. Thread pool
   b. Thread specific data
   c. Scheduler activation
7. What is pre-emptive scheduling and non-pre-emptive scheduling?
8. Define the following:
   a. CPU utilization
   b. Throughput
   c. Turnaround time
   d. Waiting time
   e. Response time
9. Explain scheduling algorithms with examples.
10. Explain multilevel and multilevel feedback queue.
11. For the following set of process find the avg. waiting time and avg. turn around using Gantt chart for a) FCFS b) SJF (primitive and non-primitive) c) RR (quantum= 4)

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

12. What are semaphores? Explain two primitive semaphore operations. What are its advantages?
13. Explain any one synchronization problem for testing newly proposed sync scheme
14. Explain three requirements that a solution to critical –section problem must satisfy.

15. State Dining Philosopher's problem and give a solution using semaphores. Write structure of philosopher.
16. What do you mean by binary semaphore and counting semaphore? With C struct, explain implementation of wait() and signal. Semaphore as General Synchronization Tool.
17. Describe term monitor. Explain solution to dining philosophers.
18. What are semaphores? Explain solution to producer-cons umer problem using semaphores
19. What is critical section ? Explain the various methods to implement process synchronization.
20. Explain the various classical synchronization problems.