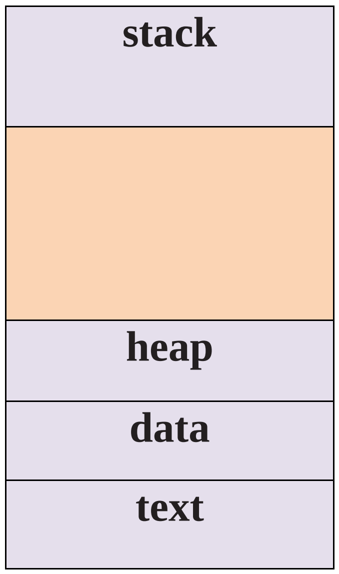
Unit II

Process management

Processes :

- A process is a program in execution.
- A processis more than the program code, which is sometimes known as the text section.
- It also includes the current activity, as represented by the value of the programcounter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data and a data section, whichcontains global variables.
- A process may also include a heap, which ismemory that is dynamically allocated during process run time.
- The structure of a process in memory is



- A program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an executable file).
- Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process.

A process may be in one of the following states:

1. **New**. The process is being created.
2. **Running**. Instructions are being executed.
3. **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
4. **Ready**. The process is waiting to be assigned to a processor.
5. **Terminated**. The process has finished execution.

These names are arbitrary, and they vary across operating systems.

Process scheduling algorithms :

- ✓ The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- ✓ The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- ✓ To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.
- ✓ For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

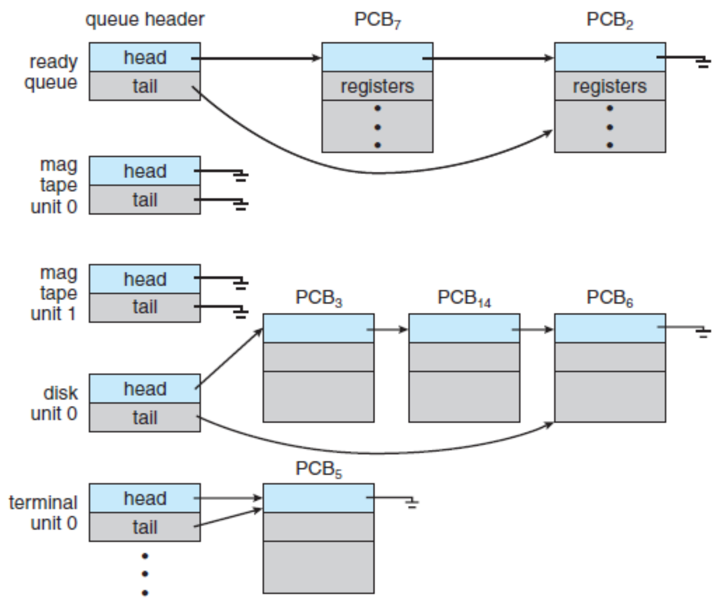


Figure 3.5 The ready queue and various I/O device queues.

➢ Scheduling Queues

The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.This queue is generally stored as a linked list.

A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues.

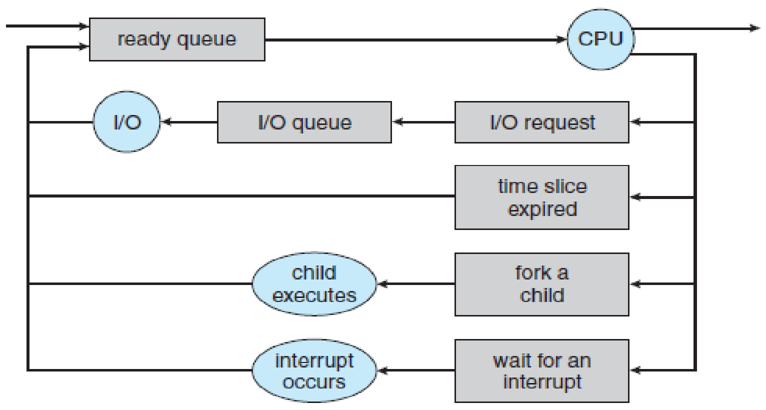


Figure 3.6 Queueing-diagram representation of process scheduling.

- Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.

The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

Interprocess Communication:

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

A process is *independent* if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is *cooperating* if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- ✓ Information sharing. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- ✓ Computation speedup. Ifwewant a particular task to run faster,wemust break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- ✓ Modularity. We may want to construct the systemin a modular fashion, dividing the system functions into separate processes or threads, as we discussed in Chapter 2.
- ✓ Convenience. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: shared memoryand message passing.

1. **Shared Memory**

   Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

2. **Message Queue**

   Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

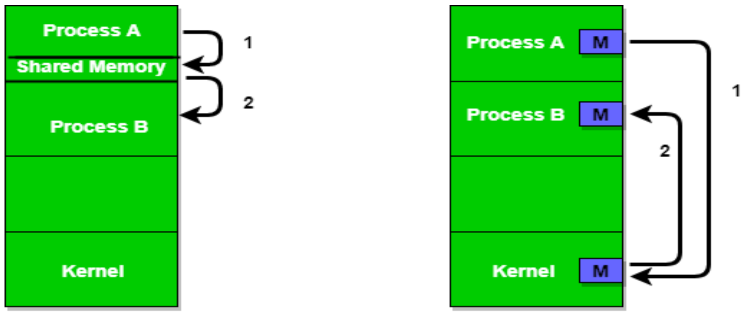


Figure 1 - Shared Memory and Message Passing

Examples Of IPC Systems:

- Posix : uses shared memory method.
- Mach : uses message passing.
- Windows XP : uses message passing using local procedural calls.

POSIX:

This interprocess communication (IPC) is a variation of System V interprocess communication. Like System V objects, POSIX IPC objects have read and write, but not execute, permissions for the owner, the owner's group, and for others. There is no way for the owner of a POSIX IPC object to assign a different owner. POSIX IPC includes the following features:

- Messages allow processes to send formatted data streams to arbitrary processes.
- Semaphores allow processes to synchronize execution.
- Shared memory allows processes to share parts of their virtual address space.

Unlike the System V IPC interfaces, the POSIX IPC interfaces are all multithread safe.

Mach IPC: Messages

- Segregated capabilities:
    - threads refer to them via local indices.
    - kernel marshalls capabilities in messages.
    - message format must identify caps
- Message contents:
    - Send capability to destination port (mandatory)
        - used by kernel to validate operation;
    - optional send capability to reply port
        - for use by receiver to send reply
    - possibly other capabilities;
    - ``in-line'' (by-value) data;
    - ``out-of-line'' (by reference) data, using copy-on-write,
        - may contain whole address spaces;

Windows XP:

- Message-passing centric via advanced local procedure call (LPC) facility.
- Only works between processes on the same system.
- Uses ports (like mailboxes) to establish and maintain communication channels.
- Communication works as follows:

    - ✓ The client opens a handle to the subsystem's connection port object.
    - ✓ The client sends a connection request.
    - ✓ The server creates two private communication ports and returns the handle to one of them to the client.

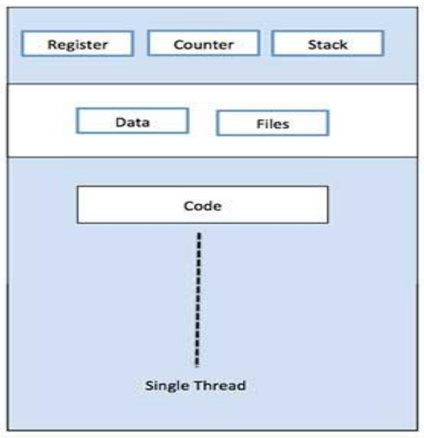
- ✓ The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Thread:

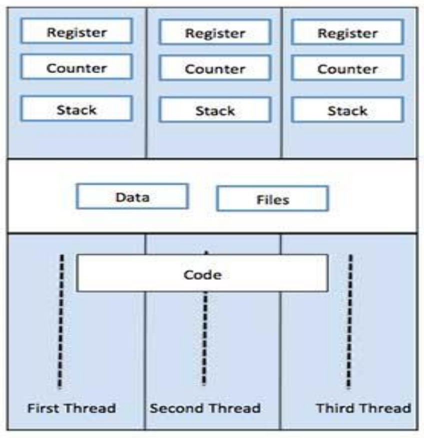
A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism



Single Process P with single thread



Single Process P with three threads

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Multi core Programming:

Multicore programming helps to create concurrent systems for deployment on multicore processor and multiprocessor systems. A multicore processor system is basically a single processor with multiple execution cores in one chip.

It has multiple processors on the motherboard or chip. A Field-Programmable Gate Array (FPGA) is might be included in a multiprocessor system.

A FPGA is an integrated circuit containing an array of programmable logic blocks and a hierarchy of reconfigurable interconnects. Input data is processed by to produce outputs. It can be a processor in a multicore or multiprocessor system, or a FPGA.

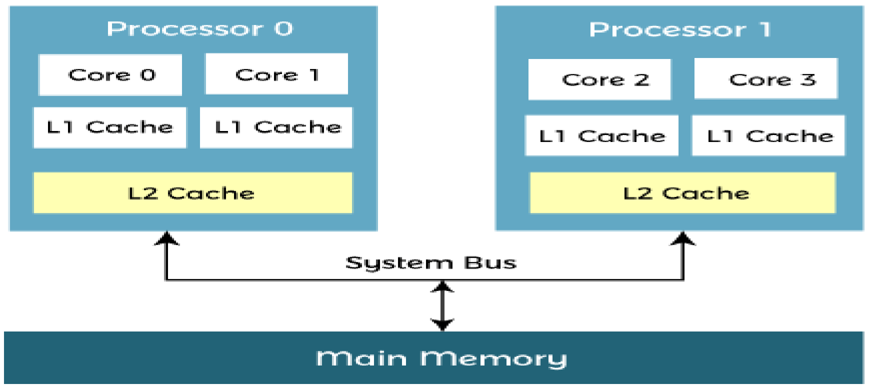
The multicore programming approach has following advantages &minus;

- Multicore and FPGA processing helps to increase the performance of an embedded system.
- Also helps to achieve scalability, so the system can take advantage of increasing numbers of cores and FPGA processing power over time.

This is known as concurrent execution. When multiple parallel tasks are executed by a processor, it is known as multitasking.

A CPU scheduler, handles the tasks that execute in parallel. The CPU implements tasks using operating system threads. So that tasks can execute independently but have some data transfer between them, such as data transfer between a data acquisition module and controller for the system.

Data transfer occurs when there is a data dependency.



Multithreading Models:

Some operating system provide a combined user level thread and Kernel level thread facility.

Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.

Multithreading models are three types

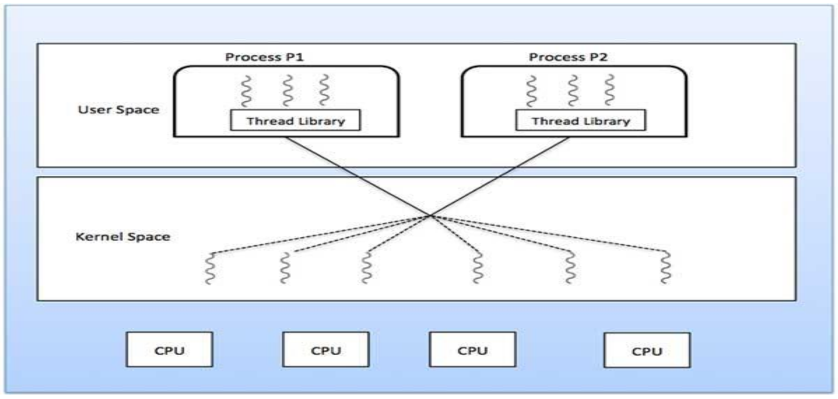
- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine.

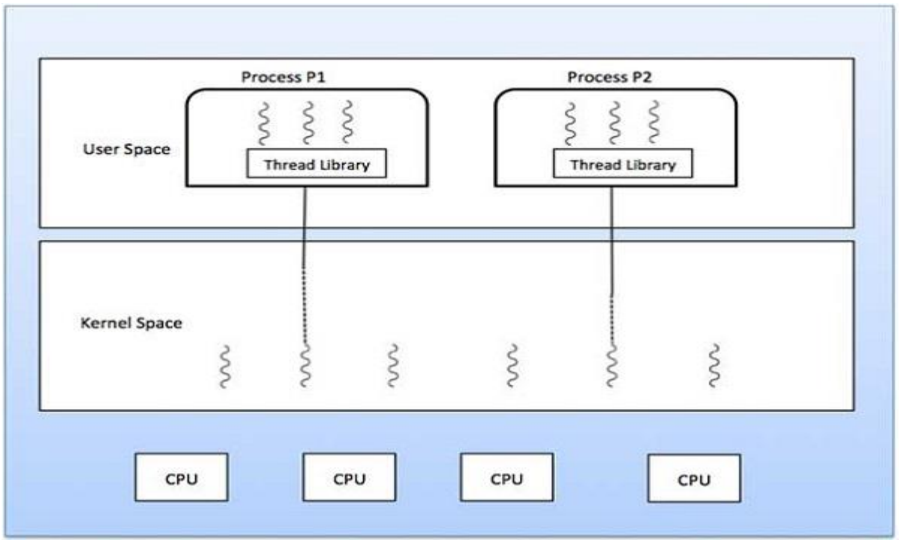
This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many to One Model:

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library.

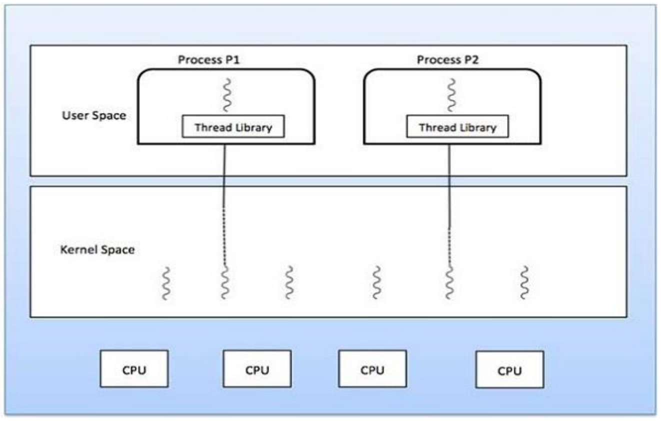
When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.



One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model.

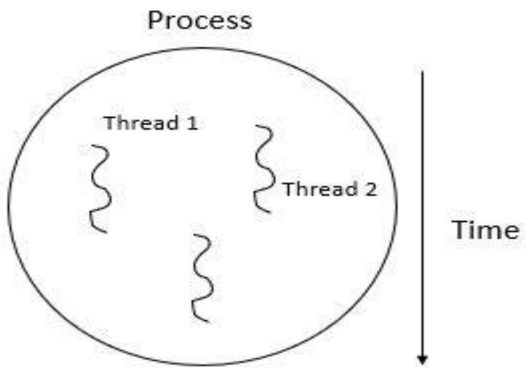
It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.



Thread Library:

A thread is a lightweight of process and is a basic unit of CPU utilization which consists of a program counter, a stack, and a set of registers.

Given below is the structure of thread in a process −



.

A thread library provides the programmer with an Application program interface for creating and managing thread.

Ways of implementing thread library

There are two primary ways of implementing thread library, which are as follows −

- The first approach is to provide a library entirely in user space with kernel support. All code and data structures for the library exist in a local function call in user space and not in a system call.
- The second approach is to implement a kernel level library supported directly by the operating system. In this case the code and data structures for the library exist in kernel space.

Invoking a function in the application program interface for the library typically results in a system call to the kernel.

The main thread libraries which are used are given below −

- **POSIX threads** − Pthreads, the threads extension of the POSIX standard, may be provided as either a user level or a kernel level library.
- **WIN 32 thread** − The windows thread library is a kernel level library available on windows systems.
- **JAVA thread** − The JAVA thread API allows threads to be created and managed directly as JAVA programs.

Thread Issues:

- **The fork() and exec() system calls**

The fork() is used to create a duplicate process. The meaning of the fork() and exec() system calls change in a multithreaded program.

If one thread in a program which calls fork(), does the new process duplicate all threads, or is the new process single-threaded? If we take, some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.

If a thread calls the exec() system call, the program specified in the parameter to exec() will replace the entire process which includes all threads.

- **Signal Handling**

Generally, signal is used in UNIX systems to notify a process that a particular event has occurred. A signal received either synchronously or asynchronously, based on the source of and the reason for the event being signalled.

All signals, whether synchronous or asynchronous, follow the same pattern as given below −

- A signal is generated by the occurrence of a particular event.
- The signal is delivered to a process.
- Once delivered, the signal must be handled.

- **Cancellation**

Thread cancellation is the task of terminating a thread before it has completed.

For example − If multiple database threads are concurrently searching through a database and one thread returns the result the remaining threads might be cancelled.

A target thread is a thread that is to be cancelled, cancellation of target thread may occur in two different scenarios −

- **Asynchronous cancellation** − One thread immediately terminates the target thread.
- **Deferred cancellation** − The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an ordinary fashion.

Some of the problems that arise in creating a thread are as follows −

- The amount of time required to create the thread prior to serving the request together with the fact that this thread will be discarded once it has completed its work.
- If all concurrent requests are allowed to be serviced in a new thread, there is no bound on the number of threads concurrently active in the system.
- Unlimited thread could exhaust system resources like CPU time or memory.

A thread pool is to create a number of threads at process start-up and place them into a pool, where they sit and wait for work.

Process Synchronization:

Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

- Independent Process
- Cooperative Process

Independent Processes:

Two processes are said to be independent if the execution of one process does not affect the execution of another process.

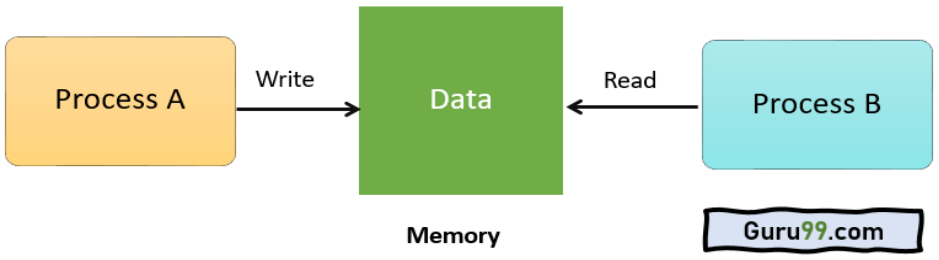
Cooperative Processes

Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

Process Synchronization Task:

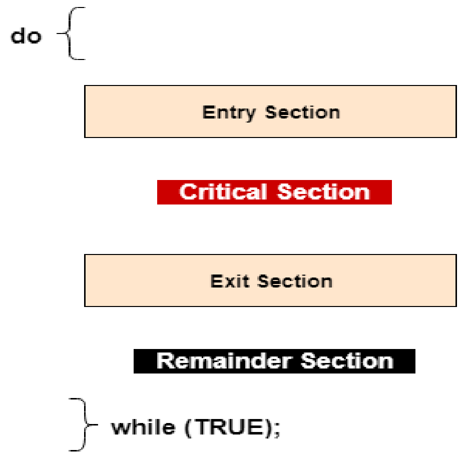
It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

- It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
- In order to synchronize the processes, there are various synchronization mechanisms.
- Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time.



Critical-Section Problem:

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.



The entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

Solution to the Critical Section Problem:

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

- **Mutual Exclusion**
  Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.
- **Progress**
  Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
- **Bounded Waiting**
  Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

Peterson's Solution:

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.

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   /* critical section */
   flag[i] = false;
   /* remainder section */
}
while (true);
```

Explanation:

The structure of process Pi in Peterson’s solution. This solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The processes are numbered P0 and P1. We use Pj for convenience to denote the other process when Pi is present; that is, j equals 1 − I, Peterson’s solution requires the two processes to share two data items −

int turn;

boolean flag[2];

The variable turn denotes 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. If a process is ready to enter its critical section, the flag array is used to indicate that.

For E.g., if flag[i] is true, this value indicates that Pi is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in above.

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 do so.

Turn will be set to both i and j at roughly the same time, if both processes try to enter at the same time. Only one of these assignments will occur ultimately; the other will occur but will be overwritten immediately. The final value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that −

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded-waiting requirement is met.

To prove 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true.

These two observations indicate that P0 and P1 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 — say, 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 Pj is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that if a process is stuck in the while loop with the condition flag[j] == true and turn == j, process Pi can be prevented from entering the critical section only; this loop is the only one possible. flag[j] will be == false, and Pi can enter its critical section if Pj is not ready to enter the critical section.

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, Pi will enter the critical section then. Pj will enter the critical section, If turn == j.

Although once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. Pj must also set turn to i, if Pj resets flag[j] to true. Hence, 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 (bounded waiting).

Disadvantage:

- Peterson's solution works for two processes, but this solution is best scheme in user mode for critical section.
- This solution is also a busy waiting solution so CPU time is wasted. So that **"SPIN LOCK"** problem can come. And this problem can come in any of the busy waiting solution.

Synchronization Hardware:

Process Synchronization refers to coordinating the execution of processes so that no two processes can have access to the same shared data and resources. A problem occurs when two processes running simultaneously share the same data or variable.

There are three hardware approaches to solve process synchronization problems:

1. Swap
2. Test() and Set()
3. Unlock and lock

✓ Test and Set

In Test and Set the shared variable is a lock that is initialized to false.

The algorithm returns whatever value is sent to it and sets the lock to true. Mutual exclusion is ensured here, as till the lock is set to true, other processes will not be able to enter and the loop continues.

However, after one process is completed any other process can go in as no queue is maintained.

```
boolean test and set(boolean *target)
{
 boolean rv = *target;
 *target = true;
 return rv;
 }
```

The definition of the test and set() instruction

✓ Swap

In this algorithm, instead of directly setting the lock to true, the key is first set to true and then swapped with the lock.

Similar to Test and Set, when there are no processes in the critical section, the lock turns to false and allows other processes to enter.

Hence, mutual exclusion and progress are ensured but the bound waiting is not ensured for the very same reason.

```
int compare and swap(int *value, int expected, int new value)
{
 int temp = *value;
if (*value == expected)
*value = new value;
return temp;
}
```

The definition of the compare and swap() instruction.

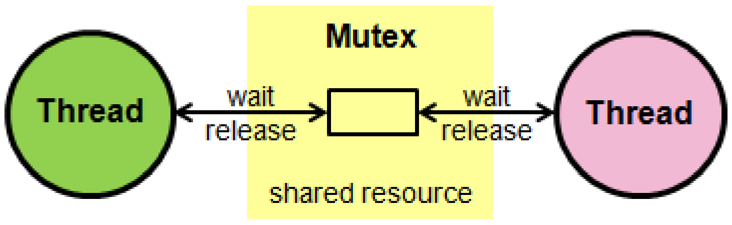
- ✓ **Unlock and Lock**

In addition to Test and Set, this algorithm uses waiting[i] to check if there are any processes in the wait. The processes are set in the ready queue with respect to the critical section.

Unlike the previous algorithms, it doesn't set the lock to false and checks the ready queue for any waiting processes. If there are no processes waiting in the ready queue, the lock is then set to false and any process can enter.

Mutex Locks:

A mutex is a binary variable used to provide a locking mechanism. It offers mutual exclusion to a section of code that restricts only one thread to work on a code section at a given time.



Working of Mutex

- Suppose a thread executes a code and has locked that region using a mutex.

- If the scheduler decides to perform context switching, all the threads ready to execute in the same region are unblocked.
- Out of all the threads available, only one will make it to the execution, but if it tries to access the piece of code already locked by the mutex, then this thread will go to sleep.
- Context switching will take place again and again, but no thread will be able to access the region that has been locked until the lock is released.
- Only the thread that has locked the region can unlock it as well.

Deadlock in Mutex

In a multiprogramming environment where two or more processes are executed, deadlock can occur using mutex during process synchronization. Deadlock occurs when threads are holding each other locks and are waiting for each other to unlock first. The four conditions for deadlock are:

1. **Mutual exclusion**: There must exist at least one resource which can be used by only one resource.
2. **No preemption:** The thread must release the resource by itself. Another thread can't snatch it.
3. **Hold and wait:** There must exist a resource holding a resource and waiting for another resource to be released.
4. **Circular wait:** All the processes must wait in a circular pattern.

Advantages of Mutex

- ✓ Since there is only one thread present in the critical section at a given time, there are no race conditions, and the data always remains consistent.
- ✓ Only one thread will be able to access the critical section.
- ✓ Solves the race condition problem.

Semaphores:

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

- **Wait**
  The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
  while (S<=0);


  S--;
}
```

- **Signal**
  The signal operation increments the value of its argument S.

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

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

- **Counting Semaphores**
  These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.
- **Binary Semaphores**

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Classic Problems of Synchronization:

The classical problems of synchronization are as follows:

- ➢ Bound-Buffer problem
- ➢ Sleeping barber problem
- ➢ Dining Philosophers problem
- ➢ Readers and writers problem

Bound-Buffer problem

Also known as the **Producer-Consumer problem**. In this problem, there is a buffer of n slots, and each buffer is capable of storing one unit of data.

There are two processes that are operating on the buffer – Producer and Consumer. The producer tries to insert data and the consumer tries to remove data.

If the processes are run simultaneously they will not yield the expected output.

The solution to this problem is creating two semaphores, one full and the other empty to keep a track of the concurrent processes.

Sleeping Barber Problem

This problem is based on a hypothetical barbershop with one barber. When there are no customers the barber sleeps in his chair. If any customer enters he will wake up the barber and sit in the customer chair. If there are no chairs empty they wait in the waiting queue.

Dining Philosopher's problem

This problem states that there are K number of philosophers sitting around a circular table with one chopstick placed between each pair of philosophers. The philosopher will be able to eat if he can pick up two chopsticks that are adjacent to the philosopher.

This problem deals with the allocation of limited resources.

Readers and Writers Problem

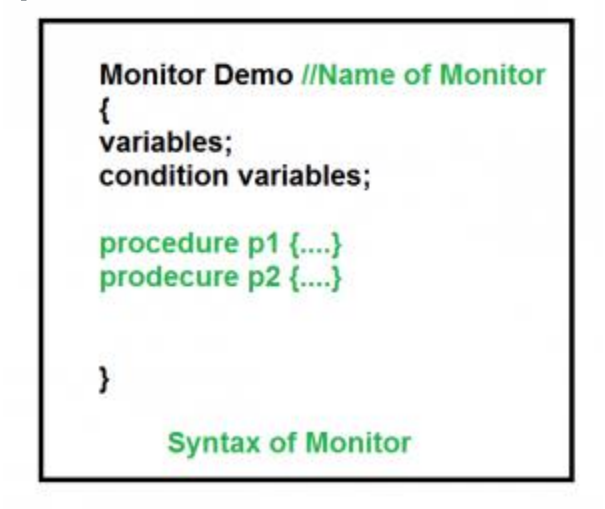
This problem occurs when many threads of execution try to access the same shared resources at a time. Some threads may read, and some may write. In this scenario, we may get faulty outputs.

Monitor Process Synchronization:

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can’t access the internal variable of the monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

Syntax:



```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

Condition Variables:
Two different operations are performed on the condition variables of the monitor.
Wait.

signal.

let say we have 2 condition variables
condition x, y; // Declaring variable

Wait operation
x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Signal operation
x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

```
If (x block queue empty)
  // Ignore signal
else
  // Resume a process from block queue.
```

Advantages of Monitor:
Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.

Disadvantages of Monitor:
Monitors have to be implemented as part of the programming language . The compiler must generate code for them.
This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes.
Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid.

CPU Scheduling:

- Operating System has to define which process the CPU will be given.
- **In Multiprogramming systems**, the Operating system schedules the processes on the CPU to have the maximum utilization of it and this procedure is called **CPU scheduling**. The Operating System uses various scheduling algorithm to schedule the processes.

CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler**, or CPU scheduler.

➢ The scheduler selects a processfrom the processes in memory that are ready to execute and allocates the CPU.to that process.

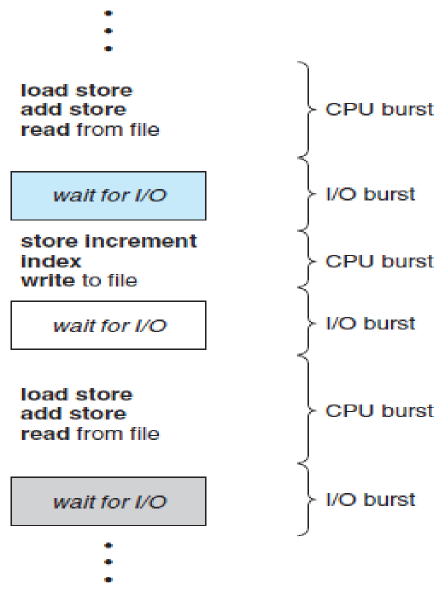


Figure 6.1 Alternating sequence of CPU and I/O bursts.

Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

➢ When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process).

➢ When a process switches from the running state to the ready state (for example, when an interrupt occurs).

➢ When a process switches from the waiting state to the ready state (for example, at completion of I/O).

➢ When a process terminates.

Scheduling Criteria:

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **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 loaded system).
- **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 ratemay be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time**. From the point of view of a particular process, the important criterion is 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/O.
- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. 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.
- **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.

Thread Scheduling:

Scheduling of threads involves two boundary scheduling,

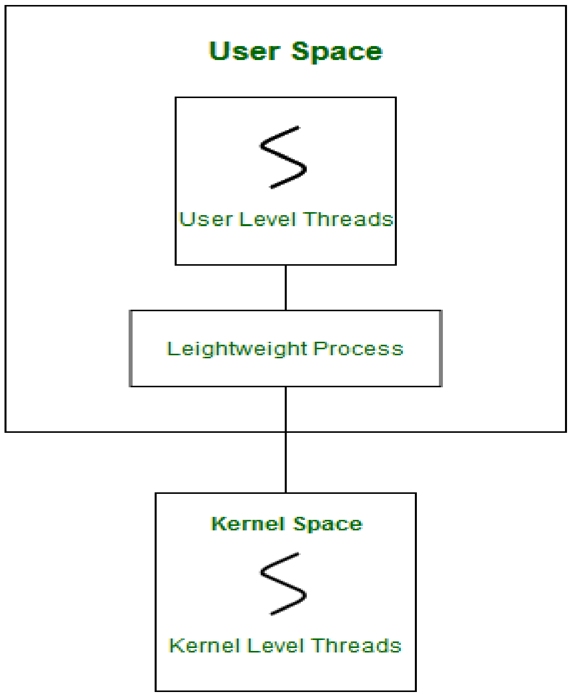
- Scheduling of user level threads (ULT) to kernel level threads (KLT) via lightweight process (LWP) by the application developer.
- Scheduling of kernel level threads by the system scheduler to perform different unique os functions.

Lightweight Process (LWP) :

Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources. Thread library schedules which thread of a process to run on which LWP and how long.

The number of LWP created by the thread library depends on the type of application. In the case of an I/O bound application, the number of LWP depends on the number of user-level threads.

This is because when an LWP is blocked on an I/O operation, then to invoke the other ULT the thread library needs to create and schedule another LWP. Thus, in an I/O bound application, the number of LWP is equal to the number of the ULT. In the case of a CPU bound application, it depends only on the application. Each LWP is attached to a separate kernel-level thread.



In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority. It requires two controls to be specified for the User level threads: Contention scope, and Allocation domain. These are explained as following below.

1. Contention Scope :
The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library. Depending upon the extent of contention it is classified as **Process Contention Scope** and **System Contention Scope**.

1. **Process Contention Scope (PCS) –**
   The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).

2. **System Contention Scope (SCS) –**
   The contention takes place among **all threads in the system**. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.
   In LINUX and UNIX operating systems, the POSIX Pthread library provides a function *Pthread_attr_setscope* to define the type of contention scope for a thread during its creation.

int Pthread_attr_setscope(pthread_attr_t *attr, int scope)

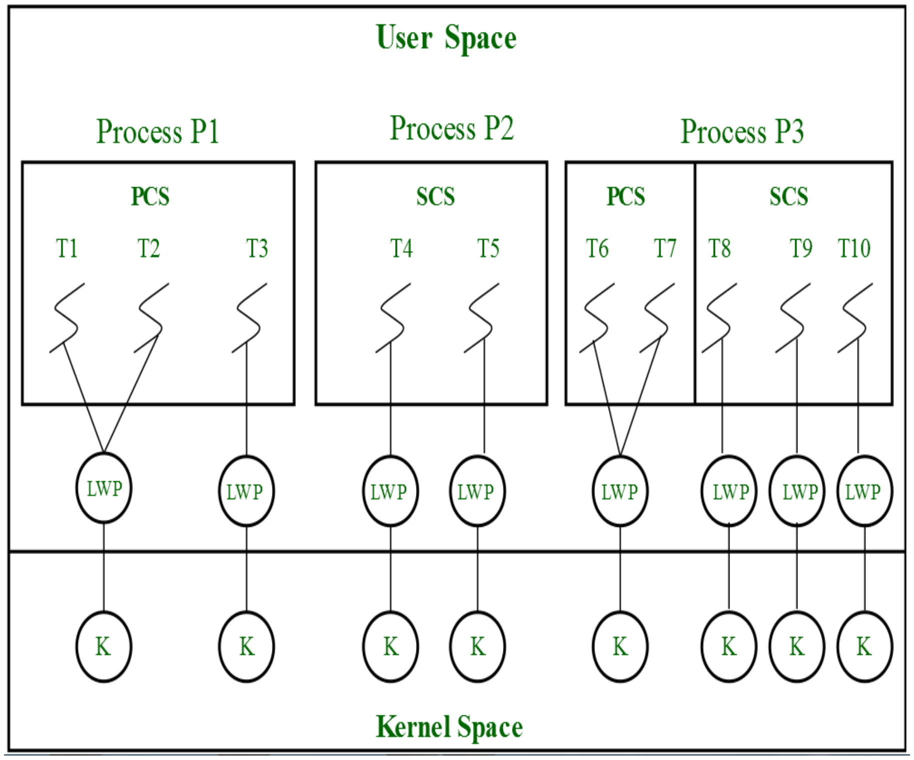
1. The first parameter denotes to which thread within the process the scope is defined.
   The second parameter defines the scope of contention for the thread pointed. It takes two values.

PTHREAD_SCOPE_SYSTEM

PTHREAD_SCOPE_PROCESS

1. If the scope value specified is not supported by the system, then the function returns *ENOTSUP*.

2. Allocation Domain :
The allocation domain is **a set of one or more resources** for which a thread is competing. In a multicore system, there may be one or more allocation domains where each consists of one or more cores. One ULT can be a part of one or more allocation domain. Due to this high complexity in dealing with hardware and software architectural interfaces, this control is not specified.



1. **Process P1:**
   All PCS threads T1, T2, T3 of Process P1 will compete among themselves. The PCS threads of the same process can share one or more LWP. T1 and T2 share an LWP and T3 are allocated to a separate LWP. Between T1 and T2 allocation of kernel resources via LWP is based on preemptive priority scheduling by the thread library. A Thread with a high priority will preempt low priority threads. Whereas, thread T1 of

process p1 cannot preempt thread T3 of process p3 even if the priority of T1 is greater than the priority of T3. If the priority is equal, then the allocation of ULT to available LWPs is based on the scheduling policy of threads by the system scheduler(not by thread library, in this case).

2. **Process P2:**
   Both SCS threads T4 and T5 of process P2 will compete with processes P1 as a whole and with SCS threads T8, T9, T10 of process P3. The system scheduler will schedule the kernel resources among P1, T4, T5, T8, T9, T10, and PCS threads (T6, T7) of process P3 considering each as a separate process. Here, the Thread library has no control of scheduling the ULT to the kernel resources.

3. **Process P3:**
   Combination of PCS and SCS threads. Consider if the system scheduler allocates 50% of CPU resources to process P3, then 25% of resources is for process scoped threads and the remaining 25% for system scoped threads. The PCS threads T6 and T7 will be allocated to access the 25% resources based on the priority by the thread library. The SCS threads T8, T9, T10 will divide the 25% resources among themselves and access the kernel resources via separate LWP and KLT. The SCS scheduling is by the system scheduler.