BCA Fourth Semester

Operating Systems

Unit -3 Process Management [15 Hrs]

Introduction to Process

- **A process is an instance of a program in execution.**
- **A program by itself is not a process;** a program is a **passive entity**, such as a file containing a list of instructions stored on disks. (often called an executable file), whereas a **process is an active entity**, with a program counter specifying the next instruction to execute and a set of associated resources.
- **A program becomes a process when an executable file is loaded into memory.**

Program: A set of instructions a computer can interpret and execute.

Process:

- Dynamic
- Part of a program in execution
- a live entity, it can be created, executed and terminated.
- It goes through different states wait running Ready etc
- Requires resources to be allocated by the OS
- one or more processes may be executing the same code.

Program:

- static
- no states

Introduction to Process (contd..)

This example illustrate the difference between a process and a program:

```
main () {
        int i , prod =1;
        for (i=0;i<100;i++)
                prod = pord*i;
}
```

- It is a program containing one multiplication statement (prod = prod * i) but the process will execute 100 multiplication, one at a time through the 'for' loop.
- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.
- For instance several users may be running different copies of mail program, or the same user may invoke many copies of web browser program.
- Each of these is a separate process, and although the text sections are equivalent, the data, heap and stack section may vary.

The Process Model

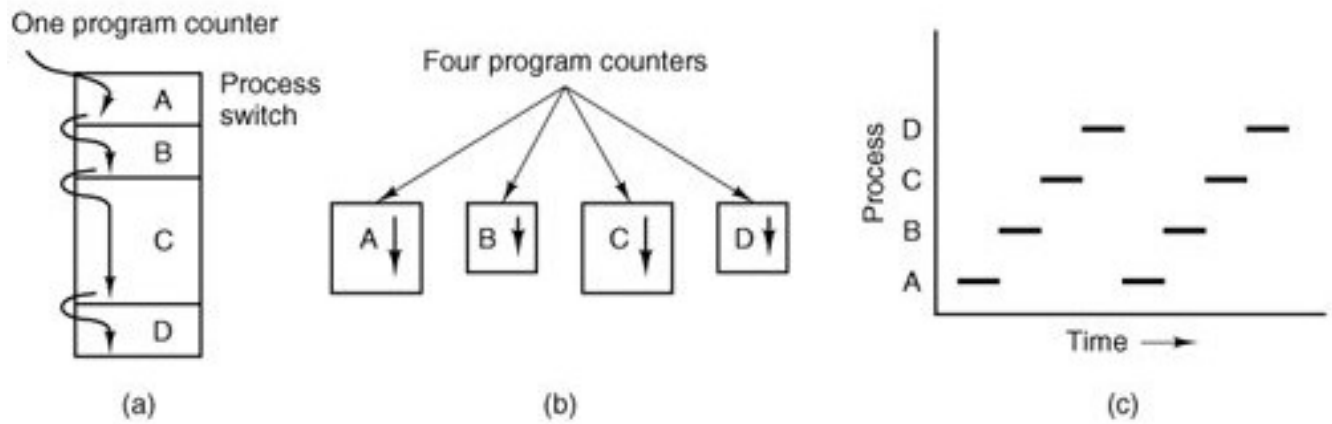


Fig: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

- A process is just an executing program, including the current values of the program counter, registers, and variables.
- Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program.
- **This rapid switching back and forth is called multiprogramming.**

Process Creation:

There are four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

- Parent process create children processes, which, in turn create other processes, forming a tree of processes. Generally, process identified and managed via a process identifier (pid).
- When an operating system is booted, often several processes are created.
- Some of these are foreground processes, that is, processes that interact with (human) users and perform work for them.
- Others are background processes, which are not associated with particular users, but instead have some specific function. For example, a background process may be designed to accept incoming requests for web pages hosted on that machine, waking up when a request arrives to service the request. Processes that stay in the background to handle some activity such as web pages, printing, and so on are called daemons.
- **In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job.**
- In interactive systems, users can start a program by typing a command or double clicking an icon.

Process Control Block:

- **In operating system each process is represented by a process control block(PCB) or a task control block.**
- **Its a data structure that physically represent a process in the memory of a computer system.**

- It contains many pieces of information associated with a specific process that includes the following.
  - **Identifier**: A unique identifier associated with this process, to distinguish it from all other processes.
  - **State**: If the process is currently executing, it is in the running state.
  - **Priority**: Priority level relative to other processes.
  - **Program counter**: The address of the next instruction in the program to be executed.
  - **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.

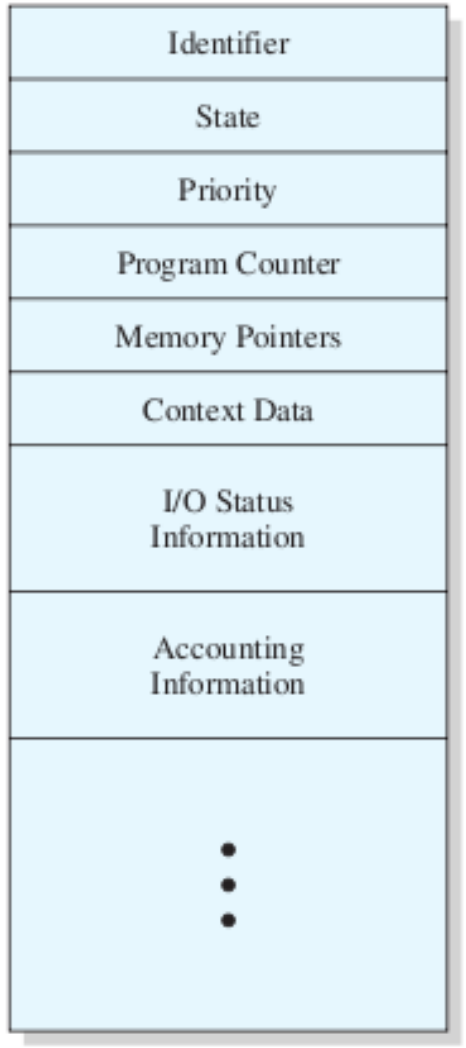


Fig: PCB

Process Control Block (contd.):

- **Context data**: These are data that are present in registers in the processor while the process is executing.

- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files in use by the process, and so on.

- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

Process Termination:

- After a process has been created, it starts running and does whatever its job is: After some time it will terminate due to one of the following conditions.

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Process States:

Each process may be in one of the following states:

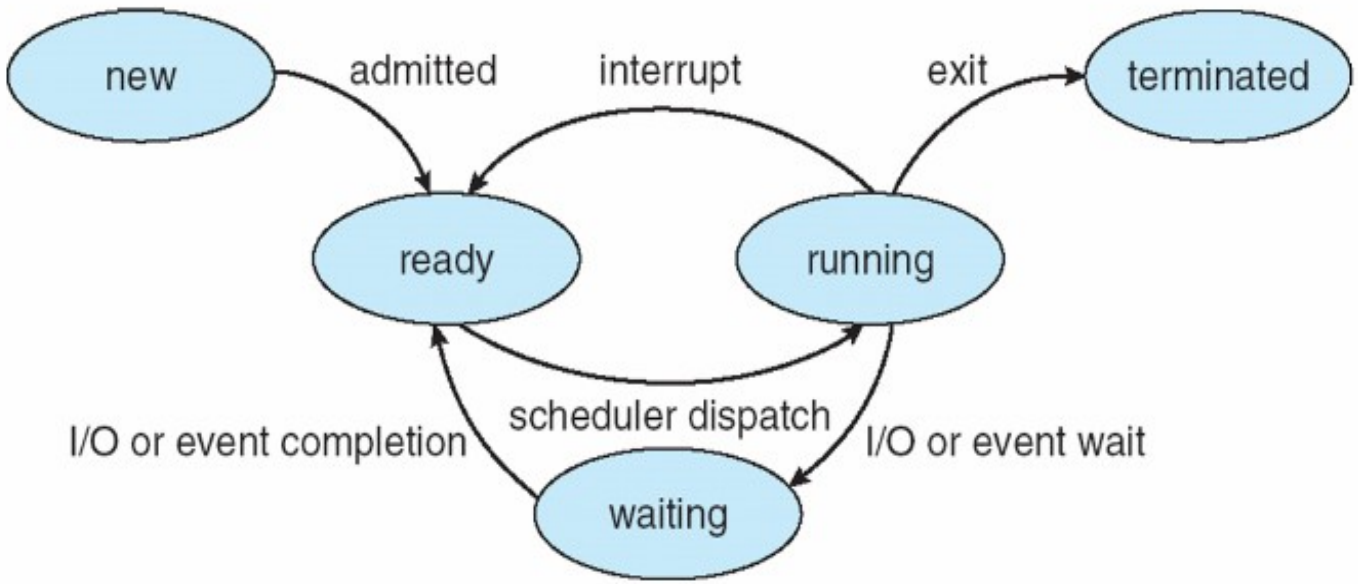


Fig: Process State Transition Diagram

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

Context Switching:

- **Switching the CPU to another process requires performing a state save of the current process and a state restore restore of a different process. This task is known as context switch.**
- When a context switch occurs, the kernel saves the the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.

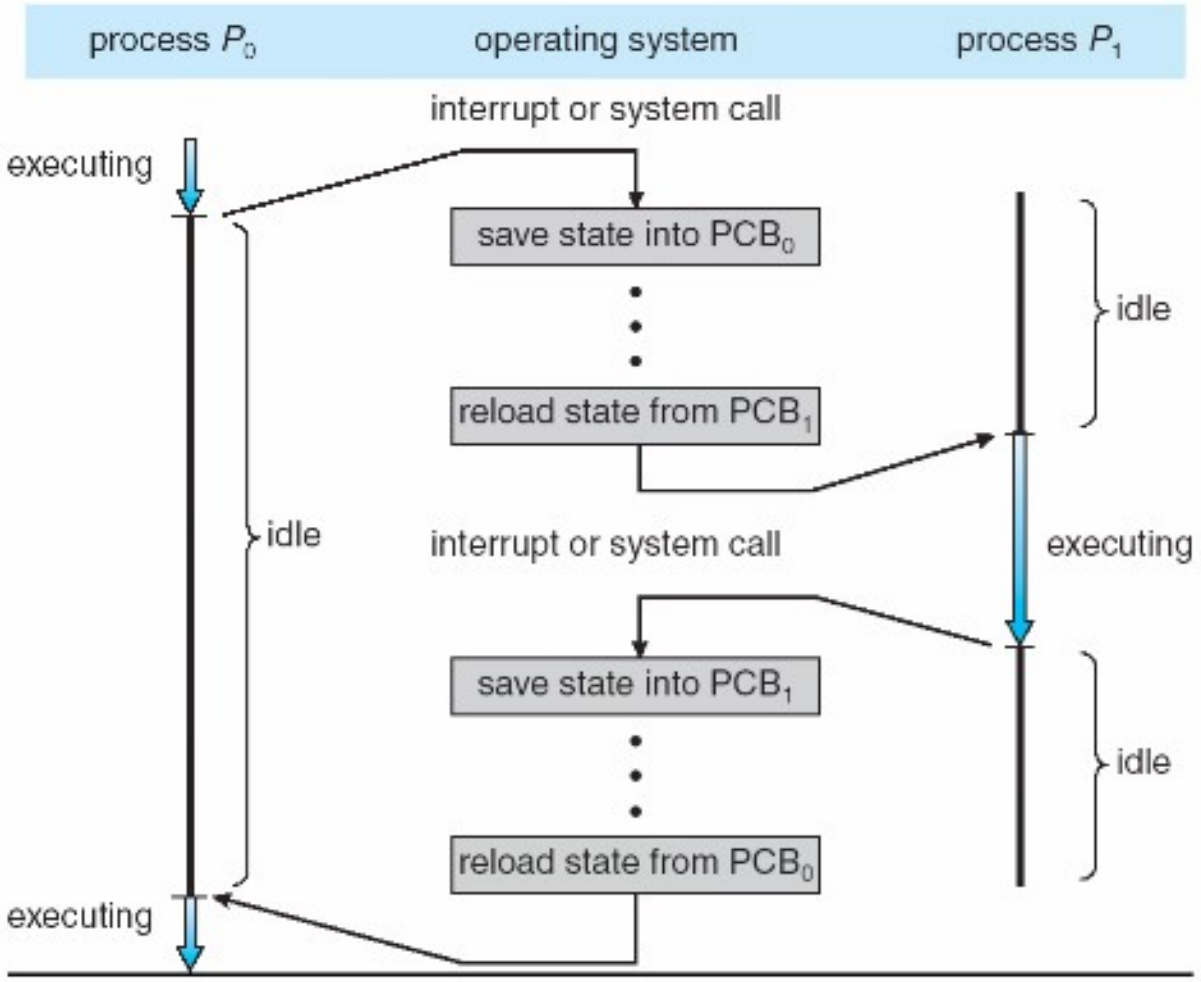


Fig: *CPU switch from one process to another*

Threads:

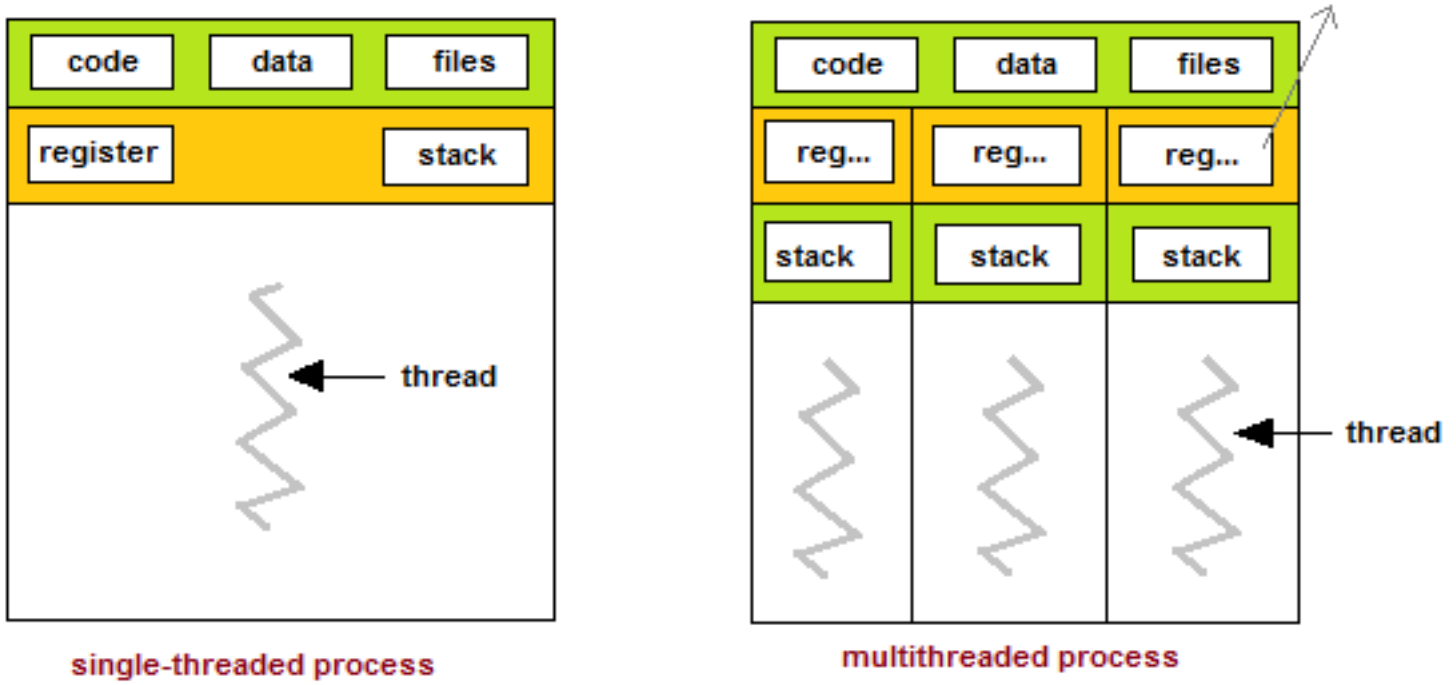
- **A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.**
- A thread is also called a **lightweight process**.
- Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.
- A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.

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

Multithreading:

- **In computer architecture, multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system.**
- As each thread has its own independent resource for process execution, multpile processes can be executed parallelly by increasing number of threads.



Benefits of Multithreading:

1) **Responsiveness:** Multithreaded interactive application continues to run even if part of it is blocked or performing a lengthy operation, thereby increasing the responsiveness to the user.

2) **Resource Sharing:** By default, threads share the memory and the resources of the process to which they belong. It allows an application to have several different threads of activity withing the same address space.

3) **Economy:** Allocating memory and resources for process creation is costly. Since thread shares the resources of the process to which they belong, it is more economical to create and context switch threads. It is more time consuming to create and manage process than threads.

4) **Utilization of multiprocessor architecture:** The benefits of multi threading can be greatly increased in multiprocessor architecture, where threads may be running in parallel on different processors. Multithreading on a multi-CPU increases concurrency.

Process VS Thread:

S.N.	Process	Thread
1	Program in execution.	Basic unit of CPU utilization.
2	Heavy weight	Light weight
3	Unit of Allocation – Resources, privileges, etc .	Unit of Execution – PC, SP, registers PC—Program counter, SP—Stack pointer
4	Inter-process communication is expensive: need to context switch Secure: one process cannot corrupt another process	Inter-thread communication cheap: can use process memory and may not need to context switch Not secure: a thread can write the memory used by another thread.
5	Process are Typically independent.	Thread exist as subsets of a process.
6	Process carry considerable state information.	Multiple thread within a process share state as well as memory and other resources.
7	Processes have separate address space .	Thread share their address space.
8	Processes interact only through system-provided inter-process communication mechanisms.	Context switching between threads in the same process is typically faster than context switching between processes.

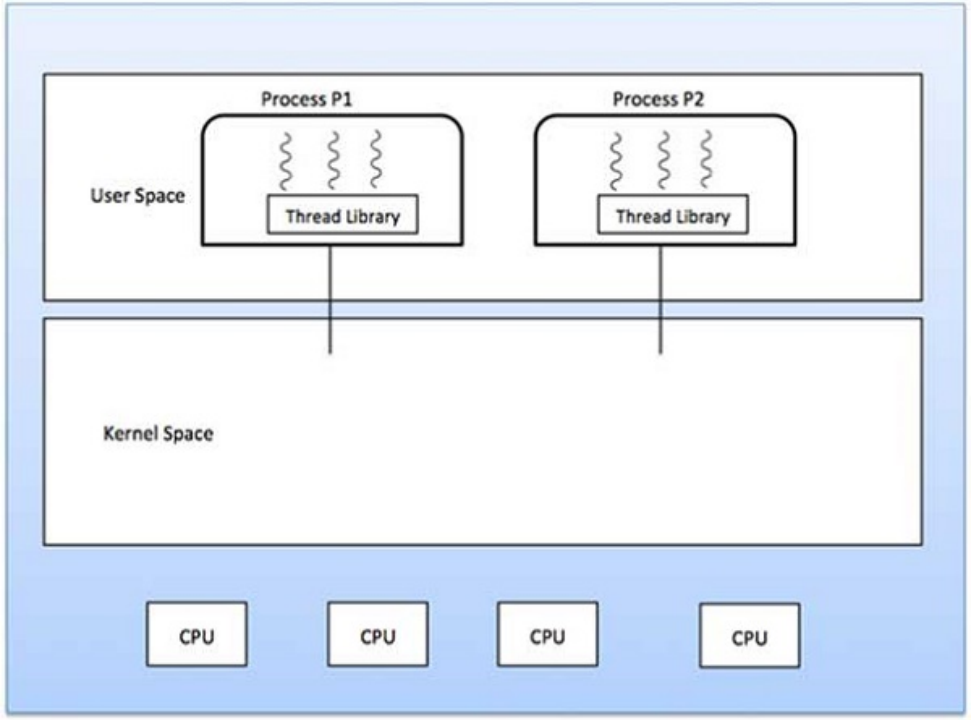
Types of Thread

There are two types of threads:

- User Threads
- Kernel Threads

User threads, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.



User Level Threads

- **In this case, the thread management kernel is not aware of the existence of threads**.
- The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.
- **The application starts with a single thread.**

Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

- **In this case, thread management is done by the Kernel.** There is no thread management code in the application area. **Kernel threads are supported directly by the operating system**. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.
- The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. **The Kernel performs thread creation, scheduling and management in Kernel space**.
- **Kernel threads are generally slower to create and manage than the user threads.**

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Difference between User and Kernel Level Threads

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

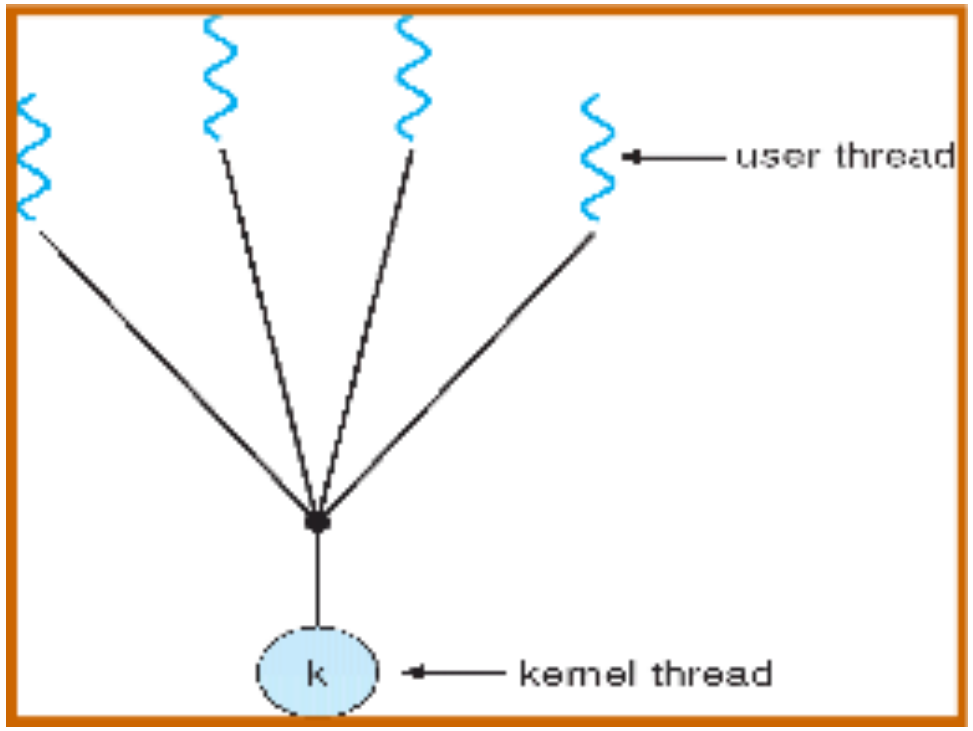
Multi Threading Models

- **Some operating system provide a combined user level thread and Kernel level thread facility**. Solaris is a good example of this combined approach.
- User thread are supported above the kernel and are managed without the kernel support whereas kernel threads are supported and are managed directly by the operating system.
- Virtually all operating-system includes kernel threads.

- **Ultimately there must exists a relationship between user threads and kernel threads. We have three models for it.**
  1. Many to many relationship.
  2. Many to one relationship.
  3. One to one relationship.

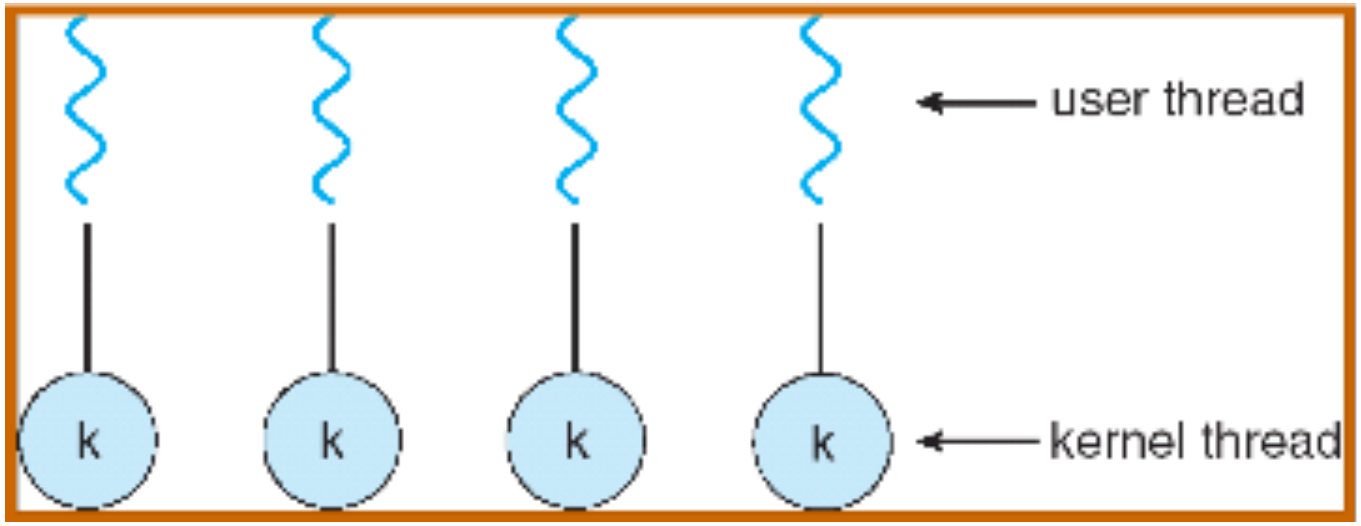
Many-to-one model

- **This model maps many user level threads to one kernel thread.**
- Thread management is done by the thread library in user space so it is efficient; but the entire process will block if a thread makes a blocking system call.
- Also only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- **Green Threads - a thread library available for Solaris use this model.**



One-to-one model

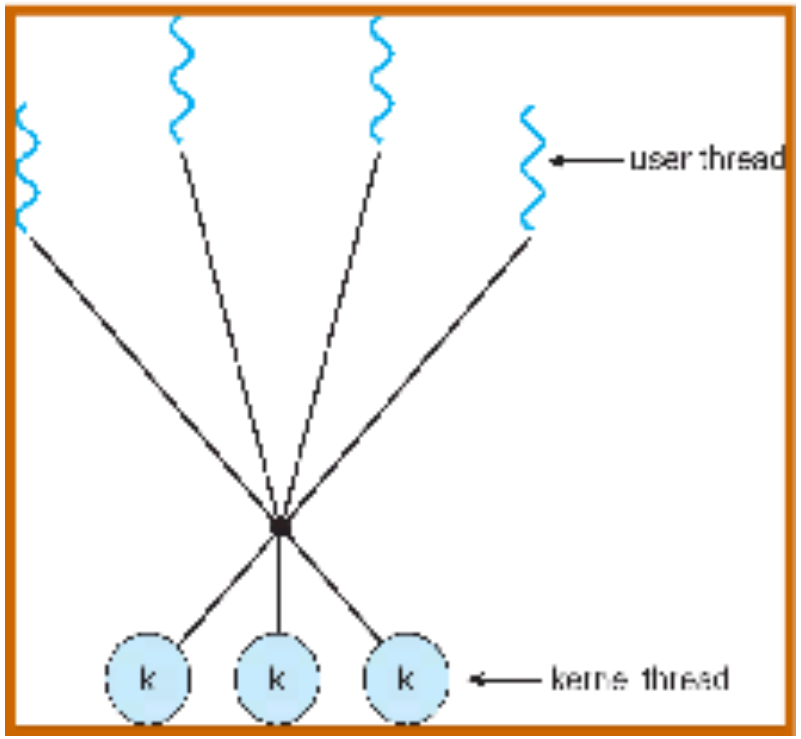
- **This model maps each user thread to a kernel thread.**
- It provides more concurrency than many to one model by allowing another thread to run when a thread makes a blocking system call.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- **Linux along with families of Windows operating system use this model.**



Many-to-many model

- **This model multiplexes many user level thread to a smaller or equal number of kernel threads.**
- The number of kernel thread may be specific to either a particular application or a particular machine.

- Many-to-many model allows the users to create as many threads as he wishes but the true concurrency is not gained because the kernel can schedule only one thread at a time.



Inter Process Communication (IPC)

- Processes frequently needs to communicate with each other.
- **Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves synchronizing their actions and managing shared data.**

- IPC enables one application to control another application, and for several applications to share the same data without interfering with one another.
- Inter-process communication (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes.
- Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for **message passing, synchronization, shared memory, and remote procedure calls (RPC).**

Co-Operating Process

- A process is independent if it can't affect or be affected by another process.
- **A process is co-operating if it can affects other or be affected by the other process. Any process that shares data with other process is called co-operating process.**
- There are many reasons for providing an environment for process co-operation.

1. **Information sharing:** Several users may be interested to access the same piece of information(for instance a shared file). We must allow concurrent access to such information.
2. **Computation Speedup**: Breakup tasks into sub-tasks.
3. **Modularity**: construct a system in a modular fashion.
4. **Convenience**:

Co-Operating Process (cont..)

- Co-operating process requires IPC. There are two fundamental ways of IPC.

  a. Shared Memory

  b. Message Passing

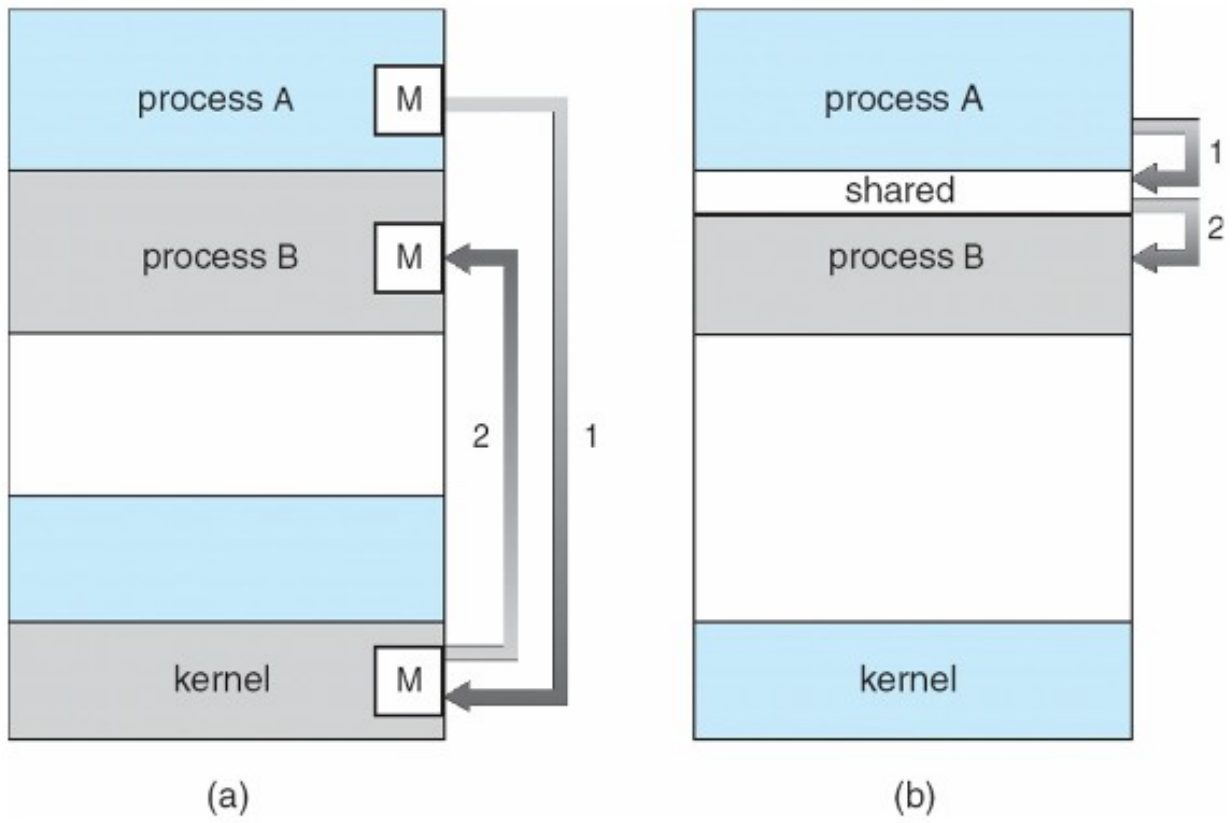


Fig: Communication Model a. Message Passing b. Shared Memory

Co-Operating Process (cont..)

Shared Memory

- **Here a region of memory that is shared by co-operating process is established.**
- Process can exchange the information by reading and writing data to the shared region.
- Shared memory allows maximum speed and convenience of communication as it can be done at the speed of memory within the computer.
- System calls are required only to establish shared memory regions. Once shared memory is established no assistance from the kernel is required, all access are treated as routine memory access.

Message Passing

- **Communication takes place by means of messages exchanged between the co-operating process.**
- Message passing is useful for exchanging the smaller amount of data since no conflict need to be avoided.
- Easier to implement than shared memory.
- Slower than that of Shared memory as message passing system are typically implemented using system call which requires more time consuming task of Kernel intervention.

Race Condition

- Concurrent access to shared resources can lead to race condition.
- **A race condition may be defined as the occurring of a condition when two or more threads can access shared data and then try to change its value at the same time**.
- Due to this, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.
- **A Race Condition typically occurs when two or more threads try to read, write and possibly make the decisions based on the memory that they are accessing concurrently.**

- **A race condition occurs when two or more threads can access shared data and they try to change it at the same time.** Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. **both threads are "racing" to access/change the data**.

Example of Race Condition

- **Problems often occur when one thread does a "check-then-act"** (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act".

- For Example,

```
if (x == 5) // The "Check"
{
    y = x * 2; // The "Act"
    // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
    // y will not be equal to 10.
}
```

- The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

Solution

- In order to prevent race conditions from occurring, you would **typically put a lock around the shared data to ensure only one thread can access the data at a time**.
- This would mean something like this:

```
// Obtain lock for x
if (x == 5)
{
    y = x * 2;      // Now, nothing can change x until the lock is released.
                    // Therefore y = 10
}
// release lock for x
```

Another Example for Race Condition:

- When two or more concurrently running threads/processes access a shared data item or resources and final results depends on the order of execution, we have race condition.
- **Suppose we have two threads A and B as shown below. Thread A increase the share variable count by 1 and Thread B decrease the count by 1.**

Thread A	Thread B
.....	.....
Count++;	Count--;
.....	.....

- If the current value of Count is 10, both execution orders yield 10 because Count is increased by 1 followed by decreased by 1 for the former case, and Count is decreased by 1 followed by increased by 1 for the latter.
- However, if Count is not protected by mutual exclusion, we might get difference results.

Avoiding Race Condition

Critical Section

- To avoid race condition we need **Mutual Exclusion.**
- **Mutual Exclusion** is someway of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

- That part of the program where the shared memory is accessed is called the **critical region or critical section**.

- **If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions.**
- Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.

Avoiding Race Condition

Critical Section (cont..)

(Rules for avoiding Race Condition) Solution to Critical section problem:

- No two processes may be simultaneously inside their critical regions. (Mutual Exclusion)
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

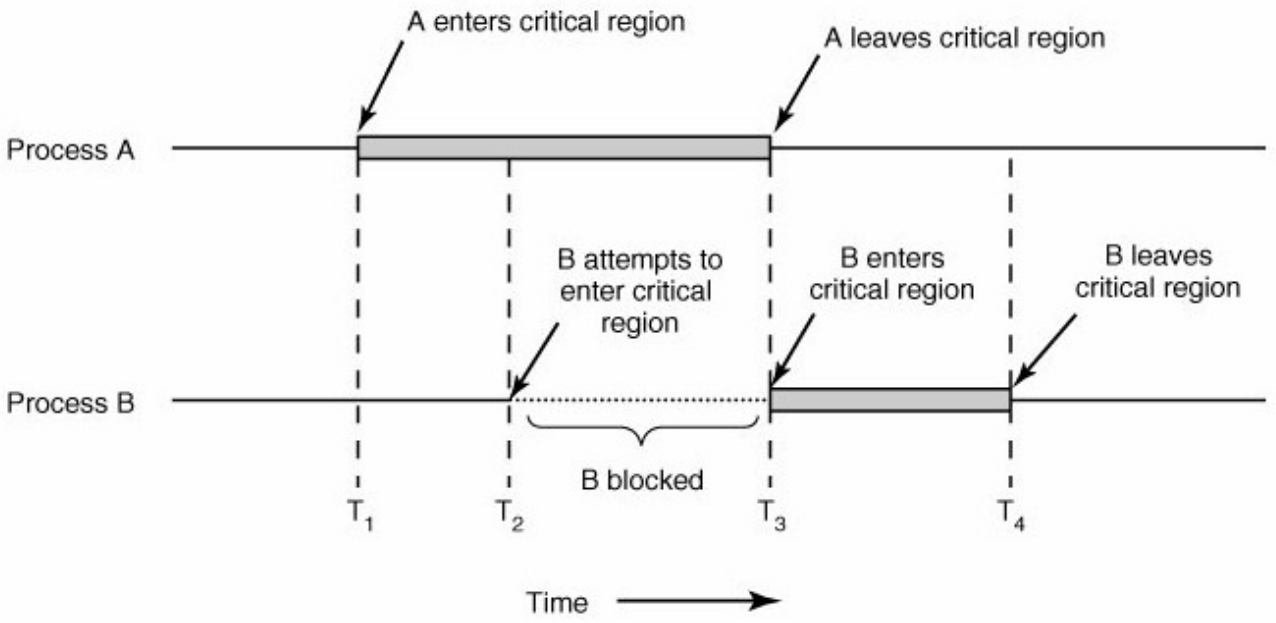


Fig: Mutual Exclusion using Critical Region

Techniques for avoiding Race Condition:

1. Disabling Interrupts
2. Lock Variables
3. Strict Alteration
4. Peterson's Solution
5. TSL instruction
6. Sleep and Wakeup
7. Semaphores
8. Monitors
9. Message Passing

1. Disabling Interrupts

- **The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.**
- With interrupts disabled, no clock interrupts can occur.
- The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

Advantages:

- It is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.

Disadvantages:

- It is unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did, and then never turned them on again?
- Furthermore, if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

2.Lock Variables

- a single, shared, (lock) variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.
- Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Drawbacks:

- Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1.
- When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

3.Strict Alteration:

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

- Integer variable turn is initially 0.
- It keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects turn, finds it to be 0, and enters its critical region.
- Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.

3.Strict Alteration (cont.):

- Continuously testing a variable until some value appears is called busy waiting. It should usually be avoided, since it wastes CPU time.
- Only when there is a reasonable expectation that the wait will be short is busy waiting used.
- A lock that uses busy waiting is called a spin lock. When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region.
- This way no two process can enters critical region simultaneously.

Drawbacks:

- Taking turn is is not a good idea when one of the process is much slower than other.
- This situation requires that two processes strictly alternate in entering their critical region.

4.Peterson's Solution:

- Initially neither process is in critical region.
- Now process 0 calls enter_region. It indicates its interest by setting its array element and sets turn to 0.
- Since process 1 is not interested, enter_region returns immediately.
- If process 1 now calls enter_region, it will hang there until interested[0] goes to FALSE, an event that only happens when process 0 calls leave_region to exit the critical region.

```
#define FALSE 0
#define TRUE  1
#define N     2                       /* number of processes */
int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially 0 (FALSE)*/
void enter_region(int process)        /* process is 0 or 1 */
{
    int other;                        /* number of the other process */
    other = 1 - process;              /* the opposite of process */
    interested[process] = TRUE;       /* show that you are interested */
    turn = process;                   /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}
void leave_region(int process)        /* process: who is leaving */
{
    interested[process] = FALSE;      /* indicate departure from critical region */
}
```

- Now consider the case that both processes call enter_region almost simultaneously.
- Both will store their process number in turn. Whichever store is done last is the one that counts; the first one is lost. Suppose that process 1 stores last, so turn is 1.
- When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region.

5. The TSL Instruction

TSL RX,LOCK

(Test and Set Lock) that works as follows:

- It reads the contents of the memory word LOCK into register RX and then stores a nonzero value at the memory address LOCK.
- The operations of reading the word and storing into it are guaranteed to be indivisible no other processor can access the memory word until the instruction is finished.
- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

- One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls enter_region, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls leave_region, which stores a 0 in LOCK. As with all solutions based on critical regions, the processes must call enter_region and leave_region at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was nonzero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```