(1) *What is Process? Give the difference between Process and Program.*

Process:

- Process is a program under execution.
- It is an instance of an executing program, including the current values of the program counter, registers & variables.
- Process is an abstraction of a running program.

Process	Program
A process is program in execution.	**A program is set of instructions.**
A process is an active/ dynamic entity.	A program is a passive/ static entity.
A process has a limited life span. It is created when execution starts and terminated as execution is finished.	A program has a longer life span. It is stored on disk forever.
A process contains various resources like memory address, disk, printer etc… as per requirements.	A program is stored on disk in some file. It does not contain any other resource.
A process contains memory address which is called address space.	A program requires memory space on disk to store all instructions.

(2) *What is multiprogramming?*

- 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, 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 and the number of processes loaded simultaneously in memory is called degree of multiprogramming.

(3) *What is context switching?*

- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process.
- This task is known as a context switch.

- The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state and memory-management information.
- When a context switch occurs, the kernel saves 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, and the existence of special instructions.

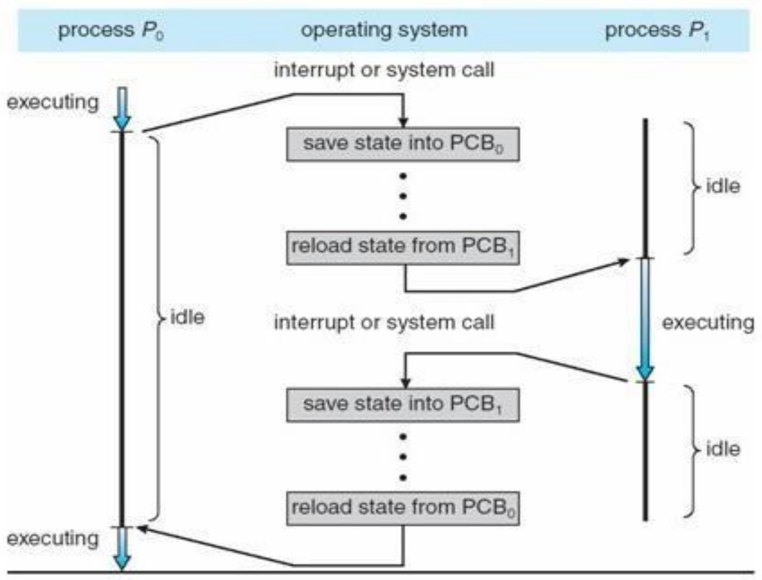


Figure 2-1. Context Switching

(4) *Explain Process Model in brief.*

- In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of sequential processes.
- 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, 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.

- In Fig. 2-2 (a) we see a computer multiprogramming four programs in memory.
- In Fig. 2-2 (b) we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones.
- There is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter.
- When it is finished for the time being, the physical program counter is saved in the process' logical program counter in memory.
- In Fig. 2-2 (c) we see that over a long period of time interval, all the processes have made progress, but at any given instant only one process is actually running.
- With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again.
- Thus, processes must not be programmed with built-in assumptions about timing.

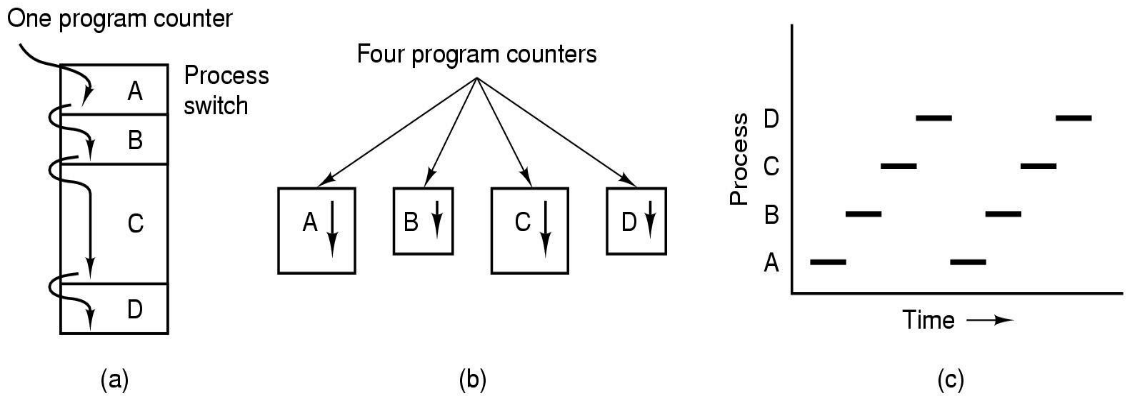


Figure 2-2. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

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.

Process Termination

- After a process has been created, it starts running and does whatever its job is.

- However, nothing lasts forever, not even processes. Sooner or later the new process will terminate, usually 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 Hierarchies

- In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways.
- The child process can itself create more processes, forming a process hierarchy.
- An example of process hierarchy is in UNIX, which initializes itself when it is started.
- A special process, called **init**, is present in the boot image.
- When it starts running, it reads a file telling how many terminals there are.
- Then it forks off one new process per terminal. These processes wait for someone to log in.
- If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth.
- Thus, all the processes in the whole system belong to a single tree, with **init** at the root.
- In contrast, Windows does not have any concept of a process hierarchy. All processes are equal.
- The only identification for parent child process is that when a process is created, the parent is given a special token (called a **handle)** that it can use to control the child.
- However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

(5) What is process state? Explain state transition diagram. OR

What is process state? Explain different states of a process with various queue generated at each stage.

Process state:

- The state of a process is defined by the current activity of that process.
- During execution, process changes its state.
- The process can be in any one of the following three possible states.
  1) **Running** (actually using the CPU at that time and running).

2) **Ready** (runnable; temporarily stopped to allow another process run).
3) **Blocked** (unable to run until some external event happens).

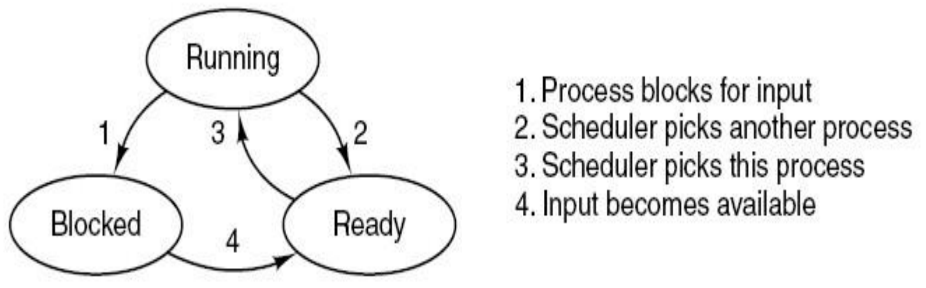

Running
Blocked
Ready
1
2
3
4
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-3. Process state transition diagram

- Figure 2-3 shows the state transition diagram.
- Logically, the first two states are similar. In both cases the process is willing to run, but in the ready state there is no CPU temporarily available for it.
- In blocked state, the process cannot run even if the CPU is available to it as the process is waiting for some external event to take place.
- There four possible transitions between these three states.
- Transition 1 occurs when the operating system discovers that a process cannot continue right now due to unavailability of input. In other systems including UNIX, when a process reads from a pipe or special file (e.g. terminal) and there is no input available, the process is automatically blocked.
- Transition 2 and 3 are caused by the process scheduler (a part of the operating system), without the process even knowing about them.
- Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time.
- Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again. The subject of scheduling, that is, deciding which process should run when and for how long, is an important one.
- Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that time, transition 3 will be triggered and the process will start running. Otherwise it may have to wait in ready state for a little time until the CPU is available and its turn comes.

Scheduling queues generated at each stage:

- As the process enters the system, it is kept into a job queue.

- The processes that are ready and are residing in main memory, waiting to be executed are kept in a ready queue.
- This queue is generally stored as a linked list.
- The list of processes waiting for a particular I/O device is called a device queue.
- Each device has its own queue. Figure 2-4 shows the queuing diagram of processes.
- In the figure 2-4 each rectangle represents a queue.
- There are following types of queues
  - ✓ **Job queue** : set of all processes in the system
  - ✓ **Ready queue** : set of processes ready and waiting for execution
  - ✓ **Set of device queues** : set of processes waiting for an I/O device
- The circle represents the resource which serves the queues and arrow indicates flow of processes.
- A new process is put in ready queue. It waits in the ready queue until it is selected for execution and is given the CPU.

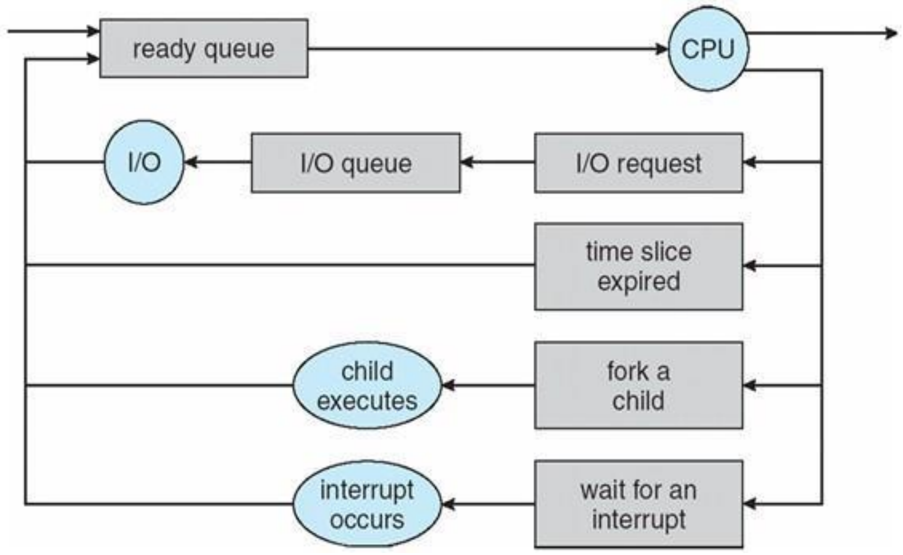


Figure 2-4. Queuing diagram representation of Process Scheduling

- Once the process starts execution one of the following events could occur.
  - ✓ The process issues an I/O request and then be placed in an I/O queue.
  - ✓ The process creates a new sub process and wait for its termination.
  - ✓ The process is removed forcibly from the CPU and is put back in ready queue.
- A process switches from waiting state to ready state and is put back in ready queue.

- The cycle continues unless the process terminates, at which the process is removed from all queues and has its PCB and resources de-allocated.

(6) Explain Process Control Block (PCB).

PCB (Process Control Block):

- To implement the process model, the operating system maintains a table (an array of structures) called the process table, with one entry per process, these entries are known as Process Control Block.
- Process table contains the information what the operating system must know to manage and control process switching, including the process location and process attributes.
- Each process contains some more information other than its address space. This information is stored in PCB as a collection of various fields. Operating system maintains the information to manage process.
- Various fields and information stored in PCB are given as below:
  - **Process Id:** Each process is given Id number at the time of creation.
  - **Process state:** The state may be ready, running, and blocked.
  - **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
  - **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
  - **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
  - **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
  - **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
  - **Status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

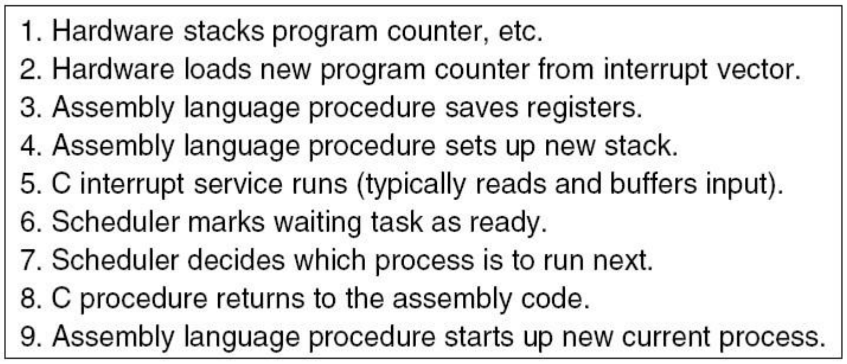
Process management	Memory management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit maps for signals	
	Various flag bits	

Figure 2-5. Some of the fields of a typical process table entry.

- Figure 2-5 shows some of the more important fields in a typical system.
- The fields in the first column relate to process management.
- The other two columns relate to memory management and file management, respectively.
- In practice, which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.

(7) What is interrupt? How it is handled by an OS?

- A software interrupt is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed.
- The former is often called a trap or exception and is used for errors or events occurring during program execution that is exceptional enough that they cannot be handled within the program itself.
- Interrupts are a commonly used technique for process switching.
- Associated with each I/O device class (e.g., floppy disks, hard disks etc…) there is a location (often near the bottom of memory) called the interrupt vector.
- It contains the address of the interrupt service procedure.
- Suppose that user process 3 is running when a disk interrupt occurs.
- User process 3's program counter, program status word, and possibly one or more registers are pushed onto the (current) stack by the interrupt hardware.
- The computer then jumps to the address specified in the disk interrupt vector.
- That is all the hardware does.



1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler marks waiting task as ready.
7. Scheduler decides which process is to run next.
8. C procedure returns to the assembly code.
9. Assembly language procedure starts up new current process.

Figure 2-6. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

- From here on, it is up to the software, in particular, the interrupt service procedure.
- All interrupts start by saving the registers, often in the process table entry for the current process.
- Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler.
- When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type.
- When it has done its job, possibly making some process now ready, the scheduler is called to see who to run next.
- After that, control is passed back to the assembly language code to load up the registers and memory map for the now-current process and start it running.
- Interrupt handling and scheduling are summarized in Figure 2-6.

(8) What is thread? Explain thread structure. Explain different types of thread. OR Explain thread in brief.

Thread

- A program has one or more locus of execution. Each execution is called a thread of execution.
- In traditional operating systems, each process has an address space and a single thread of execution.
- It is the smallest unit of processing that can be scheduled by an operating system.
- A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams.

Thread Structure

- Process is used to group resources together and threads are the entities scheduled for execution on the CPU.
- The thread has a program counter that keeps track of which instruction to execute next.
- It has registers, which holds its current working variables.
- It has a stack, which contains the execution history, with one frame for each proce- dure called but not yet returned from.
- Although a thread must execute in some process, the thread and its process are dif- ferent concepts and can be treated separately.
- What threads add to the process model is to allow multiple executions to take place in the same process environment, to a large degree independent of one another.
- Having multiple threads running in parallel in one process is similar to having multi- ple processes running in parallel in one computer.

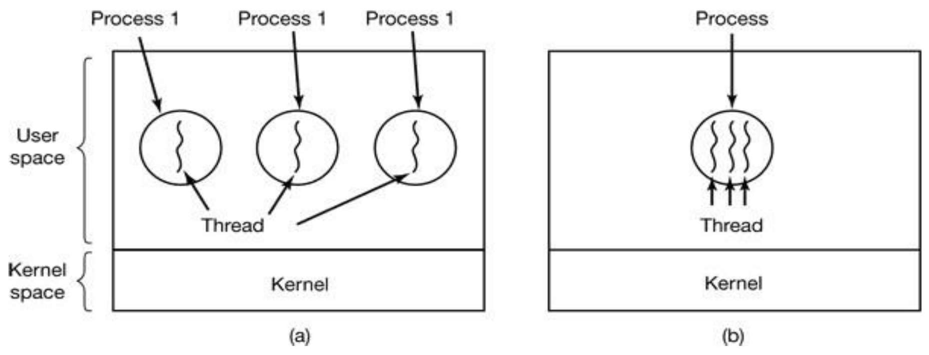


Figure 2-7. (a) Three processes each with one thread. (b) One process with three threads.

- In former case, the threads share an address space, open files, and other resources.
- In the latter case, process share physical memory, disks, printers and other resources.
- In Fig. 2-7 (a) we see three traditional processes. Each process has its own address space and a single thread of control.
- In contrast, in Fig. 2-7 (b) we see a single process with three threads of control.

- Although in both cases we have three threads, in Fig. 2-7 (a) each of them operates in a different address space, whereas in Fig. 2-7 (b) all three of them share the same address space.
- Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: running, blocked, ready, or terminated.
- When multithreading is present, processes normally start with a single thread present.

  This thread has the ability to create new threads by calling a library procedure

  thread_create.
- When a thread has finished its work, it can exit by calling a library procedure

  thread_exit.
- One thread can wait for a (specific) thread to exit by calling a procedure ***thread_join***.

  This procedure blocks the calling thread until a (specific) thread has exited.
- Another common thread call is ***thread_yield***, which allows a thread to voluntarily give up the CPU to let another thread run.

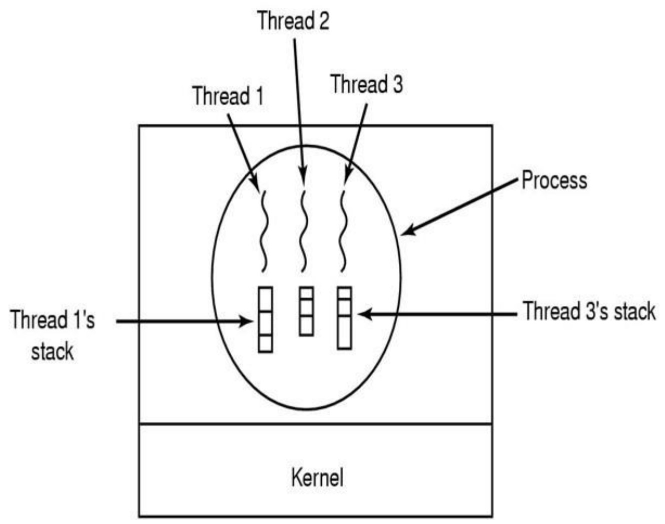


Figure 2-8. Each thread has its own stack.

Types of thread

1. User Level Threads
2. Kernel Level Threads

User Level Threads

- User level threads are implemented in user level libraries, rather than via systems calls.
- So thread switching does not need to call operating system and to cause interrupt to the kernel.
- The kernel knows nothing about user level threads and manages them as if they were single threaded processes.
- When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process.
- This table keeps track only of the per-thread properties, such as each thread's pro-
  gram counter, stack pointer, registers, state, and so forth.
- The thread table is managed by the run-time system.
- Advantages
    - It can be implemented on an Operating System that does not support threads.
    - A user level thread does not require modification to operating systems.
    - **Simple Representation**: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
    - **Simple Management**: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
    - **Fast and Efficient**: Thread switching is not much more expensive than a pro- cedure call.
    - User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm.
- Disadvantages
    - There is a lack of coordination between threads and operating system ker- nel. Therefore, process as a whole gets one time slice

irrespective of wheth- er process has one thread or 1000 threads within. It is up to each thread to give up control to other threads.

- Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
- A user level thread requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will be blocked in the kernel, even if a sin- gle thread is blocked but other runnable threads are present. For example, if one thread causes a page fault, the whole process will be blocked.

Kernel Level Threads

- In this method, the kernel knows about threads and manages the threads.
- No runtime system is needed in this case.
- Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the tradi- tional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.
- Advantages
  - Because kernel has full knowledge of all threads, scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
  - Kernel threads do not require any new, non-blocking system calls. Blocking of one thread in a process will not affect the other threads in the same pro- cess as Kernel knows about multiple threads present so it will schedule other runnable thread.
- Disadvantages
  - The kernel level threads are slow and inefficient. As thread are managed by system calls, at considerably greater cost.
  - Since kernel must manage and schedule threads as well as processes. It re- quires a full thread control block (TCB) for each thread to maintain infor- mation about threads. As a result there is significant overhead and increased

in kernel complexity.

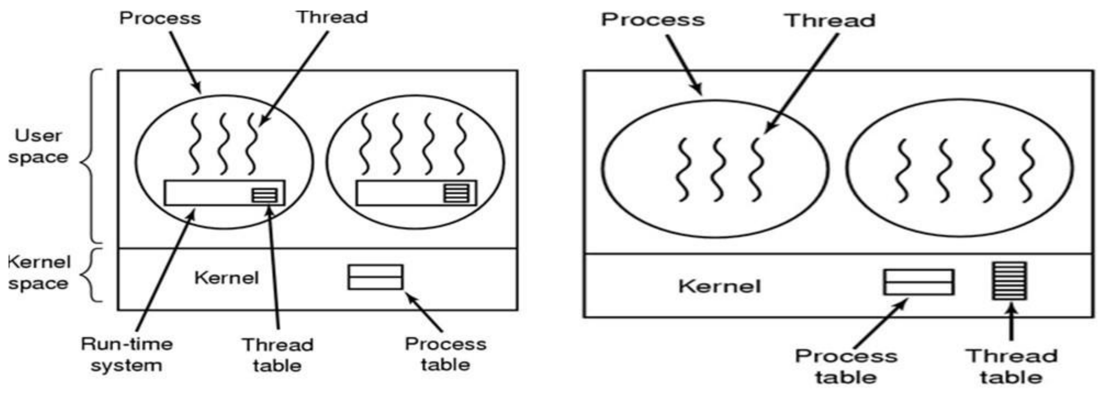


Figure 2-9. (a) A user-level threads package. (b) A threads package managed by the kernel.

Hybrid Implementations

- To combine the advantages of user-level threads with kernel-level threads, one way is to use the kernel-level threads and then multiplex user-level threads onto some
  or all of the kernel threads, as shown in Fig. 2-10.
- In this design, the kernel is aware of only the kernel-level threads and schedules those.
- Some of those threads may have multiple user-level threads multiplexed on top of them.
- These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capa- bility.
- In this model, each kernel-level thread has some set of user-level threads that take turns using it.
- This model gives the ultimate in flexibility as when this approach is used, the pro-
  grammer can determine how many kernel threads to use and how many user-level threads to multiplex on each one.

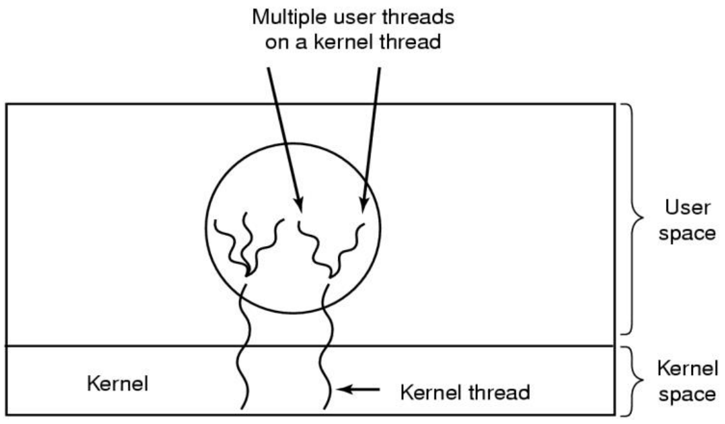


Figure 2-10. Multiplexing user-level threads onto kernel-level threads.

(9) *Write the short note on Multithreading and Multitasking.*

Multithreading

- The ability of an operating system to execute different parts of a program, called *threads* simultaneously, is called multithreading.
- The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other.
- On a single processor, multithreading generally occurs by time division multiplexing (as in multitasking) the processor switches between different threads.
- This context switching generally happens so speedy that the user perceives the threads or tasks as running at the same time.

Multitasking

- The ability to execute more than one *task* at the same time is called multitasking.
- In multitasking, only one CPU is involved, but it switches from one program to an- other so quickly that it gives the appearance of executing all of the programs at the same time.
- There are two basic types of multitasking.
  1) *Preemptive:* In preemptive multitasking, the operating system assign CPU *time slices* to each program.

2) *Cooperative:* In cooperative multitasking, each program can control the CPU for as long as it needs CPU. If a program is not using the CPU, however, it can allow another program to use it.

(10) Write the similarities and dissimilarities (difference) between process and thread.

Similarities

- Like processes threads share CPU and only one thread is active (running) at a time.
- Like processes threads within a process execute sequentially.
- Like processes thread can create children.
- Like a traditional process, a thread can be in any one of several states: running, blocked, ready, or terminated.
- Like process threads have Program Counter, stack, Registers and state.

Dissimilarities

- Unlike processes threads are not independent of one another, threads within the same process share an address space.
- Unlike processes all threads can access every address in the task.
- Unlike processes threads are design to assist one other. Note that processes might or might not assist one another because processes may be originated from different users.