MCA Part II

Paper-XIII

Topic: The Process

Prepared by: Dr. Kiran Pandey
(School of Computer Science)

Email-id: kiranpandey.nou@gmail.com

PROCESS CONCEPTS

A program in the execution is called a Process. Process is not the same as program. A process is more than a program code. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

Process memory is divided into four sections for efficient working :

- The text section is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The data section is made up the global and static variables, allocated and initialized prior to executing the main.
- The heap is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.

- The stack is used for local variables. Space on the stack is reserved for local variables when they are declared.

Formally, we can define a **process** is an executing program, including the current values of the program counter, registers, and variables. The subtle difference between a process and a program is that the program is a group of instructions whereas the process is the activity. In multiprogramming systems, processes are performed in a pseudo-parallelism as if each process has its own processor. In fact, there is only one processor but it switches back and forth from process to process. Henceforth, by saying execution of a process, we mean the processor's operations on the process like changing its variables, etc. and I/O work means the interaction of the process with the I/O operations like reading something or writing to somewhere. They may also be named as "processor (CPU) burst" and "I/O burst" respectively. According to these definitions, we classify programs as:

- **Processor- bound program**: A program having long processor bursts (execution instants)
- **I/O- bound program**: A program having short processor bursts.

The operating system works as the computer system software that assists hardware in performing process management functions. Operating system keeps track of all the active processes and allocates system resources to them according to policies devised to meet design performance objectives. To meet process requirements OS must maintain many data structures efficiently. The process abstraction is fundamental means for the OS to manage concurrent program execution. OS must interleave the execution of a number of processes to maximize processor use while providing reasonable response time. It must allocate resources to processes in conformance with a specific policy. In general, a process will need certain resources such as CPU time, memory, files, I/O devices etc. to accomplish its tasks. These resources are allocated to the process when it is created. A single processor may be shared among several processes with some scheduling algorithm being used to determine when to stop work on one process and provide service to a different one which we will discuss later in this unit. Operating systems must provide some way to create all the processes needed. In simple systems, it may be possible to have all

the processes that will ever be needed be present when the system comes up. In almost all systems however, some way is needed to create and destroy processes as needed during operations. In UNIX, for instant, processes are created by the fork system call, which makes an identical copy of the calling process. In other systems, system calls exist to create a process, load its memory, and start it running. In general, processes need a way to create other processes. Each process has one parent process, but zero, one, two, or more child processes. For an OS, the process management functions include:

- Process creation
- Termination of the process
- Controlling the progress of the process
- Process Scheduling
- Dispatching
- Interrupt handling / Exceptional handling
- Switching between the processes
- Process synchronization
- Inter-process communication support
- Management of Process Control Blocks.

Process States

As defined, a process is an independent entity with its own input values, output values, and internal state. A process often needs to interact with other processes. One process may generate some outputs that other process uses as input. For example, in the shell command

cat file1 file2 file3 | grep tree

- The first process, running cat, concatenates and outputs three files. Depending on the relative speed of the two processes, it may happen that grep is ready to run, but there is no input waiting for it. It must then block until some input is available. It is also possible for a process that is ready and able to run to be blocked because the operating system is decided to allocate

the CPU to other process for a while. A process state may be in one of the following:

- **New :** The process is being created.
- **Ready :** The process is waiting to be assigned to a processor
- **Running :** Instructions are being executed.
- **Waiting/Suspended/Blocked :** The process is waiting for some event to occur.
- **Halted/Terminated :** The process has finished execution.

The transition of the process states are shown in Figure 1.and their corresponding transition are described below:

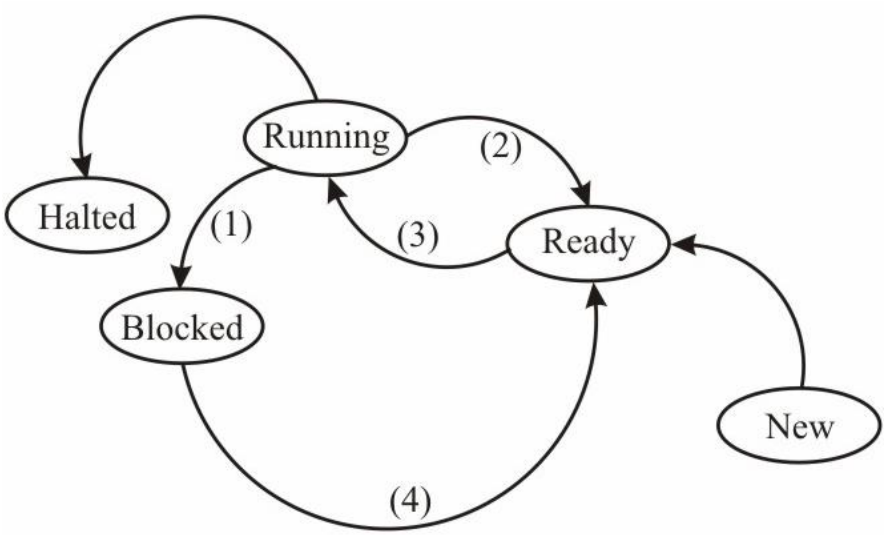

Figure 1: PROCESS STATES

As shown in Figure 1, four transitions are possible among the states. Transition 1 appears when a process discovers that it cannot continue. In order to get into blocked state, some systems must execute a system call block. In other systems, when a process reads from a pipe or special file 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. 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 other processes have had their share and it is time for the first process to run again.

Transition 4 appears when the external event for which a process was waiting was happened. If no other process is running at that instant, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in ready state for a little while until the CPU is available.

Using the process model, it becomes easier to think about what is going on inside the system. There are many processes like user processes, disk processes, terminal processes, and so on, which may be blocked when they are waiting for something to happen. When the disk block has been read or the character typed, the process waiting for it is unblocked and is ready to run again.

The process model, an integral part of an operating system, can be summarized as follows. The lowest level of the operating system is the scheduler with a number of processes on top of it. All the process handling, such as starting and stopping processes are done by the scheduler. More on the schedulers can be studied is the subsequent sections.

Process control block

To implement the process model, the operating system maintains a table, an array of structures, called the process table or process control block (PCB) or Switch frame. Each entry identifies a process with information such as process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information. In other words, it must contain everything about the process that must be saved when the process is switched from the running state to the ready state so that it can be restarted later as if it had never been stopped. The following is the information stored in a PCB. Each process is represented in the OS by a process control block. It is also by a process control block. It is also known as task control block.

A process control block contains many pieces of information associated with a specific process.

It includes the following information.

- **Process state:** The state may be new, ready, running, waiting or terminated state.
- **Process number**: each process is identified by its process number, called process ID;
- **Program counter:** it indicates the address of the next instruction to be executed for this purpose.
- **CPU registers:** The registers vary in number & type depending on the computer architecture. It includes accumulators, index registers, stack pointer & general purpose registers, plus any condition- code information must be saved when an interrupt occurs to allow the process to be continued correctly after- ward.
- **CPU scheduling information:** This information includes process priority pointers to scheduling queues & any other scheduling parameters.
- **Memory management information:** This information may include such information as the value of the bar & limit registers, the page tables or the segment tables, depending upon the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account number, job or process numbers and so on.
- **I/O Status Information:** This information includes the list of I/O devices allocated to this process, a list of open files and so on. The PCB simply serves as the repository for any information that may vary from process to process.
- List of open files.

Context switch

A *context switch* also sometimes referred to as a *process switch* or a *task switch* is the switching of the CPU (central processing unit) from one *process* or *thread* to another.

A process also sometimes referred to as a *task* is an *executing* i.e., running) instance of a program. In Linux, threads are lightweight processes that can run in parallel and share an *address space* i.e., a range of memory locations and other resources with their *parent* processes i.e., the processes that created them.

A *context* is the contents of a CPU's *registers* and *program counter* at any point in time. A register is a small amount of very fast memory inside of a CPU (as opposed to the slower RAM main memory outside of the CPU) that is used to speed the execution of computer programs by providing quick access to commonly used values, generally those in the midst of a calculation. A program counter is a specialized register that indicates the position of the CPU in its instruction sequence and which holds either the address of the instruction being executed or the address of the next instruction to be executed, depending on the specific system.

Context switching can be described in slightly more detail as the *kernel* (i.e., the core of the operating system) performing the following activities with regard to processes (including threads) on the CPU: (1) suspending the progression of one process and storing the CPU's *state* (i.e., the context) for that process somewhere in memory, (2) retrieving the context of the next process from memory and restoring it in the CPU's registers and (3) returning to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume the process.

A context switch is sometimes described as the kernel suspending *execution of one process* on the CPU and resuming *execution of some other process* that had previously been suspended. Although this wording can help clarify the concept, it can be confusing in itself because a process *is*, by definition, an executing instance of a program. Thus the wording *suspending progression of a process* might be preferable.

Context Switches and Mode Switches

Context switches can occur only in *kernel mode*. Kernel mode is a privileged mode of the CPU in which only the kernel runs and which provides access to all memory locations and all other system resources. Other programs, including applications, initially operate in *user mode*, but they can run portions of the kernel code via *system calls*. A system call is a request in a Unix-like operating system by an *active process* (i.e., a process currently progressing in the CPU) for a service performed by the kernel, such as *input/output* (I/O) or *process creation* (i.e., creation of a new process). I/O can be defined as any movement of information to or from the combination of the CPU and main memory (i.e. RAM), that is, communication between this combination and the computer's users (e.g., via the keyboard or mouse), its storage devices (e.g., disk or tape drives), or other computers.

The existence of these two modes in Unix-like operating systems means that a similar, but simpler, operation is necessary when a system call causes the CPU to shift to kernel mode. This is referred to as a *mode switch* rather than a context switch, because it does not change the current process.

Context switching is an essential feature of *multitasking* operating systems. A multitasking operating system is one in which multiple processes execute on a single CPU seemingly simultaneously and without interfering with each other. This illusion of *concurrency* is achieved by means of context switches that are occurring in rapid succession (tens or hundreds of times per second). These context switches occur as a result of processes voluntarily relinquishing their time in the CPU or as a result of the *scheduler* making the switch when a process has used up its CPU *time slice*.

A context switch can also occur as a result of a *hardware interrupt*, which is a signal from a hardware device (such as a keyboard, mouse, modem or system clock) to the kernel that an *event* (e.g., a key press, mouse movement or arrival of data from a network connection) has occurred.

Intel 80386 and higher CPUs contain hardware support for context switches. However, most modern operating systems perform *software context switching*,

which can be used on any CPU, rather than *hardware context switching* in an attempt to obtain improved performance. Software context switching was first implemented in Linux for Intel-compatible processors with the 2.4 kernel.

One major advantage claimed for software context switching is that, whereas the hardware mechanism saves almost all of the CPU state, software can be more selective and save only that portion that actually needs to be saved and reloaded. However, there is some question as to how important this really is in increasing the efficiency of context switching. Its advocates also claim that software context switching allows for the possibility of improving the switching code, thereby further enhancing efficiency, and that it permits better control over the validity of the data that is being loaded.

1. Get into kernel mode.
2. Quickly save the old program's registers somewhere, and the address of the next instruction it was about to execute, so they can be restored later. This storage area is known as a **Process Control Block**, or **PCB**, and it is stored in kernel-mode memory somewhere.
3. Save the mappings in the current page map, also into the PCB.
4. Unmap all of the old program's pages from the page map.
5. Choose a new program to re-start. Find its PCB.
6. Re-map the pages of the new program from its PCB.
7. Re-load the registers of the new program from its PCB.
8. Return to the next instruction of the new program, and return to user-mode at the same time.

Process Hierarchies

Modern general purpose operating systems permit a user to create and destroy processes.

- In Unix this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process.

- After a fork both parent and child keep running (indeed they have the *same* program text) and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.
- A process can *choose* to wait for children to terminate. For example, if C issued a wait() system call it would block until G finished.

Old or primitive operating system like MS-DOS are not multi-programmed so when one process starts another, the first process is *automatically* blocked and waits until the second is finished.

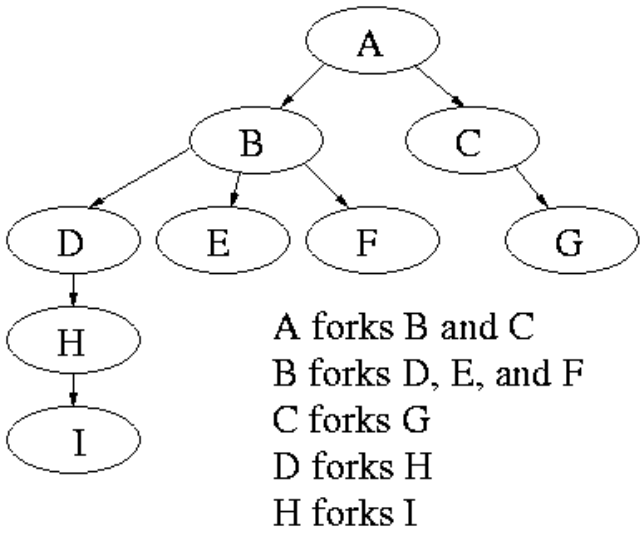

Figure2 : Process fork()

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 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. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.

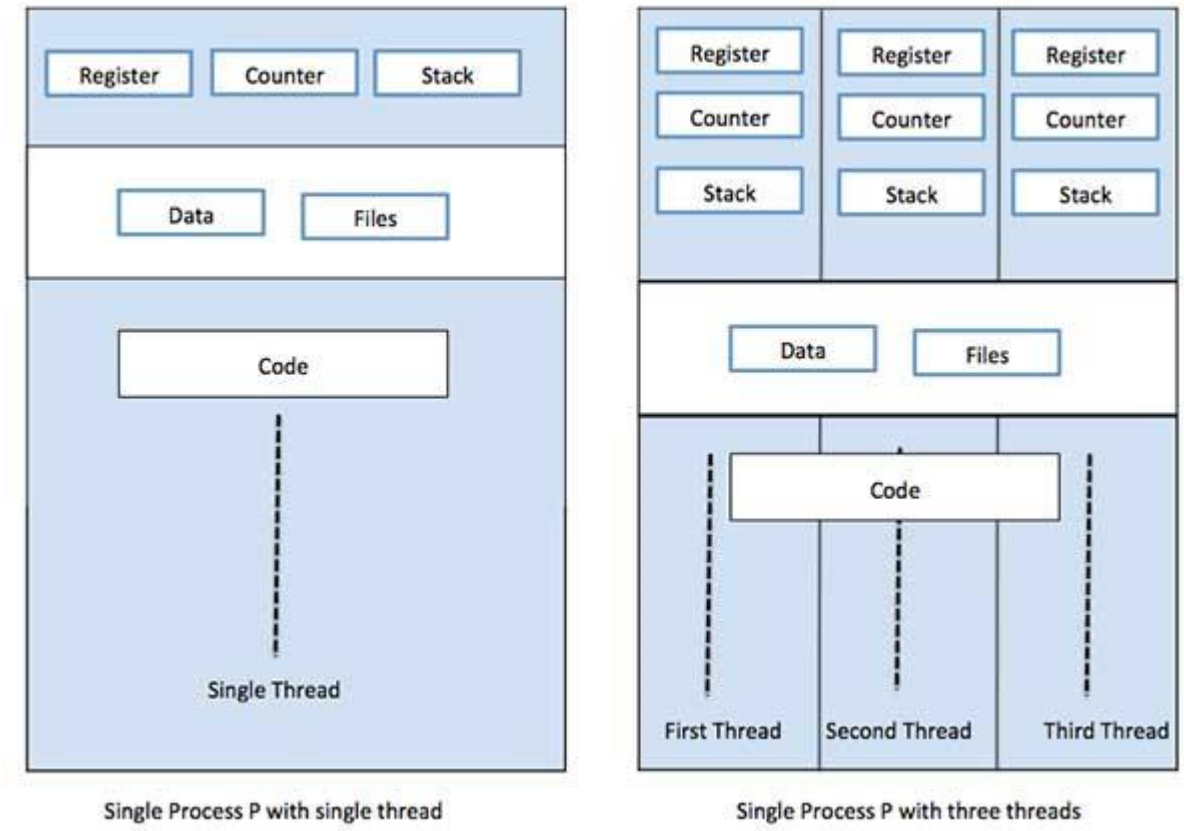

Figure 3: Threads

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.

Types of Thread

Threads are implemented in following two ways –

- **User Level Threads** – User managed threads.
- **Kernel Level Threads** – Operating System managed threads acting on kernel, an operating system core.

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.

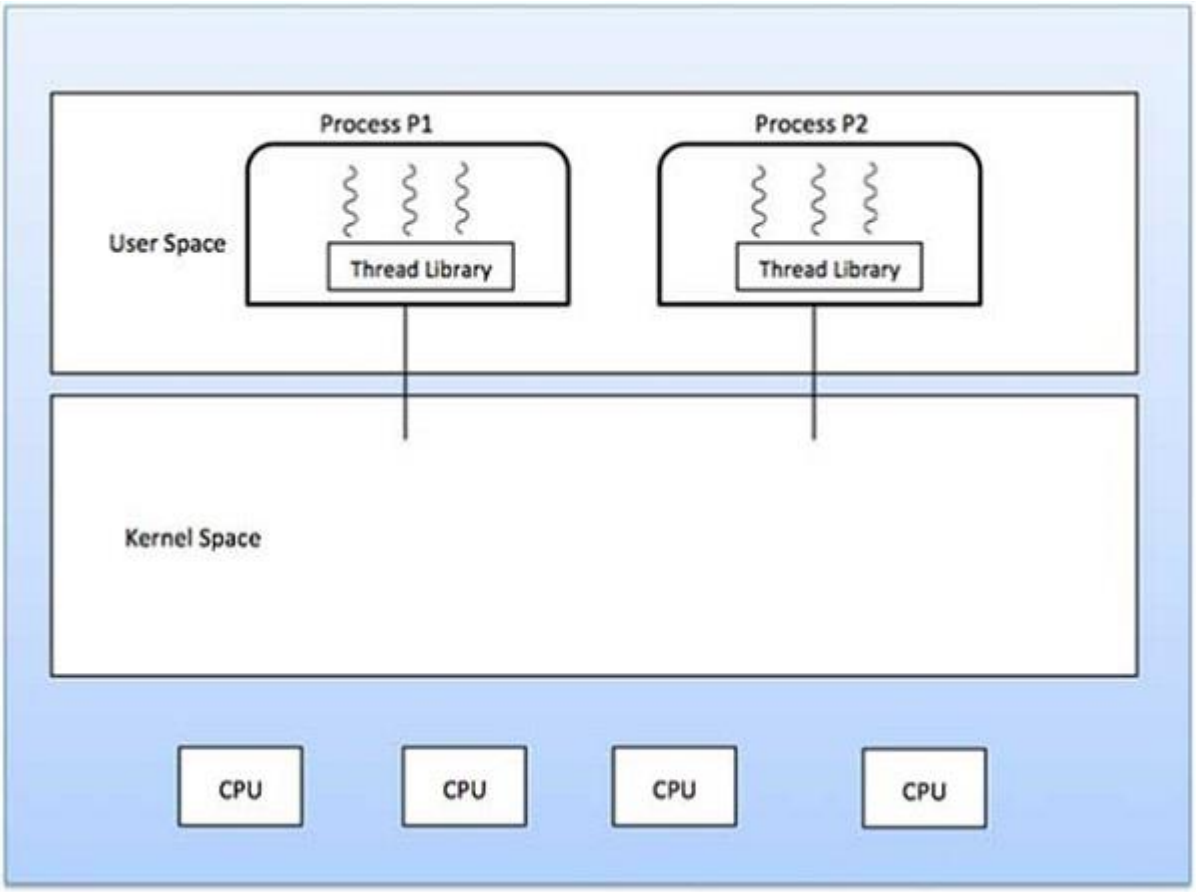

Figure 4: User level thread

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.

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.

The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level 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.

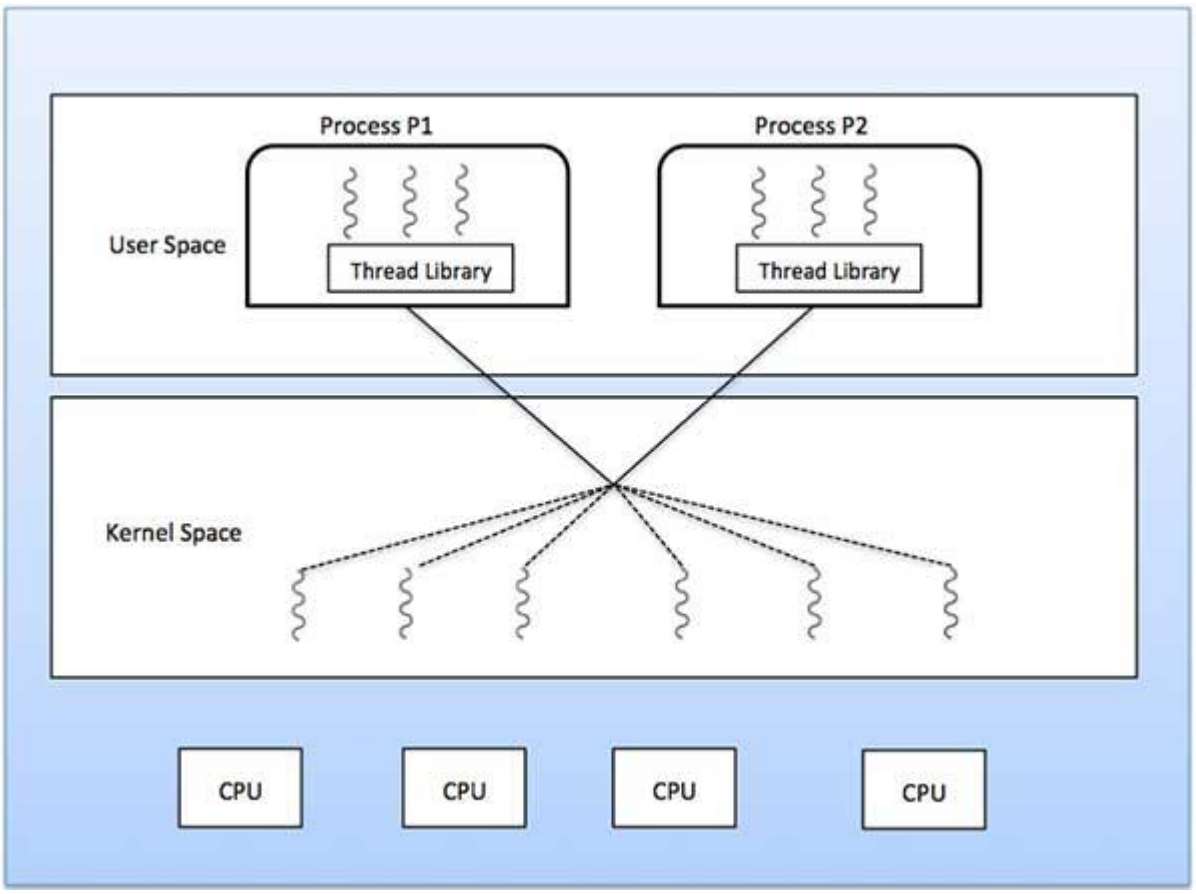

Figure 5: Many-to-many model

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.

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.

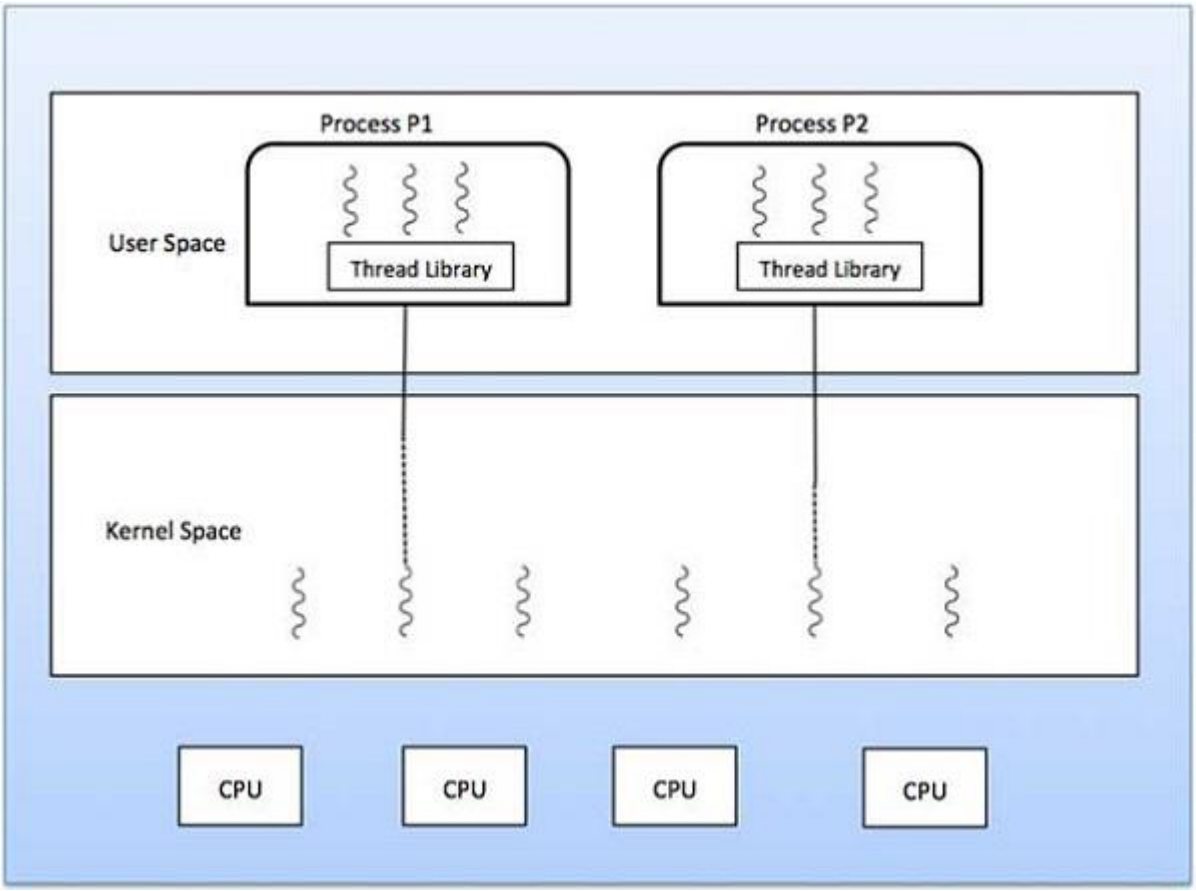

Figure 6: Many to one model

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.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

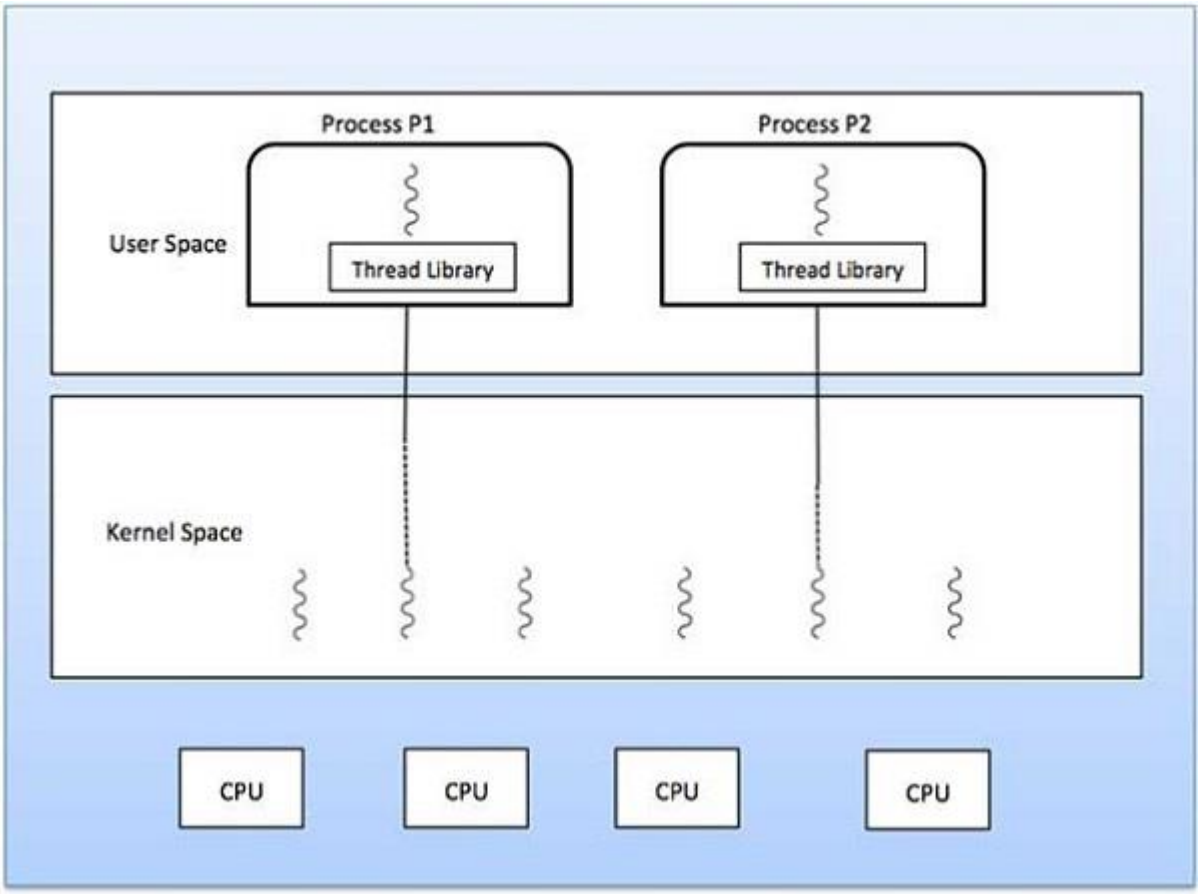

Figure 7: One to one model

Difference between User-Level & Kernel-Level Thread

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	Kernel routines themselves can

	advantage of multiprocessing.	be multithreaded.

Some important implementations of threads are:

- The Mach System / OSF1 (user and system level)
- Solaris 1 (user level) Solaris 2 (user and system level)
- OS/2 (system level only)
- NT threads (user and system level)
- IRIX threads
- POSIX standardized user threads interface

OPERATION ON PROCESS

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

Process creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique process identifier (or **pid),** which is typically an integer number. In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes-including pageout and fsflush. These processes are responsible for managing memory and file systems. The sched process also creates the ini t process, which serves as the root parent process for all user processes. In Figure 3.9, we see two children of ini t-inetd and dtlogin. inetd is responsible for networking services such as telnet and ftp; dtlogin is the process representing a user login screen. When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt_shel process. Below sdLshel, a user's command-line shell-the C-shell or csh-is created. In this command-line interface, the user can then invoke various child processes, such as the ls and cat commands. We also see a csh process with pid of 7778 representing a user who has logged onto the system using telnet. This user has started the Netscape browser (pid of 7785) and the emacs editor (pid of 8105).(The tree of process on a typical Solaris system is shown below in figure 11).

On UNIX, we can obtain a listing of processes by using the ps command. For example, the command ps -el will list complete information for all processes currently active in the system.

In general, a process will need certain resources (CPU time, memory, files, I/0 devices) to accomplish its task. When a process creates a sub process, that sub process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many sub processes.

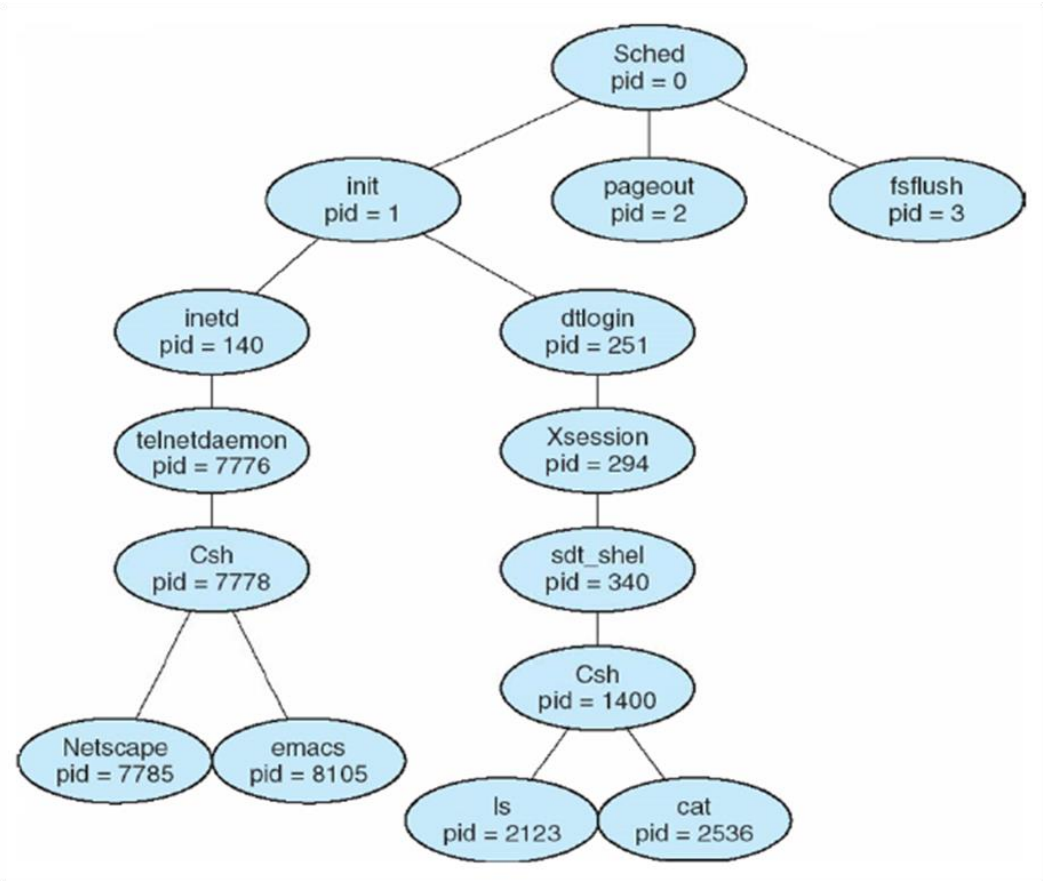

Figure 8: A tree of process on atypical Solaris system

In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process. For example, consider a process whose function is to display the contents of a file-say, *img.jpg-on* the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file *img.jpg,* and it will use that file name, open the file, and write the contents out. It may also get the name of the output device. Some operating systems pass resources to child processes. On

such a system, the new process may get two open files, *img.jpg* and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities exist in terms of execution:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

- The child process is a duplicate of the parent process (it has the same program and data as the parent).

- The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork () , with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the exec() system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program. The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue await() system call to move itself off the ready queue until the termination of the child.

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system calL At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process-including physical and virtual memory, open files, and I/0 buffers-are deallocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess () in Win32). Usually, such a system call can be invoked only by

the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)

- The task assigned to the child is no longer required.

- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems, including VMS, do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in UNIX, we can terminate a process by using the exit() system call; its parent process may wait for the termination of a child process by using the wait() system call. The wait() system call returns the process identifier of a terminated child so that the parent can tell which of its children has terminated. If the parent terminates, however, all its children have assigned as their new parent the init process. Thus, the children still have a parent to collect their status and execution statistics.

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: If we want a particular task to run faster, we must 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 elements (such as CPUs or I/O channels).

Modularity: We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Convenience: Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, 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: (1) shared memory and message passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure

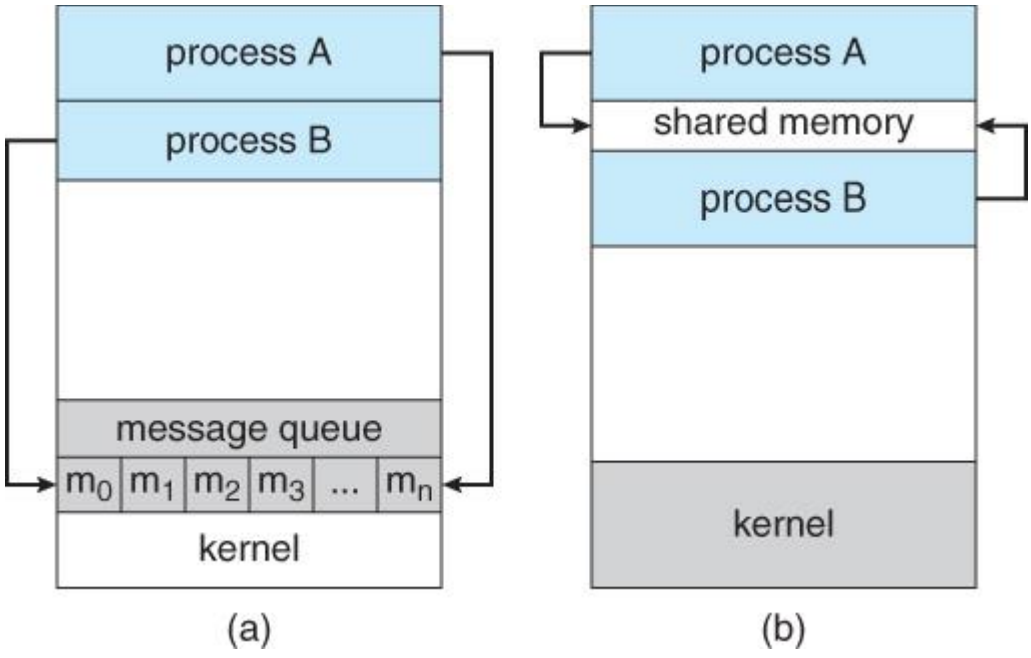

Figure 9: Communication models: (a) Message passing

(b) Shared memory

Both of the models discussed above are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to

implement than is shared memory for inter-computer communication. Shared memory allows maximum speed and convenience of communication. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Questions

1. What is a process?

2. How process is different from a thread?

3. Explain different states of a process with a diagram.

4. What is context switch. Explain the reason why many systems use two level scheduling algorithms.