Unit I

Chapter 2: Process and Threads

Introduction:

Processes are one of the oldest and most important abstractions that operating systems provide. They support the ability to have (pseudo) simultaneous operation even when there is only one CPU available. They turn a single CPU into multiple virtual CPUs. Without the process abstraction, modern computing could not exist.

PROCESSES

All latest computers frequently do various things at the same time. People used to working with personal computers may not be fully aware of this fact, so a few examples may make the point clearer. First examine a Web server. Requests come in from all over asking for Web pages. When a request comes in, the server checks to see if the page needed is in the cache. If it is, it is sent back; if it is not, a disk request is started to fetch it. However, from the CPU's point of view, disk requests take a very long time. While waiting for the disk request to complete, many more requests may come in. If there are multiple disks present, some or all of them may be fired off to other disks long before the first request is satisfied. Obviously some way is required to model and control this concurrency. Processes (and especially threads) can help here.

Now examine a user PC. When the system is booted, many processes are secretly started, often unknown to the user. For instance, a process may be started up to wait for incoming e-mail. Another process may run on behalf of the antivirus program to check from time to time if any new virus definitions are available. In addition, explicit user processes may be running, printing files and burning a CDROM, all while the user is surfing the Web. All this activity has to be managed, and a multiprogramming system supporting multiple processes comes in very useful here.

In any multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is running only one process, in the course of 1 second, it may work on several of them, giving the false impression of parallelism. Sometimes people

speak of pseudoparallelism in this perspective, to compare it with the true hardware parallelism of multiprocessor systems (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have developed a conceptual model (sequential processes) that makes parallelism easier to handle.

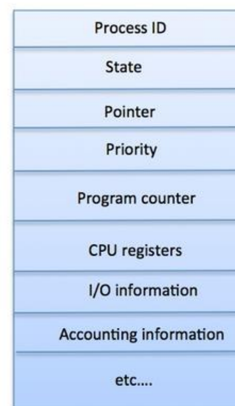
Process Control Block (PCB)

- A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

S.N.	Information & Description
1	Process State The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	Process privileges This is required to allow/disallow access to system resources.
3	Process ID Unique identification for each of the process in the operating system.
4	Pointer A pointer to parent process.
5	Program Counter Program Counter is a pointer to the address of the next instruction to be executed for this process.

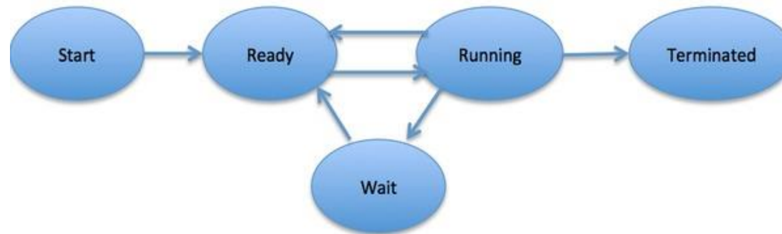
6	CPU registers Various CPU registers where process need to be stored for execution for running state.
7	CPU Scheduling Information Process priority and other scheduling information which is required to schedule the process.
8	Memory management information This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	Accounting information This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	IO status information This includes a list of I/O devices allocated to the process.

- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



Process Life Cycle

- When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.
- In general, a process can have one of the following five states at a time.



S.N.	State & Description
1	Start This is the initial state when a process is first started/created.
2	Ready The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	Running Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	Waiting Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

5	Terminated or Exit Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

When an operating system is booted, normally various 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 specific users, but instead have some particular function. For instance, one background process may be designed to accept incoming e-mail, sleeping most of the day but suddenly springing to life when incoming e-mail arrives. Another 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 e-mail, Web pages, news, printing, and so on are called daemons. Large systems generally have dozens of them. In UNIX, the ps program can be used to list the running processes. In Windows, the task manager can be used.

Process Creation

Operating systems require some way to make processes. In very simple systems, or in systems designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is required to create and finish processes as required during operation. We will now look at some of the issues.

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 performs whatever its job is. However, nothing lasts forever, not even processes. Sooner or later the new process will end, generally 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 connected in certain ways. The child process can itself create more processes, forming a process hierarchy. Note that unlike plants and animals that use sexual reproduction, a process has only one parent (but zero, one, two, or more children).

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently connected with the keyboard (generally all active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

Implementation of Processes

To implement the process model, the operating system maintains a table (an array of structures), called the process table, with one entry per process. (Some authors call these entries process control blocks.) This entry includes important information about the process state, containing its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped.

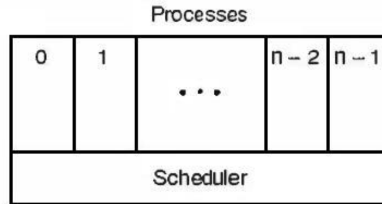


Figure (a). The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

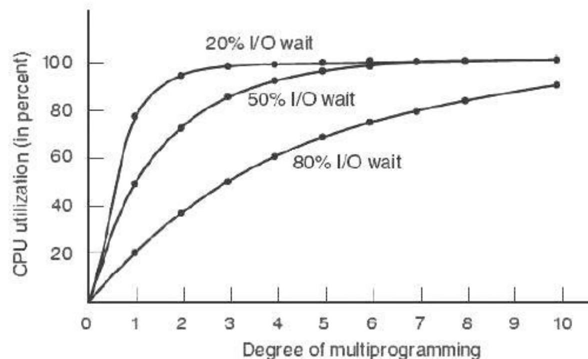
Modeling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20% of the time it is sitting in memory, with five processes in memory at once, the CPU should be busy all the time. This model is unrealistically hopeful, however, since it tacitly assumes that all five processes will never be waiting for I/O at the same time.

A better model is to look at CPU usage from a probabilistic viewpoint. Assume that a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is then given by the formula

$$\text{CPU utilization} = 1 - p^n$$

The following figure shows the CPU utilization as a function of n, which is called the degree of multiprogramming.



CPU utilization as a function of the number of processes in memory

From the figure it is clear that if processes spend 80% of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10%. When

you understand that an interactive process waiting for a user to type something at a terminal is in I/O wait state, it should be clear that I/O wait times of 80% and more are not abnormal. But even on servers, processes doing a lot of disk I/O will sometimes have this percentage or more.

What is Thread?

- A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.
- A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

Thread Usage

Why would anyone want to have a kind of process within a process? It turns out there are various reasons for having these miniprocesses, called threads. Let us now look at some of them. The major reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

In usual operating systems, each process has an address space and a single thread of control. In reality, that is almost the definition of a process. However, there are often situations in which it is desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were (almost) separate processes (except for the shared address space). In the following sections we will talk about these situations and their implications.

Thread Usage

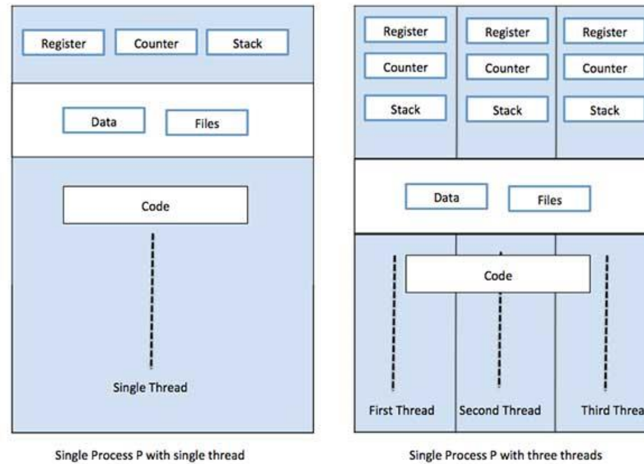
Why would anyone want to have a kind of process within a process? It turns out there are various reasons for having these miniprocesses, called threads. Let us now look at some of them. The major reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

We have seen this argument before. It is specifically the argument for having processes. Instead of thinking about interrupts, timers, and context switches, we can think about parallel processes. Only now with threads we add a new element: the ability for the parallel entities to share an address space and all of its data among themselves. This ability is necessary for certain applications, which is why having multiple processes (with their separate address spaces) will not work.

A second argument for having threads is that since they are lighter weight than processes, they are easier (i.e., faster) to create and destroy than processes. In many systems, creating a thread goes 10-100 times faster than creating a process. When the number of threads required changes dynamically and rapidly, this property is useful to have.

A third reason for having threads is also a performance argument. Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, hence speeding up the application.

Finally, threads are useful on systems with multiple CPUs, where real parallelism is possible. We will come back to this issue in "MULTIPLE PROCESSOR SYSTEMS".



Single Process P with single thread   Single Process P with three threads

Difference between Process and Thread

1. Threads are easier to create than processes since they don't require a separate address space.

2. Multithreading requires careful programming since threads share data structures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.

3. Threads are considered lightweight because they use far less resources than processes.

4. Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other.

5. A process can consist of multiple threads.

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but	All threads can share same set of open files, child processes.

	has its own memory and file resources.	
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

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.

Disadvantages

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

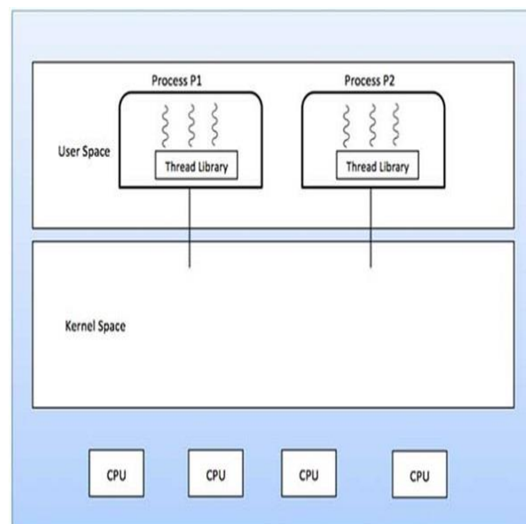
Types of Thread

- Threads are implemented in following two ways −
- 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.

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-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 advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

One way to organize the Web server is shown in Fig. 2(a). Here one thread, the dispatcher, reads incoming requests for work from the network. After examining the request, it chooses an idle (i.e., blocked) worker thread and hands it the request, possibly by writing a pointer to the message into a special word linked with each thread. The dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state.

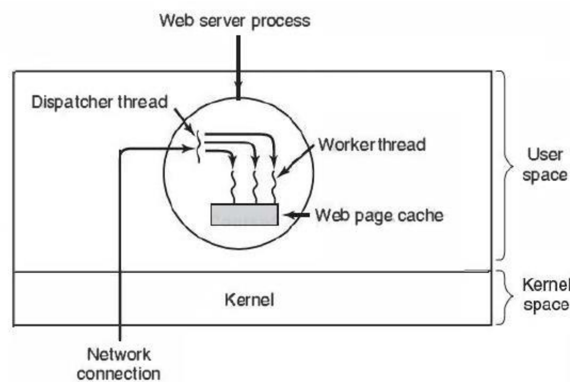


Figure 2. A multithreaded Web server

When the worker wakes up, it checks to see if the request can be satisfied from the Web page cache, to which all threads have access. If not, it starts a read operation to get the page from the disk and blocks until the disk operation completes. When the thread

blocks on the disk operation, another thread is chosen to run, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

The Classical Thread Model

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 multiple processes running in parallel in one computer. In the former case, the threads share an address space and other resources. In the latter case, processes share physical memory, disks, printers, and other resources. Because threads have some of the properties of processes, they are sometimes called lightweight processes. The term multithreading is also used to explain the situation of allowing various threads in the same process.

In Fig.1.(a) we see three usual processes. Each process has its own address space and a single thread of control. On the contrary, in Fig.1.(b) we see a single process with three threads of control. Though in both cases we have three threads, in Fig.1.(a) each of them operates in a different address space, whereas in Fig.1.(b) all three of them share the same address space.

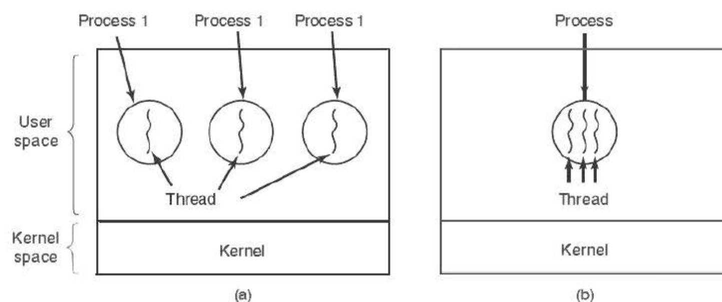


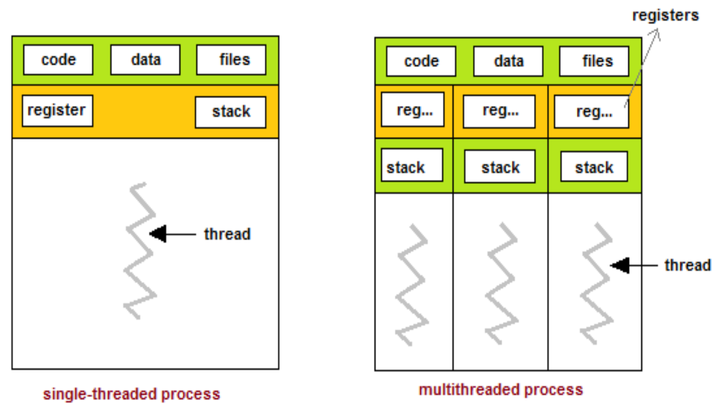
Figure 1. (a) Three processes each with one thread. (b) One process with three threads

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

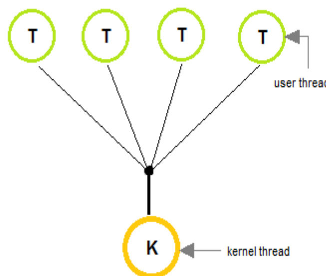
- Many to many relationship.
- Many to one relationship.
- One to one relationship.



single-threaded process multithreaded process

Many-To-One Model

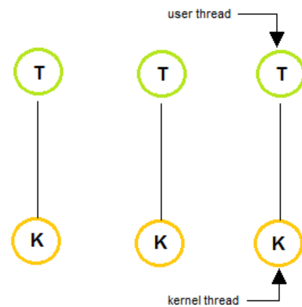
- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is efficient in nature.



One-To-One Model

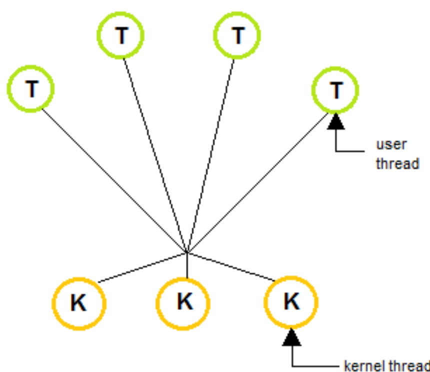
- The one-to-one model creates a separate kernel thread to handle each and every user thread.

- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.



Thread Libraries

- Thread libraries provides programmers with API for creating and managing of threads.
- Thread libraries may be implemented either in user space or in kernel space. The user space involves API functions implemented solely within user space, with no kernel support. The kernel space involves system calls, and requires a kernel with thread library support.

There are three types of thread :

- POSIX Pthreads, may be provided as either a user or kernel library, as an extension to the POSIX standard.
- Win32 threads, are provided as a kernel-level library on Windows systems.
- Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Implementing Threads in the Kernel

The kernel's thread table holds each thread's registers, state, and other information. The information is the same as with user-level threads, but now kept in the kernel instead of in user space (inside the run-time system). This information is a subset of the information that traditional kernels keep up about their single threaded processes, that is, the process state. In addition, the kernel also maintains the traditional process table to keep track of processes.

Benefits of Multithreading

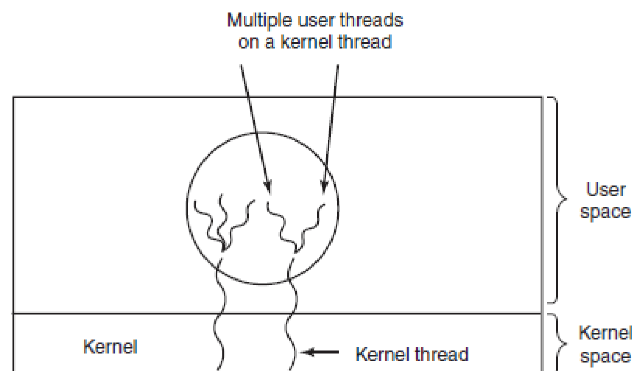
- Responsiveness
- Resource sharing, hence allowing better utilization of resources.
- Economy. Creating and managing threads becomes easier.
- Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.
- Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

Multithreading Issues

- Thread Cancellation. Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is Asynchronous cancellation, which terminates the target thread immediately. The other is Deferred cancellation allows the target thread to periodically check if it should be cancelled.
- Signal Handling. Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.
- fork() System Call. fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied or not?
- Security Issues because of extensive sharing of resources between multiple threads.

Hybrid Implementations

- Various ways have been investigated to try to combine the advantages of user level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of them.
- When this approach is used, the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one. This model gives the ultimate in flexibility.



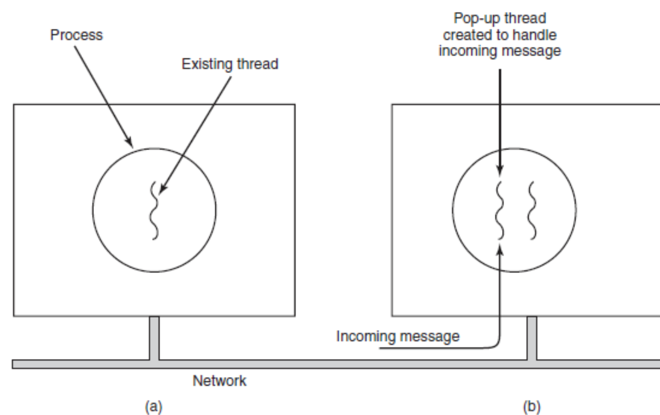
Scheduler Activations

- The goals of the scheduler activation work are to mimic the functionality of kernel threads, but with the better performance and greater flexibility.
- User threads should not have to make special nonblocking system calls or check in advance if it is safe to make certain system calls. Nevertheless, when a thread blocks on a system call or on a page fault, it should be possible to run other threads within the same process, if any are ready.
- When scheduler activations are used, the kernel assigns a certain number of virtual processors to each process and lets the (user-space) run-time system allocate threads to processors.

Pop-Up Threads

- Threads are frequently useful in distributed systems. An important example is how incoming messages, for example requests for service, are handled.

- When a message arrives, it accepts the message, unpacks it, examines the contents, and processes it.
- However, a completely different approach is also possible, in which the arrival of a message causes the system to create a new thread to handle the message. Such a thread is called a pop-up thread
- A key advantage of pop-up threads is that since they are brand new, they do not have any history— registers, stack, whatever—that must be restored. Each one starts out fresh and each one is identical to all the others.
- This makes it possible to create such a thread quickly. The new thread is given the incoming message to process. The result of using pop-up threads is that the latency between message arrival and the start of processing can be made very short.



INTERPROCESS COMMUNICATION

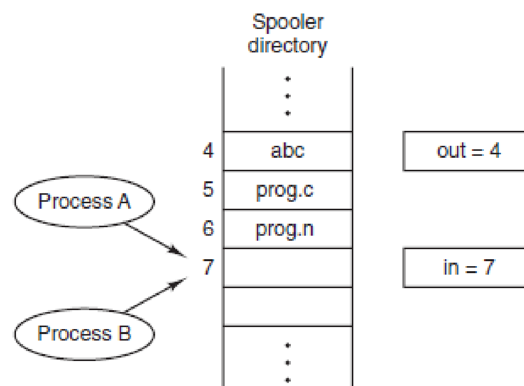
- Processes frequently need to communicate with other processes.
- Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts.
- Very briefly, there are three issues here. The first was alluded to above: how one process can pass information to another. The second has to do with making sure two or more processes do not get in each other’s way, for example, two processes in an airline reservation system each trying to grab the last seat on a

plane for a different customer. The third concerns proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print.

- It is also important to mention that two of these issues apply equally well to threads. The first one—passing information—is easy for threads since they share a common address space (threads in different address spaces that need to communicate fall under the heading of communicating processes).

Race Conditions

- To see how interprocess communication works in practice, let us now consider a simple but common example:
- At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes *A* and *B* decide they want to queue a file for printing.



Critical Regions

- How do we avoid race conditions? The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is mutual exclusion, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

- 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 races. We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block any process.
4. No process should have to wait forever to enter its critical region.

Mutual Exclusion with Busy Waiting

- Disabling Interrupts: On a single-processor system, 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.

Synchronization Hardware

- Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.
- In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

- Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.
- This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

- As the synchronization hardware solution is not easy to implement fro everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.
- As the resource is locked while a process executes its critical section hence no other process can access it.

Semaphores

- In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called semaphore. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P() and V() respectively.
- The classical definition of wait and signal are :
- Wait: decrement the value of its argument S as soon as it would become non-negative.
- Signal: increment the value of its argument, S as an individual operation.

Properties of Semaphores

- Simple
- Works with many processes

- Can have many different critical sections with different semaphores
- Each critical section has unique access semaphores
- Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

- Semaphores are mainly of two types:
- Binary Semaphore: It is a special form of semaphore used for implementing mutual exclusion, hence it is often called *Mutex*. A binary semaphore is initialized to 1 and only takes the value 0 and 1 during execution of a program.
- Counting Semaphores: These are used to implement bounded concurrency.

Limitations of Semaphores

- Priority Inversion is a big limitation os semaphores.
- Their use is not enforced, but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called Deadlock. We will be studying deadlocks in details in coming lessons.

SCHEDULING

When a computer is multiprogrammed, it often has multiple processes or threads competing for the CPU simultaneously. This situation happens whenever two or more of them are in the ready state at the same time. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the scheduler and the algorithm it uses is called the scheduling algorithm. These topics form the subject matter of the following sections.

Many of the same problems that apply to process scheduling also apply to thread scheduling, though some are different. When the kernel manages threads,

scheduling is generally done per thread, with little or no regard to which process the thread belongs. At first we will focus on scheduling issues that apply to both processes and threads. Later on we will explicitly look at thread scheduling and some of the unique issues it raises.

Process Scheduling

- The act of determining which process in the ready state should be moved to the running state is known as Process Scheduling.
- The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.
- Schedulers fell into one of the two general categories :
- Preemptive scheduling: The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.

  Non-Preemptive scheduling: When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.

CPU Scheduling

- CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

Scheduling Criteria

- There are many different criteria to check when considering the "best" scheduling algorithm :

- CPU utilization: To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
- Throughput: It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.
- Turnaround time: It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).
- Waiting time: The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.
- Load average: It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.
- Response time: Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).
- In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling Queues

- All processes when enters into the system are stored in the job queue.
- Processes in the Ready state are placed in the ready queue.
- Processes waiting for a device to become available are placed in device queues. There are unique device queues for each I/O device available.

Types of Schedulers

- There are three types of schedulers available :
- Long Term Scheduler :Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution. Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.
- Short Term Scheduler :This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.
- Medium Term Scheduler :During extra load, this scheduler picks out big processes from the ready queue for some time, to allow smaller processes to execute, thereby reducing the number of processes in the ready queue.

Categories of Scheduling Algorithms

Not amazingly, in different environments different scheduling algorithms are required. This situation happens because different application areas (and different kinds of operating systems) have different objectives. In other words, what the scheduler should optimize for is not the same in all systems. Three environments worth distinguishing are

1. Batch.
2. Interactive.
3. Real time.

Batch systems are still in extensive use in the business world for doing payroll, inventory, accounts receivable, accounts payable, interest calculation (at banks), claims processing (at insurance companies), and other periodic tasks. In batch systems, there are no users anxiously waiting at their terminals for a quick response to a short request.

As a result, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process, are frequently acceptable. This approach reduces process switches and hence improves performance. The batch algorithms are in fact fairly general and often applicable to other situations as well, which makes them worth studying, even for people not involved in corporate mainframe computing.

In an environment with interactive users, preemption is necessary to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, one process might shut out all the others indefinitely due to a program bug. Preemption is required to prevent this behavior. Servers also fall into this category, since they usually serve multiple (remote) users, all of whom are in a big hurry.

In systems with real-time restrictions, preemption is, oddly enough, sometimes not required because the processes know that they may not run for long periods of time and generally do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further the application at hand. Interactive systems are general purpose and may run arbitrary programs that are not cooperative or even malicious.

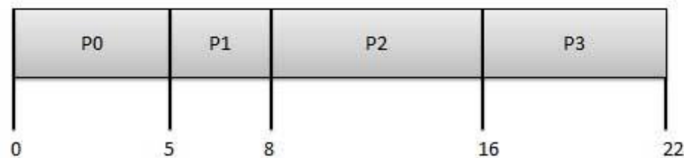
Scheduling in Batch Systems

- There are six popular process scheduling algorithms which we are going to discuss in this chapter −
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows –

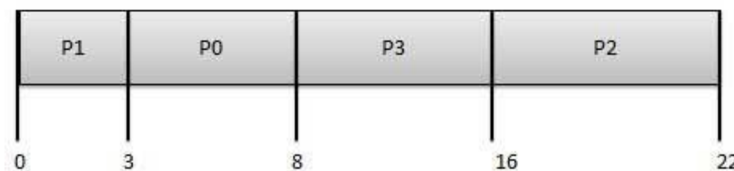
Process	Wait Time : Service Time - Arrival Time
P0	**0 - 0 = 0**
P1	**5 - 1 = 4**
P2	**8 - 2 = 6**
P3	**16 - 3 = 13**

Average Wait Time: (0+4+6+13) / 4 = 5.75

Shortest Job Next (SJN)

- This is also known as shortest job first, or SJF
- This is a non-preemptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	3
P1	1	3	0
P2	2	8	16
P3	3	6	8



Wait time of each process is as follows

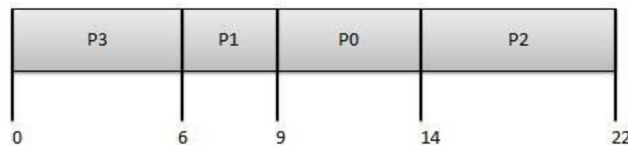
Process	**Wait Time : Service Time - Arrival Time**
P0	**3 - 0 = 3**
P1	**0 - 0 = 0**
P2	**14 - 2 = 12**
P3	**8 - 3 = 5**

Average Wait Time: (3+0+12+5) / 4 = 5

Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0



Wait time of each process is as follows

Process	Wait Time : Service Time - Arrival Time
P0	**9 - 0 = 9**
P1	**6 - 1 = 5**
P2	**14 - 2 = 12**
P3	**0 - 0 = 0**

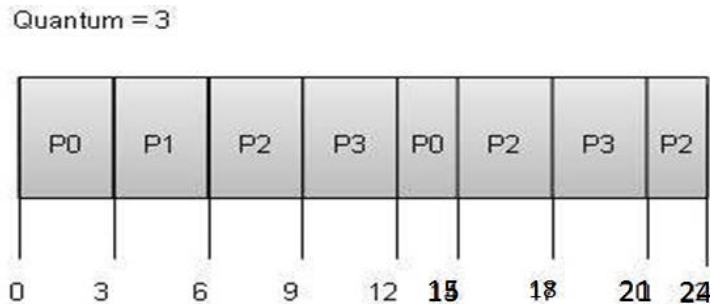
Average Wait Time: (9+5+12+0) / 4 = 6.5

Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.



Wait time of each process is as follows

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (15 - 9) + (21 - 17) = 14
P3	(9 - 3) + (18 - 12) = 12

Average Wait Time: (9+2+14+12) / 4 = 9.25

Multiple-Level Queues Scheduling

- Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.
- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.
- For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Thread Scheduling

When many processes each have multiple threads, we have two levels of parallelism present: processes and threads. Scheduling in such systems differs considerably depending on whether user-level threads or kernel-level threads (or both) are supported.

Let us examine user-level threads first. Since the kernel is not aware of the existence of threads, it functions as it always does, picking a process, say, A, and giving A control for its quantum. The thread scheduler inside A determines which thread to run, say A1. Since there are no clock interrupts to multiprogram threads, this thread may continue running as long as it wants to. If it uses up the process entire quantum, the kernel will choose another process to run.

When the process A finally runs again, thread A1 will resume running. It will continue to consume all of A's time until it comes to an end. On the other hand its antisocial

behavior will not affect other processes. They will get whatever the scheduler considers their appropriate share, no matter what is going on inside process A.

A main difference between user-level threads and kernel-level threads is the performance. Doing a thread switch with user-level threads takes a handful of machine instructions. With kernel-level threads it requires a full context switch, changing the memory map and invalidating the cache, which is several orders of magnitude slower. However, with kernel-level threads, having a thread block on I/O does not suspend the complete process as it does with user-level threads.

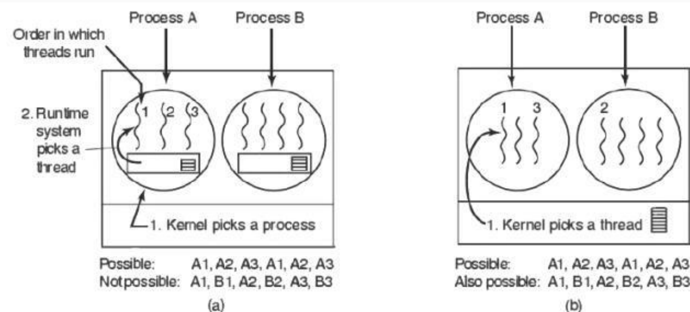


Figure 1. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Since the kernel knows that switching from a thread in process A to a thread in process B is more expensive than running a second thread in process A (due to having to change the memory map and having the memory cache spoiled), it can consider this information when making a decision. For instance, given two threads that are otherwise equally important, with one of them belonging to the same process as a thread that just blocked and one belonging to a different process, preference could be given to the former.

IPC Problems:

Classical Problem of Synchronization

- Bounded Buffer Problem : This problem is generalized in terms of the Producer-Consumer problem.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.

The Readers Writers Problem

- In this problem there are some processes(called readers) that only read the shared data, and never change it, and there are other processes(called writers) who may change the data in addition to reading or instead of reading it.
- There are various type of the readers-writers problem, most centered on relative priorities of readers and writers

Dining Philosophers Problem

- The dining philosopher's problem involves the allocation of limited resources from a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks kept beside them and a bowl of rice in the center, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Dijkstra's algorithm

- Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.
- Dijkstra's algorithm to find the shortest path between *a* and *b*. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.