BTCOC402

Operating System

Teaching Notes

Lecture Number	Topic to be covered (Unit 2)
	Introduction (7 Hrs)
1	Process Concept, Process Scheduling, Operation on process
2	Inter-process Communication, Cooperating processes, Threads,
3	Multithreading model, Scheduling criteria,
4	Scheduling Algorithms, Thread Scheduling,
5	Multiple-Processor Scheduling, Scheduling Algorithms evaluation.

: Submitted by:

Prof. A. S. Aher

Operating System

Unit 2:- PROCESSES AND CPU SCHEDULING

Process Concept:

- The concept of a **process** helps us understand how programs execute in an operating system.
- A process is an execution of a program using a set of resources. We emphasize “an” because several executions of the same program may be present in the operating system at the same time; these executions constitute different processes.
- That happens when users initiate independent executions of the program, each on its own data.
- It also happens when a program that is coded with concurrent programming techniques is being executed. The kernel allocates resources to processes and schedules them for use of the CPU.
- This way, it realizes execution of sequential and concurrent programs uniformly.
- A thread is also an execution of a program but it functions in the environment of a process—that is, it uses the code, data, and resources of a process.
- It is possible for many threads to function in the environment of the same process; they share its code, data, and resources.

- Switching between such threads requires less overhead than switching between processes.
- Process is a program in execution. This is a typical definition of a process. A process does not mean only program but it could contain some part called as text section.
- It may also contain the current activity, represented by the value of the program counter and the contents of CPU register.
- The main difference between program and process is that program is user written and process is generated by the operating system to run small part of the program.
- The program is converted into different processes to execute by the CPU. Multiple processes represent a program at the time of execution.
- Another part of process is a stack. Stack is basically used to store the temporary values. The temporary values can be parameter or return values of functions, local variables, addresses of the return values, etc.
- There is another part called data section. Data section stores the global variable. Global variables are accessible throughout the program.
- Sometimes it is necessary to use dynamic memory allocation while executing program. The Heap is a part which is used for memory allocation.
- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched. The figure below depicts the memory state.

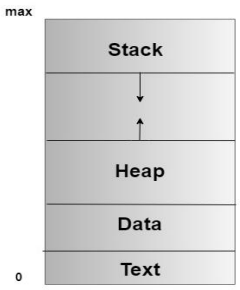


Figure: Process in the Memory

- Process is also refereed as an active entity. A program cannot be directly referred as a process e.g. a file contains the instructions or commands given to machine.
- This file is present on the disk. The file called as executable file. It cannot be directly said that this executable file is a process.
- In case when executable file gets loaded into memory and start executing instructions one by one using program counter, it can be called as process.
- Two common techniques to run executable files are double clicking on the file name icon and run it or run it on the command prompt by inputting its name.
- There could be two processes associated with the same program. But remember that these two processes are treated as separate processes.
- This may happen that same user may run multiple copies of a same program at a time .e.g.: One user may load multiple copies of web browser or mail at a time. Though the program is same, its different copies will be created.
- For all these copies different processes will be created in the memory. Processes created are same but there is difference in the sections of processes. The text section will be same but other sections like stack, data and heap will be different.
- There is one more case that when a process runs in the memory it may generate multiple processes.
- A Process has various attributes associated with it. Some of the attributes of a Process are:
- **Process Id:** Every process will be given an id called Process Id to uniquely identify that process from the other processes.

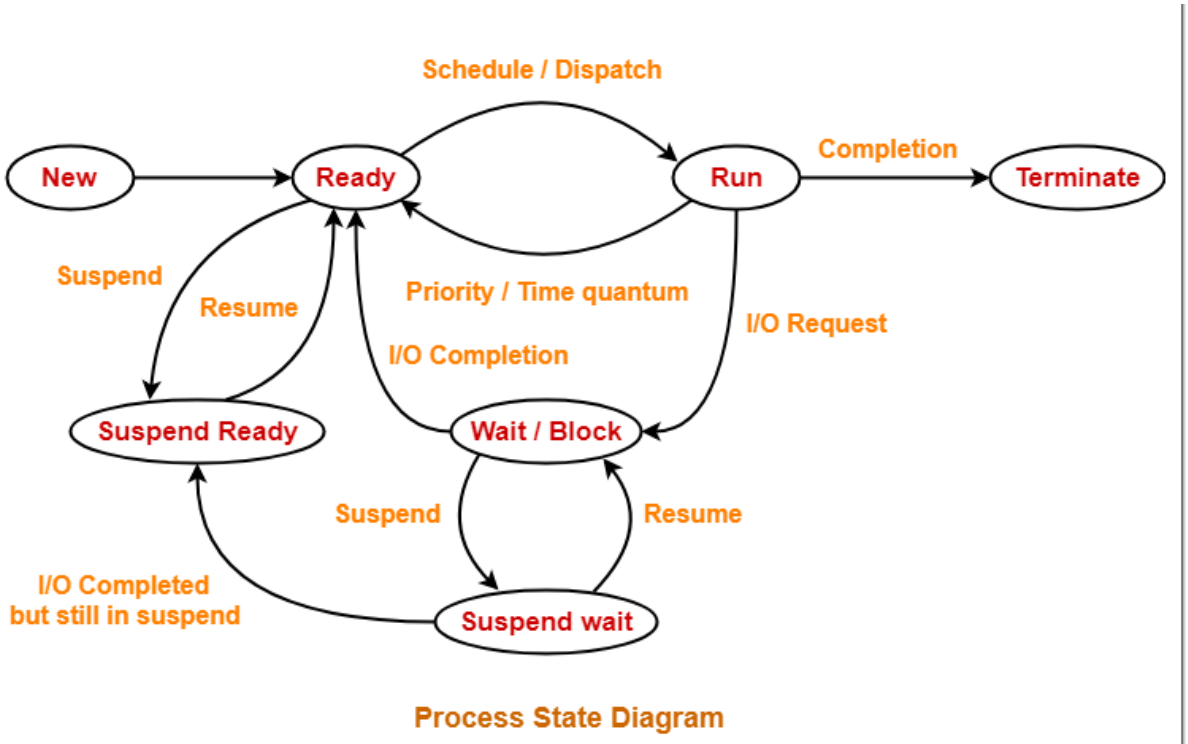
- **Process state:** Each and every process has some states associated with it at a particular instant of time. This is denoted by process state. It can be ready, waiting, running, etc.
- **CPU scheduling information:** Each process is executed by using some process scheduling algorithms like FCSF, Round-Robin, SJF, etc.
- **I/O information:** Each process needs some I/O devices for their execution. So, the information about device allocated and device need is crucial.

Difference between Program and Process

Sr. No.	Program	Process
1.	Program contains a set of instructions designed to complete a specific task.	Process is an instance of an executing program.
2.	Program is a passive entity as it resides in the secondary memory.	Process is a active entity as it is created during execution and loaded into the main memory.
3.	Program exists at a single place and continues to exist until it is deleted.	Process exists for a limited span of time as it gets terminated after the completion of task.
4.	Program is a static entity.	Process is a dynamic entity.
5.	Program does not have any resource requirement; it	Process has a high resource requirement; it needs resources

Sr. No.	Program	Process
	only requires memory space for storing the instructions.	like CPU, memory address, I/O during its lifetime.
6.	Program does not have any control block.	Process has its own control block called Process Control Block.
7.	Program has two logical components: code and data.	In addition to program data, a process also requires additional information required for the management and execution.
8.	Program does not change itself.	Many processes may execute a single program. There program code may be the same but program data may be different. These are never same.

Process States:



Process State Diagram

- As a process runs, it changes its states. Process
- **New State:** This is the state when the process is just created. It is the first state of a process.
- **Ready State:** After the creation of the process, when the process is ready for its execution then it goes in the ready state. In a ready state, the process is ready for its execution by the CPU but it is waiting for its turn to come. There can be more than one process in the ready state.
- **Ready Suspended State:** There can be more than one process in the ready state but due to memory constraint, if the memory is full then some process from the ready state gets placed in the ready suspended state.
- **Running State:** Amongst the process present in the ready state, the CPU chooses one process amongst them by using some CPU scheduling

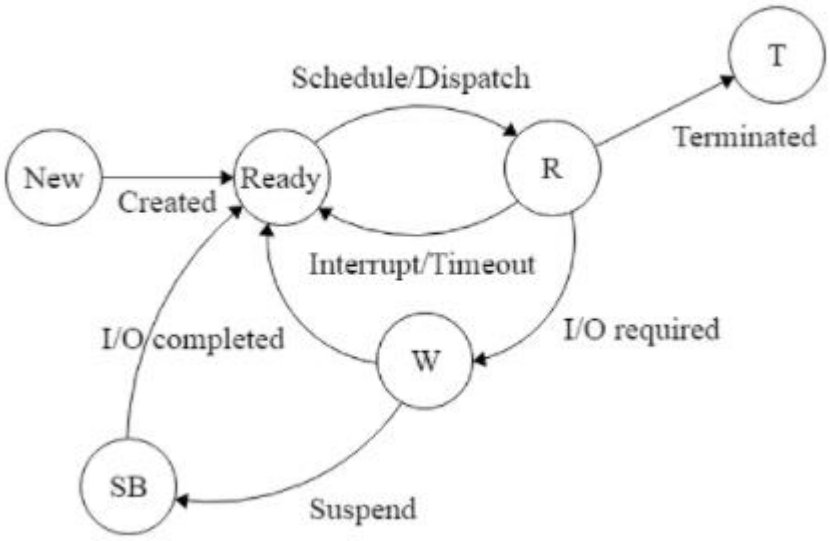
algorithm. The process will now be executed by the CPU and it is in the running state.

- **Waiting or Blocked State:** During the execution of the process, the process might require some I/O operation like writing on file or some more priority process might come. In these situations, the running process will have to go into the waiting or blocked state and the other process will come for its execution. So, the process is waiting for something in the waiting state.
- **Waiting Suspended State:** When the waiting queue of the system becomes full then some of the processes will be sent to the waiting suspended state.
- **Terminated State:** After the complete execution of the process, the process comes into the terminated state and the information related to this process is deleted.

Suspended Processes:

Characteristics of suspend process

- Suspended process isn't immediately available for execution.
- The process may or may not be waiting on an event.
- For preventing the execution, process is suspend by OS, parent process, process itself and an agent.
- Process can't be removed from the suspended state till the agent orders the removal.
- Swapping is used to move all of a process from main memory to disk. When all the process by putting it in suspended state and transferring it to disk.
- Figure below shows the process state transition diagram with suspended state.



- Swapping is used to move all of a process from main memory to disk. When all the processes in main memory are in blocked state, the OS can suspend one process by putting it in the suspended state and transferring it to disk.

Reasons for Process Suspension:

- Swapping
- Timing
- Interactive user request.
- Parent process request.
- Swapping: OS needs to release required main memory to bring in a process that is ready to execute.
- Timing: Process may be suspended next time interval.
- Parent Process Request: To modify the suspended process or to co-ordinate the activity of various descendants.
- **Example:** Explain whether the following transitions between process are possible or not. If possible, give the example.

1. Running – Ready.
2. Running – Waiting.
3. Waiting – Running.
4. Running – Terminated.

Process Control Block:

- The process control block (PCB) of a process contains three kinds of information concerning the process—identification information such as the process id, id of its parent process, and id of the user who created it; process state information such as its state, and the contents of the PSW and the general-purpose registers (GPRs); and information that is useful in controlling its operation, such as its priority, and its interaction with other processes.
- It also contains a pointer field that is used by the kernel to form PCB lists for scheduling, e.g., a list of ready processes.
- The priority and state information is used by the scheduler. It passes the id of the selected process to the dispatcher. For a process that is not in the running state, the PSW and GPRs fields together contain the CPU state of the process when it last got blocked or was preempted
- Operation of the process can be resumed by simply loading this information from its PCB into the CPU.
- This action would be performed when this process is to be dispatched.
- When a process becomes blocked, it is important to remember the reason.
- It is done by noting the cause of blocking, such as a resource request or an I/O operation, in the event information field of the PCB. Consider a process Pi that is blocked on an I/O operation on device d.
- The event information field in Pi's PCB indicates that it awaits end of an I/O operation on device d.

- When the I/O operation on device d completes, the kernel uses this information to make the transition blocked -> ready for process Pi.

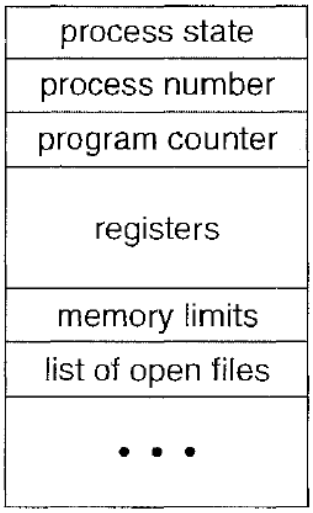


Figure 3.3 Process control block (PCB).

- **Process state:** The state may be new, ready running, waiting, halted, and so on.
- **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.
- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.
- PCB stands for Process Control Block. It is a data structure that is maintained by the Operating System for every process. The PCB should be identified by an integer Process ID (PID). It helps you to store all the information required to keep track of all the running processes.
- It is also accountable for storing the contents of processor registers. These are saved when the process moves from the running state and then returns back to it. The information is quickly updated in the PCB by the OS as soon as the process makes the state transition.
- The PCB simply serves as the repository for any information that may vary from process to process.
- When a process is created, hardware registers and flags are set to the values provided by the loader or linker.
- Whenever that process is suspended, the contents of a processor register are usually saved on the stack and the pointer to the related stack frame is stored in the PCB. In this way, the hardware state can be restored when the process is scheduled to run again.

Process Scheduling:

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.
- For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.
- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.
- The scheduling mechanism is the part of the process manager that handles the removal of running process from the CPU and the selection of another process on the basis of a particular strategy.

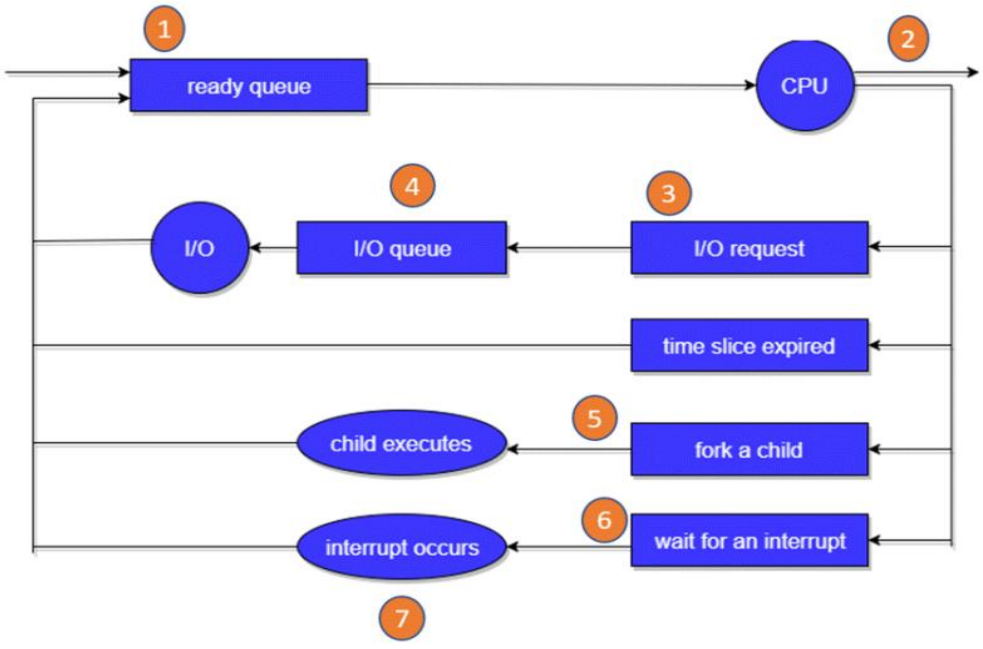
Scheduling Queues:

- Process Scheduling Queues help you to maintain a distinct queue for each and every process states and PCBs. All the process of the same execution state is placed in the same queue.
- Therefore, whenever the state of a process is modified, its PCB needs to be unlinked from its existing queue, which moves back to the new state queue.

- As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue.
- The figure represents queue using rectangular box. The circle represents the resources that serve the queues. The arrow indicates the flow of processes in the system.

Queues are of three types:

1. **Job queue** – It helps you to store all the processes in the system.
2. **Ready queue** – This type of queue helps you to set every process residing in the main memory, which is ready and waiting to execute.
3. **Device queues** – It is a process that is blocked because of the absence of an I/O device.

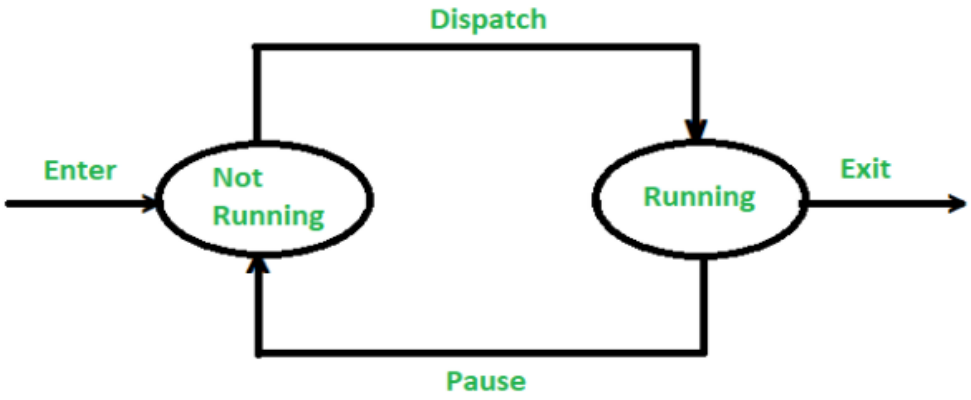


Process Scheduling Queues

1. Every new process is first put in the Ready queue. It waits in the ready queue until it is finally processed for execution. Here, the new process is put in the ready queue and wait until it is selected for execution or it is dispatched.
2. One of the processes is allocated the CPU and it is executing.
3. While executing the process, one of the several events could occur:
   - The process could issue an I/O request
   - Then, it should be placed in the I/O queue.
   - The process should create a new sub process.
   - The process should be waiting for its termination.
   - The process could be removed forcefully from the CPU, as a result of interrupt and put back in the ready queue.
4. In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

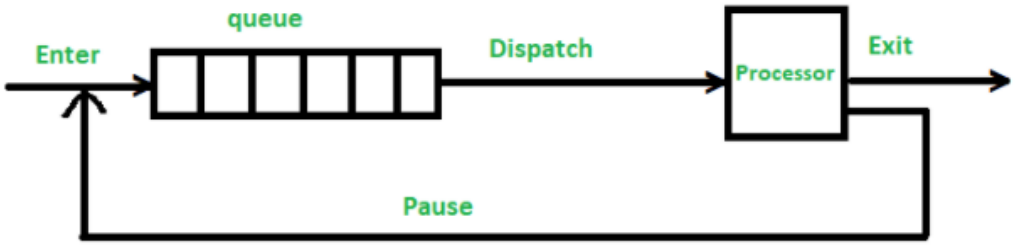
Two State Process Model:

- A process migrates among the various scheduling queues throughout its lifetime.
- While migrating among various queues, the process may be in one of the two states:
  - Running State
  - Not Running State



Two State process model

- A two-state can be created anytime no matter whether a process is being executed or not.
- Firstly, when the OS creates a new process, it also creates a process control block for the process so that the process can enter into the system in a non-running state. If any processes exit/leaves the system, then it is known to the OS.
- Once in a while, the currently running process will be interrupted or break-in and the dispatcher (a program that switches the processor from one process to another) of the OS will run any other process.
- Now, the former process(interrupted process) moves from the running state to the non-running state and one of the other processes moves to the running state after which it exits from the system.
- Processes that are not running must be kept in a sort of queue, and wait for their turn to execute.
- In the Queuing diagram, there is a single queue in which the entry is a pointer to the process control block (a block in which information like state, identifier, program counter, context data, etc., are stored in a data structure) of a particular process.



Queuing Diagram

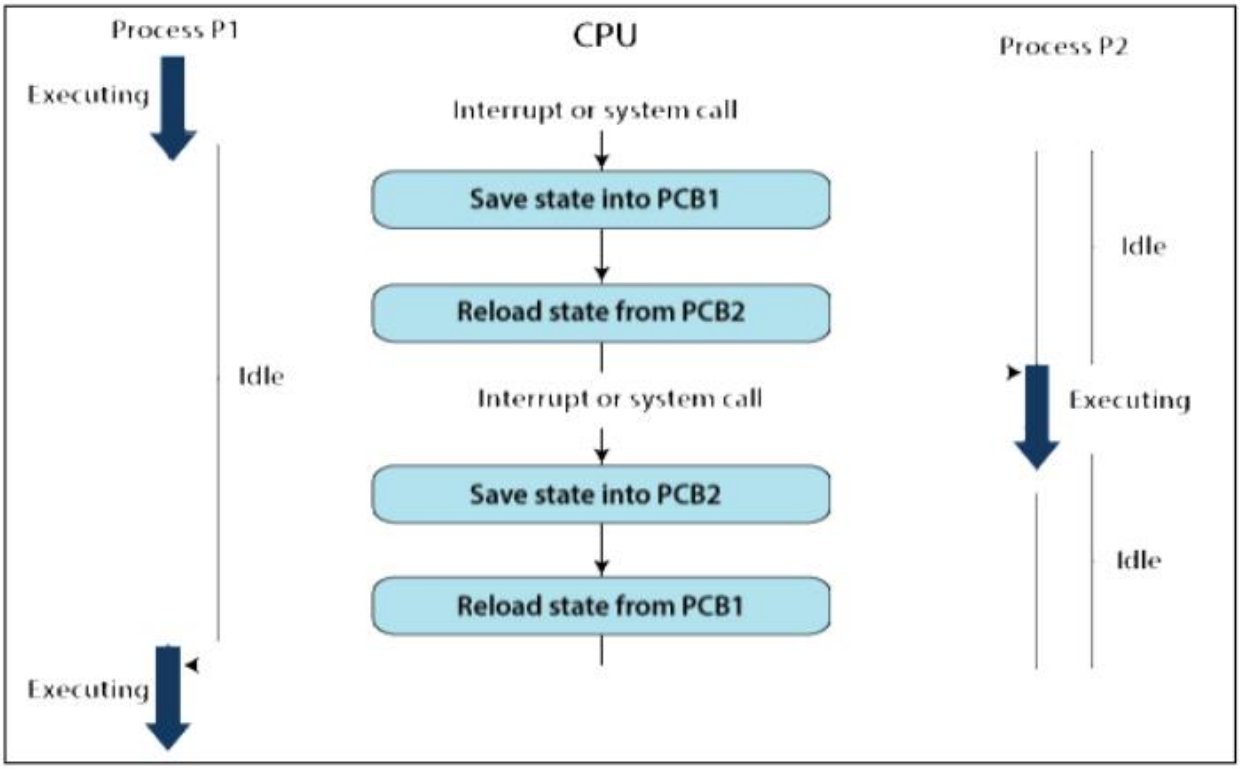
- Each entry in the queue is a printer to a particular process.
- Queues are implemented by using a linked list.
- Use of dispatcher is as follows: When a process is interrupted, that process is transferred in the waiting queue.
- If the process has completed or aborted, the process is discarded.
- In either case, the dispatcher then selects a process from the queue to execute.

Context Switch:

- The occurrence of the event is marked by an interrupt from either the hardware or software.
- Hardware may trigger an interrupt at any time by sending signal to trigger an interrupt at any time by sending signal to the CPU, by the way of system bus.
- Software can also trigger an interrupt by a special means and that is called as system call.
- Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems.

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state and memory-management information.
- Generically, we perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state resume is performed to resume a process.
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.
- 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.
- When Process is switched the information stored is:
  - Program Counter.
  - Scheduling Information.
  - Base and limit register value.
  - Currently used register.
  - Changed state.
  - I/O status.
  - Accounting
- 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 (such as a single instruction to load or store all registers). Typical speeds are a few milliseconds.

- Some processors may provide multiple CPU sets of registers. A context switch here simply requires changing the pointer to the current register set.
- Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before.
- Also, the more complex the operating system, the more work must be done during a context switch. In that case to put the extra data to and from the memory, different memory management techniques are used.
- The below given diagram gives a basic idea of context switching.



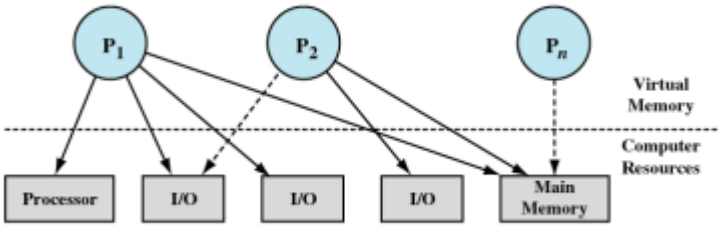
- There are various steps which are involved in the context switching:

1. The process P1 context, which is in the running state, will be stored in PCB (Program Control Block). That is called PCB1.
2. Next, PCB1 is transferred to the appropriate queue, i.e., the I/O queue, ready queue, and the waiting queue.

3. Then from the ready queue, we choose the new process which is to be executed i.e., the process P2.
4. Next, we update the PCB (Program Control Block) of the P2 process called PCB2. It includes switching the process state from one to another (ready, blocked, suspend, or exit). If the CPU previously executed process P2, then we get the location of the last executed process so that we can again proceed with the P2 process execution.
5. In the same manner, if we again need to execute the process P1, then the same procedure is followed.

Process Description:

- Operating System schedules and dispatches processes for execution by the processor, allocates resources to processes and responds to requests by user processes for basic services.
- The below given figure shows the processes and resources.

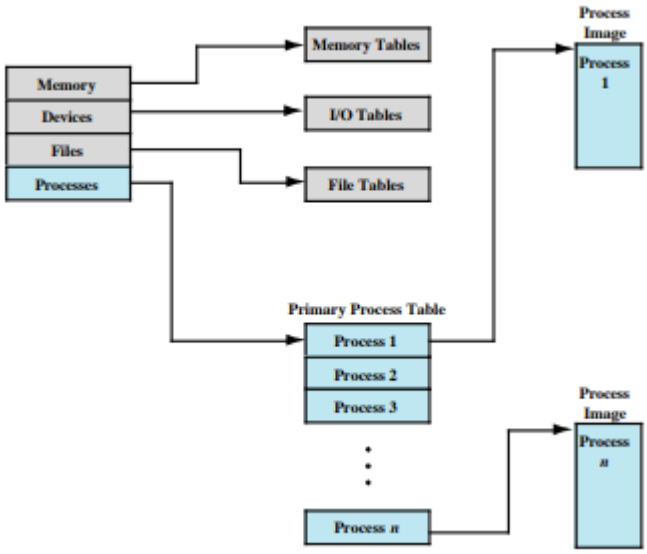


Operating System Control Structure:

- If the operating system is to manage processes and resources, it must have information about the current status of each process and resource.
- The figure shows general structure of operating system control tables.
- Operating system maintains four different tables:

1. Memory

2. I/O
3. File
4. Process



1. Memory Tables:

- Memory Table is used to keep track of both main and secondary memory. Some of main memory is reserved for operating system.
- Memory table must include the following information:
- Allocation of memory to processes.
- Allocation of secondary memory to processes.
- Any protection attributes of blocks of main or virtual memory.
- Any information needed to manage virtual memory.

2. I/O tables:

- I/O tables are used by operating system, the I/O devices and channels of the computer system.

- At any time, an I/O device may be available or assigned to a particular process.
- OS needs to know the status of I/O operation.

3. File Tables:

- File table provides information about the existence of files, their location on secondary memory, their current status and other attributes.
- All above information is maintained and used by a file management system.

4. Process Tables:

- Operating system uses process tables to manage processes.

Operations on Processes:

- The processes in most systems can execute concurrently and different operations can be performed on the processes.
- Processes can be created and deleted dynamically.
- The systems must provide a mechanism for process creation and termination. Here we are going to focus on two main operations i.e. creation of process and deletion of processes.

Process Creation:

- As we have discussed in last sections that processes are created by the operating systems to simplify the job of execution of the program by the CPU.
- One process which is created may create several new processes when it runs.

- These processes are created by the original process using create process system call.
- 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.
- In many operating systems, processes are identified by the unique process identifier which is a unique number given to it. It is an integer value assigned to the process. It is referred as **PID**.

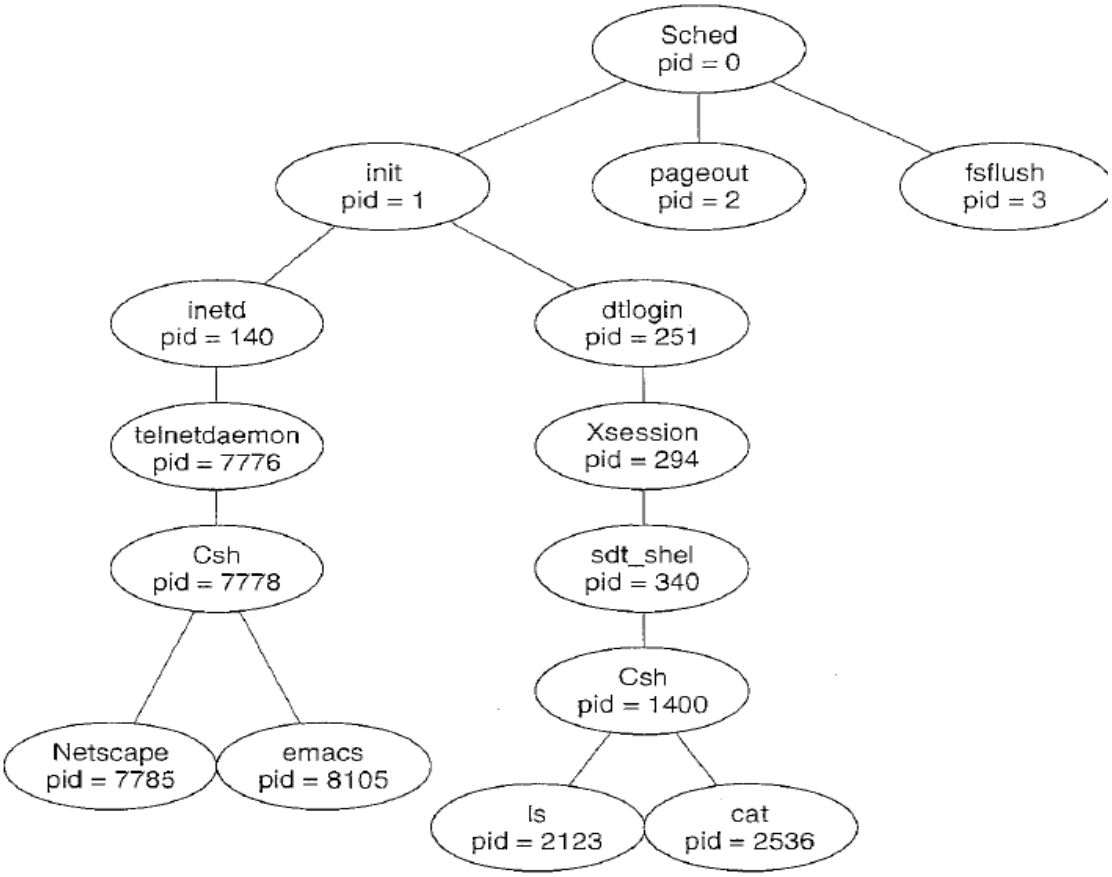


Figure 3.9 A tree of processes on a typical Solaris system.

- The above figure illustrates a typical process tree for the Solaris operating system, showing the name of each process and its pid.

- 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 init process, which serves as the root parent process for all user processes.
- In this Figure, we see two children of init-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 sdt_shel, 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).
- 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.
- It is easy to construct a process tree similar to what is shown in Figure by recursively tracing parent processes all the way to the init process.
- In general, a process requires certain resources like CPU time, memory, files, I/0 devices to accomplish its task. When parent process creates the process it is called as sub process.

- Operating system may provide required resources to the sub process or it may have to use the resources available to its 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.
- If child processes are restricted to become subset of parent's processes then we can avoid children processes to overload the system by creating many sub processes.
- Apart from the logical and physical resources which process share with its children processes, it even shares the input i.e. the input received to the parent process could be shared by its children.
- 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:

1. The parent continues to execute concurrently with its children.
2. 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:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

- Consider the following program written in the Unix to understand the relation of parent process and children process: C Program Forking Separate Process.

```
int main()
{
pid_t pid;
/*fork anther process*/
pid = fork();
if (pid < 0) {/*Error Occured*/}
fprintf(stderr, "Fork Failed");
exit(-1);
}
else if(pid==0) {/*child process*/
execlp("/bin/ls","ls",NULL);
}
else {/* parent process */}
 /* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
exit(0);
}
}
```

- From the program we come to know that pid is an unique id given to the process. 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 the parent as well as child processes get executed when fork() system call occurs. The main difference is when new process is created as a child the return code is zero but when fork() is executed for the parent the return code is nonzero number i.e. child's process code is returned to parent.
- The exec() system call is used after 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 a wait() system call to move itself off the ready queue until the termination of the child.

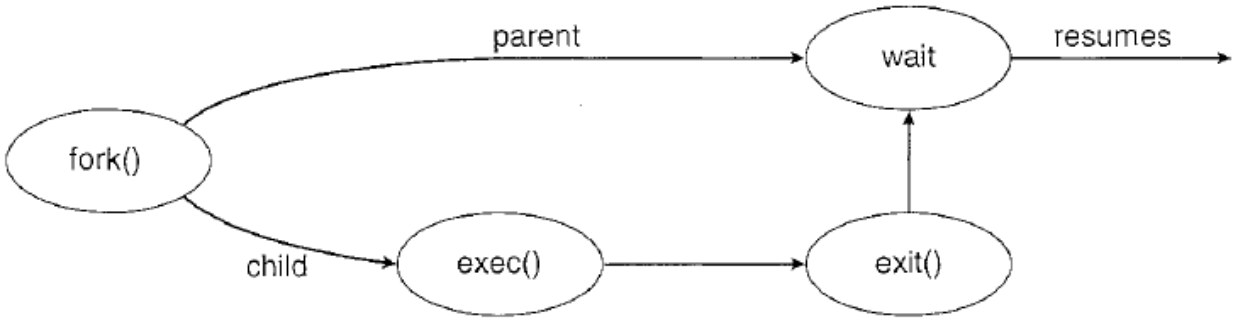


Figure 3.11 Process creation using `fork()` system call.

- The figure above as well as the C program, which we have written, shows that when fork() system call is create a parent and child processes are created. Parent process waits till the child process runs and once the child process is terminated using exec(), it removes the address space of parent process. Then exit() system call exits from the child process and parent process will be resumed.

Process Termination:

- A process is terminated when it finishes by executing an exit() system call. When its last statement gets executed the operating system deletes it using exit() system call.
- At this point process may return a status value to its parent process via the wait() system call. This return status value is generally an integer. All the logical and physical resources including open files, I/0 buffers, memory etc. allocated to the process is de-allocated by an operating system.

- Termination can even occur in different situations. A process can be terminated by another process via appropriate system call. Such a system call can be invoked by only parent of the process that has to be terminated.
- In the absence of this rule, 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:
  1. 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.)
  2. The task assigned to the child is no longer required.
  3. 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.
- In case of UNIX operating systems, the processes are terminated by exit() system call. In this case parent process waits till child process gets executed.
- If parent process terminates then child processes do not terminate. They are given a parent process terminates children process continues with new parent, which keeps track of children processes.

Reasons for Process Creation:

New batch job	The operating system is provided with a batch job control stream, usually on tape or disk. When the operating system is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The operating system can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Reasons for Process Termination:

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource or a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.

Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Inter-Process 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 anything with any other process is also known as an independent process.

- On other hand the processes, which are cooperating with each other or which gets affected by the other processes executing in the system are called as Co-operating process.
- Clearly, any process that shares data with other processes is termed as a cooperating process.
- There are several reasons for providing an environment that allows process cooperation:

1. **Information sharing**: Since several users may be interested in the same piece of information (for instance, a shared file), OS must provide an environment to allow concurrent or simultaneous access to such information.
2. **Computation speedup:** It is necessary to run the computation activities as fast as possible. 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).
3. **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
4. **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.

- There should be a mechanism for cooperating processes to communicate with each other. This mechanism is called as inter-process communication (IPC). IPC allows processes to exchange data and information.
- There are two fundamental models of inter-process communication:
  1. Shared memory and
  2. 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 diagram below gives an idea of IPC using shared memory and message passing.

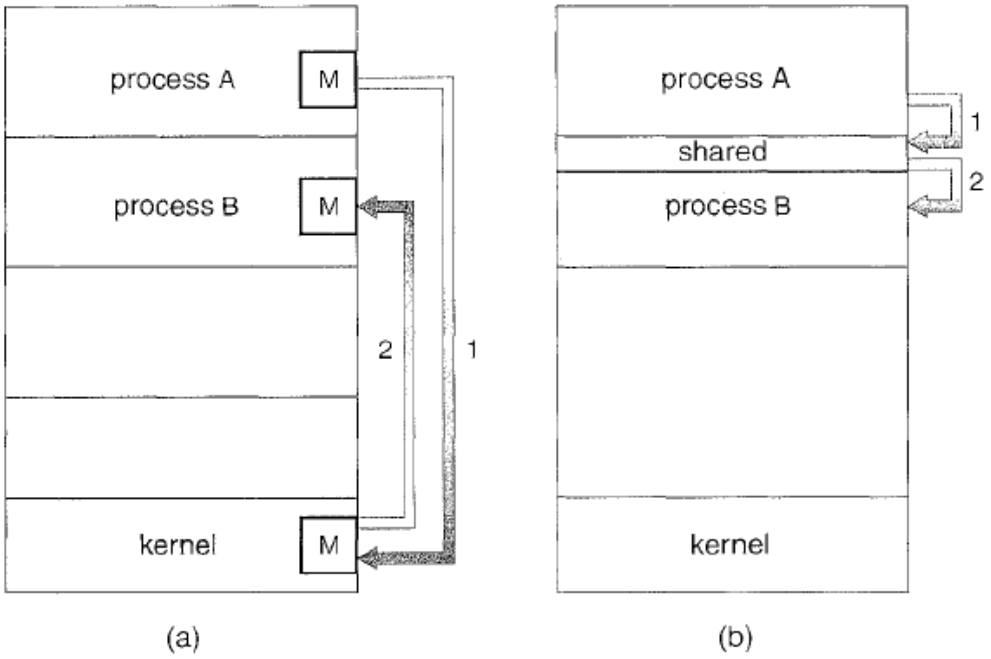


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

- Shared memory as well as message passing models is very common and popular communication models used very common and popular communication models used in an operating system.
- Message passing models are easy for exchanging small amount of data, as there is no much conflict.
- Message passing phenomenon is easier than shared memory model for inter-computer communication.

- Shared memory allows maximum speed and convenience of communication as it can be done at memory speeds when within a computer.
- 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.

Shared Memory Systems:

- Shared Memory systems allow processes to share data or information using shared memory region. A shared-memory region resides in the address space of the process creating the shared memory segment. Other processes that wish to communicate using this shared memory segment must attach it to their address space.
- Generally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes.

- A producer process produces information that is consumed by a consumer process.
- For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.
- The producer-consumer problem also provides a useful metaphor for the client-server paradigm.
- We generally think of a server as a producer and a client as a consumer. For example, a Web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client Web browser requesting the resource.
- One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- The buffer used could be of two types as follows:
  1. Unbounded buffer.
  2. Bounded Buffer.
- The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

- The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- Let's look more closely at how the bounded buffer can be used to enable processes to share memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in== out; the buffer is full when ((in+ 1)% BUFFER_SIZE) == out.
- **Producer Code:**

```
item nextProduced;

while (true) {
      /* produce an item in nextProduced */
      while (((in + 1) % BUFFER_SIZE) == out)
         ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.14 The producer process.

- **Consumer Code:**

```
item nextConsumed;

while (true) {
     while (in == out)
        ; // do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

Figure 3.15 The consumer process.

- The code described above is producer's and consumer's code. Producer uses local variable nextProduced in which newly produced item will be stored. nextConsumed is a local variable in consumers code, which shows the item consumed. There could be a issue if consumer as well as producer tries to share the shared buffer at a time.

Message Passing Systems:

- In the last section we studied that cooperating processes uses shared memory technique to communicate with each other. In this technique processes uses shared memory region. Another way is message passing where individual processes passes messages to each other as a means of communication.
- In message passing mechanism processes can just pass the messages to each other without sharing of their address spaces. This type of communication is mainly used in distributed environments.
- In distributed computing the processes may reside even on different computers connected by the network. The best example to illustrate

message passing system is the chat which takes place on world wide web where multiple users chat from different locations. They exchange messages among them from different nodes or locations.

- A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward.
- Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.
- If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation but rather with its logical implementation. This link should be useful to execute send () or receive () operations.
- There are certain methods to above said operations.

1. Direct or indirect communication.
2. Synchronous or asynchronous communication.
3. Automatic or explicit buffering.

- There are certain issues which are related to this method like.
  1. Naming
  2. Synchronization.
  3. Buffering

1. Naming.

- Processes that want to communicate must have a way to refer to each other. In order to communicate with each other the need is to know each

other with the name for identification. There are two types of communications as follows:

1. Direct Communication.
2. Indirect Communication.

- Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:
- send(P, message) -Send a message to process P.
- receive (Q, message)-Receive a message from process Q.
- A communication link in this scheme has the following properties:
- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.
- This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.
- send(P, message) -Send a message to process P.
- receive (id, message) -Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.
- The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

- All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection.
- With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification.
- In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox, however.
- The send() and receive 0 primitives are defined as follows:
- send (A, message) -Send a message to mailbox A.
- receive (A, message)-Receive a message from mailbox A.
- In this scheme, a communication link has the following properties:
- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.
- A mailbox may be owned either by a process or by the operating system.
- If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox).
- Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any

process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

- In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process.
- The operating system then must provide a mechanism that allows a process to do the following:

1. Create a new mailbox.
2. Send and receive messages through the mailbox.
3. Delete a mailbox.

- The process that creates a new mailbox is that mailbox's owner by default.
- Initially, the owner is the only process that can receive messages through this mailbox.
- However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

2. Synchronization.

- Communication between processes takes place through calls to send() and receive () primitives. There are different design options for implementing each primitive.
- Message passing may be either blocking or non-blocking also known as synchronous and asynchronous.
- Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox.
- Non-blocking send: The sending process sends the message and resumes operation.
- Blocking receive: The receiver blocks until a message is available.

- Non-blocking receive. The receiver retrieves either a valid message or a null.
- Different combinations of send () and receive () are possible. When both send () and receive () are blocking, we have a rendezvous between the sender and the receiver. The solution to the producer-consumer problem becomes trivial when we use blocking send () and receive () statements.
- The producer merely invokes the blocking send () call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive (), it blocks until a message is available.

3. Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
- **Zero capacity**: The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**: The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**: The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.
- The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

Cooperating Processes:

- 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.
- A process is cooperating if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a cooperating process.
- Cooperating processes are those that can affect or are affected by other processes running on the system. Cooperating processes may share data with each other.

Reasons for needing cooperating processes

There may be many reasons for the requirement of cooperating processes. Some of these are given as follows −

- **Modularity**

  Modularity involves dividing complicated tasks into smaller subtasks. These subtasks can complete by different cooperating processes. This leads to faster and more efficient completion of the required tasks.

- **Information Sharing**

  Sharing of information between multiple processes can be accomplished using cooperating processes. This may include access to the same files. A mechanism is required so that the processes can access the files in parallel to each other.

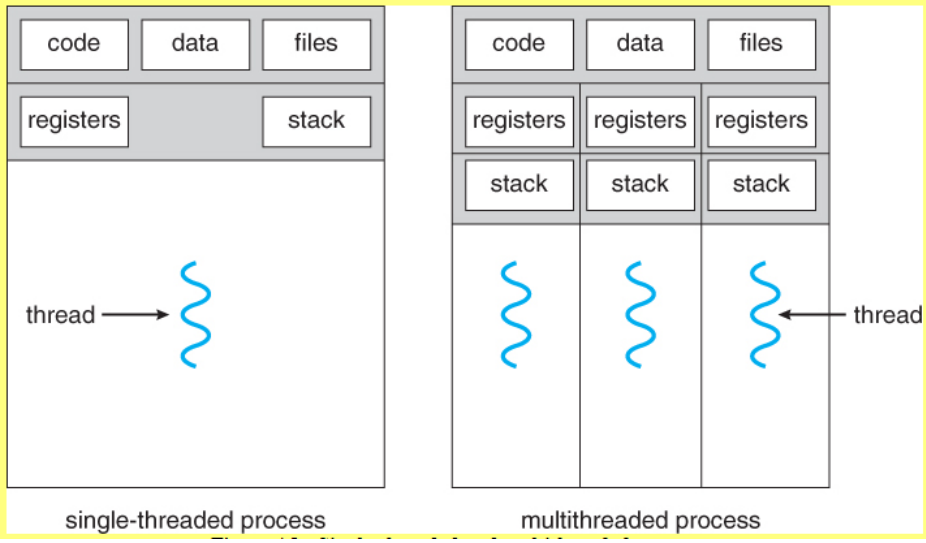
- **Convenience**

- There are many tasks that a user needs to do such as compiling, printing, editing etc. It is convenient if these tasks can be managed by cooperating processes.
- **Computation Speedup**

  Subtasks of a single task can be performed parallel using cooperating processes. This increases the computation speedup as the task can be executed faster. However, this is only possible if the system has multiple processing elements.

Threads:

- A thread is a basic unit of CPU utilization. Thread is associated with thread ID, a program counter, a register set, and a stack. Thread shares its code section, data section, and other operating system resources such as open files and signals with other threads which belong to the same type of process.
- Generally, a traditional process has signal thread but process can be multithreaded also. If the system is multithreaded control system then it is very powerful as it can also perform multiple tasks at a time. Observe the figure given below.



- The figure depicts multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.
- Modern operating systems are multithreaded operating systems and that makes them very powerful. One process if have multiple threads, then multiple tasks takes place. Just take an example of web browser which has many threads. One thread may display images, some may display text, some may be other links like data from network, etc
- Another example is word processor where one thread will display key strokes, one may show image, or one may do any other operating etc.
- In case of a server which is single processor system, it receives request from the different clients for the service. Now server can create different processes to serve each request. Creating process creates an overhead.
- Solution to this problem is creating a single process with multiple threads. The threads so not create overhead or time delay. So here multithreaded system can perform better than single threaded system.
- Threads plays very important role in case of Remote process call as server can collect requests as threads and can serve many requests at a time. Example of this type is java's RMI.
- The conclusion is that now most of the operating systems have become multithreaded systems. The kernels have many threads which performs tasks like interrupt handling, device management, free memory management, etc.

Benefits of Multithreaded Programming:

The benefits of multithreading programming are as follows:

1. Responsiveness.
2. Resource Sharing.

3. Economy.
4. Utilization of multiprocessor architecture.

- **Responsiveness** –
- Multithreading operating system are interactive systems or applications. This environment may allow program to execute or run continuously even if the part of it is blocked or it may be performing a lengthy operation. This is known as responsiveness to the user.
- Due to multithreading system is user is using web browser one thread displays image while other thread does other operation. This again part of responsiveness.
- One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
- **Resource sharing** –
- Thread belongs to a particular process. The resources allotted or allocated to the process can be used by its threads. So many threads can share resources of its process to which they belong.
- By default threads share common code, data, and other resources, which allow multiple tasks to be performed simultaneously in a single address space.
- **Economy** –
- As discussed earlier, process creation is costly due to memory allocation and resource allocation. As threads shares all resources required from its process to which they belong, it is economical.
- Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
- **Scalability, i.e. Utilization of multiprocessor architectures** –

- A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

User Thread and Kernel Thread

- Support for creating thread is done on two levels i.e. user level and kernel level. User level threads are called as user threads. Kernel level threads are called as kernel threads.
- Support for the threads may be provided either at the user level called user threads. Kernel level threads provided by OS are called as Kernel Threads.
- User threads are supported above the kernel and are managed without kernel support. On the other hand operating system supports and manages kernel threads. The kernel thread and user threads are related to each other.
- There are different models which shows the relationship between kernel thread and user threads as follows:

  1. Many to many model.

  2. One to One model.

  3. Many to many model.

- We will understand the multithreading model in the Operating system.

1. **Many to many model.**

- In many to one model one kernel thread is connected with multiple user level threads. Thread management is done by thread library in user space. So it is efficient. The whole process will block if a thread makes a blocking system call.

- As only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- E.g: Green thread is the library available in Solaris operating system which implements the same model.

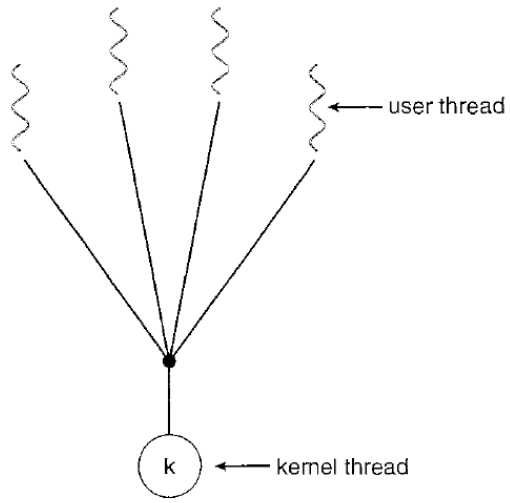


Figure 4.5 Many-to-one model.

Advantages:

1. Use of library.
2. Efficient system in terms of performance.
3. One kernel thread controls multiple user threads.

Disadvantages:

1. Cannot run multiple user threads parallel.
2. One block call blocks all user threads.

2. **One to One model.**

- In one to one model each user thread is mapped to kernel thread. This model provides more concurrency by allowing other thread to run when a thread makes a blocking system call. This model also allows multiple threads to run in parallel on multiprocessors.

- The drawback of this model is that when user thread gets created then corresponding kernel thread should be created.
- Creating kernel thread is overhead for the system. So this model has this restriction that every time when user thread is created burden of creating kernel thread is there. This reduces the performances of an application.

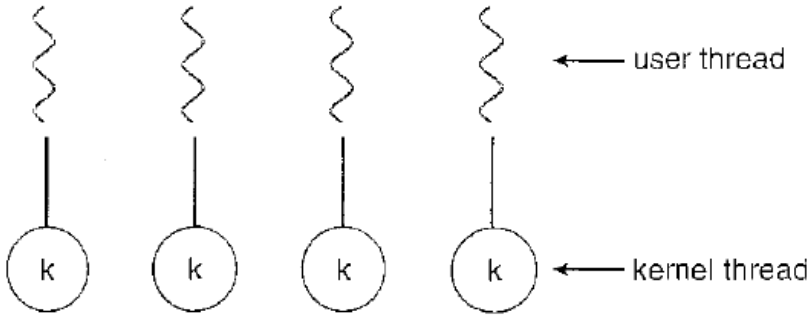


Figure 4.6 One-to-one model.

- Some popular examples are the operating system which has implemented this model e.g. UNIX, Windows XP, etc.

Advantages:

1. More concurrency.
2. Multiple threads can run parallel.
3. Less complication in the processing.

Disadvantages:

1. Every time with user's thread, kernel thread is created.

2. Kernel thread is an overhead.

3. It reduces the performance of system.

3. Many-to-Many Models:

- In many to many models, many user threads are attached with same number of kernel threads. The number of kernel threads dependent on the application or even on particular machine.
- Many to Many models allows developer to create many threads as user wishes. In this model true concurrency cannot be achieved because one kernel thread executes one thread at a time.
- On a multiprocessor system many kernel threads can run parallel. When one thread calls block system call, other kernel thread calls for another system call.

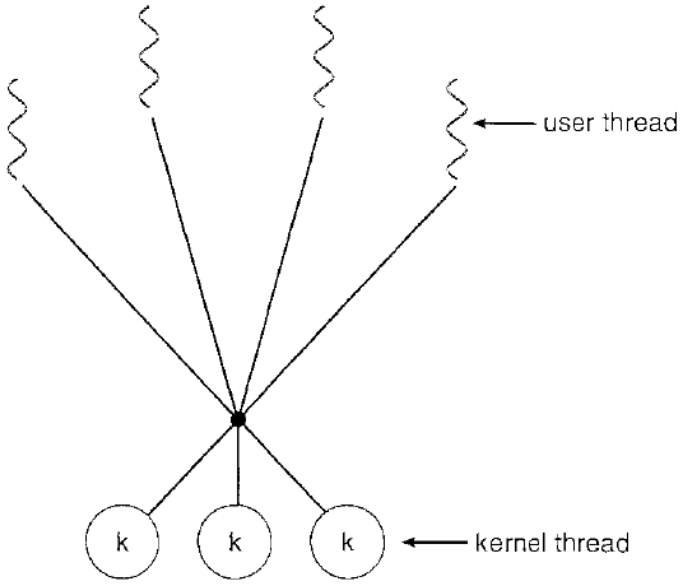


Figure 4.7 Many-to-many model.

- There is one popular variation on the many to many model still multiplexes many user-level threads to a smaller or equal number of kernel threads. It also allows a user-level thread to be bound to a kernel thread.

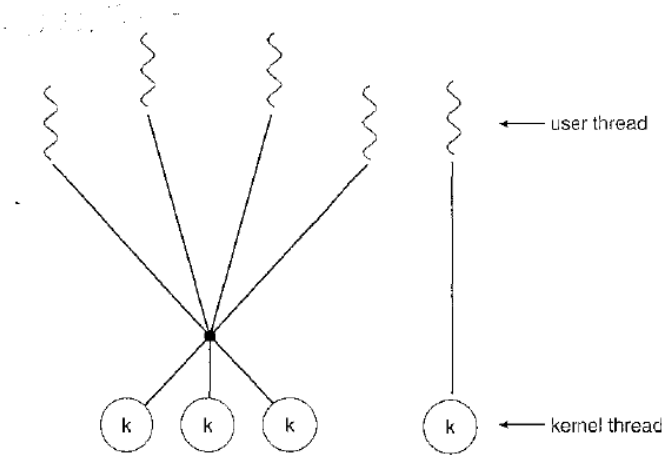


Figure 4.8 Two-level model.

- This variation is called as two level model which is supported by operating system such as IRX, HP-UX.

Advantages:

1. Many threads can be created as per user's requirement.
2. Multiple kernel or equal to user threads can be created.

Disadvantages:

1. True concurrency cannot be achieved.

2. Multiple threads of kernel are an overhead for operating system.

3. Performance is less.

CPU SCHEDULING CRITERIA:

- Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a

substantial difference in which algorithm is judged to be best. The criteria include the following:

1. CPU Utilization.
2. Throughput
3. Turnaround time
4. Waiting time
5. Response time

1. CPU Utilization

- It is important to keep CPU always busy, so that it will be fully utilized. Theoretically CPU can be utilized in percentage from 0 to 100.
- In the real system, CPU utilization falls in the range of 40% to 90%, 40% is mainly for lightly loaded systems and 90% is for heavy loaded systems.

2. Throughput

- If we try to keep CPU busy all the time, then expected work will be done. There has to be certain unit which calculates the actual work done by CPU.
- There is one measure to calculate work done by CPU i.e. number of processes that are completed per time unit.
- This is called as throughput. If the processes are long then time could be one process per hour or for short processes it could be 10 processes per second.

3. Turnaround time

- From the point of view of a particular process, the important criterion is how long it takes to execute that process.
- The interval from the time of submission of a process to the time of completion is the turnaround time.

- Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/0.

4. Waiting time

- The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0; it affects only the amount of time that a process spends waiting in the ready queue.
- Waiting time is the sum of the periods spent waiting in the ready queue

5. Response time

- In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user.
- Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.
- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average.

Scheduling Algorithms:

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms.

    1. First Come First Serve.
    2. Shortest Job First.

3. Priority Scheduling.
4. Round Robin Scheduling.

1. First Come First Serve.

- FCFS is non-preemptive algorithm. It is easy and simplest CPU scheduling. The process which requests the CPU first is allocated to the CPU first. The FIFO stands for first in first out.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The concept of FCFS is very easy to understand from the following example we can calculate the performance of FCFS algorithms. It is poor.
- E.G:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:

- In FCFS, if there is a process which is CPU bound and many other processes having I/O bound, then CPU bound process will hold CPU for its execution for more time.

- At the same time, the I/O bound processes finishes their I/O operations and enters in ready queue for an execution. This movement CPU gets idle state.
- In this way performance of FCFS is poor. Because CPU and I/O are not effectively utilized in this method.
- Especially, in the case of time sharing operating system FCFS is not a good solution for the CPU scheduling.
- The reason behind this is, in time sharing operating system it is necessary that each user should get CPU share after certain interval. It would be terrific to allow one process to keep the CPU for long time.

2. Shortest Job First (SJF).

-

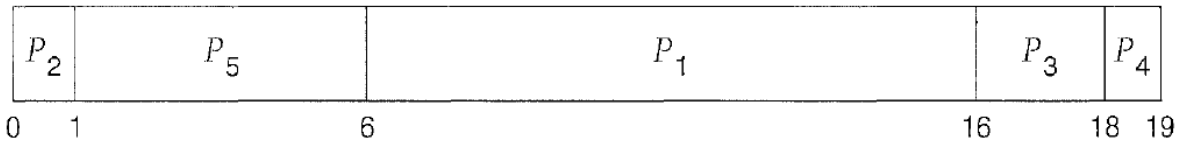
3. Priority Scheduling.

- The SJF algorithm is a special case of the general priority scheduling algorithm.
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- In priority scheduling the range of priorities can be represented as 0 to 7 or 0 to 4,095 However, there is no general agreement on whether 0 is the highest or lowest priority.
- Some systems use low numbers to represent low priority; others use low numbers for high priority.

- This difference can lead to big confusion for the user in order to understand the numbering of priority.
- As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, Ps, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Using priority scheduling, we would schedule these processes according to the following Gantt chart:



- The average waiting time is 8.2 milliseconds.

Advantages:

- Easy to use scheduling method
- Processes are executed on the basis of priority so high priority does not need to wait for long which saves time
- This method provides a good mechanism where the relative important of each process may be precisely defined.
- Suitable for applications with fluctuating time and resource requirements.

Disadvantages:

- If the system eventually crashes, all low priority processes get lost.
- If high priority processes take lots of CPU time, then the lower priority processes may starve and will be postponed for an indefinite time.
- This scheduling algorithm may leave some low priority processes waiting indefinitely.
- A process will be blocked when it is ready to run but has to wait for the CPU because some other process is running currently.
- If a new higher priority process keeps on coming in the ready queue, then the process which is in the waiting state may need to wait for a long duration of time.

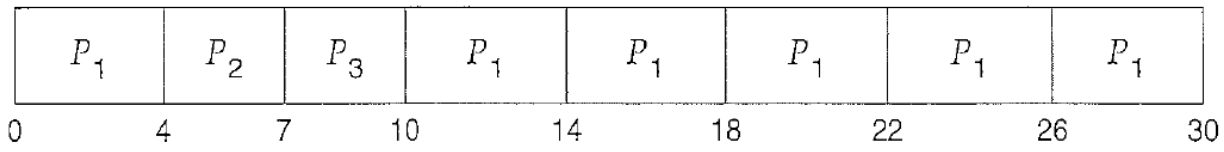
4. Round Robin Scheduling.

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. In this case, there are two possibilities.

- If time quantum of processor is greater than process burst time then, process itself releases CPU, otherwise interrupt interrupts the CPU. Then CPU stops the execution and the process is shifted to the tail of the ready process queue. After this, CPU scheduler selects next job for execution.
- The average waiting time in case of RR algorithm is generally longer.
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2.
- Process P2 does not need 4 milliseconds, so it quits before its time quantum expires.
- The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum.
- The resulting RR schedule is as follows:



- Let's calculate the average waiting time for the above schedule. P1 waits for 6 milliseconds (10- 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.
- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).
- If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.
- The performance of the RR algorithm depends heavily on the size of the time quantum.
- At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing and (in theory) creates the appearance that each of 11 processes has its own processor running at $1/n^{th}$ the speed of the real processor

Advantage:

The average waiting time is minimal (negligible).

Disadvantage:

If the time quantum is too large then it is as good as FCFS policy

Thread Scheduling:

Multiple - Processor Scheduling:

- Multilevel queue scheduling deals with a single processor. In case of multiprocessor system it is complicated to schedule processes. In this scheduling we use homogeneous (uniform) processors in terms of their functionality. In this type of scheduling we use common ready queues.

- All the processes are getting joined to ready queue and queue is assigned to available processor.
- The two approaches are used to schedule the job. In first approach each processor is self-scheduling. CPU itself checks the common ready queue and makes selection of the process to execute. It is known as symmetric multiprocessing system i.e. SMP.
- In other approach two processors cannot choose the same process and process cannot lost from queues. In real time scheduling, this is to be done by using special processor as scheduler for other processors. It is referred as asymmetric multiprocessing system.
- The scheduling facility is required to support real time computing within a general purpose computer system. Real time computing is divided into two types:

1. Hard real time systems.
2. Soft real time computing.

Scheduling Algorithms evaluation:

Prof. A. S. Aher
(Subject In-charge)