## Process Concept

Process: A process can be thought of as a program in execution (or) A process is the unit of work in a modern time-sharing system. A process will need certain resources such as CPU time, memory, files and I/O devices to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

The process concept includes the following:

1. Process
2. Process state
3. Process Control Block
4. Threads

The below figure shows the structure of process in memory:

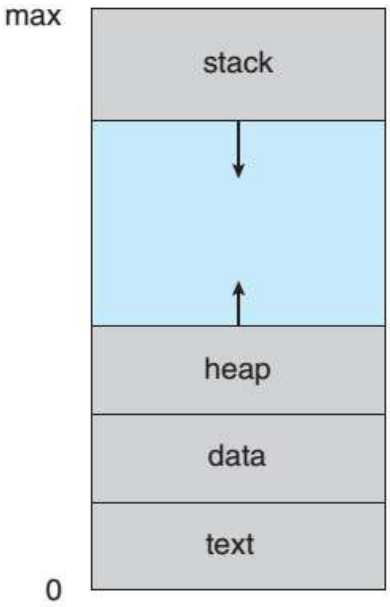



Fig: Process in memory

The process contains several sections: **Text, Data, Heap and Stack.**

· **Text Section** contains the program code. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.

· **Process stack** contains temporary data such as function parameters, return addresses and local variables.

· **Data section** contains global variables.

· **Heap** is memory that is dynamically allocated during process run time.

**Difference between Program and Process**

- A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk often called an **executable file**.
- A process is an ***active*** entity with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line as in **prog.exe** or **a.out.**

Although two processes may be associated with the same program, they are considered as two separate execution sequences. For instance, several users may be running different copies of the mail program or the same user may invoke many copies of the web browser program. Each of these is considered as a separate process.

**Process State**

As a process executes, it changes **state**. The process state defines the current activity of that process.

A process may be in one of the following states:

· **New**: The process is being created.

· **Ready**: The process is waiting to be assigned to a processor.

· **Running**: Instructions are being executed.

· **Waiting**: The process is waiting for some event to occur such as an I/O completion or reception of a signal.

· **Terminated**: The process has finished execution.

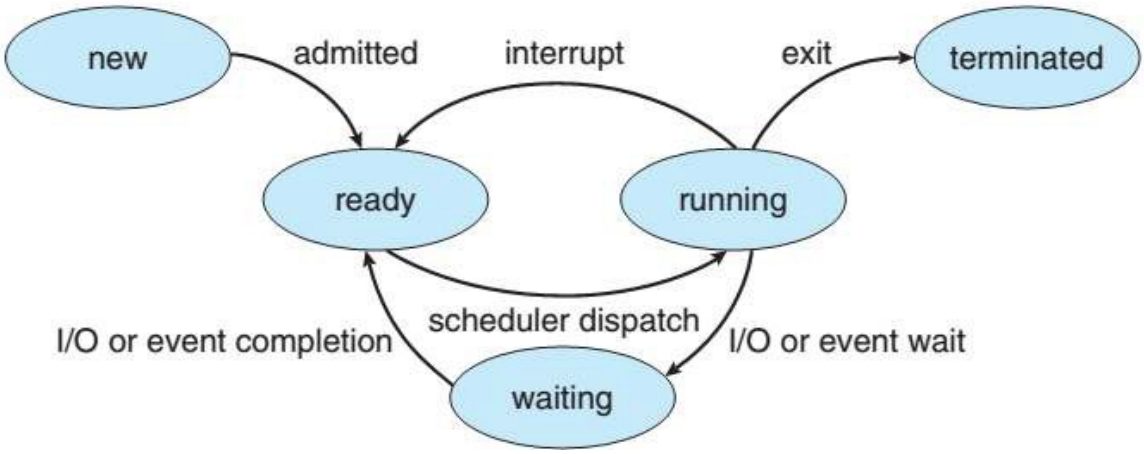



Fig: Process State

**Note:** Only one process can be *running* on any processor at any instant of time.

**Process Control Block**

Each process is represented in the operating system by a **Process Control Block (PCB)**. It is also called a **Task Control Block**.

PCB serves as the repository for any information that may vary from process to process.

**The PCB contains information related to process such as:**

**Process state**: The state may be new, ready, running, waiting and terminated.

**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 etc. Along with the program counter, this state information must be

saved when an interrupt occurs, to allow the process to be continued correctly afterward.

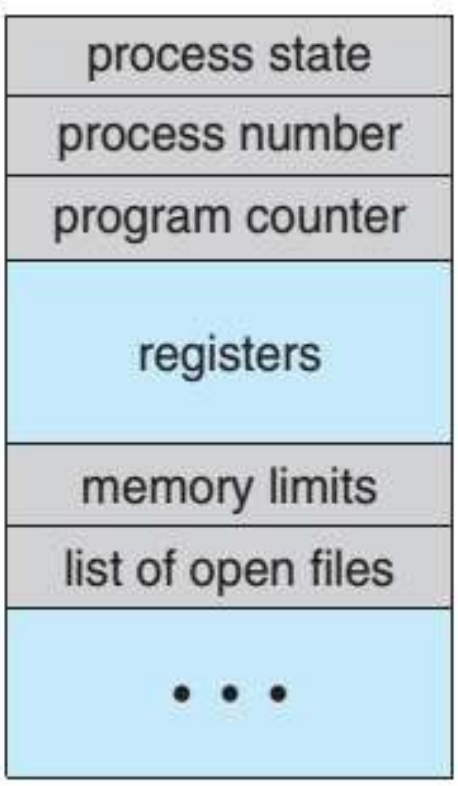



Fig: Process Control Block

**Process Control Block Information**

- **CPU-scheduling information**: This information includes a process priority, pointers to scheduling queues and any other scheduling parameters.
- **Memory-management information**: This information includes the base and limit registers values, 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.

**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 for program execution on the CPU.

Process scheduling involves three things:

1. Scheduling Queues
2. Schedulers
3. Context Switch

## Process Scheduling

The objective of multiprogramming is to have some process always running, 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 for program execution on the CPU.

Process scheduling involves three things:

1. Scheduling Queues
2. Schedulers
3. Context Switch

**Scheduling Queues**

There are several queues are implemented in operating system such as Job Queue, Ready Queue, Device Queue.

- **Job Queue:** It consists of all processes in the system. As processes enter the system, they are put into a **job queue**.
- **Ready Queue:** The processes that are residing in main memory and they are ready and waiting to execute are kept on a list called the **Ready Queue**. Ready queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- **Device Queue:** Each device has its own device queue. It contains the list of processes waiting for a particular I/O device.

Consider the given Queuing Diagram:

- Two types of queues are present: the **Ready Queue** and a set of **Device Queues**.
- CPU and I/O are the resources that serve the queues.

- A new process is initially put in the ready queue. It waits there until it is selected for execution or **dispatched**.
- Once the process is allocated the CPU and is executing, one of several events could occur:
    1. The process could issue an I/O request and then be placed in an I/O queue.
    2. The process could create a new child process and wait for the child's termination.
    3. The process could be removed forcibly from the CPU, because of an interrupt and be put back in the ready queue.

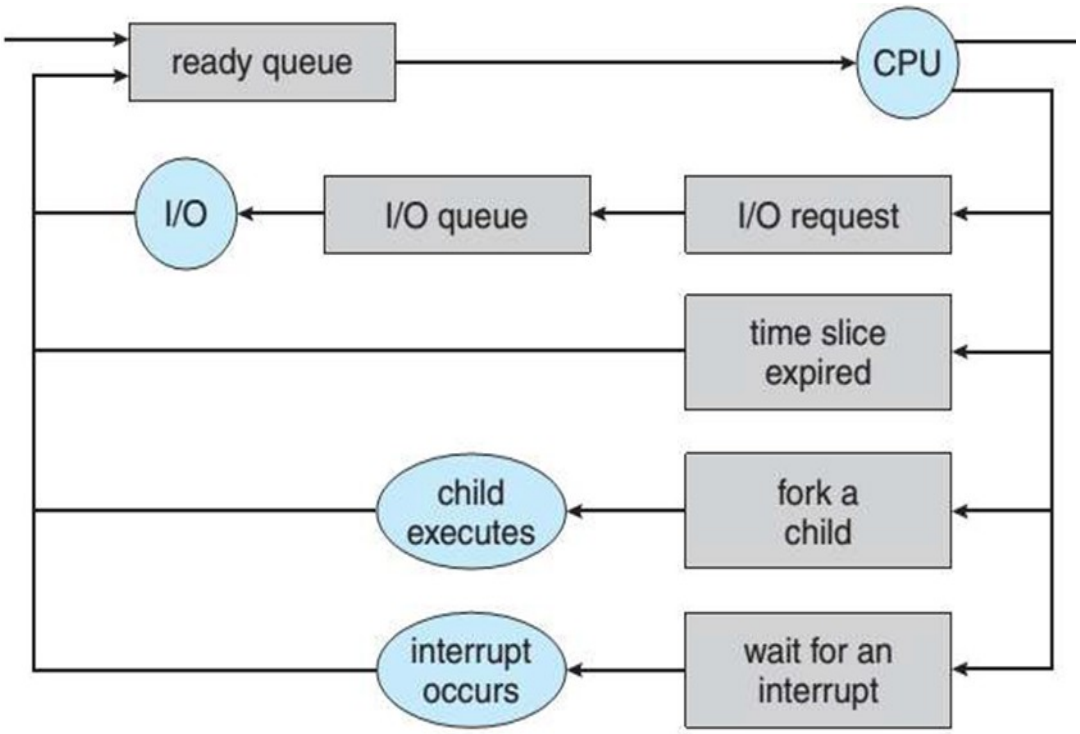



Fig: Queuing diagram representation of process scheduling

**Schedulers**

A process migrates among the various scheduling queues throughout its lifetime. For scheduling purpose, the operating system must select processes from these queues. The selection process is carried out by the **Scheduler.**

There are three types of Schedulers:

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

**Long Term Scheduler (New to ready state)**

- Initially processes are spooled to a mass-storage device (i.e Hard disk), where they are kept for later execution.
- Long-term scheduler or job scheduler selects processes from this pool and loads them into main memory for execution. (i.e. from Hard disk to Main memory).
- The long-term scheduler executes much less frequently, there may be minutes of time between creation of one new process to another process.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).

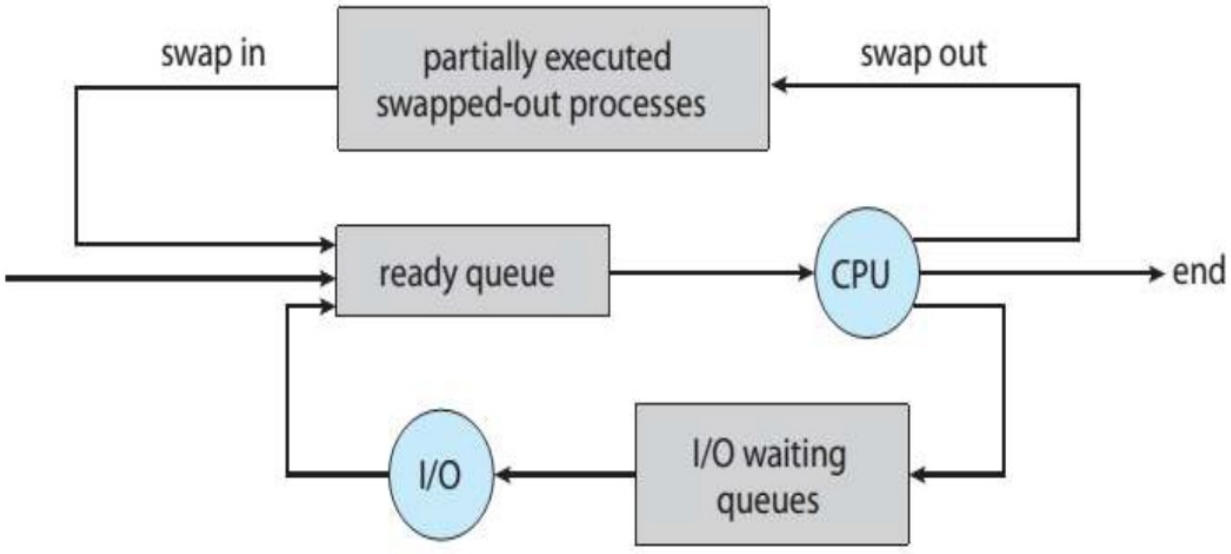


Fig: Process Scheduler

**Short Term Scheduler (Ready to Running)**

- Short-term scheduler or CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them. (i.e. a process that resides in main memory will be taken by CPU for execution).
- The short-term scheduler must select a new process for the CPU frequently.
- The short term scheduler must be very fast because of the short time between executions of processes.

**Medium Term Scheduler**

Medium Term Scheduler does two tasks:

1. **Swapping:** Medium-term scheduler removes a process from main memory and stores it into the secondary storage. After some time, the process can be reintroduced into main memory and its execution can be continued where it left off. This procedure is called Swapping.
2. Medium Term Scheduler moves a process from CPU to I/O waiting queue and I/O queue to ready queue.

**The processes can be described as two types:**

1. I/O bound process is one that spends more of its time doing I/O than it spends doing computations.
2. CPU Bound process using more of its time doing computations and generates I/O requests infrequently.

**The long-term scheduler selects a good *process mix* of I/O-bound and CPU-bound processes**.

- If all processes are I/O bound, the ready queue will almost always be empty, and the CPU will remain idle for long time because I/O device processing takes a lot of time.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty. I/O devices will be idle, and CPU is busy for most of the time.
- Thus, if the system maintains the combination of CPU bound and I/O bound processes then the system performance will be increased.

**Note:** Time-sharing systems such as UNIX and Microsoft Windows systems often have no

long-term scheduler but simply put every new process in memory for the short-term scheduler.

**Context Switching**

- Switching the CPU from one process 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**.
- 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.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure overhead, because the system does no useful work while switching. Context switch time may be in few milliseconds.

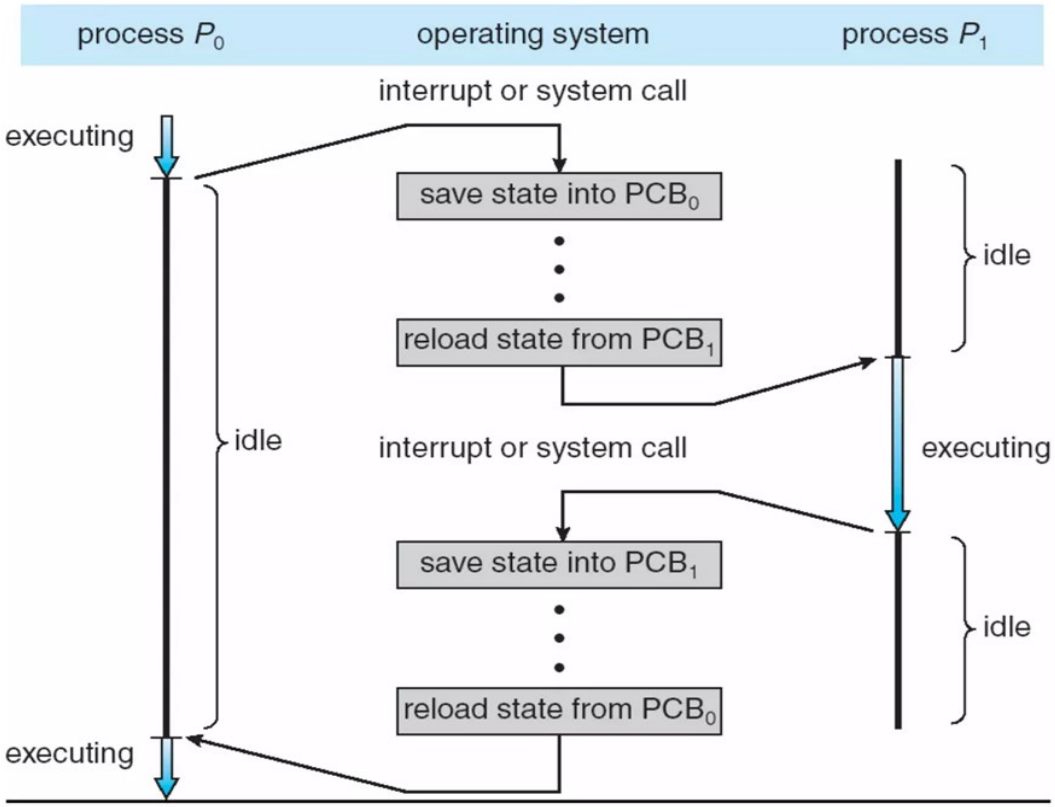



Fig: CPU Switch from Process to Process

**Operations on Processes**

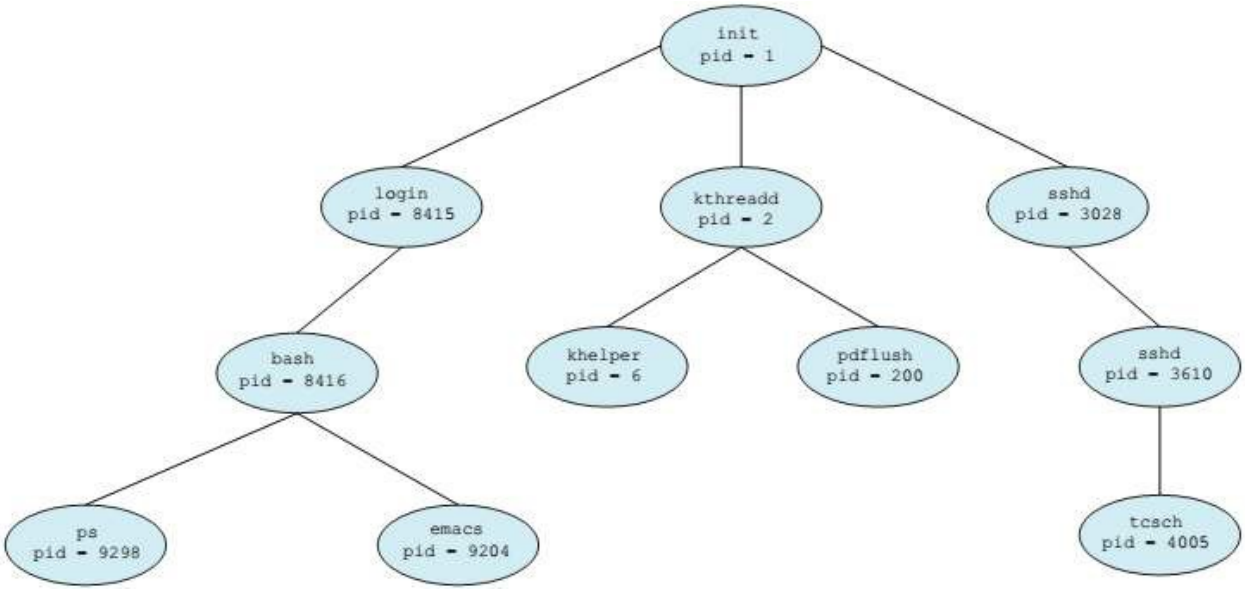
1. Process Creation
2. Process Termination

**Process Creation**

- During the execution of a process in its life time, a process may create several new processes.
- The creating process is called a parent process and the new processes are called children process.
- Each of these new processes may create other processes forming a **tree** of processes.
- Operating system identifies processes according to **process identifier** (**pid**).

- Pid provides an unique integer number for each process in the system.
- Pid can be used as an index to access various attributes of a process within the kernel.

The below figure shows the process tree for the Linux OS that shows the name of each process and its process ID (pid) . In Linux process is called task.




- The init process always has a pid of 1. The init process serves as the root parent process for all user processes.

- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server etc.
- *kthreadd* and *sshd* are child processes of init.
- The *kthreadd* process is responsible for creating additional processes that perform tasks on behalf of the kernel.
- The *sshd* process is responsible for managing clients that connect to the system by using secure shell (ssh).
- **ps** command is used to obtain a list of processes:

***ps –el***

The command will list complete information for all processes currently active in the system.

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

When a process creates a new process there exist two possibilities for execution:

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

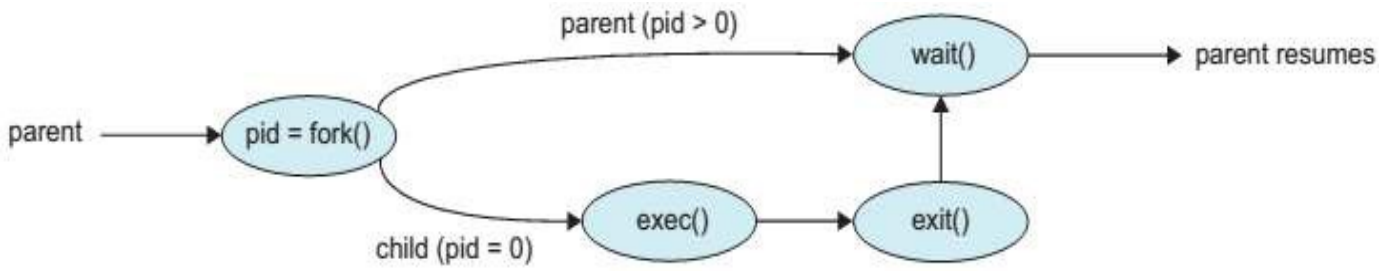
There are also two address-space possibilities for the new process:

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

**Process System calls in Unix/ Linux: fork( ), exec( ), wait( ), exit( )**

- **fork( )**: In UNIX OS 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 and the child processes continue execution at the instruction after the fork( ).
- For the new child process (i.e. Child Process) the return code for the fork( ) is zero.
- The nonzero process identifier of the child is returned to the parent.
- **exec( )**: After a fork( ) system call, one of the two processes typically uses the **exec( )** system call to replace the process's memory space with a new program.
- The exec( ) system call loads a binary file into memory and starts its execution.
- In this way, the two processes are able to communicate and then go their separate ways.
- **wait( ):** The parent can create more children or if the parent has nothing else to do while the child process is running then the parent process can issue a **wait( )** system call to move itself out of the Ready Queue until the child process terminates

- The call to exec( ) overlays the process's address space with a new program or the call to exec( ) does not return control unless an error occurs.




**Program for Creating a separate process using the UNIX fork( ) system call**

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main( )
{
pid_t pid;
/* fork a child process */
pid = fork( );
if (pid < 0)
{ /* error occurred */
fprintf(stderr, "Fork Failed");
return 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");
}
return 0;
}
```

The C program in the previous slide shows the UNIX system calls fork, exec, wait. Two different processes

are running copies of the same program.

- The only difference is that the value of pid for the child process is zero, while the value of pid for the parent is an integer value greater than zero (i.e. the actual pid of the child process).
- The child process inherits privileges and scheduling attributes from the parent, as well as certain resources such as open files.
- The child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp( ) system call (execlp( ) is a version of the exec( ) system call).
- The parent waits for the child process to complete with the wait( ) system call.

- When the child process completes by either implicitly or explicitly invoking exit( ), the parent process resumes from the call to wait( ), where it completes using the exit( ) system call.

**Process Termination: exit( )**

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

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

1. The child has exceeded its usage of some of the resources that it has been allocated.
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.

**Operations on Processes**

**Cascading Termination**

- If a parent process terminates either normally or abnormally then all its children must also be terminated is referred as Cascading Termination. It is normally initiated by operating system.
- In Linux and UNIX systems, a process can be terminated by using the *exit( )* system call providing an exit status as a parameter:

/* exit with status 1 */

**exit(1);**

- Under normal termination, exit( ) may be called either directly (i.e. exit(1)) or indirectly (i.e. by a return statement in main( ) ).
- A parent process may wait for the termination of a child process by using the wait( ) system call.
- The wait( ) system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

**pid_t pid;**

**int status;**

**pid = wait(&status);**

**Zombie process**

A process that has terminated but whose parent has not yet called wait( ) is known as a **zombie** process.

- When a process terminates, its resources are deallocated by the operating system. Its entry in the process table must remain there until the parent calls wait( ), because the process table contains the process's exit status.
- Once the parent calls wait( ), the process identifier of the zombie process and its entry in the process table are released.

**Orphan Processes**

If a parent did not invoke wait( ) and instead terminated, thereby leaving its child processes as **orphans** are called Orphan processes.

- Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes.

- The init process periodically invokes wait( ), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

**Process Scheduling**

Process scheduling is the basis of Multi-programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive

- In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled.
- The CPU will sit idle and waiting for a process that needs an I/O operation to complete. If the I/O operation completes then only the CPU will start executing the process. A lot of CPU time has been wasted with this procedure.
- The objective of multiprogramming is to have some process always running to maximize CPU utilization.
- When several processes are in main memory, if one processes is waiting for I/O then the operating system takes the CPU away from that process and gives the CPU to another process. Hence there will be no wastage of CPU time.

**Concepts of Process Scheduling**

1. CPU–I/O Burst Cycle
2. CPU Scheduler
3. Pre-emptive Scheduling
4. Dispatcher

**CPU–I/O Burst Cycle**

Process execution consists of a **cycle** of CPU execution and I/O wait.

- Process execution begins with a **CPU burst**. That is followed by an **I/O burst.**
- Processes alternate between these two states.
- The final CPU burst ends with a system request to terminate execution.
- Hence the **First cycle** and **Last cycle** of execution must be CPU burst.

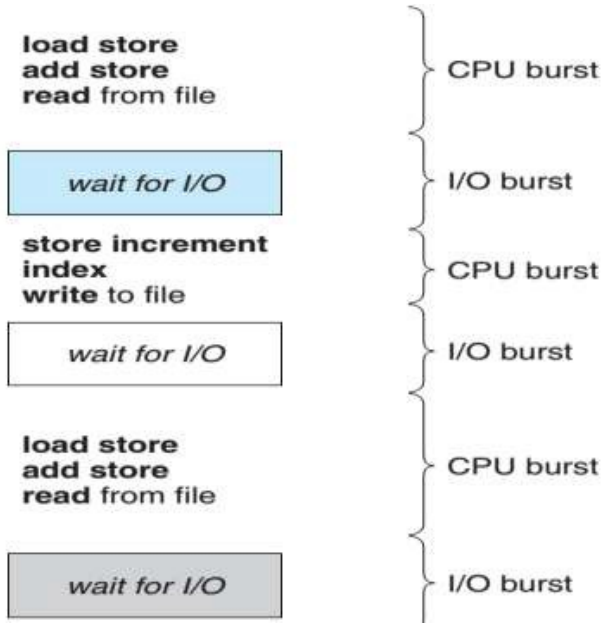



Fig: Process Scheduler

**CPU Scheduler**

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **Short-Term Scheduler** or **CPU scheduler**.

**Preemptive Scheduling**

CPU-scheduling decisions may take place under the following four cases:

1. When a process switches from the running state to the waiting state. Example: as the result of an I/O request or an invocation of wait( ) for the termination of a child process.
2. When a process switches from the running state to the ready state. Example: when an interrupt occurs
3. When a process switches from the waiting state to the ready state. Example: at completion of I/O.
4. When a process terminates. For situations 2 and 4 are considered as **Pre-emptive scheduling** situations. Mach OS X,WINDOWS 95 and all subsequent versions of WINDOWS are using Preemptive scheduling.

**Dispatcher**

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. Dispatcher function involves:

1. Switching context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart that program.

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another process running is known as the **Dispatch Latency**.

**Process Scheduling**

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

Many criteria have been suggested for comparing CPU-scheduling algorithms:

- **CPU utilization:** CPU must be kept as busy as possible. CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 to 90 percent.
- **Throughput:** The number of processes that are completed per time unit.
- **Turn-Around Time:** It is the interval from the time of submission of a process to the time of completion. 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/O.
- **Waiting time:** It is the amount of time that a process spends waiting in the ready queue.
- **Response time:** It is the time from the submission of a request until the first response is produced. Interactive systems use response time as its measure.

***Note:*** It is desirable to maximize CPU utilization and Throughput and to minimize Turn- round Time, Waiting time and Response time.

## Process Scheduling Algorithms

Process/CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. Different CPU-scheduling algorithms are:

1. First-Come, First-Served Scheduling (FCFS)
2. Shortest-Job-First Scheduling (SJF)
3. Priority Scheduling
4. Round Robin Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

**First-Come, First-Served Scheduling (FCFS)**

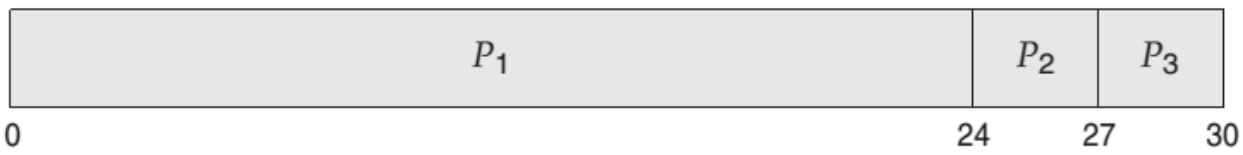
In FCFS, the process that requests the CPU first is allocated the CPU first.

- FCFS scheduling algorithm is Non-preemptive.
- Once the CPU has been allocated to a process, it keeps the CPU until it releases the CPU.
- FCFS can be implemented by using FIFO queues.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

**Example:1** Consider the following set of processes that arrive at time 0. The processes arearrived in the order P1, P2, P3, with the length of the CPU burst given in milliseconds.

| Process | Burst Time |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

Gantt Chart for FCFS is:



The average waiting time under the FCFS policy is often quite long.

- The waiting time is 0 milliseconds for process *P*1, 24 milliseconds for process *P*2 and 27 milliseconds for process *P*3.
- Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

**Convoy Effect in FCFS**

Convoy effect means, when a big process is executing in CPU, all the smaller processes must have to wait until the big process execution completes. This will affect the performance of the system.

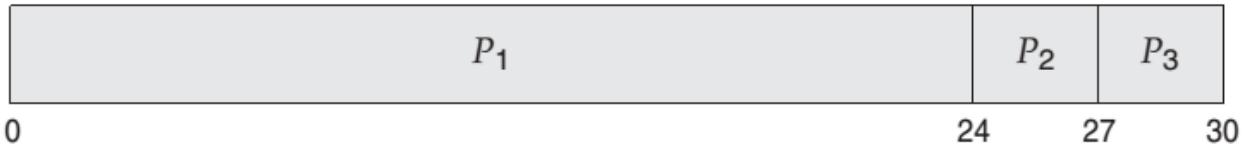
**Example:2** Let us consider same example above but with the processes arrived in the order

P2, P3, P1.

The processes coming at P2, P3, P1 the average waiting time (6 + 0 + 3)/3 = 3 milliseconds whereas the processes are coming in the order P1, P2, P3 the average waiting time is 17 milliseconds.

**Disadvantage of FCFS:**

FCFS scheduling algorithm is Non-preemptive, it allows one process to keep CPU for long time. Hence it is not suitable for time sharing systems.




**Shortest-Job-First Scheduling(SJF)**

**Shortest-Job-First Scheduling (SJF)**

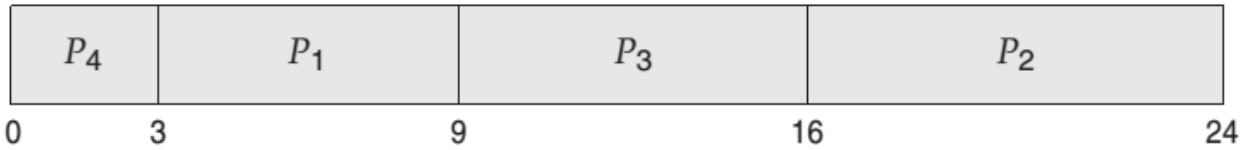
SJF algorithm is defined as **"**when the CPU is available, it is assigned to the process that has the smallest next CPU burst**".** If the next CPU bursts of two processes are the same, FCFS scheduling is used between two processes.

SJF is also called as **Shortest-Next CPU-Burst** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

**Example:** Consider the following processes and CPU burst in milliseconds:

| Process | Burst Time |
|---|---|
| P1 | 6 |
| P2 | 8 |
| P3 | 7 |
| P4 | 3 |

**Gantt Chart of SJF algorithm:**




- **By looking at the table the average waiting time by using SJF algorithm is 7ms.**
- **SJF gives the minimum average waiting time for a given set of processes. SJF is optimal.**
- **The average waiting time decreases because moving a short process before long process decrease the waiting time of the short process more than it increases the waiting time ofthe long process.**

**Difficulty with SJF**

**The difficulty with the SJF algorithm is —knowing the length of the next CPU request. With Short-Term Scheduling, there is no way to know the length of the next CPU burst. It is not implemented practically.**

| Process | Burst Time(ms) | Waiting Time |
|---|---|---|
| P1 | 6 | 3 |
| P2 | 8 | 16 |
| P3 | 7 | 9 |
| P4 | 3 | 0 |
| Average Waiting Time | | 7ms |

**Shortest Remaining Time First Scheduling (SRTF)**

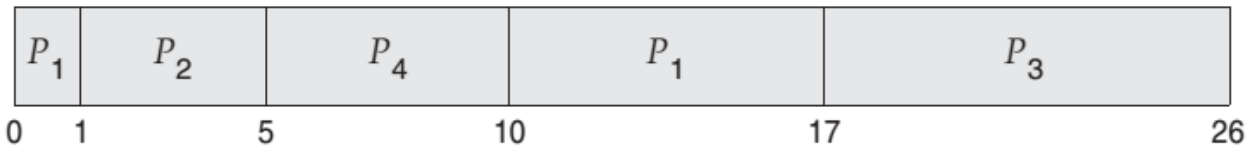
SRTF is the pre-emptive SJF algorithm.

- A new process arrives at the ready queue, while a previous process is still executing
- The next CPU burst of the newly arrived process may be shorter than the currently executing process.
- SRTF will preempt the currently executing process and executes the shortest job.

Consider the four processes with arrival times and burst times in milliseconds:

| Process | Arrival Time | Burst Time(ms) |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 1 | 4 |
| P3 | 2 | 9 |
| P4 | 3 | 5 |

**Gantt Chart for SRTF:**




**Process P1 is started at time 0, since it is the only process in the queue.**

**· Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger**

**than the time required by process P2 (4 milliseconds), so process P1 is preempted and**

**process P2 is scheduled.**

**· The average waiting time = 26/4 = 6.5 milliseconds.**

**Priority Scheduling**

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.

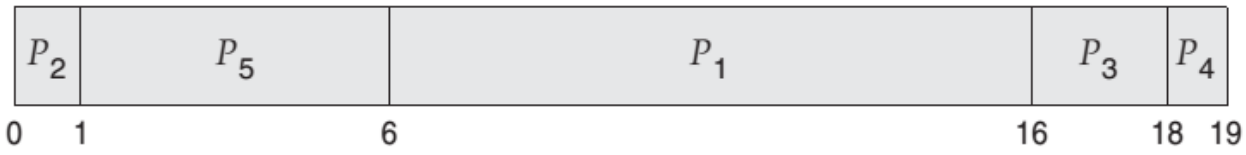
- An SJF algorithm is special kind of priority scheduling algorithm where small CPU burst will have higher priority.
- Priorities can be defined based on time limits, memory requirements, the number of open files etc.

**Example:** Consider the following processes with CPU burst and Priorities. All the processes are arrived at time t=0 in the same order. Low numbers are having higher priority.

| Process | Burst Time(ms) | Priority |
|---|---|---|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 4 |
| P4 | 1 | 5 |

| | | |
|---|---|---|
| P5 | 5 | 2 |

**Gantt chart for Priority Scheduling:**




Priority scheduling can be either Preemptive or Non-preemptive.

A Preemptive Priority Scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

Problem: Starvation or Indefinite Blocking

- In priority Scheduling when there is a continuous flow of higher priority processes has come to ready queue then all the lower priority processes must have to wait for the CPU until the all the higher priority processes execution completes.
- This leads to lower priority processes blocked from getting CPU for long period of time. This situation is called Starvation or Indefinite blocking.
- In worst case indefinite blocking may take years to execute the process.

Solution: Aging

Aging involves gradually increasing the priority of processes that wait in the system for a long time.

| Process | Burst Time(ms) | Waiting Time |
|---|---|---|
| P1 | 10 | 6 |
| P2 | 1 | 0 |
| P3 | 2 | 16 |
| P4 | 1 | 18 |
| P5 | 5 | 1 |
| Average Waiting Time | | 8.2 ms |
| | | |

**Round Robin(RR) Scheduling**

Round-Robin Scheduling (RR)

Round-Robin (RR) scheduling algorithm is designed especially for Timesharing systems.

- RR 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. New processes are added to the tail of the ready queue.

- The CPU scheduler goes around the ready queue by allocating the CPU to each process for a time interval of up to 1 time quantum and dispatches the process.
- If a process CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.

In RR scheduling one of two things will then happen:

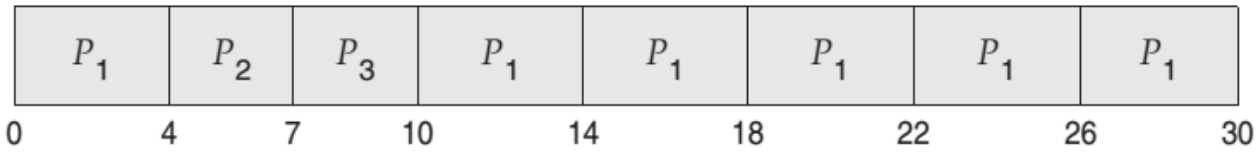
1. The process may have a CPU burst of less than 1 time quantum. The process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
2. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Consider the following set of processes that arrive at time 0 and the processes are arrived in

the order P1, P2, P3 and Time Quanta=4.

| Process | Burst Time (ms) |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

**Gantt chart of Round Robin Scheduling:**




- If we use a time quantum of 4 milliseconds, then process *P*1 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 *P*2.
- CPU burst of Process *P*2 is 3, so it does not need 4 milliseconds then it quits before its time quantum expires. The CPU is then given to the next process *P*3.
- Once each process has received 1 time quantum, the CPU is returned to process *P*1 for an additional time quantum.

The average waiting time under the RR policy is often long.

- *P*1 waits for 6 milliseconds (10 - 4), *P*2 waits for 4 milliseconds and *P*3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds. The performance of the RR algorithm depends on the size of the Time Quantum.
- If the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- If the time quantum is extremely small (i.e. 1 millisecond) the RR approach can result in many context switches.
- The time taken for context switch value should be a small fraction of Time quanta then the performance of the RR will be increased.

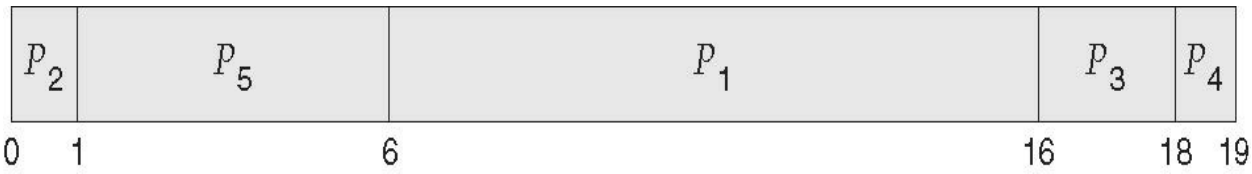
Note: *A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.*

**Priority Scheduling**

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
    - Preemptive
    - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

| Process | Arri Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

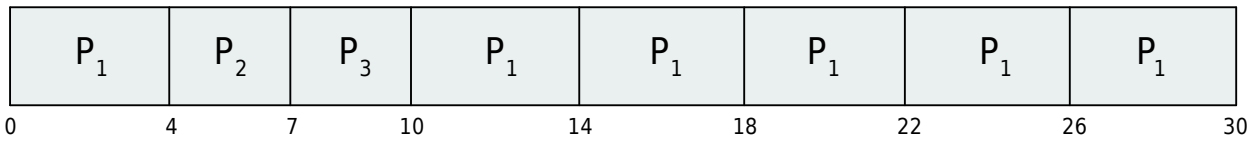
- Priority scheduling Gantt Chart
- Average waiting time = 8.2 msec




**Example of RR with Time Quantum = 4**

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- The Gantt chart is:




- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

**Multilevel Queue**

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.

- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
- 20% to background in FCFS

**Question on Multilevel Queue**

Let's take an example of a multiple queue-scheduling (MQS) algorithm that shows how the multilevel queue scheduling work. Consider the four processes listed in the table below under multilevel queue scheduling. The queue number denotes the process's queue.

| Process | Arrival Time | CPU Burst Time | Queue Number |
|---|---|---|---|
| P1 | 0 | 4 | 1 |
| P2 | 0 | 3 | 1 |
| P3 | 0 | 8 | 2 |
| P4 | 10 | 5 | 4 |

Queue 1 has a higher priority than queue **2.** Round Robin is used in queue **1 (Time Quantum = 2),** while FCFS is used in queue **2.**


**Working**

1. Both queues have been processed at the start. Therefore, **queue 1(P1, P2)** runs first (due to greater priority) in a round robin way and finished after 7 units.
2. The process in queue 2 (P3) starts running (since there is no process in queue 1), but while it is executing, P4 enters queue 1 and interrupts P3 and then P3 takes the CPU and finishes its execution.

**Multilevel Feedback Queue**

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
    - number of queues
    - scheduling algorithms for each queue
    - method used to determine when to upgrade a process
    - method used to determine when to demote a process
    - method used to determine which queue a process will enter when that process needs service
- **Example:** Consider a system that has a CPU-bound process, which requires a burst time of 40 seconds. The multilevel Feed Back Queue scheduling algorithm is used and the queue time quantum '2' seconds and in each level it is incremented by '5' seconds. Then how many times the process will be interrupted and in which queue the process will terminate the execution?
- **Solution:**
- Process P needs 40 Seconds for total execution.
- At Queue 1 it is executed for 2 seconds and then interrupted and shifted to queue 2.

- At Queue 2 it is executed for 7 seconds and then interrupted and shifted to queue 3.
- At Queue 3 it is executed for 12 seconds and then interrupted and shifted to queue 4.
- At Queue 4 it is executed for 17 seconds and then interrupted and shifted to queue 5.
- At Queue 5 it executes for 2 seconds and then it completes.
- Hence the process is interrupted 4 times and completed on queue 5.

### Processes and CPU Burst Times

| Process | Burst Time (ms) |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

### Time Quantum for Queues

- Q1 (Highest Priority): 4 ms
- Q2 (Middle Priority): 8 ms
- Q3 (Lowest Priority): 16 ms

### Table of Execution

| Process | Queue | Start Time (ms) | End Time (ms) | Burst Time Used (ms) | Remaining Time (ms) | Status |
|---|---|---|---|---|---|---|
| P2 | Q1 | 0 | 3 | 3 | 0 | Completed |
| P3 | Q1 | 3 | 6 | 3 | 0 | Completed |
| P1 | Q1 | 6 | 10 | 4 | 20 | Demoted to Q2 |
| P1 | Q2 | 10 | 18 | 8 | 12 | Demoted to Q3 |
| P1 | Q3 | 18 | 30 | 12 | 0 | Completed |

## Real Time Scheduling

Real Time systems are categorized into two types: Soft and Hard real time systems.

- **Soft Real Time System:** It provides no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes.
- **Hard real-time systems:** A task must be serviced by its deadline. Service after the deadline has expired is the same as no service at all-time.

Issues related to Real Time CPU scheduling:

1. Minimizing Latency
2. Priority Based Scheduling
3. Rate-Monotonic Scheduling
4. Earliest-Deadline-First Scheduling
5. Proportional Share Scheduling
6. POSIX Real-Time Scheduling

## Minimizing Latency

In an Event-Driven nature of **Real Time System**, the system is waiting for an event to occur. Event may arise in software or hardware. When an event occurs, the system must respond to that event and service the event as quickly as possible.

**Event Latency** is the amount of time that elapses from when an event occurs to when it is serviced.

Two types of latencies that affect the performance of the real time systems:

### 1. Interrupt latency

- Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.
- Interrupt latency must be minimized to ensure that real time tasks receive immediate action.

**2. Dispatch latency**

- The amount of time required for the scheduling dispatcher to stop one process and start another is known as dispatch latency.
- Real-time operating systems minimize dispatch latency to provide real-time tasks with immediate access to the CPU.

**Priority Based Scheduling**

In real time operating system each process is having a deadline. Hence the scheduler for a real-time operating system must support a priority-based algorithm with preemption.

- **The process which is having lowest deadline time will be given highest priority.**
- Preemptive-priority-based scheduler only guarantees soft real-time functionality.
- Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements. Hence it uses Admission control technique. In this type of scheduling a process may have to announce its deadline requirements to the scheduler.
- The Admission-control algorithm scheduler either admits the process or rejects it. If process is admitted then there is a guarantee that the process will complete on time. If a process is rejected then the task will be not serviced by its deadline.

**Rate Monotonic Scheduling**

The rate-monotonic scheduling algorithm schedules periodic tasks using a static priority policy with pre-emption.

- Upon entering the system, each periodic task is assigned a priority inversely based on itsperiod.
- **Shorter period will be given higher priority**. Longer period will be given lower priority.
- Rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst (i.e.) every time a process acquires the CPU, the duration of its CPU burst is the same.

**Scheduling Problem 1:**

P1 and P2 are two processes and Periodic Times (i.e. **Deadlines**) and processing times as given in the table. The deadline for each process requires that it complete its CPU burst by the start of its next period.

**Condition for Scheduling possibility:**

| Process | Periodic Time(p) | Processing Time(t) |
|---|---|---|
| P1 | p1=50 | t1=20 |
| P2 | p2=100 | T2=35 |

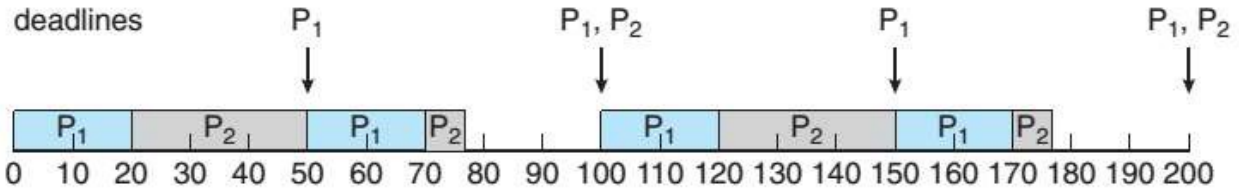
We must check whether it is possible to schedule the tasks so that each process meets its deadlines.

If we measure the CPU utilization of a process *Pi* as the ratio of its burst to its period:

- CPU utilization of *P*1 is 20/50 = 0.40 = **40%**
- CPU utilization of P2 is 35/100 = 0.35 = **35%**
- Total CPU utilization = 40 + 35= **75%**.

We can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

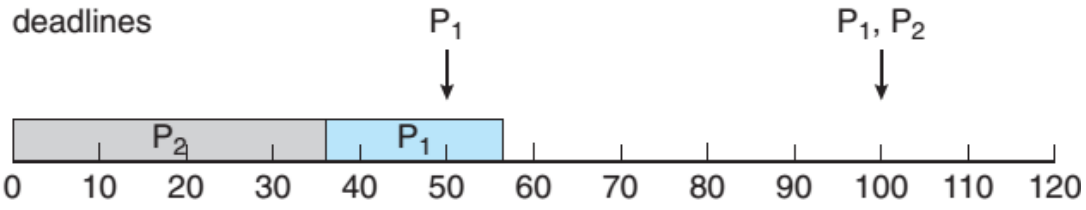
**Case 1:** By using Rate monotonic Scheduling, we assign P1 a higher priority than P2 because the period of P1 is shorter than that of P2.




i.

- P2 starts running at this point and runs until time 50.
- At this time, P2 is preempted by P1, although it still has 5 milliseconds remaining in its CPU burst.
- P1 completes its CPU burst at time 70, at which point the scheduler resumes P2.
- P2 completes its CPU burst at time 75, also meeting its first deadline.
- The system is idle until time 100, when P1 is scheduled again.

**Case 2:** We assign P2 a higher priority than P1 **(Not using Rate Monotonic Scheduling)**



- P2 starts execution first and completes at time 35.
- At this point the process P1 starts and P1 completes its CPU burst at time 55.
- The first deadline for P1 is at time 50, so the scheduler has caused P1 to miss its deadline.

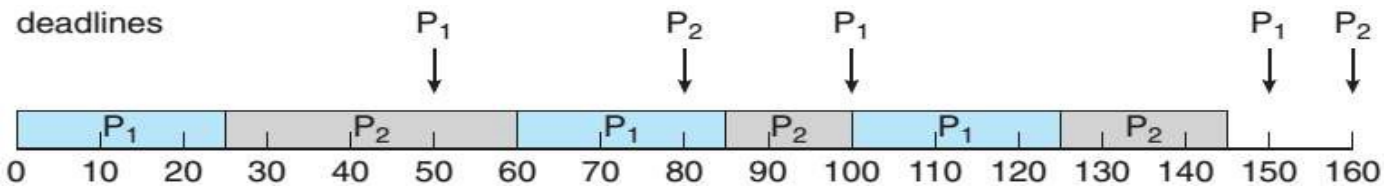
**Earliest-Deadline-First Scheduling(EDF)**

**Earliest-Deadline-First Scheduling (EDF)**

EDF scheduling dynamically assigns priorities according to deadline.

- Earlier deadline have higher priority. When a process becomes runnable, it must announce its deadline requirements to the system.
- Priorities may have to be adjusted to reflect the deadline of the newly runnable process.
- **Scheduling Problem:**

| Process | Periodic Time(p) | Processing Time(t) |
|---|---|---|
| P1 | p1=50 | t1=25 |
| P2 | p2=80 | T2=35 |



- Process P1 has the earliest deadline, so its initial priority is higher than that of process P2.
- Process P2 begins running at the end of the CPU burst for P1. However, whereas rate monotonic scheduling allows P1 to preempt P2 at the beginning of its next period at time 50, EDF scheduling allows process P2 to continue running.
- P2 now has a higher priority than P1 because its next deadline (at time 80) is earlier than that of P1 (at time 100). Thus, both P1 and P2 meet their first deadlines.
- Process P1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100.
- P2 begins running at this point, only to be preempted by P1 at the start of its next period at time 100.
- P2 is preempted because P1 has an earlier deadline (time 150) than P2 (time 160).
- At time 125, P1 completes its CPU burst and P2 resumes execution, finishing at time 145 and meeting its deadline as well.
- The system is idle until time 150, when P1 is scheduled to run once again.

**Note:** EDF scheduling is theoretically optimal but practically it is impossible.

- Theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent.
- In practice it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

**Threads: Threads and its benefits**

- **Multi-threading models**
- **Types of thread**
    - **Kernel Level thread**
    - **User level thread**
    - **Hybrid threads**

**Multithreaded Server Architecture**

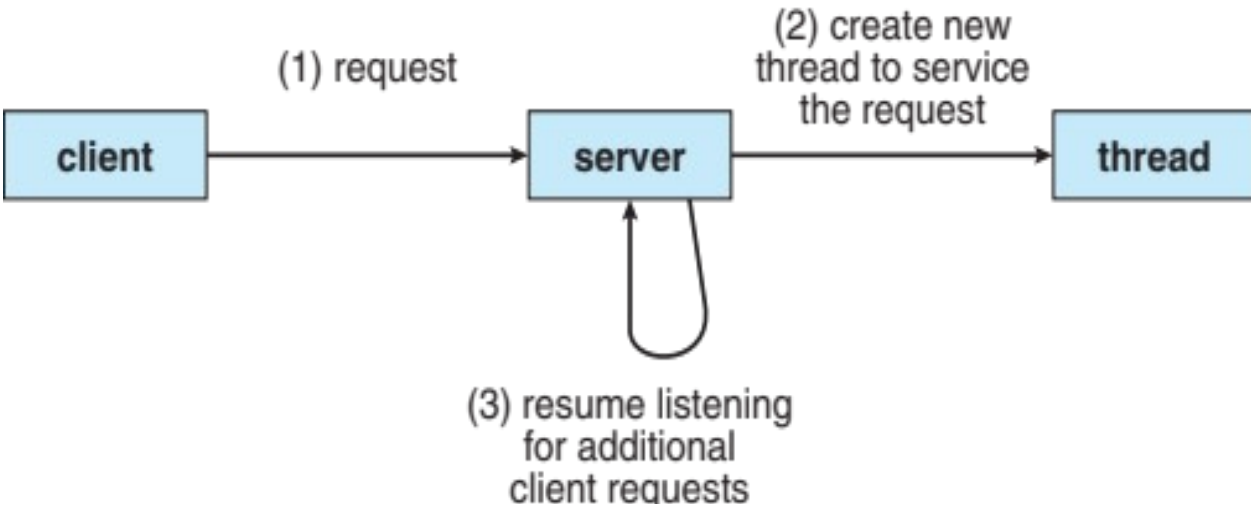



Fig 1. Multithreaded Server Architecture [1]

- Most modern applications are multithreaded Threads run within application Multiple tasks with the application can be implemented by separate threads
    - Update display

    - Fetch data
    - Spell checking
    - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

**Benefits**

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

  **Multicore Programming**

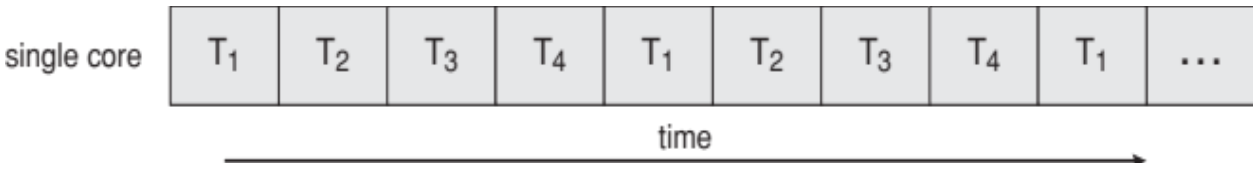
- **Multicore or multiprocessor** systems putting pressure on programmers, challenges include:
    - **Dividing activities**
    - **Balance**
    - **Data splitting**
    - **Data dependency**
    - **Testing and debugging**

- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
    - Single processor / core, scheduler providing concurrency

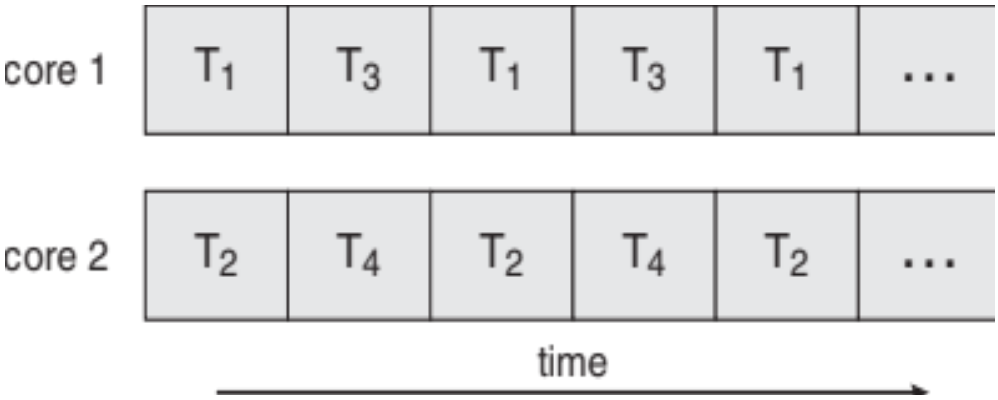
- **Types of parallelism**
    - D**ata parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
    - CPUs have cores as well as **hardware threads**
    - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

**Concurrency vs. Parallelism**

Concurrent execution on single-core system:




Parallelism on a multi-core system:




**Single and Multithreaded Processes**

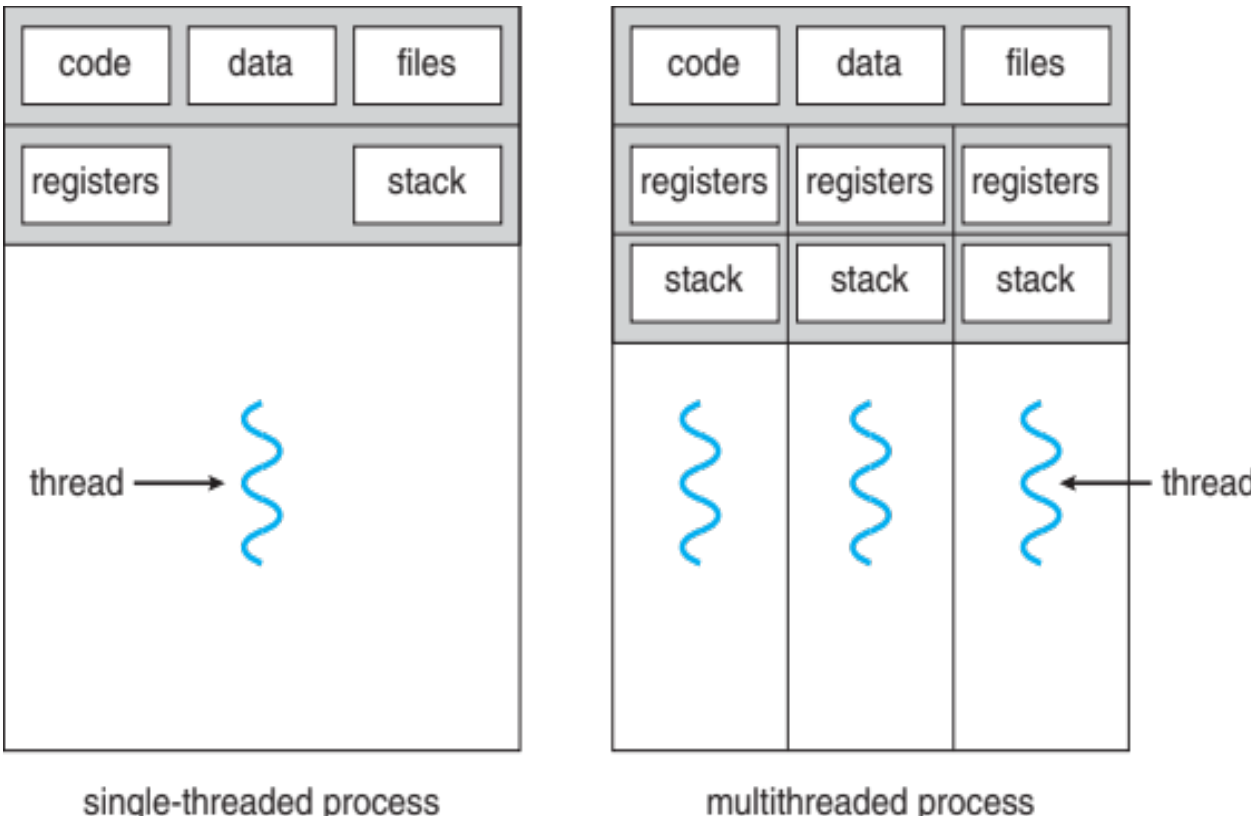



Fig . Single and Multithreaded Processes [1]

**Concurrency vs. Parallelism**

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
    - S is serial portion

- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S
- **Serial portion of an application has disproportionate effect on performance gained by adding additional cores**

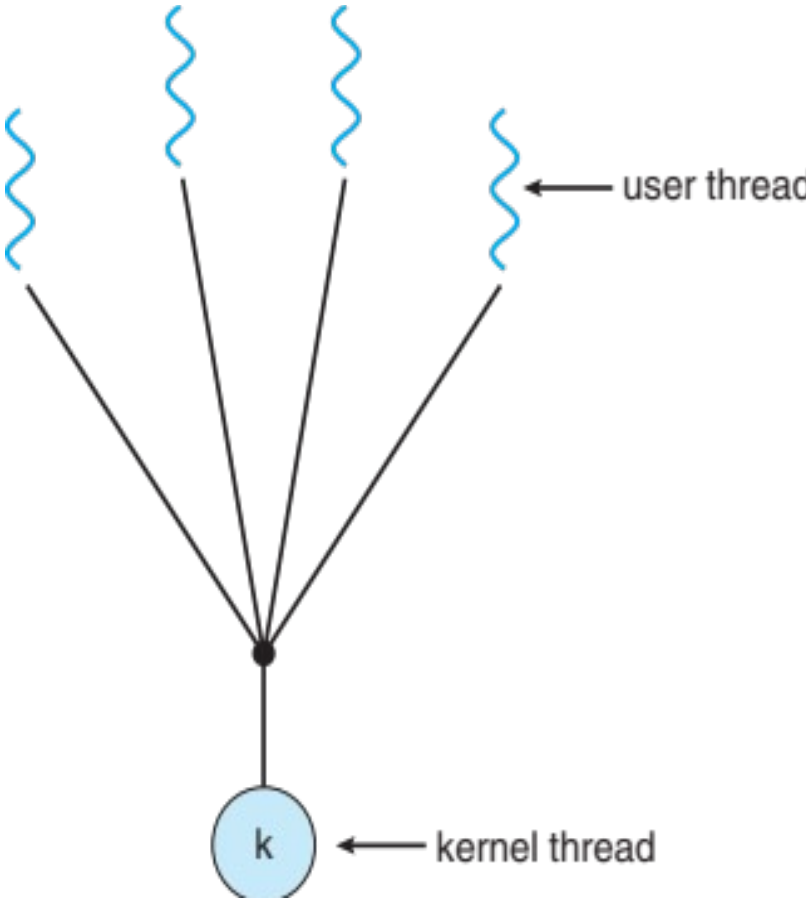
**User Threads, Kernel Threads and Hybrid Threads**

- **User threads -** management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads (Portable Operating System Interface for Unix Threads)**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
- **Hybrid thread** – Combination of both

**Multithreading Models**

- Many-to-One
- One-to-One
- Many-to-Many

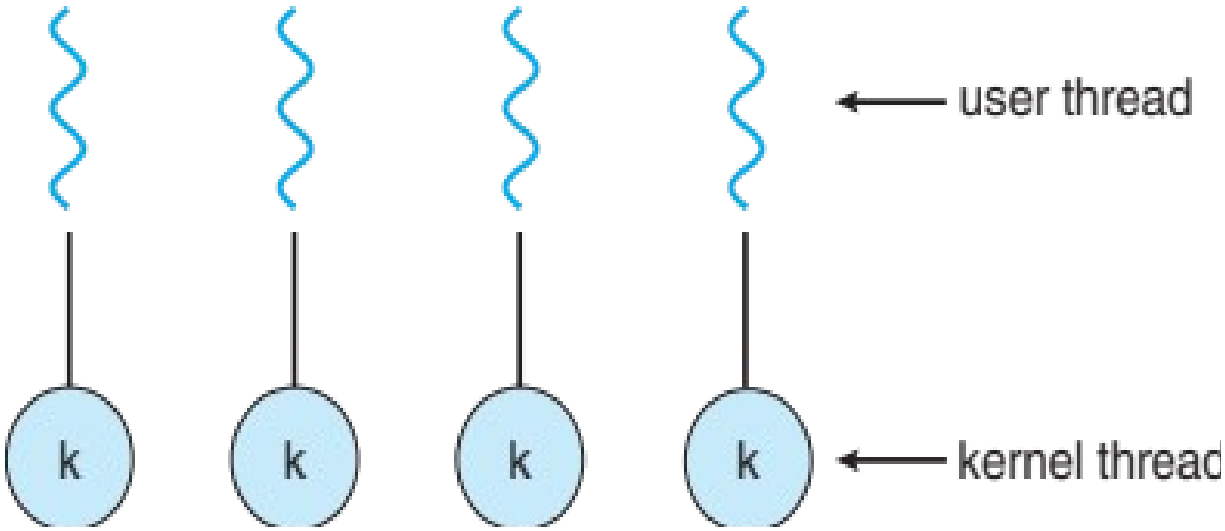
**Many-to-One Model**




- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model

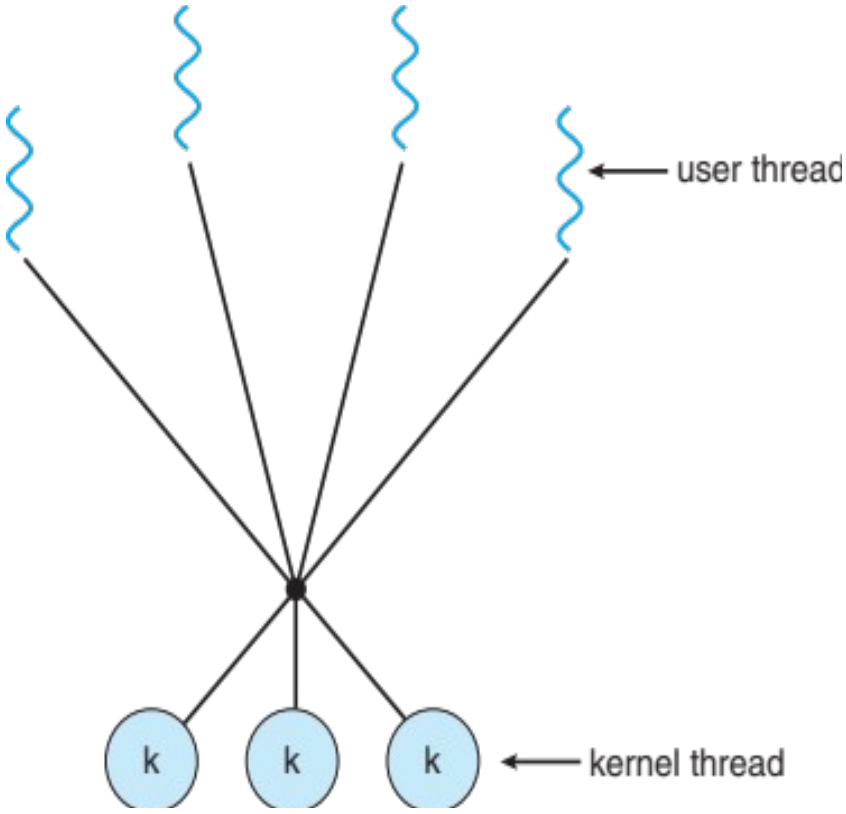
- Examples:
    - Solaris Green Threads
    - GNU Portable Threads

**One-to-One Model**



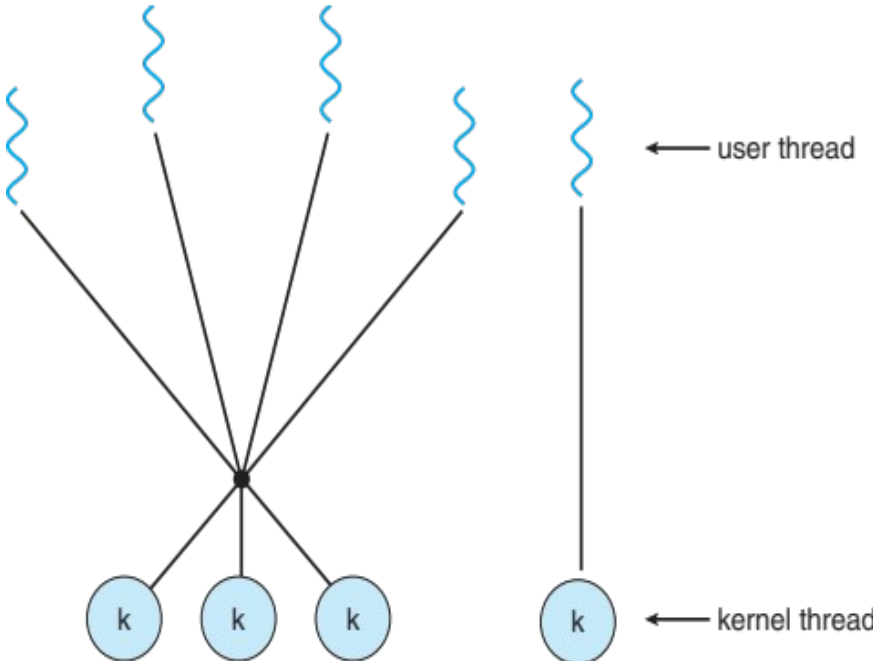

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
    - Windows
    - Linux
    - Solaris 9 and later

**Many-to-many Model**




- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the ThreadFiber package

**Two-Level Model**




- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
    - IRIX
    - HP-UX
    - Tru64 UNIX
    - Solaris 8 and earlier

**Difference between multithreading models**

| Aspect | Many-to-One Model | One-to-One Model | Many-to-Many Model |
|---|---|---|---|
| **Description** | Maps many user-level threads to one kernel thread. | Maps each user-level thread to a kernel thread. | Maps many user-level threads to many kernel threads. |
| **Concurrency** | Limited concurrency; only one thread can access the kernel at a time. | High concurrency; multiple threads can run in parallel on multiple processors. | Flexible concurrency; can handle more threads than kernel threads. |
| **Advantages** | Simple implementation; minimal overhead. | Allows true parallelism on multiprocessors; each thread can be independently scheduled. | Avoids the limitations of both many-to-one and one-to-one models; ent thread management. |
| **Disadvantages** | Blocking of one thread blocks all threads; no true parallelism. | Significant overhead for creating and managing many kernel threads. | More complex to implement than other models. |
| **Suitability** | Suitable for applications that do not require high concurrency. | Suitable for applications that require high concurrency and true parallelism. | Suitable for applications that need a balance between concurrency and resource utilization. |
| **Examples** | Green threads in early versions of Java. | Windows operating system, Linux pthreads. | Solaris operating system. |

### Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
    - Library entirely in user space
    - Kernel-level library supported by the OS

## Pthread

- May be provided either as user-level or kernel-level

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

**Common Pthread Functions**

- **pthread_create():** Creates a new thread.
- **pthread_exit():** Terminates the calling thread.
- **pthread_join():** Waits for a thread to finish.
- **pthread_mutex_init():** Initializes a mutex.
- **pthread_mutex_lock():** Locks a mutex to prevent other threads from accessing a shared resource.
- **pthread_mutex_unlock():** Unlocks a mutex.
- **pthread_cond_wait():** Waits on a condition variable.
- **pthread_cond_signal():** Signals a condition variable to wake up waiting threads

**Thread Scheduling**

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP

    - Known as **process-contention scope (PCS)** since scheduling competition is within the process
    - **Typically done via priority set by programmer**
- **Kernel thread** scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

**Pthread Scheduling**

- API allows specifying either PCS or SCS during thread creation
    - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
    - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

**Pthread Scheduling API**

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
```

```
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
/* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

**Algorithm: Checking and Setting Thread Scheduling Scope**

1. **Initialize thread attributes**:
    - Call pthread_attr_init() to get the default attributes for thread creation.
2. **Get current scheduling scope**:
    - Use pthread_attr_getscope() to retrieve the current scheduling scope.
    - Check if the returned scope is PTHREAD_SCOPE_SYSTEM (system-wide contention) or PTHREAD_SCOPE_PROCESS (process-wide contention).
3. **Set scheduling scope (if necessary)**:
    - If you need to change the scheduling scope, call pthread_attr_setscope() to set the desired scope (PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS).
4. **Create threads**:
    - Use pthread_create() to create threads using the initialized attributes.
5. **Join threads**:
    - After the threads finish, call pthread_join() to wait for all threads to complete execution.
6. **Destroy thread attributes**:
    - Once done, destroy the attribute object using pthread_attr_destroy() to release any resources.

**Threading Issues**

- Semantics of **fork()** and **exec()** system calls
- Signal handling
    - Synchronous and asynchronous
- Thread cancellation of target thread
    - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

**Semantics of fork() and exec()**

- Does **fork()**duplicate only the calling thread or all threads?
    - Some UNIXes have two versions of fork
- **exec()** usually works as normal – replace the running process including all threads

**Signal Handling**

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled by one of two signal handlers:
        - default
        - user-defined
- Every signal has **default handler** that kernel runs when handling signal
    - **User-defined signal handler** can override default
    - For single-threaded, signal delivered to process

**Thread Cancellation**

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state
- If thread has cancellation disabled, cancellation remains pending until thread enables it

•

| Mode | State | Type |
|---|---|---|
| Off | Disabled | – |
| Deferred | Enabled | Deferred |
| Asynchronous | Enabled | Asynchronous |

- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. **pthread_testcancel()**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

**Thread-Local Storage**

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **static** data
  - TLS is unique to each thread

**Scheduler Activation**

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application

- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
    - Appears to be a virtual processor on which process can schedule user thread to run
    - Each LWP attached to kernel thread
    - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads

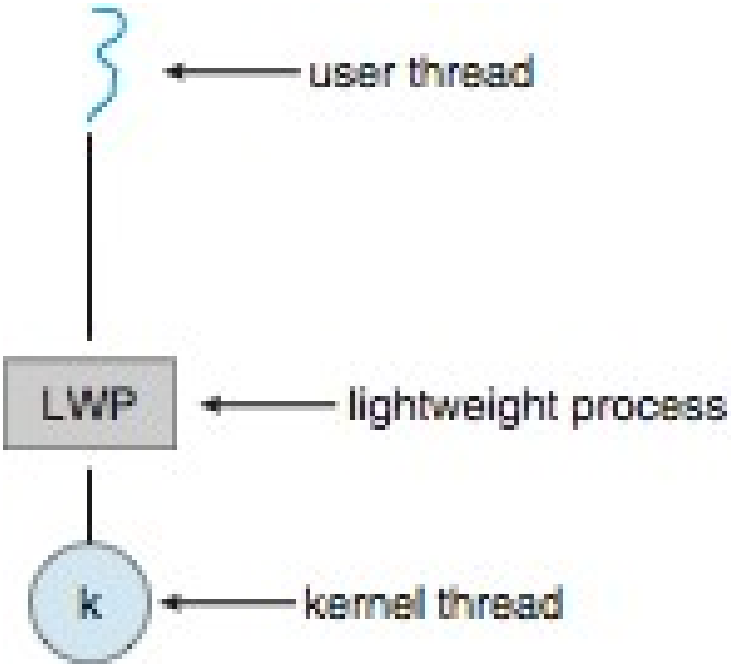



Fig 1. Scheduler Activation [1]

**Case study: Process Management in Linux**

- Linux is a modern, free operating system based on UNIX standards
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code
- Many, varying Linux Distributions including the kernel, applications, and management tools
- Version 0.01 (May 1991) had no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-drive support, and supported only the Minix file system
- Linux 1.0 (March 1994) included these new features:
    - Support for UNIX's standard TCP/IP networking protocols
    - BSD-compatible socket interface for networking programming
    - Device-driver support for running IP over an Ethernet
    - Enhanced file system
    - Support for a range of SCSI controllers for high-performance disk access
    - Extra hardware support
- Version 1.2 (March 1995) was the final PC-only Linux kernel
- Released in June 1996, 2.0 added two major new capabilities:

    - Support for multiple architectures, including a fully 64-bit native Alpha port
    - Support for multiprocessor architectures
- Other new features included:
    - Improved memory-management code
    - Improved TCP/IP performance
    - Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand
    - Standardized configuration interface
- Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems
- 2.4 and 2.6 increased SMP support, added journaling file system, preemptive kernel, 64-bit memory support
- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project
- The min system libraries were started by the GNU project, with improvements provided by the Linux community
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories

**Linux Distribution**

- Standard, precompiled sets of packages, or distributions, include the basic Linux system, system installation and management utilities, and ready-to-install packages of common UNIX tools
- The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places; modern distributions include advanced package management
- Early distributions included SLS and Slackware
- Red Hat and Debian are popular distributions from commercial and noncommercial sources, respectively
- The RPM Package file format permits compatibility among the various Linux distributions

**Linux Licensing**

- The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation
- Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product

**Design Principles**

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX **(Portable Operating System Interface**) documents; at least two Linux distributions have achieved official POSIX certification

- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

**Process Management**

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
    - The fork system call creates a new process
    - A new program is run after a call to execve
- Under UNIX, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program
- Under Linux, process properties fall into three groups: the process's identity, environment, and context

**Process Identity**

- Process ID (PID). The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process
- Credentials. Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files
- Personality. Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls

- Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX

**Stages of a Process in Linux**

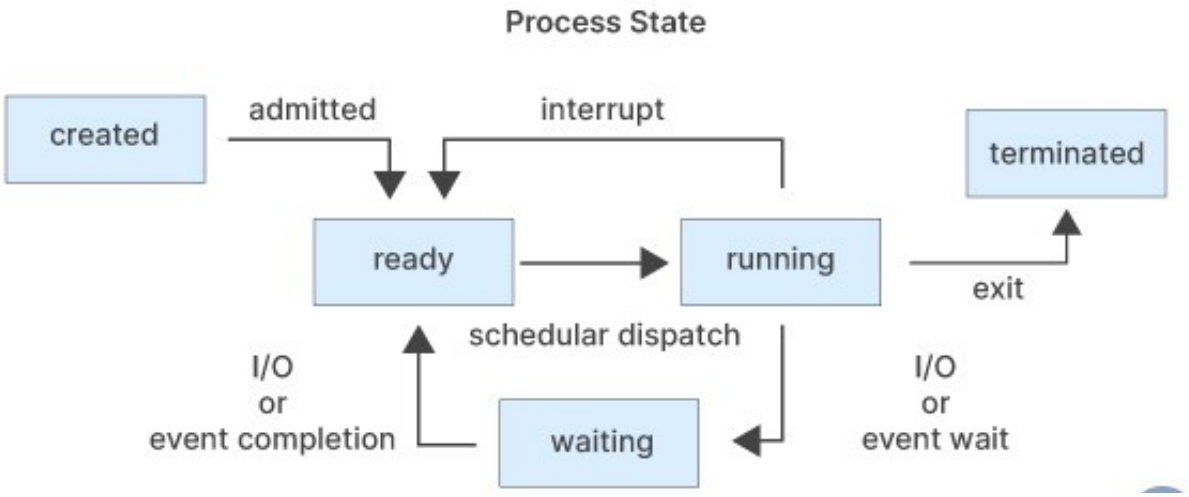



Fig 1. Stages of a Process in Linux [1]

**Stages of a Process in Linux**

1. **Created:** A process is created when a program is executed. At this stage, the process is in a "created" state, and its data structures are initialized.
2. **Ready:** The process enters the "ready" state when it is waiting to be assigned to a processor by the Linux scheduler. At this stage, the process is waiting for its turn to execute.
3. **Running:** The process enters the "running" state when it is assigned to a processor and is actively executing its instructions.

4. **Waiting:** The process enters the "waiting" state when it is waiting for some event to occur, such as input/output completion, a signal, or a timer. At this stage, the process is not actively executing its instructions.
5. **Terminated:** The process enters the "terminated" state when it has completed its execution or has been terminated by a signal. At this stage, the process data structures are removed, and its resources are freed.
6. **Zombie:** A process enters the "zombie" state when it has completed its execution but its parent process has not yet read its exit status. At this stage, the process details still have an entry in the process table, but it does not execute any instructions. The zombie process is removed from the process table when its parent process reads its exit status.

**Types of Process Management in Linux**

- **Foreground Processes:**
    - Interactive processes requiring user input.
    - Executed in the foreground and associated with a terminal.
    - Managed using:
        - Shell commands (start, stop, pause, resume).
        - "ps" command for process information.
        - Signals (e.g., "Ctrl+C").
        - Job control commands (e.g., "bg", "fg", "kill").
- **Background Processes (Non-Interactive):**
    - Run in the background without user interaction.
    - Used for system services, daemons, and long-running tasks.
    - Managed using:

- "ps" command for process information.
- "top" command for real-time monitoring.
- Methods for starting (e.g., "&" symbol, "nohup" command).
- "kill" command for termination.

**Commands Used to Manage Processes in Linux**

1. **ps:** This command is used to display information about running processes. The "ps" command can be used to list all processes or filter the list based on various criteria, such as the user who started the process, the process ID (PID), and the process status.
2. **top:** This command is used to display a real-time view of system processes. The "top" command provides information about the processes running on the system, including their resource usages, such as CPU and memory.
3. **kill:** This command is used to terminate a process. The "kill" command can be used with the process ID (PID) of the process or with a signal number to request a specific action.
4. **nice:** This command is used to adjust the priority of a process. Higher-priority processes get more CPU time than lower-priority processes. The "nice" command can be used to increase or decrease the priority of a process, which affects its CPU usage.
5. **renice:** This command is used to change or adjust the priority of a running process, which affects its CPU usage.
6. **pkill:** This command is used to send a signal to a process to request it to terminate. The "pkill" command can be used with a current process name or a regular expression to match multiple processes.
7. **top:** This command is used to display a real-time view of system processes. The "top" command provides information about the

processes running on the system, including their resource usages, such as CPU and memory.

8. **jobs:** This command is used to display a list of background jobs running in the current shell session.
9. **fg:** This command is used to move a background process to the foreground. The "fg" command can be used with the job ID of the background process.
10. **bg:** This command is used to move a suspended process to the background. The "bg" command can be used with the job ID of the suspended process.