
UNIT-II

Process Management

Syllabus:

1. **Process concept: What is the process?**
2. **Process control block**
3. **Process State Diagram**
4. **Process Scheduling- Scheduling Queues**
5. **Schedulers**
6. **Operations on Processes**
7. **Inter-process Communication(IPC)**
8. **Threading Issues**
9. **Scheduling-Basic Concepts**
10. **Scheduling Criteria**
11. **Scheduling Algorithms**

What is a Process?

- **Process** is the **execution of a program** that performs the actions specified in that program.
(or)

A program in execution is called a process.

- It can be defined as **an execution unit** where a program runs.
- **For example:**

“ when we write a program in C or Python and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the **binary code**, it becomes **a process**. “

- The **OS** helps you to **create, schedule, and terminates the processes** which is used by CPU.
- The **Process** also called as ***Job or Task***.

- Process operations can be easily controlled with the help of **PCB(Process Control Block)**.
- You can consider **PCB** as the **brain of the process**, which contains all the crucial information related to processing like **process id, priority, state, CPU registers**.
- **Process Architecture**



Text Section: A Process, sometimes known as the Text Section, also includes the **current activity** represented by the value of the **Program Counter**.

Stack: This is a special area of memory used to store information about **function calls and returns, as well as local variables**. stack contains **temporary data**.

Data Section: Contains the **global variable**. This is the **static data** used by the program, including **variables, constants, and other program data**.

Heap Section: This is a region of memory used for **dynamic allocation of memory** during **runtime**.

PCB (Process Control Block)

PCB (Process Control Block) or Task Control Block

- A process control block is a data structure used by computer operating systems to store all the information about a process. It is also known as a process descriptor.
- The PCB is a fundamental component of the operating system's process management system.
- Each process running on a system has a corresponding PCB, which contains information about the process, such as its process ID, priority level, memory allocation, CPU usage, I/O status, and state.
- As the operating system supports multiprogramming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status.

Process State
Process Number(PID)
Program Counter
CPU Register
CPU-Scheduling Information
Memory-Management Information
Accounting Information
I/O Status Information

Process Control Block

- **Process state:** The state may be new, ready running, waiting, halted, and so on.
- **Process ID (PID) :** A unique identifier assigned to each process by the operating system.
- **Program counter :** The counter indicates the address of the next instruction to be executed for this process.
- **CPU register:** 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.
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

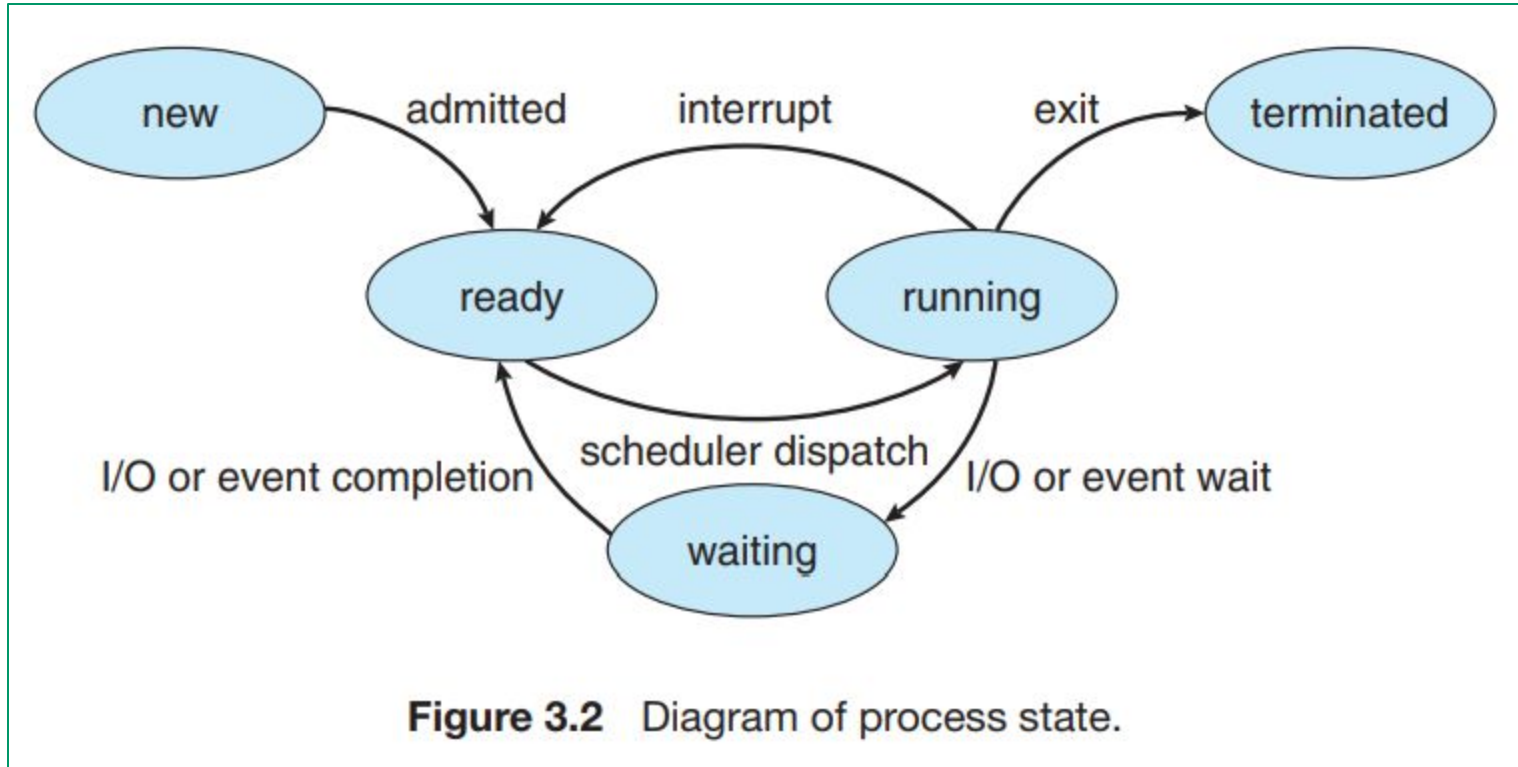
- **Memory-management information:** Includes the memory space assigned to the process, page tables, the segment tables, and virtual memory information.
- **Accounting information:** Records statistics about the process, such as the amount of CPU and real time used, time limits, I/O operations completed, and number of page fault.
- **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 States ::

Process States:

- ★ The process in an operating system passes from different states starting from its creation to its completion.
- ★ A process consists of program data and its associated data and a process control block (PCB).
- ★ A process may change its state because of the following events like I/O requests, interrupt routines, synchronization of processes, process scheduling algorithms, etc.
- ★ The process may run or may not and if it is running then that has to be maintained by the systems for appropriate progress of the process to be gained.

Process States:



Process States:

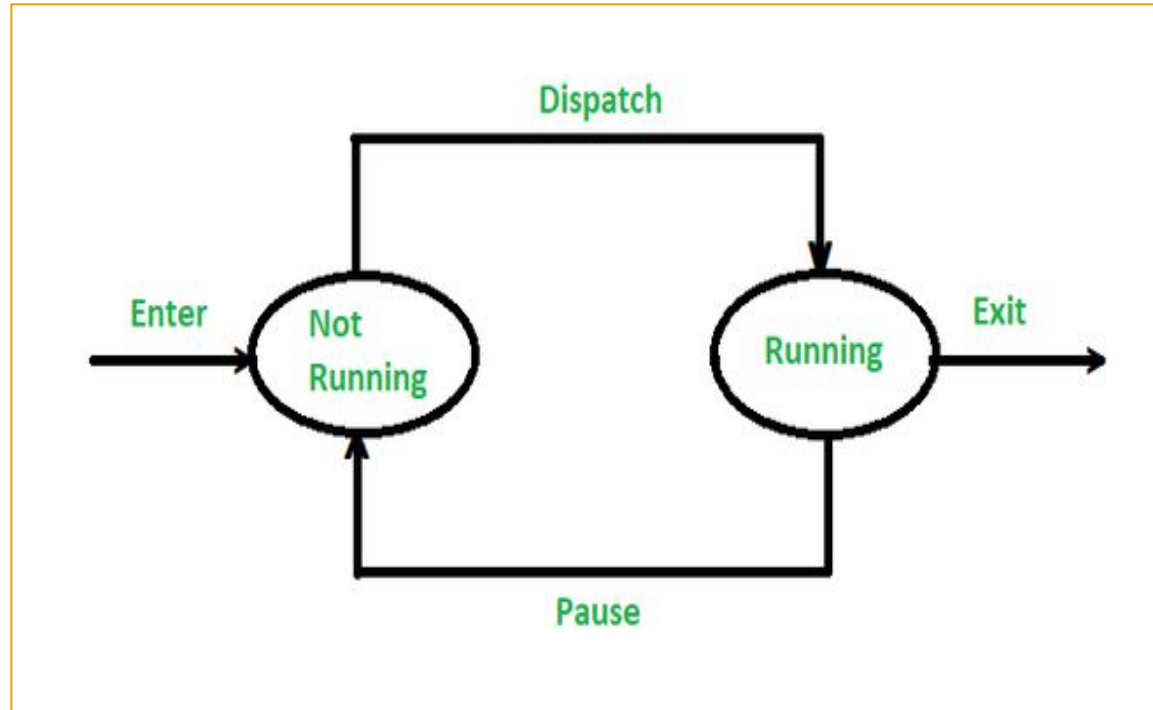
The process, from its creation to completion, passes through various states. The minimum number of states is five.

1. **New:** The process is being created
2. **Running:** Instructions are being executed
3. **Waiting:** The process is waiting for some event to occur
4. **Ready:** The process is waiting to be assigned to a processor
5. **Terminated:** The process has finished execution

Process Models:

- 1. Two-State Process Model**
- 2. Five-State Process Model**

*Two-State Process Model:

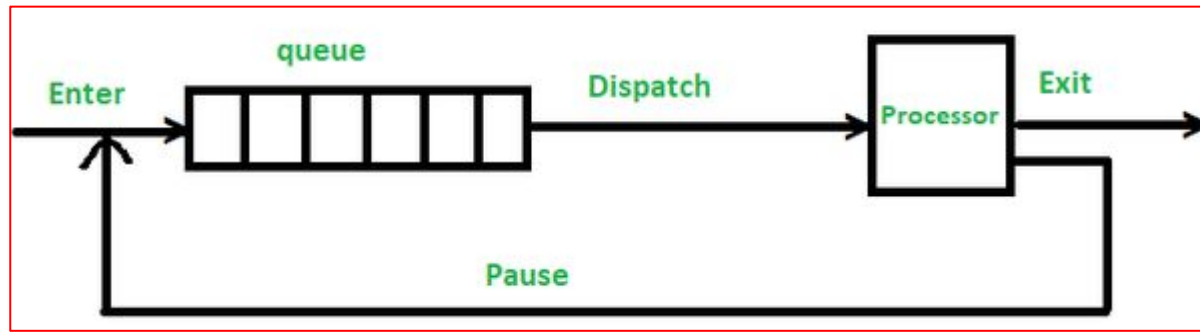


The simplest model in the process state will be a two-state model as it consists of only two states that are given below:

1. **Running State** : A state in which the process is currently being executed.
2. **Not Running State** : A state in which the process is waiting for execution.

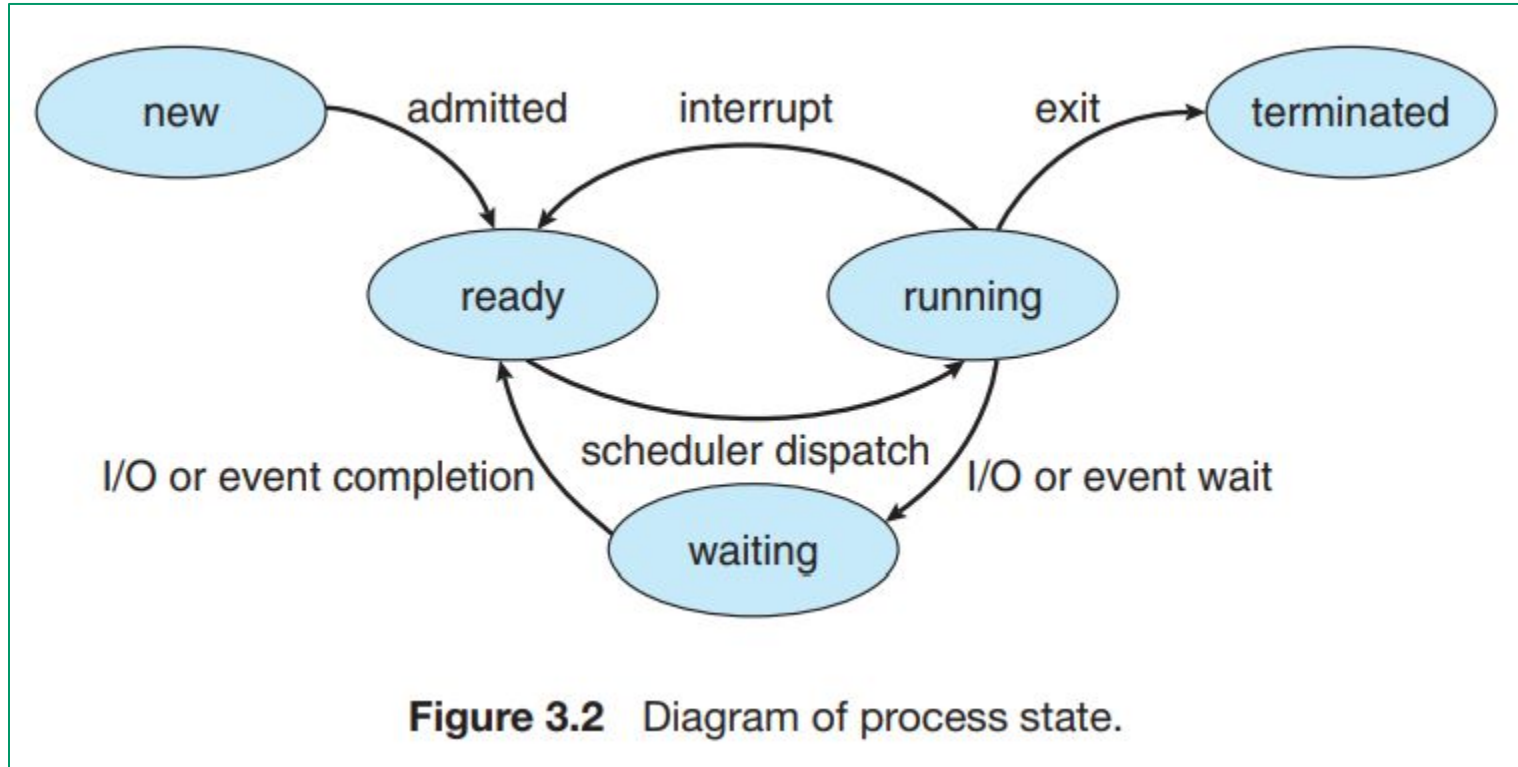
Execution of Process in Two-state Model:

- ★ 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 process 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.
- ★ a process that is interrupted is transferred to the queue waiting process, and if the process has been completed, it is terminated. After that again dispatcher takes another process from the queue and executes that process.

Five-State Process Model



1. New:

- ★ When a process is created, it is in the new state. The OS has allocated the necessary resources for the process, but it has not yet started executing.

2. Ready:

- ★ After the creation of a process, the process enters the ready state i.e. **the process is loaded into the main memory from secondary memory.**
- ★ The process here is ready to run and is waiting to get the **CPU time for its execution.**
- ★ Processes that are ready for execution by the CPU are maintained in a queue for ready processes.

3. Running :

- ★ The process is chosen by CPU for execution and **the instructions within the process are executed by any one of the available CPU cores.**

4. Blocked or wait:

- ★ Whenever the process requests access to I/O or **needs input from the user** it enters the blocked or wait state.
- ★ The process continues to **wait in the main memory and does not require CPU**.
- ★ Once the I/O operation is completed the process **goes to the ready state**.

5. Terminated or completed:

- ★ Once the process has finished execution, Process switches from Running state to Terminated state.
- ★ Process is killed as well as PCB is deleted.

Process Scheduling

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**.

Scheduling Queues:

1. **Job queue** :: As processes enter the system, they are put into a **job queue**, which consists of **all processes in the system**.
2. **Ready queue** :: The processes that are residing in **Main Memory(RAM)** and are ready and waiting to execute are kept on a list called the **ready queue**.
3. **Device queues** ::
 - When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.
 - Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk.

The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue

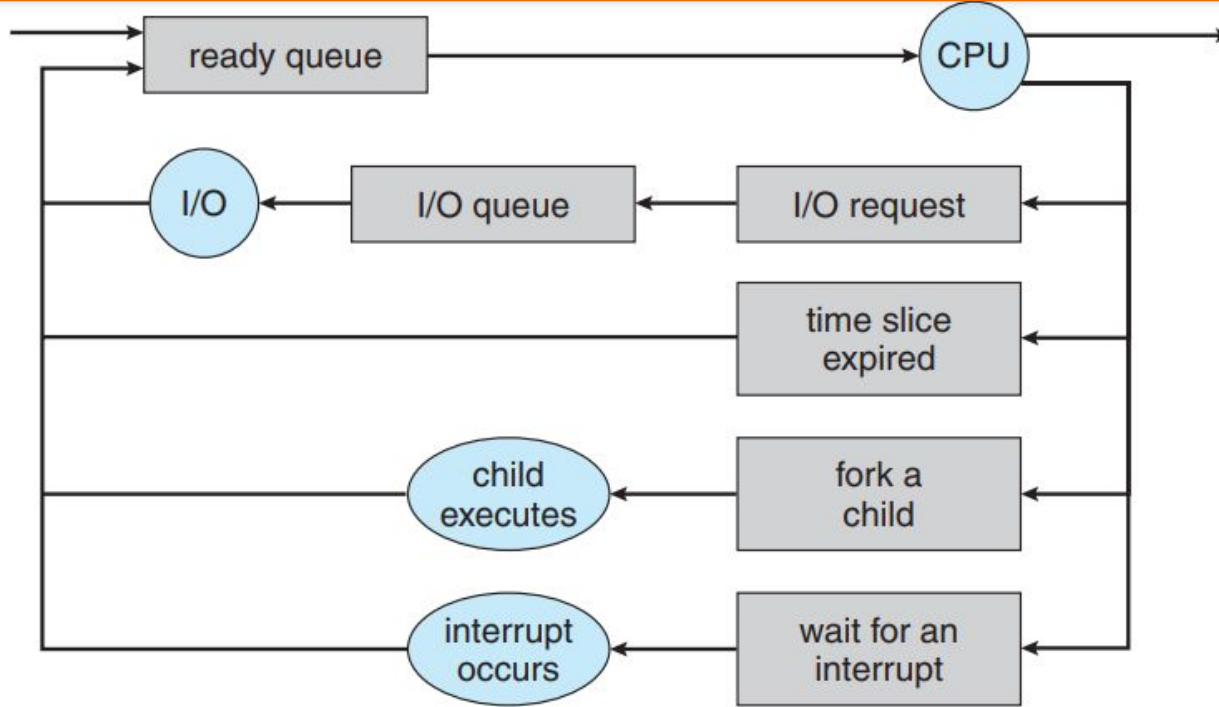


Figure 3.6 Queueing-diagram representation of process scheduling.

- Each **rectangular box** represents a queue.
- Two types of queues are present: the ready queue and a set of device queues.
- The **circles** represent the resources that serve the queues.
- The **arrows** indicate the flow of processes in the system.

- 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:
 - The process could **issue an I/O request** and then be placed in **an I/O queue**.
 - The process could **create a new child process and wait for the child's termination**.
 - The process could be removed forcibly from the CPU, as a result of **an interrupt**, and be put back in **the ready queue**.

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.

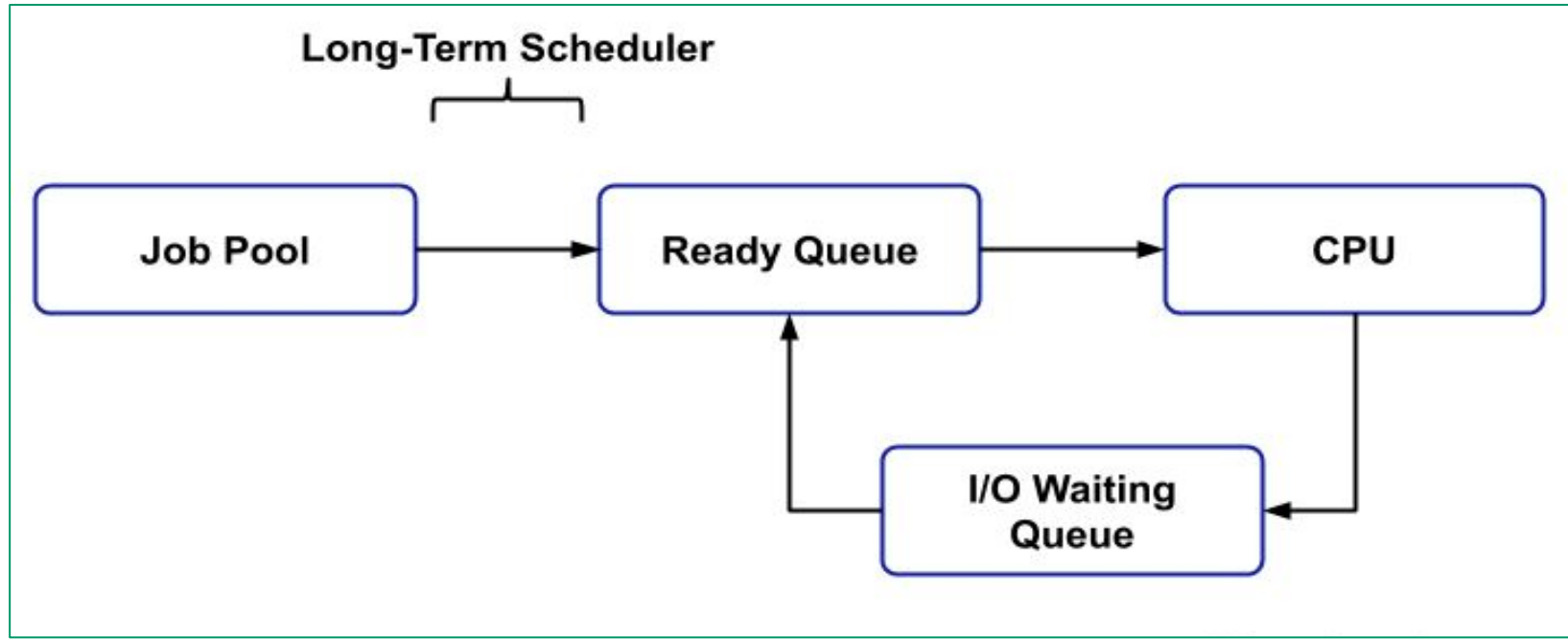
A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Process Schedulers

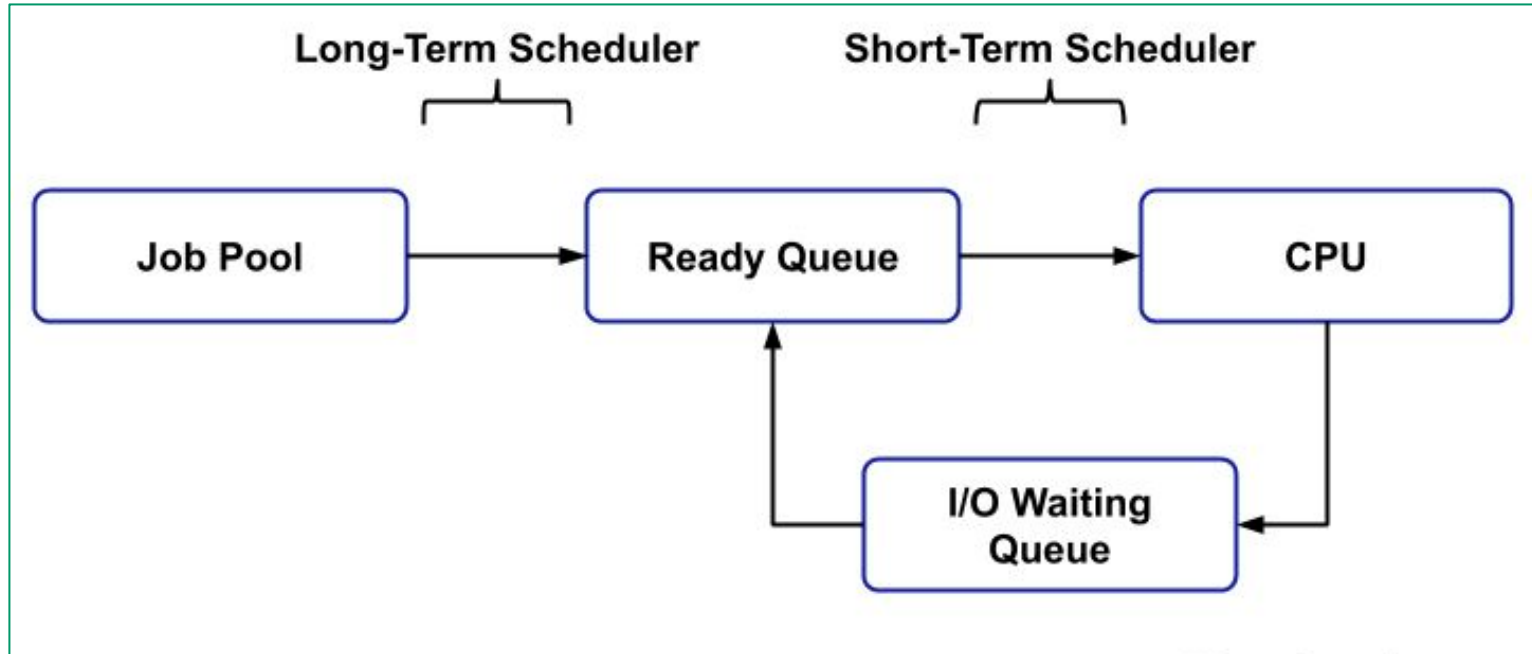
Process Schedulers

- In operating systems, a **process scheduler** is responsible for managing and scheduling the execution of processes in the system.
- The **process scheduler** is a key component of the operating system, as it ensures that the CPU is being used efficiently and fairly, and that all processes get a chance to execute.
- There are three types of process schedulers in an operating system,
 1. **Long Term Scheduler or Job Scheduler**
 2. **Short Term Scheduler or CPU Scheduler**
 3. **Medium Term Scheduler or Swapping Scheduler**

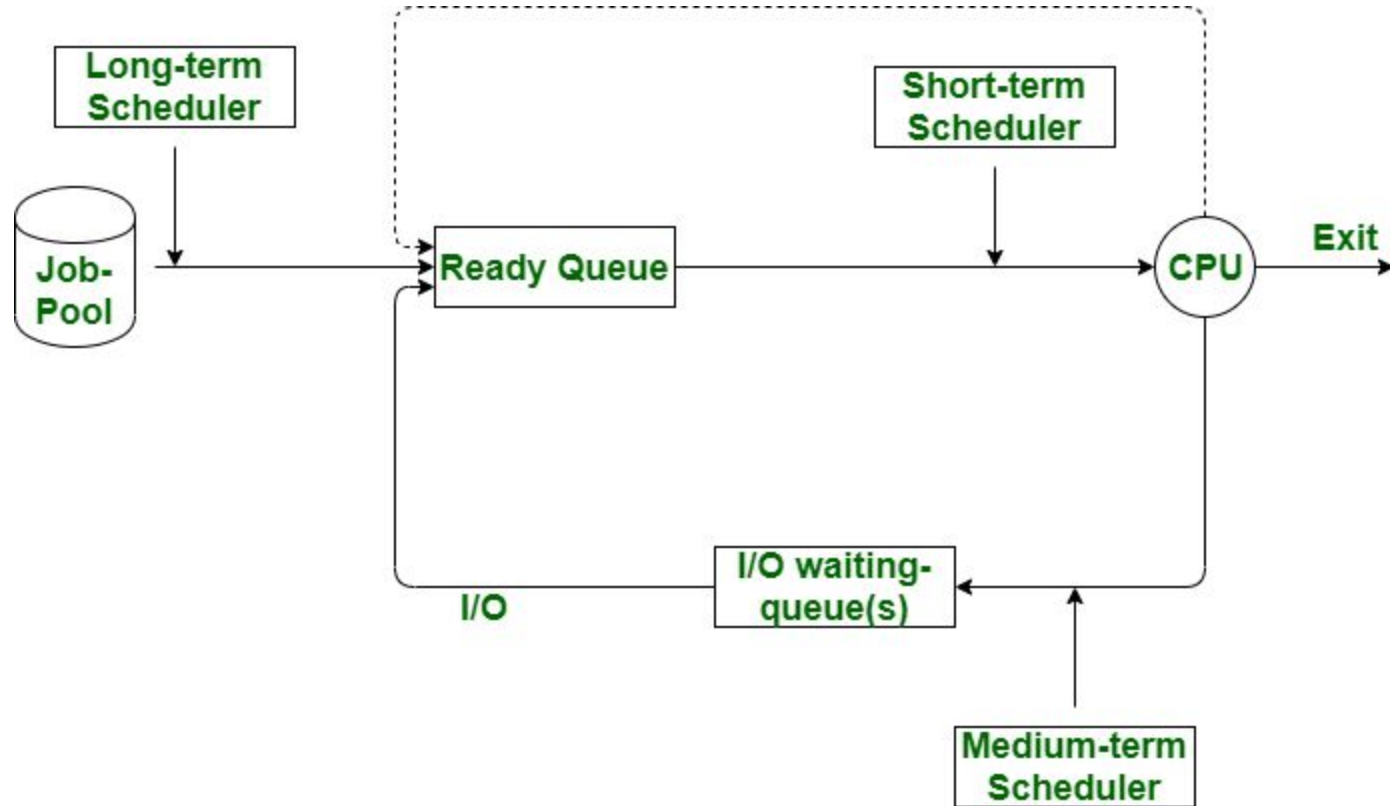
Long Term Scheduler or Job Scheduler



Long-term Scheduler vs Short-term Scheduler



Difference between Long-term, Short-term and Medium term scheduler



1.Long-Term Scheduler (Job Scheduler)

- A long-term scheduler is a scheduler that is responsible for bringing processes from the **JOB queue (or Secondary Memory) into the READY queue (or Main Memory)**
- This scheduler is responsible for selecting which processes should be admitted into the system for processing
- Its primary objective is to **maximize system throughput**, i.e., to keep the CPU busy by selecting the right processes for execution.
- The Long-term scheduler is responsible for managing **the degree of multiprogramming** in the system.i.e., managing the total processes present in the READY queue.
- It decides how many processes should be brought into the system at a time and which ones to admit.
- This decision is based on the current system load, the number of available resources, and the priority of the process.

Long-Term Scheduler (Job Scheduler)

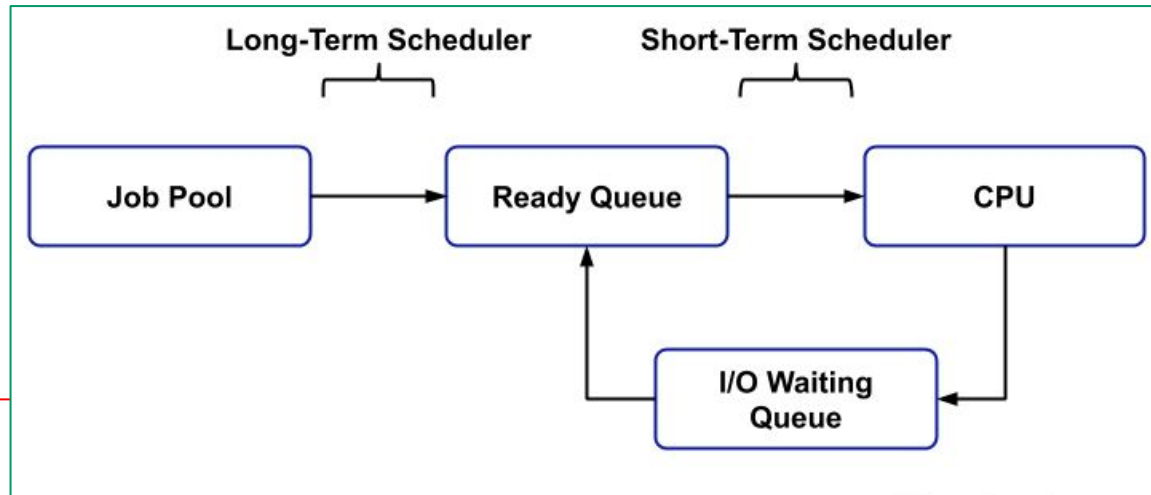
- The Long-term scheduler typically runs **less frequently** than the Short-term scheduler and Medium-term scheduler.
- Its job is to ensure that **the system is not overloaded with too many processes**, which could lead to poor performance due to excessive context switching.
- Some operating systems **do not have a long term scheduler**.
 - Example **Windows and UNIX** usually don't have a long term scheduler.
 - These systems put all the processes in **the main memory** for the short term scheduler.

2.Short-Term Scheduler (CPU Scheduler)

- The short-term scheduler is a component of an operating system (OS) that **decides which process from the ready queue should be executed next on the CPU.**
- The main goal of the short-term scheduler is to **optimize** the use of the CPU by selecting the best process to run, based on certain criteria such as the priority of the process, the length of time it has been waiting, and the amount of resources it requires.
- This scheduler **runs frequently, typically every few milliseconds**, and determines which process should be allocated the CPU for the next time slice (or quantum) of execution.
- The short-term scheduler plays a critical role in the overall performance of an operating system since it determines how efficiently the system utilizes the CPU.

3.Short-Term Scheduler (CPU Scheduler)

- The short-term scheduler is responsible for managing the CPU efficiently and ensuring that all processes receive **a fair share of CPU time**.
- It helps to minimize response time and maximize throughput by minimizing the waiting time of processes in the ready queue.



3. Medium-Term Scheduler (Swapping Scheduler)

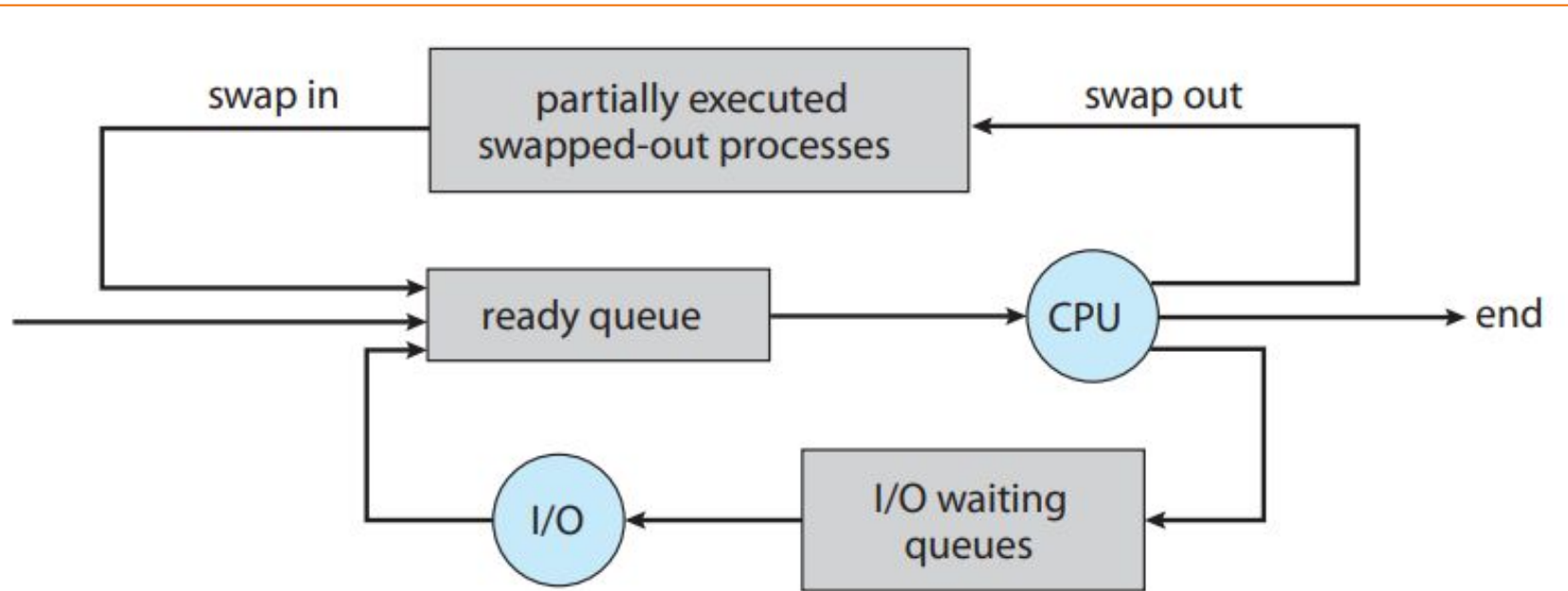


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

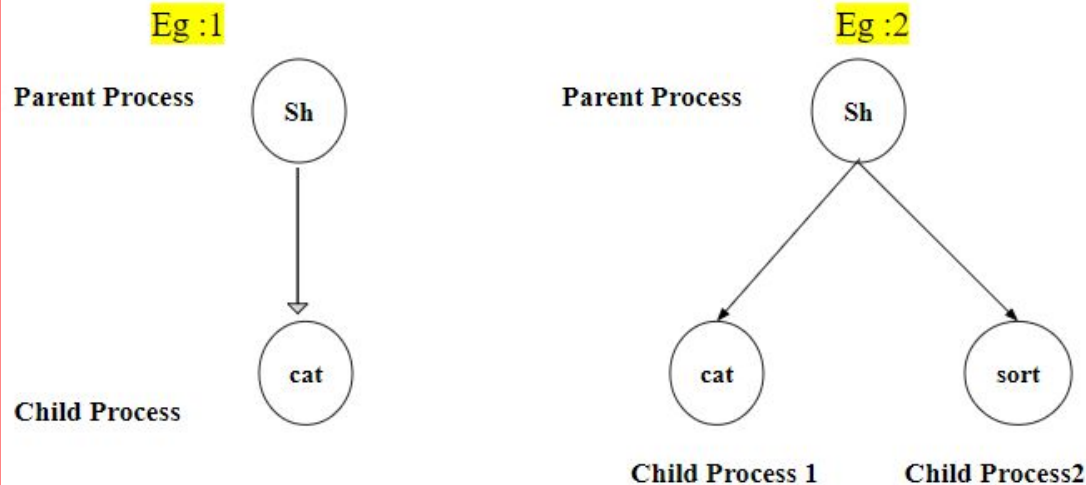
- The medium-term scheduler (also known as the "swapper") is a component of an operating system (OS) that manages the movement of processes from main memory (RAM) to secondary storage (such as a hard disk) and back.
- The medium-term scheduler selects processes from the ready queue that are ready to run, but cannot fit in the available main memory due to memory constraints, and moves them to the secondary storage. This process is known as **swapping**.
- Swapping allows the operating system to free up space in main memory for new processes to run.
- Once a process is swapped out of main memory, it is marked as **"blocked"** and is not eligible for execution until it is swapped back in.
- The medium-term scheduler is also responsible for deciding which processes to bring back into main memory when space becomes available.
- By swapping processes out of memory when they are not actively running, it helps to free up memory resources for other processes and ensure that the system runs efficiently.

S. N.	Long Term Scheduler	Short Term Scheduler	Medium Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Operations on Process

1. Process Creation :

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



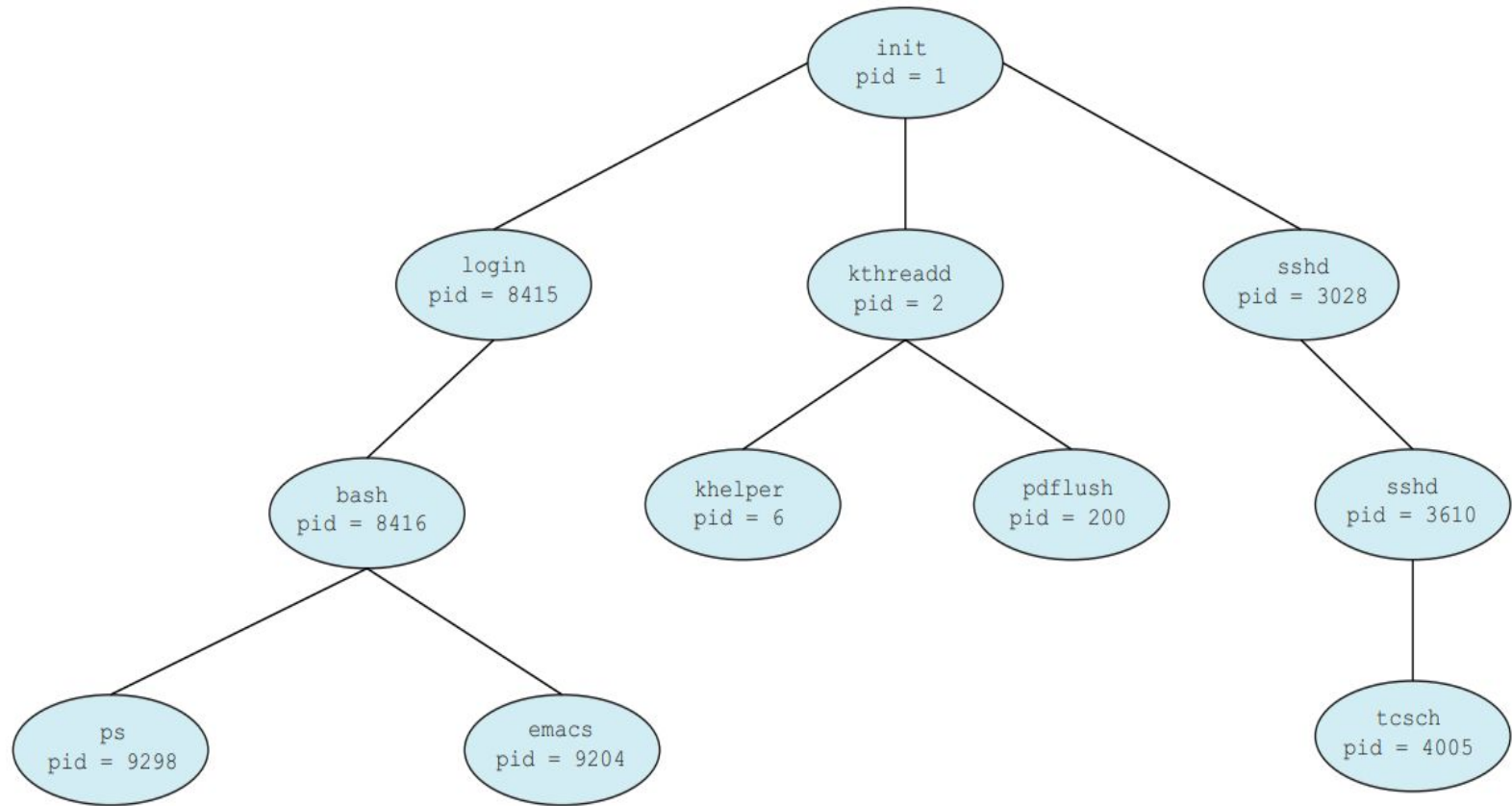


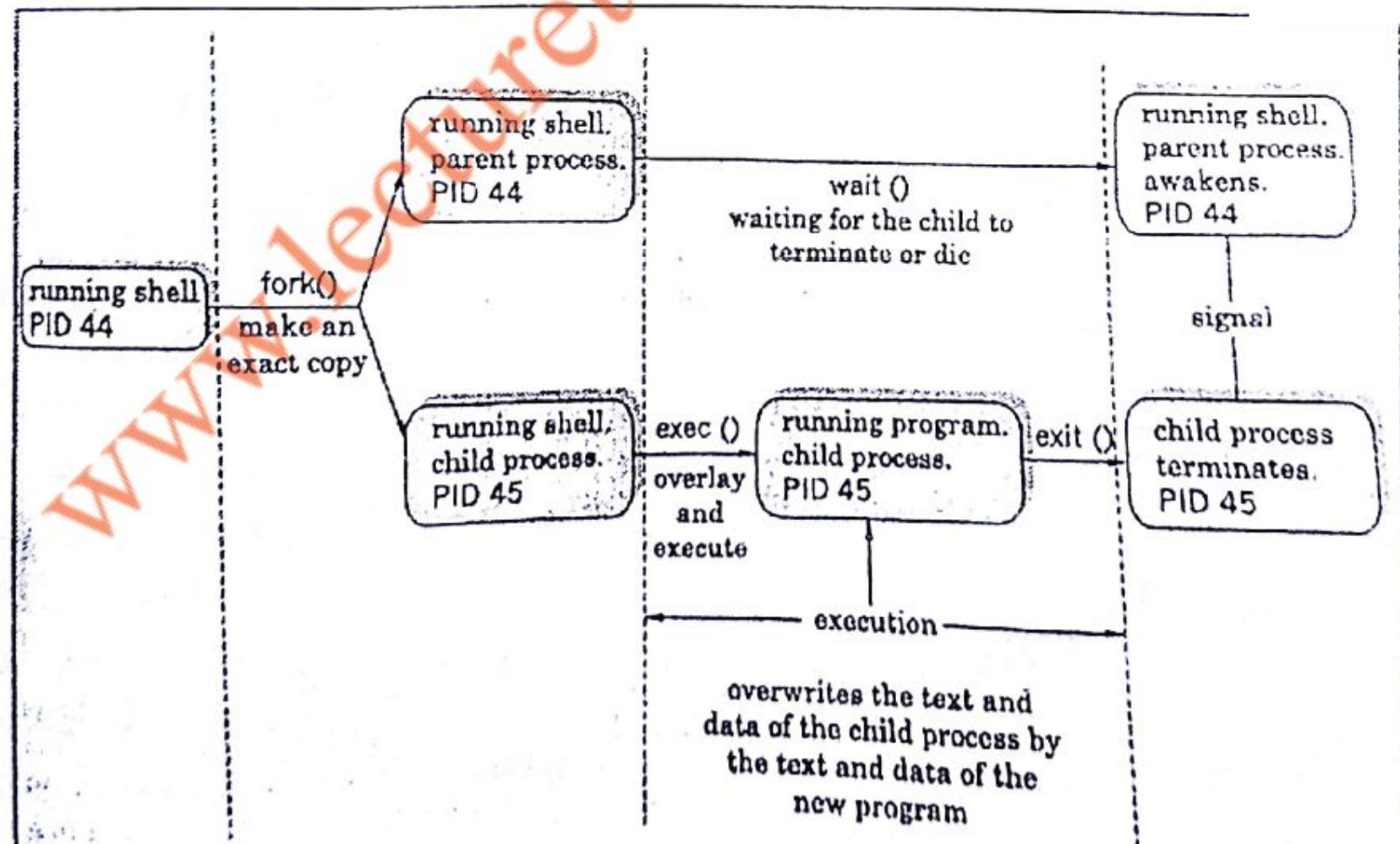
Figure 3.8 A tree of processes on a typical Linux system.

- Figure 3.8 illustrates a typical **process tree** for the Linux operating system, showing the name of each process and its **pid**.
- **The init process** (which always has a pid of 1) serves as **the root parent process for all user processes**.
NOTE: **init()** is the parent of all processes, executed by the kernel during the booting of a system.
- Once **the system has booted**, **the init process can also create various user processes**, such as a web or print server, an ssh server, and the like.
- In Figure 3.8, we see **three children of init**—(1) Login (2) kthreadd (3) sshd.
- **The kthreadd process** is responsible for creating additional processes **that perform tasks on behalf of the kernel (in this situation, khelper and pdflush)**.
- **The sshd process** is responsible for **managing clients that connect to the system by using ssh (which is short for secure shell)**.
- **The login process** is responsible for **managing clients that directly log onto the system**.
- In this example, a client has logged on and is using **the bash shell**, which has been assigned pid 8416.
- **Using the bash command-line interface**, this user has created the process **ps** as well as the **emacs** editor.

Let's first consider the UNIX operating system.

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

- Mechanism of Process Creation is depicted as,



- Forking is the first phase in the creation of a process by a process.
- The calling process (parent) makes a call to the system routine `fork()` which then makes an exact copy of itself.
- After the `fork()` there will be two processes.
- The `fork` of the parent process returns the PID of the new process, that is the child process just created
- The `fork` of the child returns a 0 (zero)
- Immediately after forking, the parent makes a system call to one of the `wait()` functions
- By doing so, the parent keeps waiting for the child process to complete its task.
- In the second phase, the `exec()` function overwrites the text and data area of the child process.
- The `exit()` function terminates the child process.
- The parent awakens only when it receives a complete signal from the child, after which it will be free to continue with its other functions

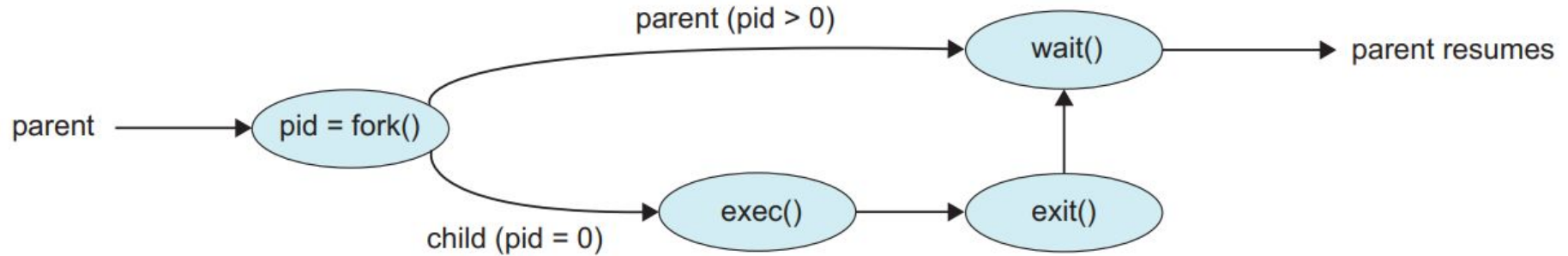


Figure 3.10 Process creation using the `fork()` system call.

- ★ 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.

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

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 address-space possibilities for 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.

2. Process Termination :

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

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 terminated**, and the operating system does not allow a child to continue if its parent terminates.

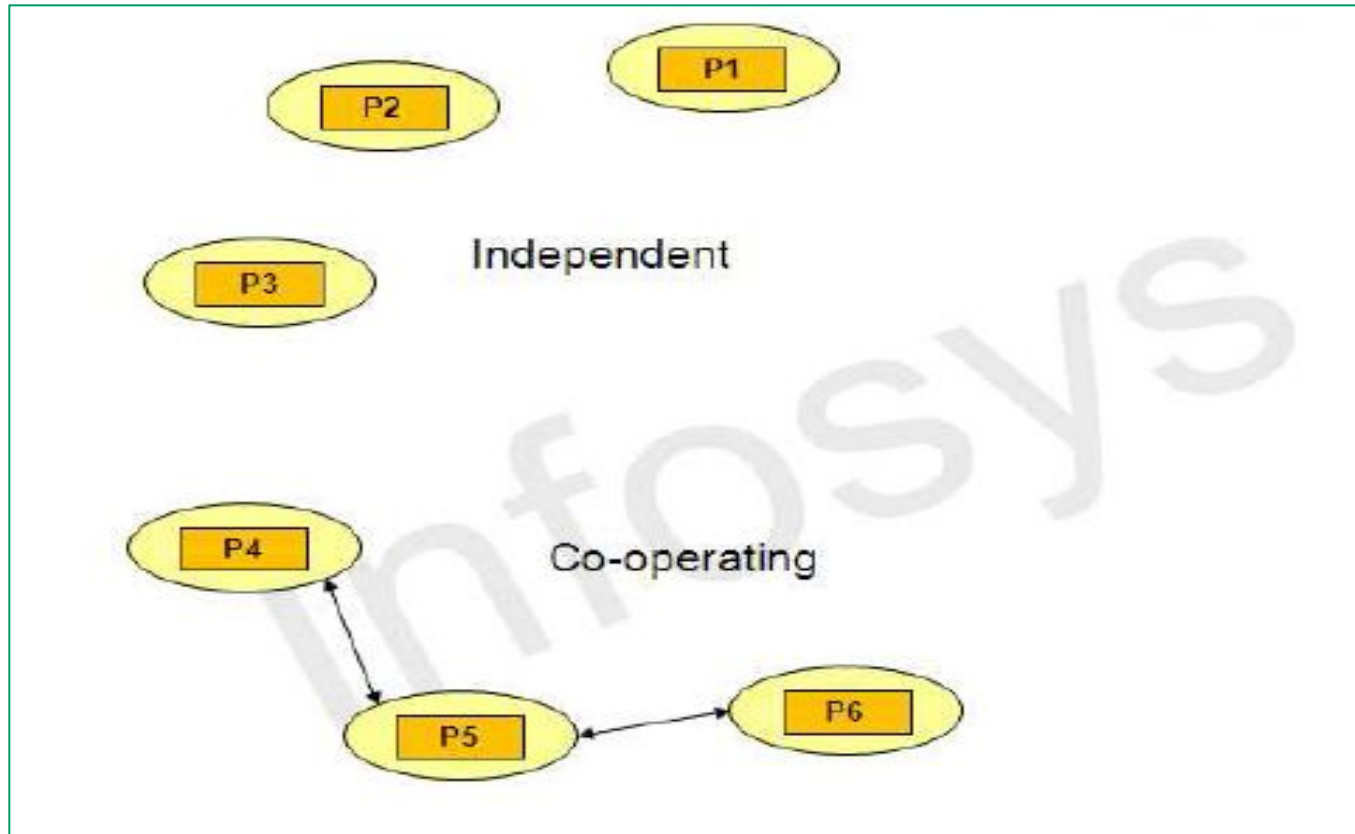


*Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.*

Interprocess Communication[IPC]

- Process executing concurrently in the operating system may be either independent process or cooperating process.
- A process can be of two types:-
 - ◆ **Independent process.**
 - ◆ **Co-operating process.**
- A process is independent if it **cannot affect or be affected** by the other process executing in the system. (does not share data)
- A process is cooperative if it **can affect or be affected** by the other processes executing in the system. (Shares data)
- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.
- Inter-process communication (IPC) refers to the mechanisms provided by the OS that enable different processes running on the same or different machines to communicate and share data with each other.
- IPC is an important aspect of modern operating systems and is used extensively in many areas, such as **distributed computing, client-server applications, and multi-core processors.**

Independent vs Co-operating 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), we must provide an environment to allow concurrent access to such information.
2. **Computation speedup**: If we want a particular task to run faster, **we must break it into subtasks, each of which will be executing in parallel with the others**. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
3. **Resource sharing**: Processes may need to share resources such as memory, CPU time, or input/output (I/O) devices. By enabling process cooperation, the operating system can ensure that resources are used efficiently and that conflicts between processes are avoided.
4. **Convenience**: Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

- ★ Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.
- ★ There are two fundamental models of interprocess communication:
 - **Shared Memory**
 - **Message Passing**

There are several methods of IPC,

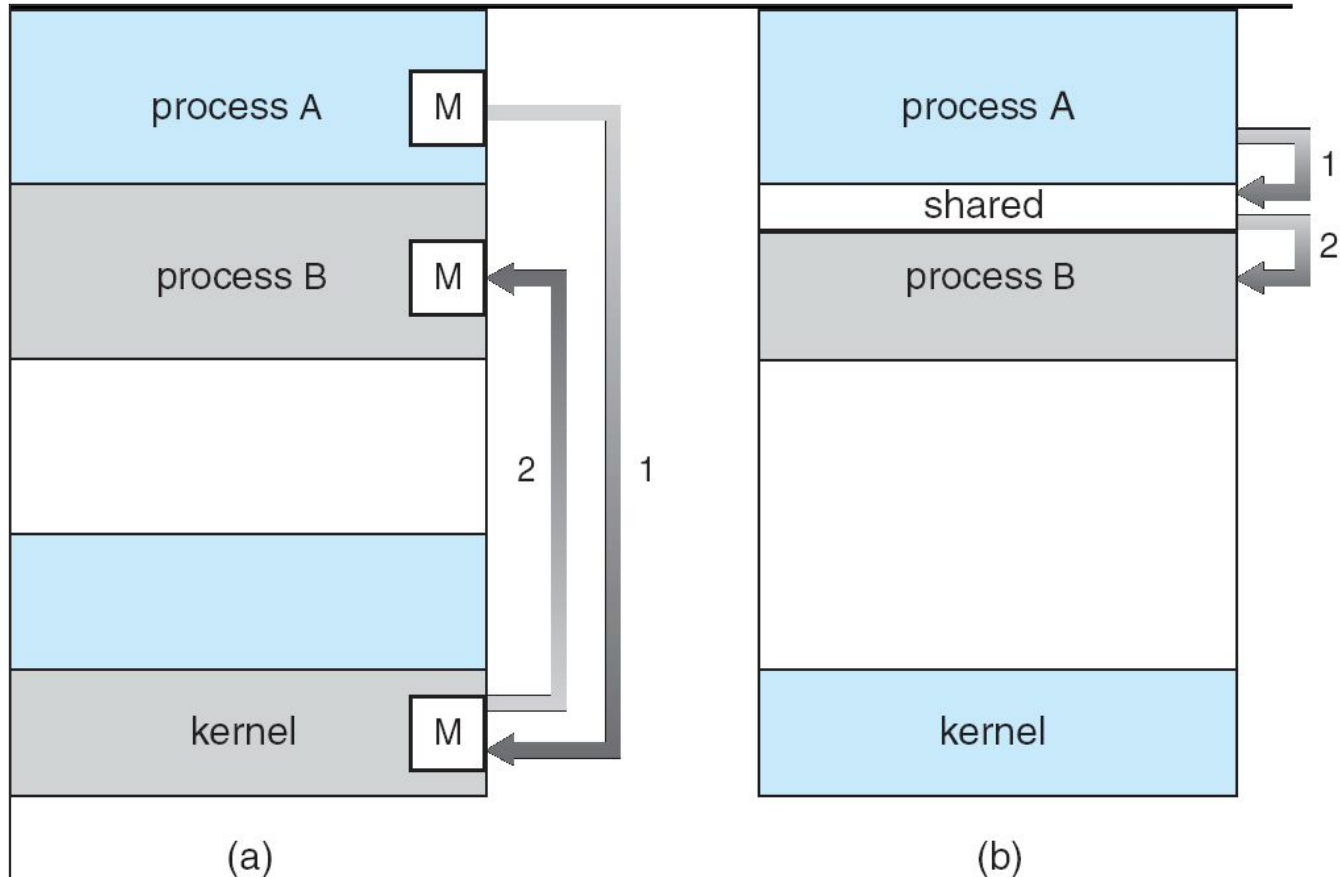
1. **Shared Memory**
2. **Message passing**
3. Sockets
4. Signals
5. pipes

Shared Memory

1. 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.
2. Shared memory can be **faster** than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
3. 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.

Message Passing

1. In the message-passing model, communication takes place **by means of messages exchanged between the cooperating processes.**
2. Message passing is useful for **exchanging smaller amounts of data**, because no conflicts need be avoided.
3. Message passing is also **easier to implement** in **a distributed system** than shared memory.



Communications models :: (a) Message passing. (b) Shared memory.

1. Shared-Memory Systems:

- ★ Shared memory is a technique used by operating systems to allow multiple processes to share a portion of the memory.
- ★ It is a form of interprocess communication that allows processes to exchange information quickly and efficiently without the overhead of copying data between them.
- ★ In shared memory, a portion of the memory is mapped into the address space of multiple processes. These processes can then read and write to the shared memory region as if it were their own private memory.
- ★ To implement shared memory, the operating system provides a set of system calls that allow processes to create, attach, and detach from shared memory regions.
- ★ Shared memory is used in many types of applications, including databases, multimedia processing, scientific computing, and network protocols.
- ★ It is particularly useful in situations where large amounts of data need to be shared between multiple processes, and copying the data would be impractical or slow.

Ex: Producer Consumer Problem

The producer-consumer problem is an example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer that shares a common fixed-size buffer use it as a queue.

- ★ The producer's job is to generate data, put it into the buffer, and start again.
- ★ At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

Below are a few points that considered as the problems occur in Producer-Consumer:

1. The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
2. Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
3. Accessing memory buffer should not be allowed to producer and consumer at the same time.

2. Message passing

- ★ Message passing is a common technique used in interprocess communication (IPC) to allow processes to exchange information with each other.
- ★ In message passing, processes send messages to each other through a communication channel, which can be implemented in various ways, such as **pipes, sockets, or message queues.**
- ★ In message passing, the sending process creates a message containing the data to be sent and sends it to the receiving process through the communication channel.
- ★ The receiving process then retrieves the message and processes the data contained in it.
- ★ There are two main types of message passing:
 - a. **Synchronous message passing**
 - b. **Asynchronous message passing**

2. Message passing

★ Synchronous Message Passing:

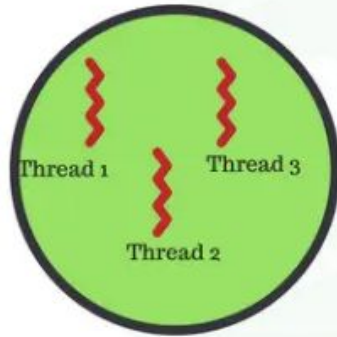
In this type of message passing, the sender process blocks until the receiver process acknowledges receipt of the message. This ensures that the sender process does not proceed until it knows that the receiver has received the message.

★ Asynchronous Message Passing:

In this type of message passing, the sender process does not wait for the receiver process to acknowledge receipt of the message. This allows the sender process to proceed immediately after sending the message without waiting for the receiver

Message passing can be used in a wide range of applications, including **client-server systems, distributed computing, and parallel processing.**

Process

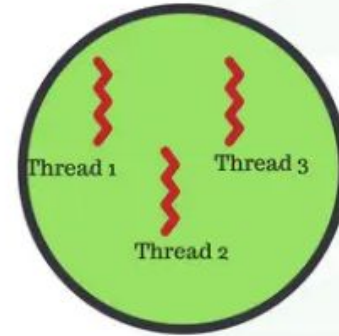


A thread has the following -

- Thread ID
- Program Counter
- Register
- Stack

Threads

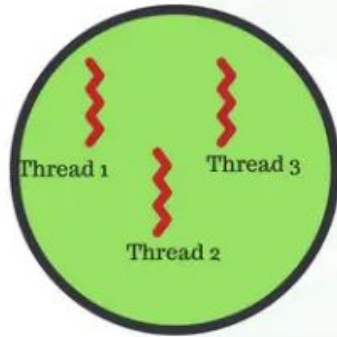
Process



A thread has the following -

- Thread ID
- Program Counter
- Register
- Stack

Process

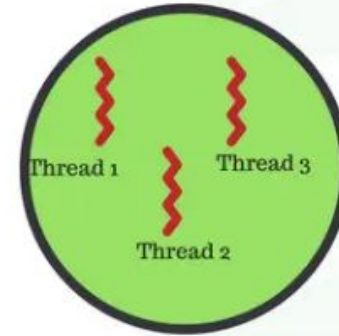


A thread has the following -

- Thread ID
- Program Counter
- Register
- Stack

Threads

Process



A thread has the following -

- Thread ID
- Program Counter
- Register
- Stack

Process means a program that is being executed. Processes are further divided into independent units also known as threads, also known as collections of threads

→ In an operating system (OS), a thread is **a unit of execution within a process**.

(or) A thread is also called **a light weight process**.

→ A process can have multiple threads, each running independently and concurrently.

→ Threads share the same memory space as the process they belong to, but each thread has its own **Thread ID, Program Counter, Stack and Registers**.

→ By dividing a task into multiple threads, each thread can execute a portion of the task simultaneously, thereby reducing the overall execution time.

→ Threads are used to achieve parallelism and improve the performance of a program. i.e

→ Threads allow a process to perform multiple tasks simultaneously or concurrently.

Example, *a web browser may use one thread to display the user interface, another thread to handle user input, and another thread to download content from the internet.*

Classification I :

There are two main types of threads in operating systems:

1. Single-Level Thread
2. Multi-Level Thread

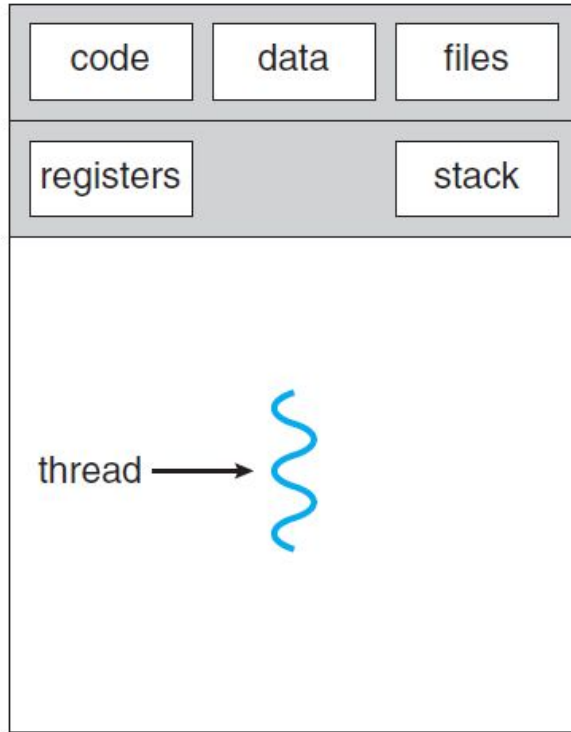
Classification II :

There are two main types of threads in operating systems:

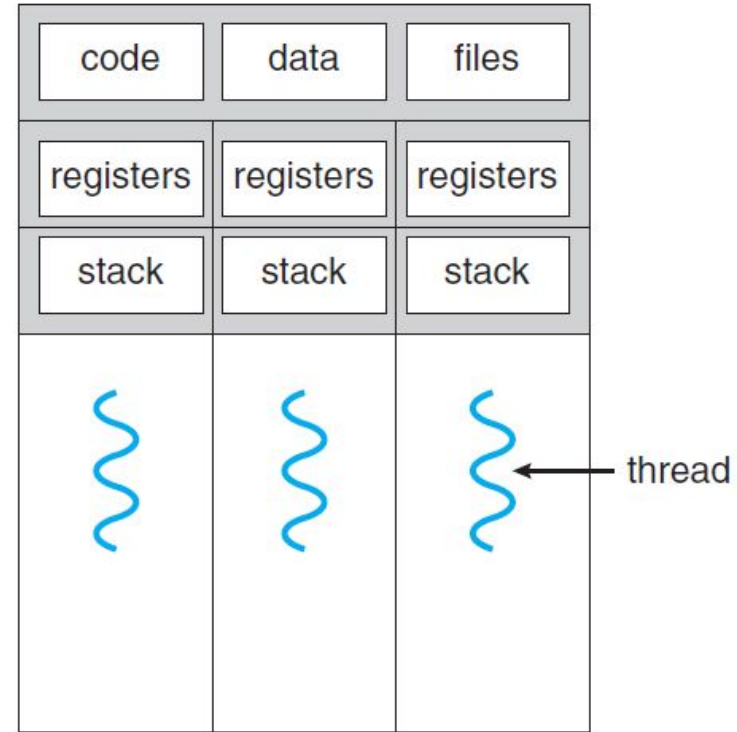
1. User-Level threads
2. Kernel-Level Thread

Classification I

Single-threaded process vs Multithreaded Process



single-threaded process

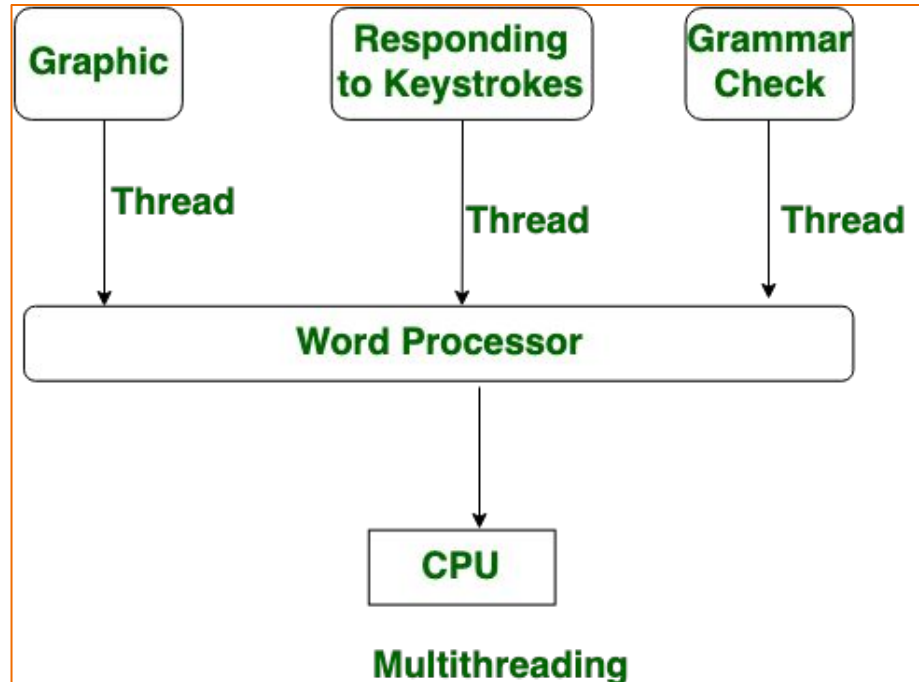


multithreaded process

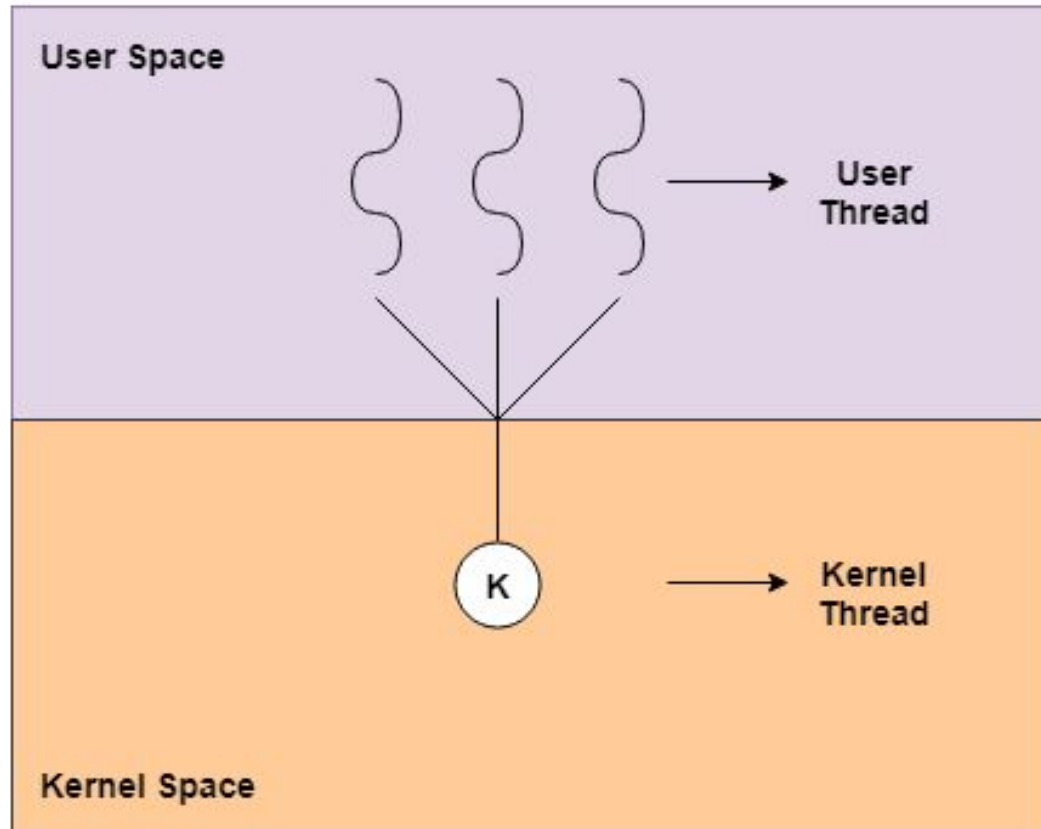
Many software packages that run on modern desktop PCs are multithreaded.

Example:

A word process may have a thread for displaying graphics, another for responding to keystrokes, and a third is for performing spelling and grammar checking in background.



Classification II :



User-Level Threads:

- User-level threads (ULTs) are threads that are managed entirely by **the application program instead of the operating system (OS)**.
- Users implement the user-level threads, and the kernel is unaware of their existence and handles them as though they are single-threaded processes.
- The User-level threads are **small and faster** as compared to **kernel-level threads**, and the OS directly supports user-level threads.
- This means that thread **creation, scheduling, and synchronization** are all handled by the **application program**, rather than the OS.
- ULTs are sometimes called **green threads** because they are implemented in user space, as opposed to kernel space.
- These threads are represented by registers, the program counter (PC), stack, and some small process control.

User-Level Threads:

- Furthermore, there is no kernel interaction in user-level thread synchronization.
- These threads are created, scheduled, and synchronized by **the user-level thread library, contains the code for thread creation, message passing, thread scheduling, and thread destroying**, which runs on top of the operating system. Three primary thread libraries:
 - ◆ **POSIX threads System Call (UNIX)**
 - ◆ **Windows threads System Call (Windows)**

Advantages:

- User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.
- User-level threads can be run on any operating system.
- There are no kernel mode privileges required for thread switching in user-level threads.

Kernel-Level Threads:

- Kernel threads are implemented by **Operating System (OS)**. i.e kernel-level threads are directly created and managed by the kernel.
- This means that the OS is responsible for creating, scheduling, and synchronizing threads, as well as providing access to system resources such as memory and I/O devices.
- KLTs are sometimes called **native threads** because they are implemented at the kernel level and can take full advantage of the OS's threading capabilities.
- KLTs have several advantages over user-level threads (ULTs). One advantage is that KLTs can take advantage of features provided by the OS, such as kernel-level synchronization primitives and I/O operations.
- KLTs are often implemented using system calls to the OS kernel, such as `fork()` or `clone()` in Unix-based systems, or `CreateThread()` in Windows. Each KLT has its own thread context, including a stack and CPU register values, which are saved and restored by the OS during thread scheduling.
- KLTs are widely used in modern operating systems, and are the default threading model in many programming languages, including C/C++, Java, and Python.
- **Examples –**
 - ◆ **Windows NT**
 - ◆ **Windows 2000**
 - ◆ **Solaris2**
 - ◆ **Tru64 UNIX**

Multithreading Models:

- ★ There must exist a relationship between user threads and kernel threads.
- ★ Some operating systems provide **a combination of both, user-level thread and kernel-level thread.**
- ★ One such example of this could be **Solaris.**
- ★ In this combined system, multiple threads run in parallel in the same system. There are 3 types of models in multithreading.

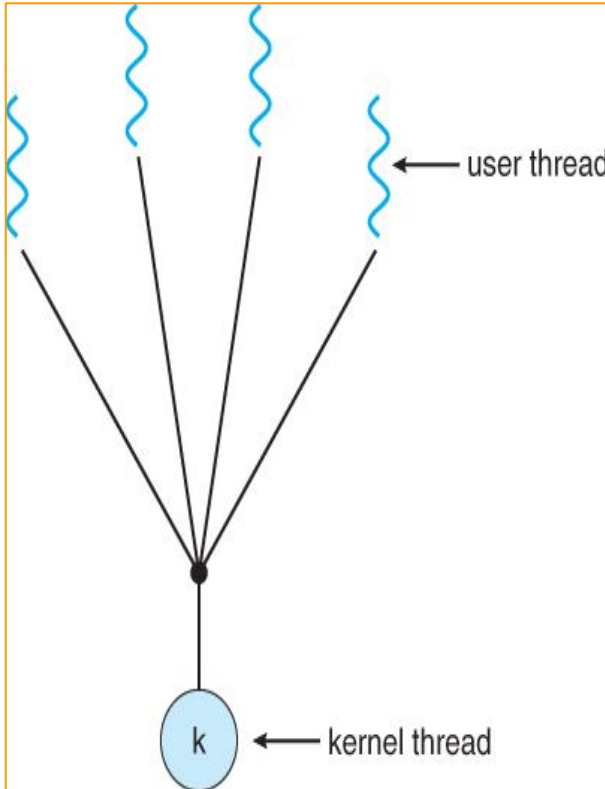
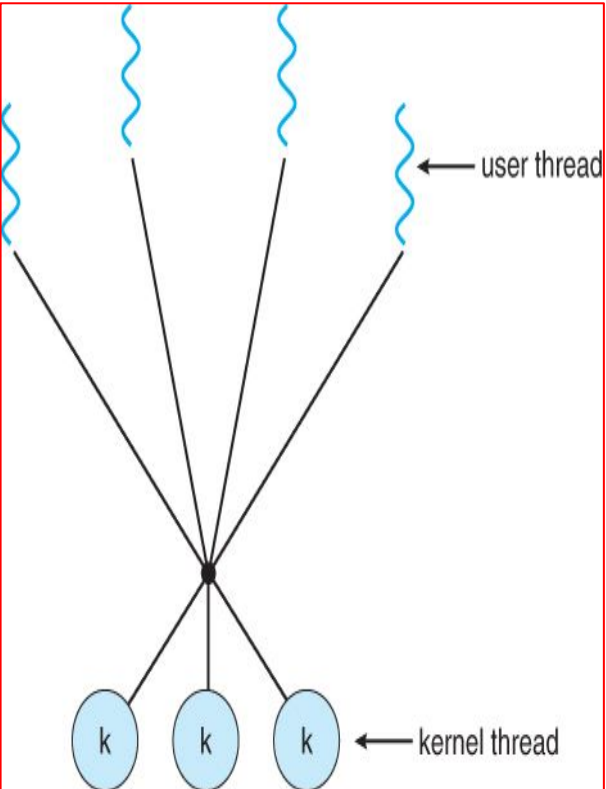
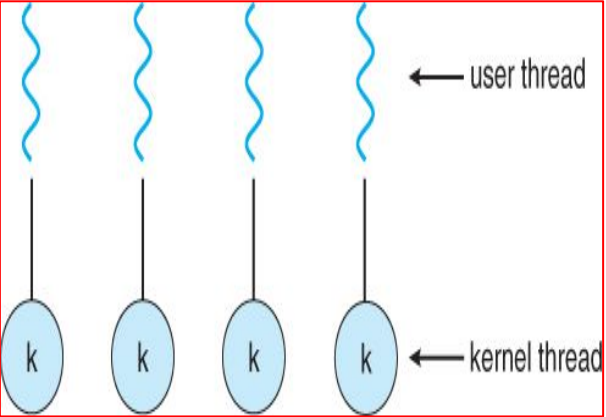
1.Many to Many Model

2.Many to One Model

3.One to One Model

Examples of Multithreading Operating Systems

- ★ Multithreading is widely used by applications. Some of the applications are processing transactions like **online bank transfers, recharge, etc.**
- ★ For instance, in the banking system, many users perform a day-to-day activities using bank servers like **transfers, payments, deposits, opening a new account, etc.**
- ★ All these activities are performed instantly without having to wait for another user to finish.
- ★ In this, all the activities get executed simultaneously as and when they arise. This is where multithreading comes into the picture, wherein several threads perform different activities without interfering with others.

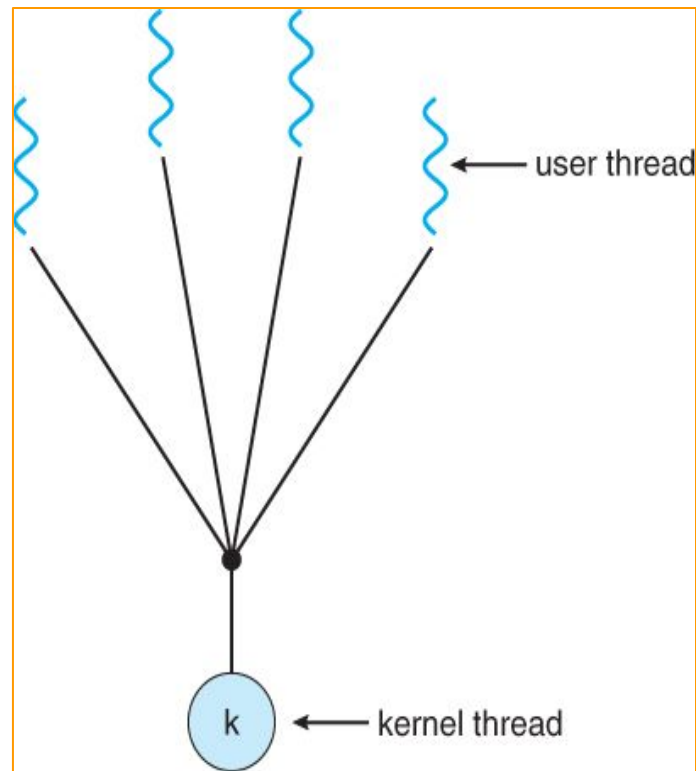


Many to One Model

- ★ In this model, **Many user threads mapped to one kernel thread.**
- ★ In this model when a user thread makes a **blocking system call** entire process blocks.
- ★ As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able access multiprocessor at the same time.
- ★ Few systems currently use this model

Examples:

- Solaris Green Threads
- GNU Portable Threads

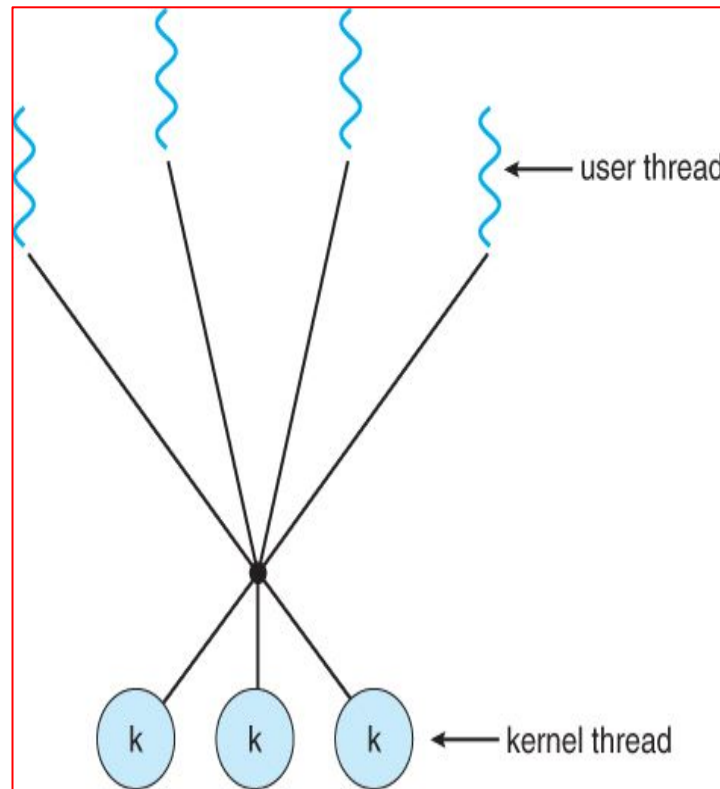


Many to Many Model

- ★ In this model, **Many user threads multiplex to same or lesser number of kernel level threads.**
- ★ Advantage of this model is if a user thread is **blocked** we can schedule others user thread to other kernel thread. Thus, **System doesn't block if a particular thread is blocked.**
- ★ It is the **best multi threading model.**

Examples:

- Windows with the Thread Fiber package
- Solaris 9



One to One Model

- ★ The one-to-one model maps each user thread to a kernel thread.
- ★ As each user thread is connected to different kernel , if any user thread makes a blocking system call, the other user threads won't be blocked.
- ★ Multiple threads to run in parallel on multiprocessors.
- ★ The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

Examples:

- Windows Family[Windows 95, 98, NT, 2000, and XP]
- Linux

