CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

UNIVERSITY INSTITUTE OF ENGINEERING

Bachelor of Engineering (Computer Science & Engineering)

Operating System

Process Concepts and Process Management

Prepared By: Er. Inderjeet Singh(e8822)

DISCOVER . LEARN . EMPOWER

Outline

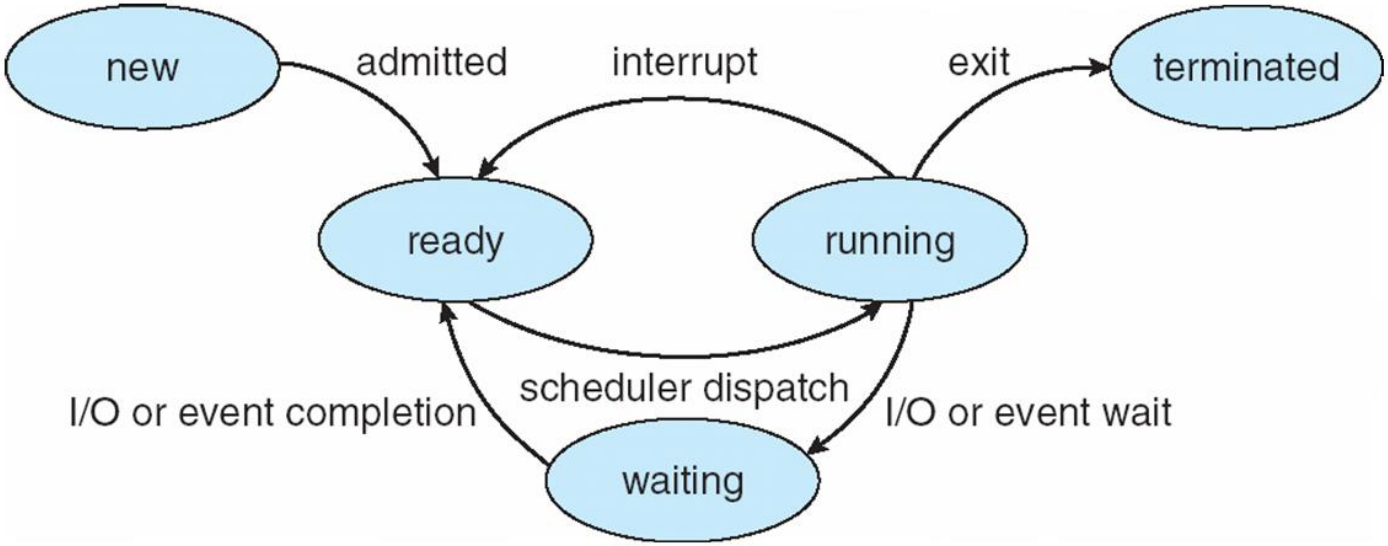
- Process Concept
- Schedulars
  - Medium Term Schedular
  - Long Term Schedular
  - Short Term Schedular
- CPU Scheduling
  - FCFS Scheduling
  - SJF Scheduling
  - SRTF Scheduling
  - Priority Scheduling
  - RR Scheduling

Process v/s Program

- A process is a program in execution. For example, when we write a program in C or C++ 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.
- A process is an ‘active’ entity, instead of a program, which is considered a ‘passive’ entity. A single program can create many processes when run multiple times; for example, when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created).

Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

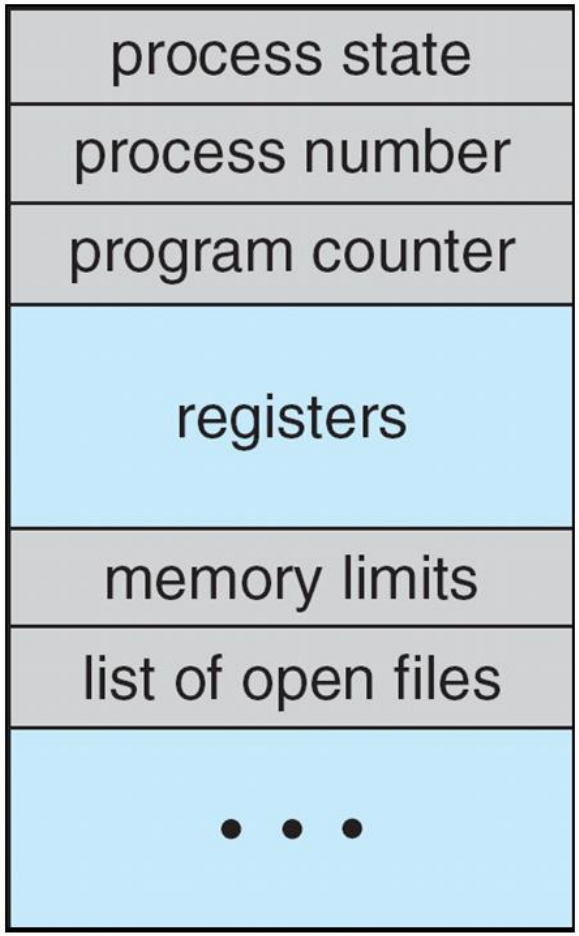


Process control block

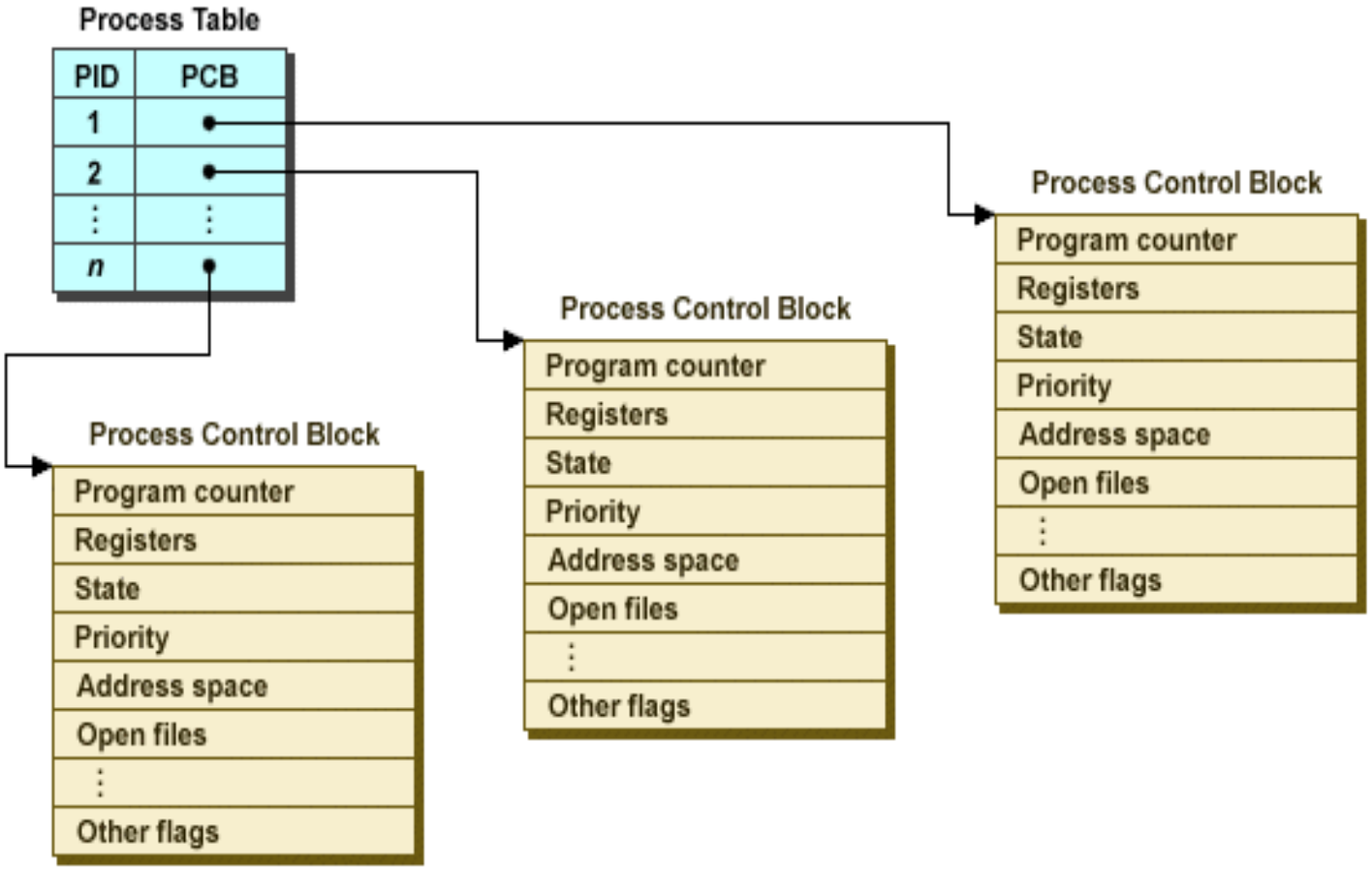
- The *process control block* (PCB) maintains information that the operating system needs in order to manage a process.
- PCBs typically include information such as
  - the process ID,
  - the current state of the process (e.g. *running*, *ready*, *blocked*, etc.),
  - the number of the next program instruction to be executed, and the starting address of the process in memory.
  - The PCB also stores the contents of various processor registers

Process Control Block (PCB)

- **Process state** – The state may be new, ready, running, waiting, halted, and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers** – The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- CPU scheduling information- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files

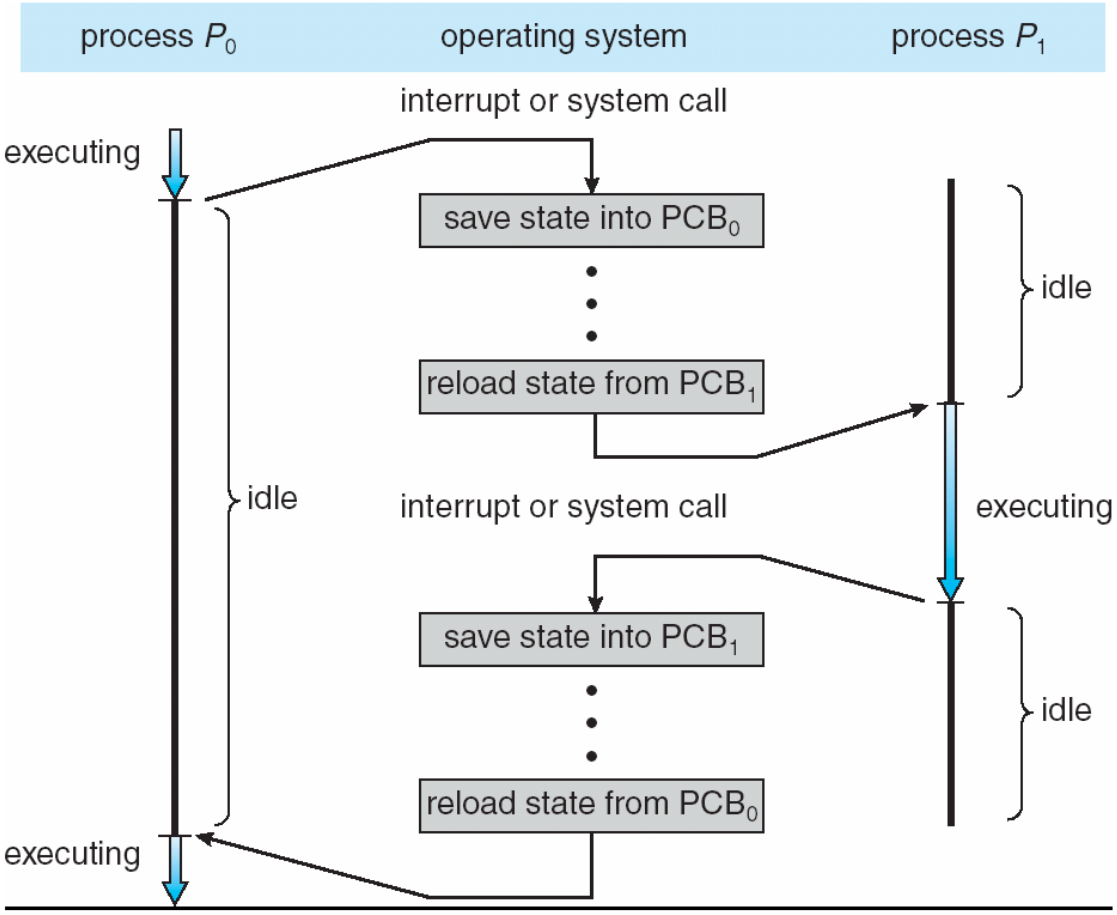


Process control block



The process table and process control blocks

Context Switching



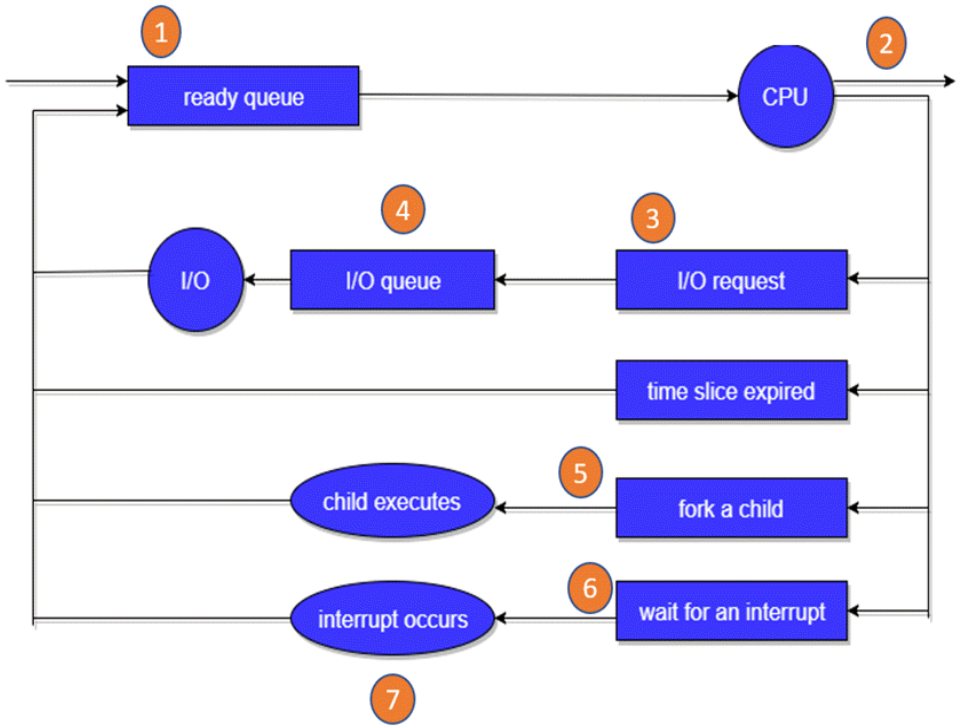
Process Scheduling

Process scheduling is a major element in process management, since the efficiency with which processes are assigned to the processor will affect the overall performance of the system. It is essentially a matter of managing queues, with the aim of minimizing delay while making the most effective use of the processor's time. The operating system carries out four types of process scheduling:

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Representation of Process Scheduling

- Every new process first put in the Ready queue .It waits in the ready queue until it is finally processed for execution. Here, the new process is put in the ready queue and wait until it is selected for execution or it is dispatched.
- One of the processes is allocated the CPU and it is executing
- The process should issue an I/O request
- Then, it should be placed in the I/O queue.
- The process should create a new subprocess
- The process should be waiting for its termination.
- It should remove forcefully from the CPU, as a result interrupt. Once interrupt is completed, it should be sent back to ready queue.

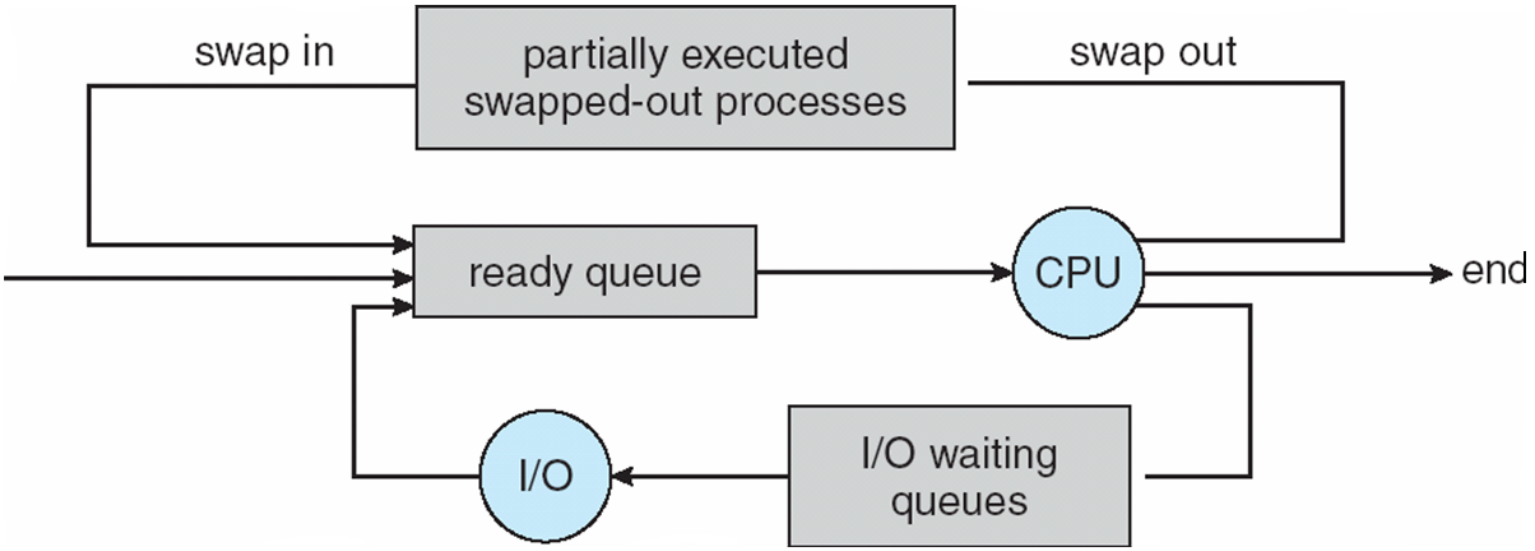


Schedulers

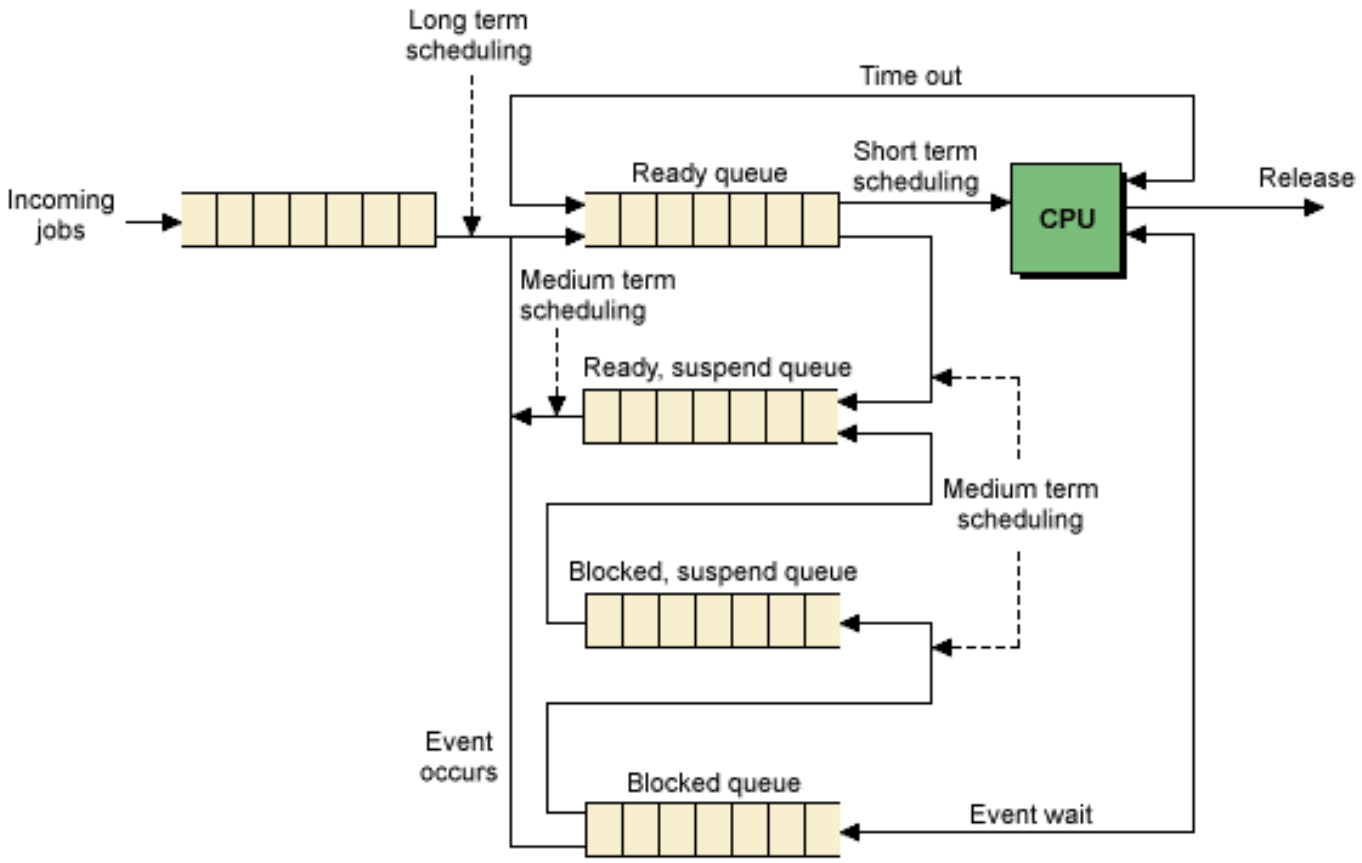
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Process Scheduling



Queuing diagram for scheduling

Process Schedulers

- The ***long-term scheduler*** determines which programs are admitted to the system for processing, and as such controls the degree of multiprogramming. Before accepting a new program, the long-term scheduler must first decide whether the processor is able to cope effectively with another process.

- ***Medium-term scheduling*** is part of the *swapping* function. The term "swapping" refers to transferring a process out of main memory and into *virtual memory* (secondary storage) or vice-versa. This may occur when the operating system needs to make space for a new process, or in order to restore a process to main memory that has previously been swapped out.

- The task of the ***short-term scheduler*** (sometimes referred to as the ***dispatcher***) is to determine which process to execute next. The short-term scheduler ,or CPU scheduler , selects from among the processes that are ready to execute and allocates the CPU to one of them.

CPU Scheduling

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

Types of Scheduling:

Preemptive scheduling: The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.

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

Preemptive Scheduling

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

1. When a process switches from the running state to the waiting state (for 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 (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

CPU Scheduling-Scheduling Criteria

Scheduling Criteria

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

CPU utilization

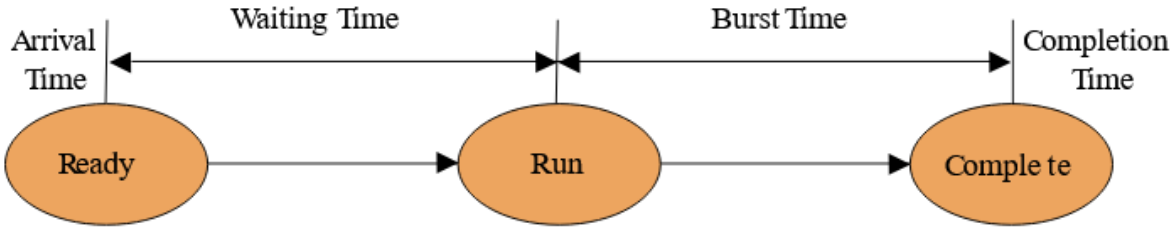
To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

CPU Scheduling

- **Turnaround time:** It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).
- **Waiting time:** The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.
- **Burst Time:** The total amount of time required by the CPU to execute the whole process is called the Burst Time. This does not include the waiting time. It is confusing to calculate the execution time for a process even before executing it hence the scheduling problems based on the burst time cannot be implemented in reality.



$$CT - AT = WT + BT$$

$$TAT = CT - AT$$

$$\text{Waiting Time} = TAT - BT$$

Scheduling Algorithms

We'll discuss four major scheduling algorithms here which are following :

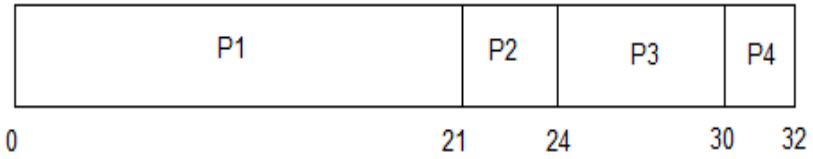
- First Come First Serve(FCFS) Scheduling
- Shortest-Job-First(SJF) Scheduling
- Priority Scheduling
- Round Robin(RR) Scheduling

First Come First Serve(FCFS) Scheduling

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.
- FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

Exercise: FCFS Scheduling

Consider the set of 5 processes whose arrival time and burst time are given below-
If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

Process Id	Arrival time	Burst time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

Hints:
Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time.

Exercise: FCFS Scheduling

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 4 = 0
P2	13	13 – 5 = 8	8 – 3 = 5
P3	2	2 – 0 = 2	2 – 2 = 0
P4	14	14 – 5 = 9	9 – 1 = 8
P5	10	10 – 4 = 6	6 – 3 = 3

Hints:
Turn Around Time = Completion Time – Arrival Time
Waiting Time = Turn Around Time – Burst Time.

Now,
Average Turn Around time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit
Average waiting time = (0 + 5 + 0 + 8 + 3) / 5 = 16 / 5 = 3.2 unit

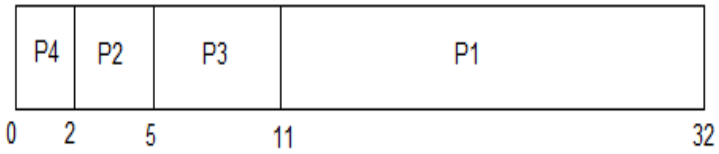
Shortest-Job-First(SJF) Scheduling

- Best approach to minimize waiting time.
- Actual time taken by the process is already known to processor.
- Impossible to implement.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

Non-Preemptive

Note: In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is preempted.

Exercise: Shortest-Job-First(SJF) Scheduling

In the following example, there are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below.
Calculate average waiting time and Turn around time?

PID	Arrival Time	Burst Time
P1	1	7
P2	3	3
P3	6	2
P4	7	10
p5	9	8

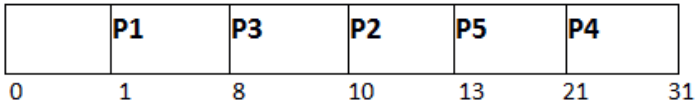
Non-Preemptive

Exercise: Shortest-Job-First(SJF) Scheduling

In the following example, there are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below.

PID	Arrival Time	Burst Time
P1	1	7
P2	3	3
P3	6	2
P4	7	10
p5	9	8

So that's how the procedure will go on in **shortest job first (SJF)** scheduling algorithm.



Avg Waiting Time = 27/5

Shortest Remaining Time First

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

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

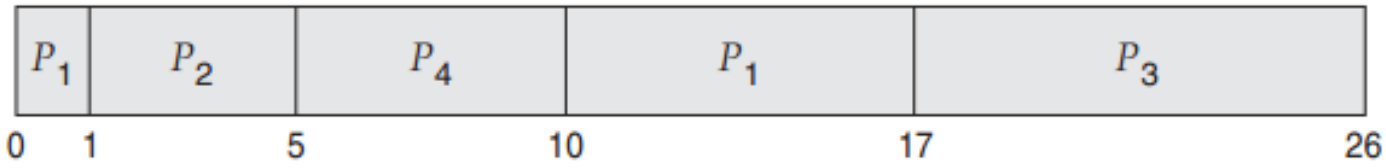
Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Solution: Shortest Remaining Time

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Exercise: Shortest Remaining Time

Example

In this Example, there are five jobs P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table.

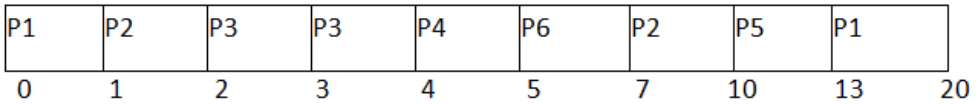
Process ID	Arrival Time	Burst Time
1	0	8
2	1	4
3	2	2
4	3	1
5	4	3
6	5	2

Exercise: Shortest Remaining Time

Example

In this Example, there are five jobs P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time	Response Time
1	0	8	20	20	12	0
2	1	4	10	9	5	1
3	2	2	4	2	0	2
4	3	1	5	2	1	4
5	4	3	13	9	6	10
6	5	2	7	2	0	5



Avg Waiting Time = 24/6

Exercise 2: Shortest Remaining Time

Example

Given the arrival time and burst time of 3 jobs in the table below. Calculate the Average waiting time of the system

.

.

Process ID	Arrival Time	Burst Time
1	0	9
2	1	4
3	2	9

Solution: Shortest Remaining Time

Example

Given the arrival time and burst time of 3 jobs in the table below. Calculate the Average waiting time of the system

.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	0	9	13	13	4
2	1	4	5	4	0
3	2	9	22	20	11

Avg Waiting Time = (4+0+11)/3 = 5 units

Priority Scheduling

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

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	4	4 – 0 = 4	4 – 4 = 0
P2	15	15 – 1 = 14	14 – 3 = 11
P3	12	12 – 2 = 10	10 – 1 = 9
P4	9	9 – 3 = 6	6 – 5 = 1
P5	11	11 – 4 = 7	7 – 2 = 5

Now,

- Average Turn Around time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

Excercise: Priority Scheduling

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, ⋯, P5 , with the length of the CPU burst given in milliseconds:
Using Priority Scheduling schedule the processes and calculate average waiting time.

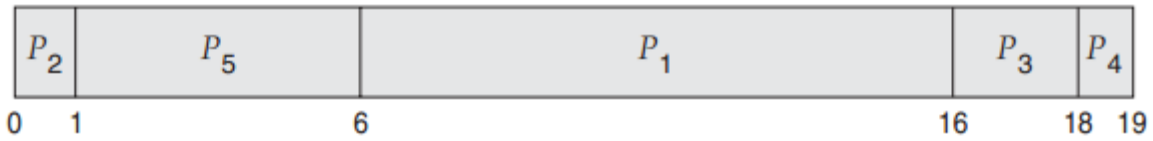
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Solution: Priority Scheduling

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, ⋯, P5 , with the length of the CPU burst given in milliseconds:

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

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



The average waiting time is 8.2 milliseconds.

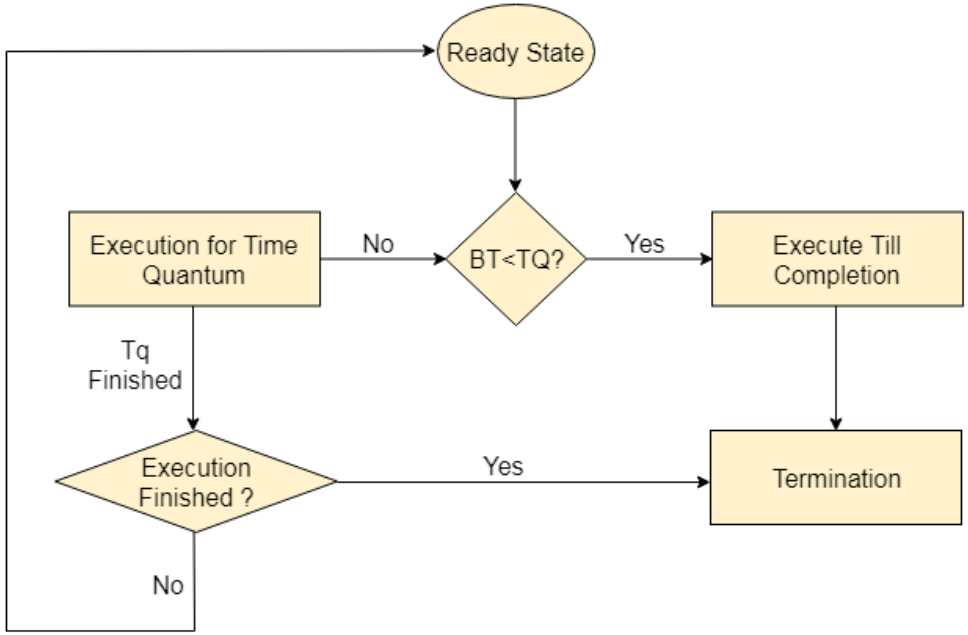
Starvation Problem in Priority Scheduling

A major problem with priority scheduling algorithms is indefinite block-ing ,or **starvation** . A process that is ready to run but waiting for the CPUcan be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely.

- A solution to the problem of indefinite blockage of low-priority processes is **aging** . Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

Round Robin Scheduling

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



Round Robin Scheduling

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Time quantum of 4 milliseconds

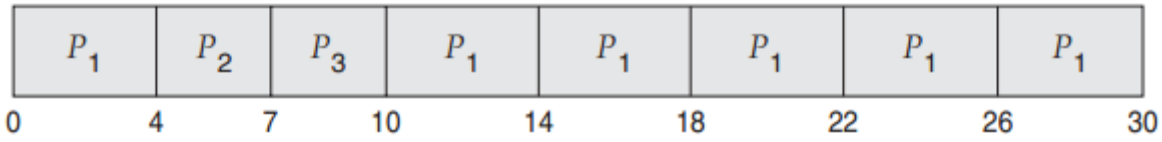
Process	Burst Time
P_1	24
P_2	3
P_3	3

Round Robin Scheduling

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Time quantum of 4 milliseconds

Process	Burst Time
P_1	24
P_2	3
P_3	3



the average waiting time is 17/3 = 5.66 milliseconds.

Exercise:Round Robin Scheduling

Here is the Round Robin scheduling example with gantt chart. Time Quantum is **5ms**.

Process	CPU Burst Time
P_1	30
P_2	6
P_3	8

Round Robin Scheduling

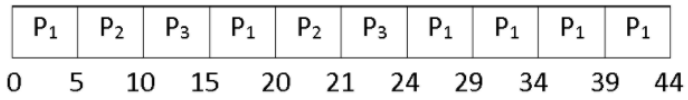
Calculate the average waiting time, average turnaround time and throughput.

Solution: Round Robin Scheduling

Quantum is 5ms.

Process	CPU Burst Time
P_1	30
P_2	6
P_3	8

Gantt Chart



Calculate the average waiting time, average turnaround time and throughput.

Average Waiting Time and Turnaround Time
Average Waiting Time
For finding Average Waiting Time, we have to find out the waiting time of each process.
Waiting Time of
P1 = 0 + (15 - 5) + (24 - 20) = 14ms
P2 = 5 + (20 - 10) = 15ms
P3 = 10 + (21 - 15) = 16ms
Therefore, Average Waiting Time = (14 + 15 + 16) / 3 = 15ms

Average Turnaround Time

Same concept for finding the Turnaround Time.
Turnaround Time of
P1 = 14 + 30 = 44ms
P2 = 15 + 6 = 21ms
P3 = 16 + 8 = 24ms
Therefore, Average Turnaround Time = (44 + 21 + 24) / 3 = 29.66ms

Throughput

Throughput = (30 + 6 + 8) / 3 = 14.66ms
In 14.66ms, one process executes.

References

1. https://www.coursehero.com/file/p5qhfoqs/Types-of-Schedulars-Long-Term-Schedular-or-Job-Schedular-or-admission-scheduler/

2. http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf

3. http://www.hexainclude.com/scheduling-criteria/

Thank You

For Any Query Contact:

Er Inderjeet Singh

Email: Inderjeet.e8822@cumail.in

M: 8699100160

University Institute of Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Bachelor of Engineering (Computer Science & Engineering)

Subject Name : Operating System

Chapter : InterProcess Communication, Threads, MultiThreading

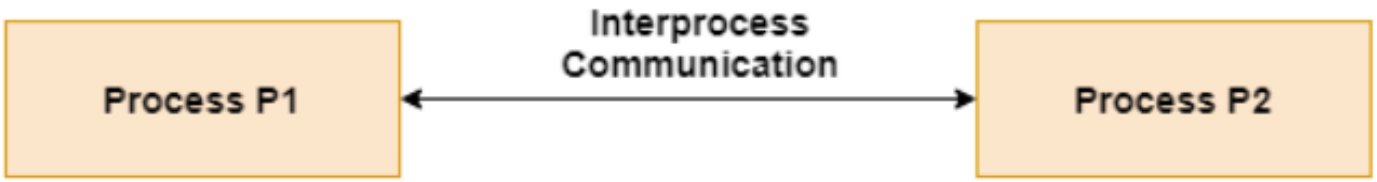
Outline

- Inter process Communication
- Processes and Threads
- Types of threads
- Concept of multithreading
- Exercise

Interprocess Communication

- Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

- "Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)."

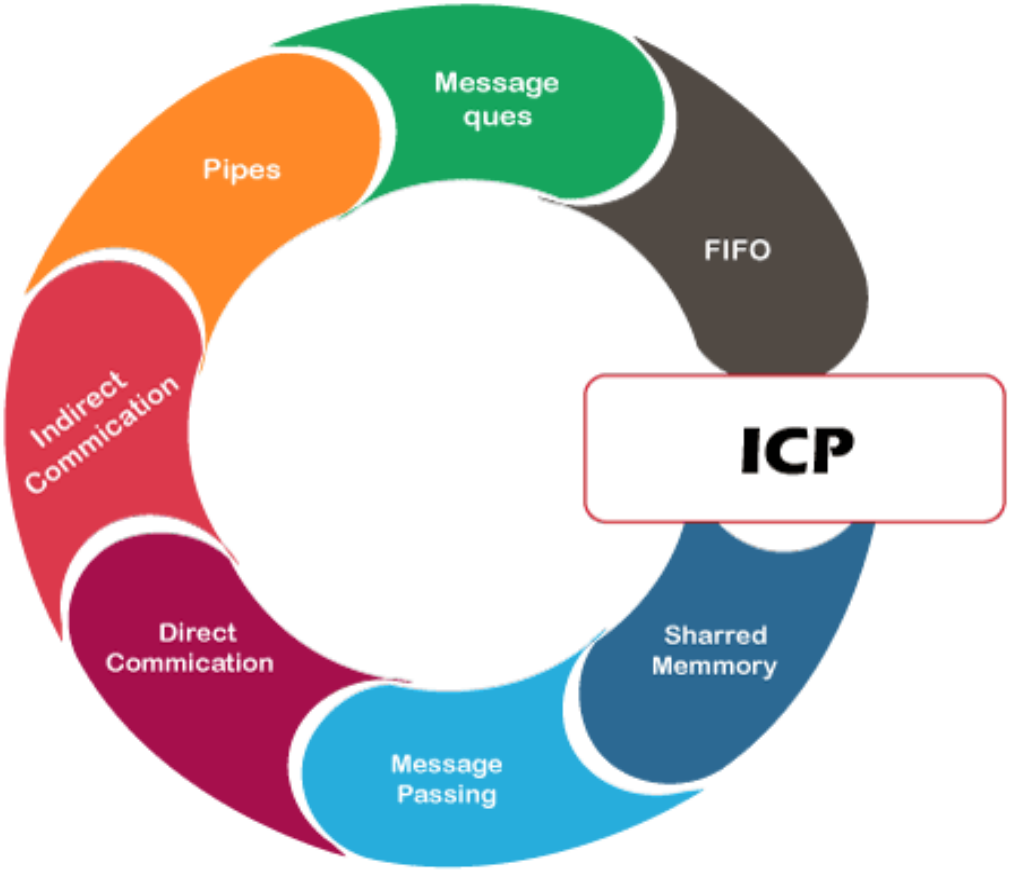


Synchronization in Inter Process Communication

- It is one of the essential parts of inter process communication. Typically, this is provided by interprocess communication control mechanisms, but sometimes it can also be controlled by communication processes.
- These are the following methods that used to provide the synchronization:

1. **Mutual Exclusion:-** It is generally required that only one process thread can enter the critical section at a time. This also helps in synchronization and creates a stable state to avoid the race condition.
2. **Semaphore:-** Semaphore is a type of variable that usually controls the access to the shared resources by several processes. Semaphore is further divided into two types : Binary Semaphore and Counting Semaphore
3. **Spinlock:-** Spinlock is a type of lock as its name implies. The processes are trying to acquire the spinlock waits or stays in a loop while checking that the lock is available or not. It is known as busy waiting because even though the process active, the process does not perform any functional operation (or task).

Approaches to Interprocess Communication



Approaches to Interprocess Communication

1. **Pipe:-**The pipe is a type of data channel that is unidirectional in nature. It means that the data in this type of data channel can be moved in only a single direction at a time. Still, one can use two-channel of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output. These pipes are used in all types of POSIX systems and in different versions of window operating systems as well.
2. **Shared Memory:-** It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously. It is primarily used so that the processes can communicate with each other. Therefore the shared memory is used by almost all POSIX and Windows operating systems as well.
3. **Message Passing:-** It is a type of mechanism that allows processes to synchronize and communicate with each other. However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

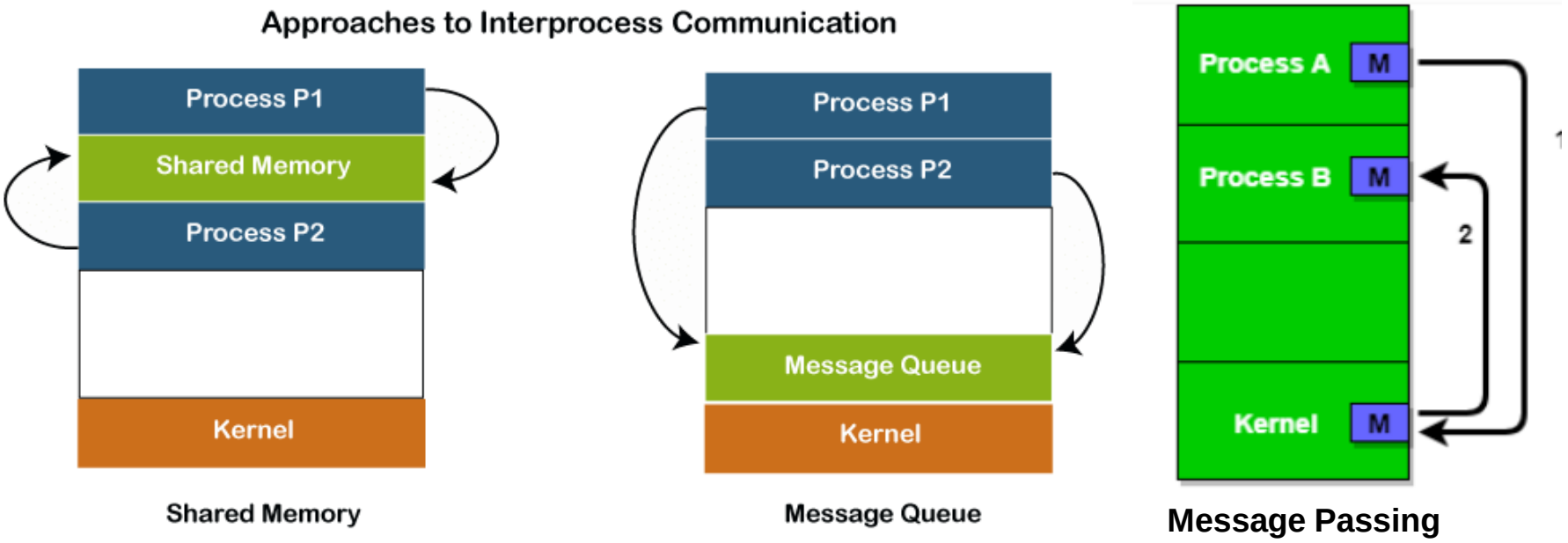
Usually, the inter-process communication mechanism provides two operations that are as follows:

- send (message)
- received (message)

Approaches to Interprocess Communication

4. Message Queue:-

In general, several different messages are allowed to read and write the data to the message queue. In the message queue, the messages are stored or stay in the queue unless their recipients retrieve them. In short, we can also say that the message queue is very helpful in inter-process communication and used by all operating systems.



Approaches to Interprocess Communication

5. Direct Communication:-

In this type of communication process, usually, a link is created or established between two communicating processes. However, in every pair of communicating processes, only one link can exist.

6. Indirect Communication

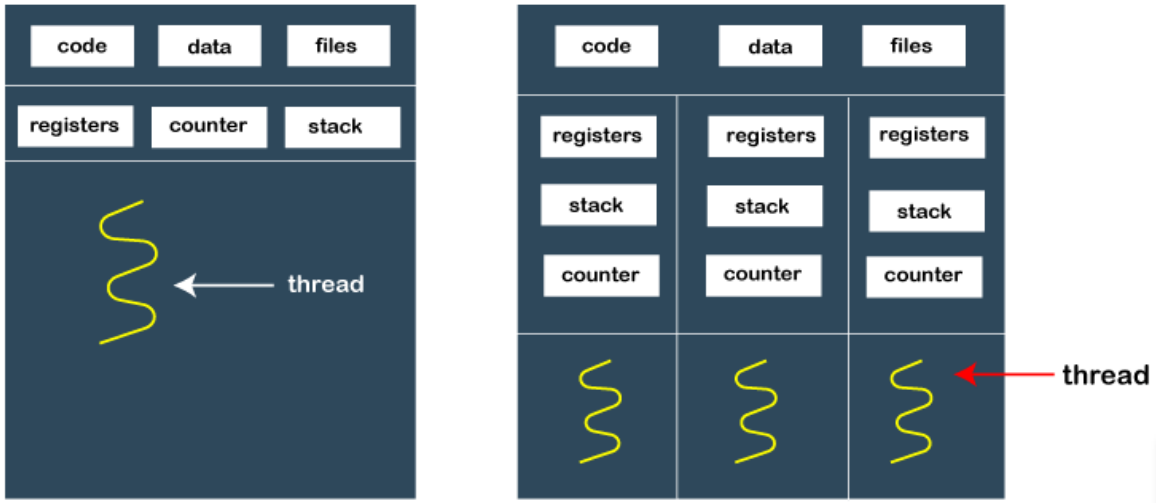
Indirect communication can only exist or be established when processes share a common mailbox, and each pair of these processes shares multiple communication links. These shared links can be unidirectional or bi-directional.

Processes and Threads

- A thread is a path of execution within a process. A process can contain multiple threads.
- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***
- The unit of resource ownership is referred to as a ***process*** or ***task***
- A **thread** is the subset of a process and is also known as the lightweight process. A process can have more than one thread, and these threads are managed independently by the scheduler. All the threads within one process are interrelated to each other. Threads have some common information, such as data segment, code segment, files, etc., that is shared to their peer threads. But contains its own registers, stack, and counter.

Processes and Threads

- ***Multithreading*** - The ability of an OS to support multiple, concurrent paths of execution within a single process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads.



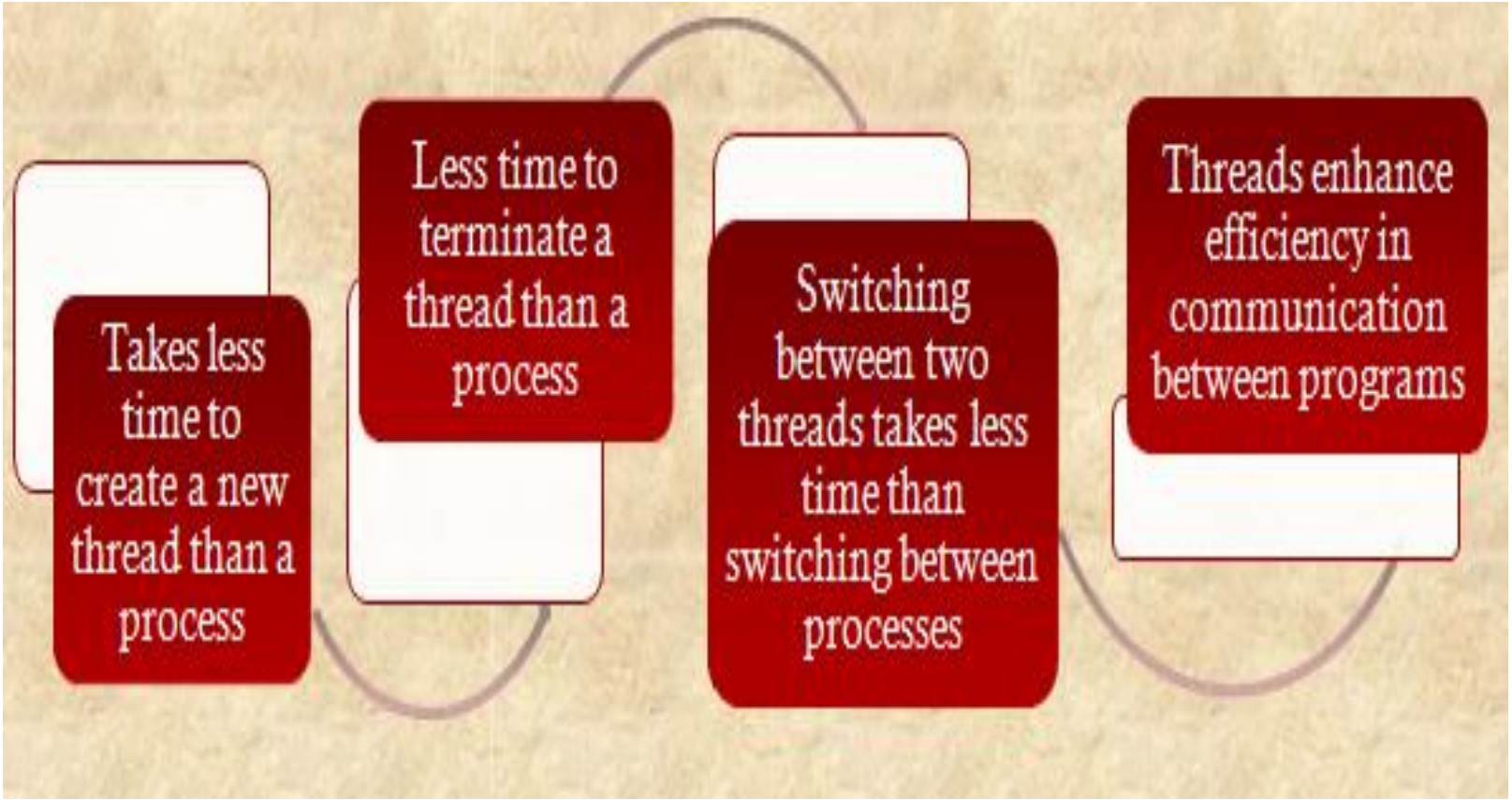
Single-threaded process

Multi-threaded process

MS-DOS is an example

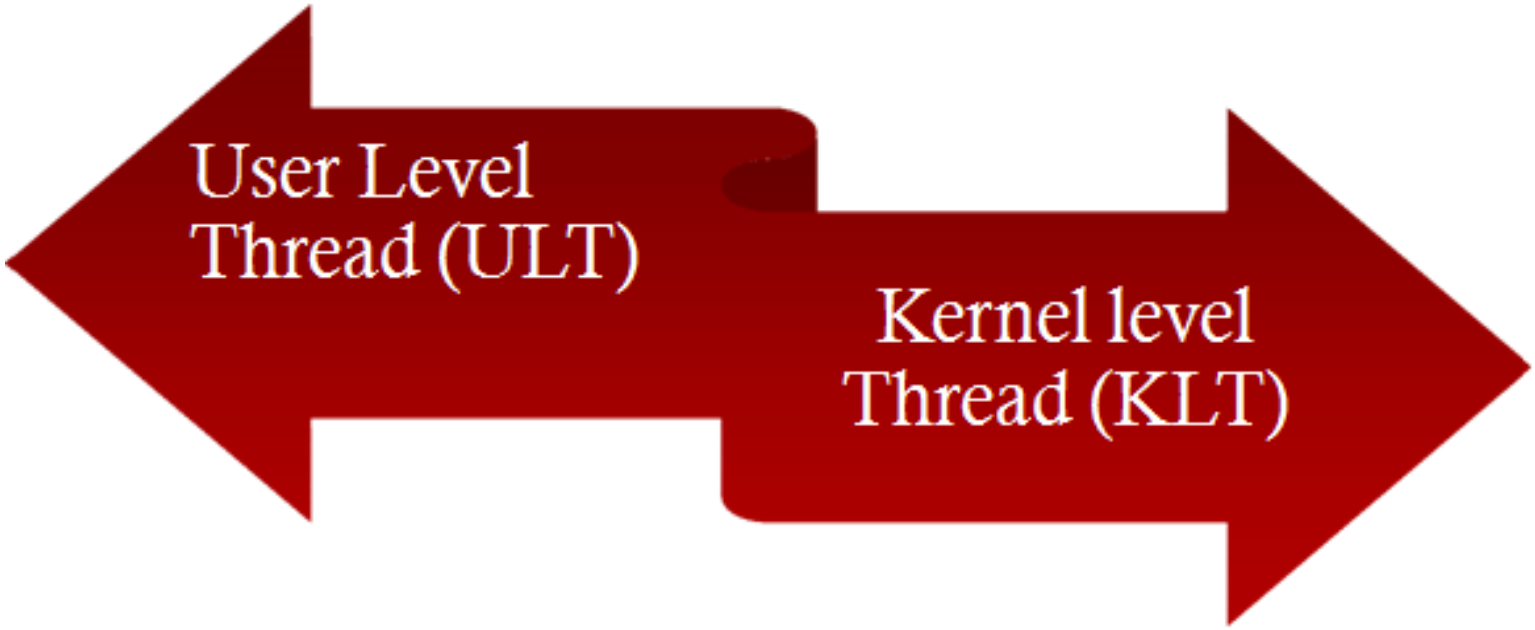
A Java run-time environment is an example of a system of one process with multiple threads

Benefits of Using Threads



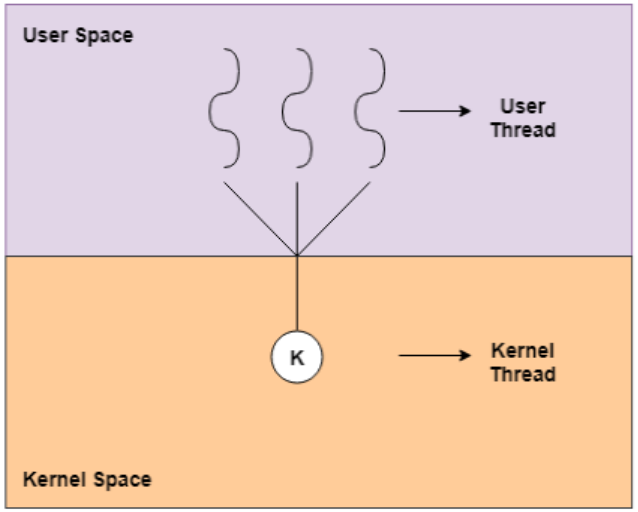
Parameter	Process	Thread
Definition	Process means a program is in execution.	Thread means a segment of a process.
Lightweight	The process is not Lightweight.	Threads are Lightweight.
Termination time	The process takes more time to terminate.	The thread takes less time to terminate.
Creation time	It takes more time for creation.	It takes less time for creation.
Communication	Communication between processes needs more time compared to thread.	Communication between threads requires less time compared to processes.
Context switching time	It takes more time for context switching.	It takes less time for context switching.
Resource	Process consume more resources.	Thread consume fewer resources.
Treatment by OS	Different process are tread separately by OS.	All the level peer threads are treated as a single task by OS.
Memory	The process is mostly isolated.	Threads share memory.
Sharing	It does not share data	Threads share data with each other.

Types of Threads



User Level Threads

- The user-level threads are implemented by users and the kernel is not aware of the existence of these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads. They are represented by a program counter(PC), stack, registers and a small process control block. Also, there is no kernel involvement in synchronization for user-level threads.



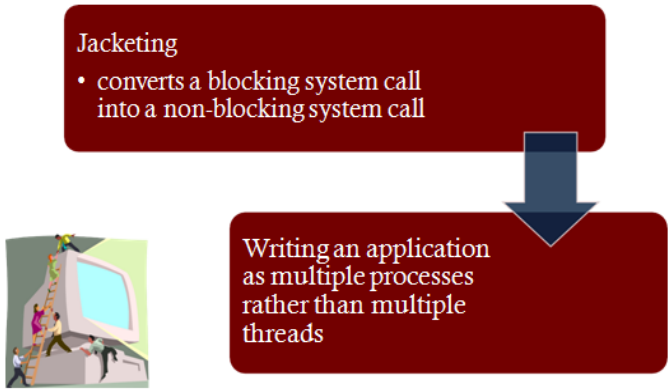
User Level Threads

Advantages of User-Level Threads

- 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.

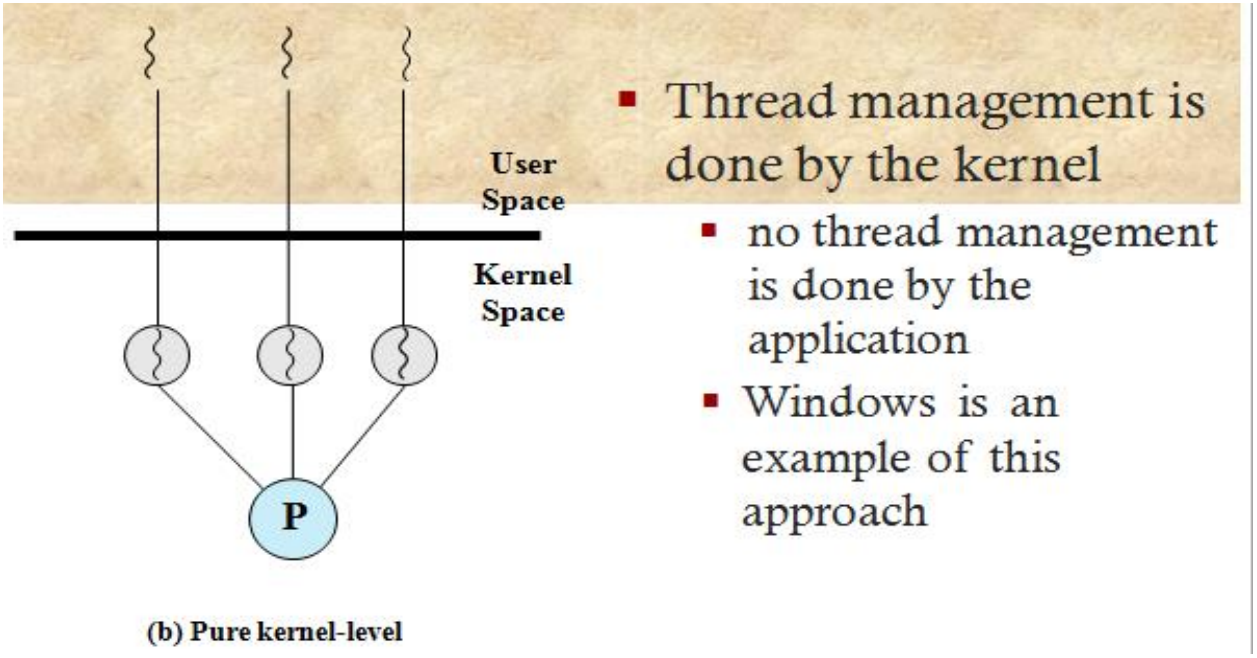
Disadvantages of User-Level Threads

- Multithreaded applications in user-level threads cannot use multiprocessing to their advantage.
- The entire process is blocked if one user-level thread performs blocking operation.
- Overcoming ULT Disadvantages



Kernel-Level Threads (KLTs)

Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel. Because of this, kernel-level threads are slower than user-level threads.



(b) Pure kernel-level

Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded

Disadvantage of KLTs

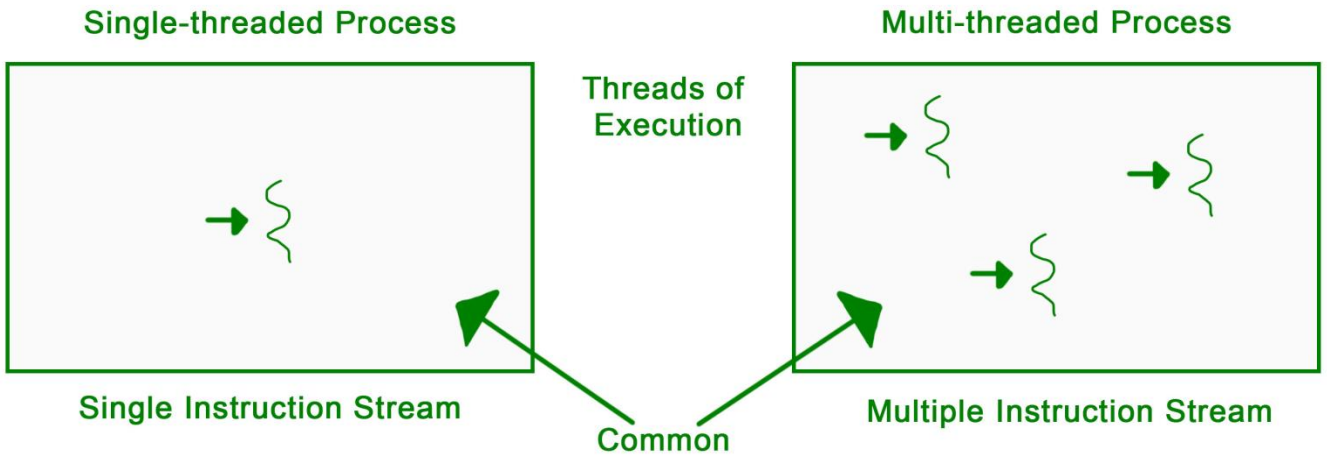
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel.
- The kernel thread manages and schedules all threads.
- The kernel-level thread is slower than user-level threads.

User-Level Thread v/s Kernel-Level Thread

User level thread	Kernel level thread
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
User level threads are designed as dependent threads. Example : Java thread, POSIX threads	Kernel level threads are designed as independent threads. Example : Window Solaris.

MultiThreading

- The concept of **multi-threading** needs proper understanding of these two terms – **a process and a thread**. A process is a program being executed. A process can be further divided into independent units known as threads.
- A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.



Single Thread and Multi Thread Process

Need for Multi-Threaded Server Architecture

- In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

- One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request.

- **If the new process will perform the same tasks as the existing process, why incur all that overhead?**

- It is generally more efficient to use one process that contains **multiple threads**. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.

Need for Multi-Threaded Server Architecture

- Most operating-system kernels are now multithreaded. Several threads operate in the kernel, and **each thread performs a specific task**, such as managing devices, managing memory, or interrupt handling. **For example,** Solaris has a set of threads in the kernel specifically for interrupt handling; Linux uses a kernel thread for managing the amount of free memory in the System.

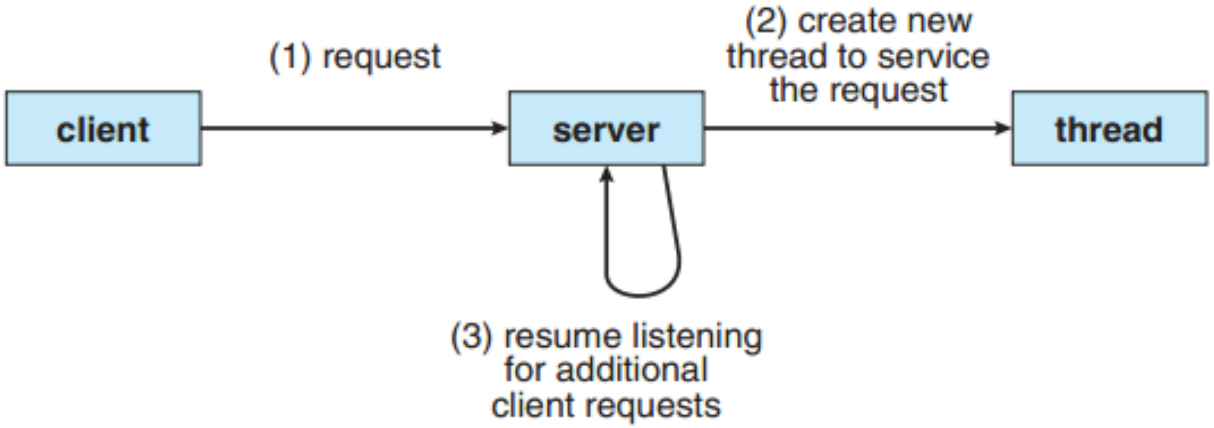


Figure 4.2 Multithreaded server architecture.

Benefits of Multi-Threading

- **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces.
- **Resource sharing:** Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default.
- **Economy:** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Scalability:** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Multithreading Models

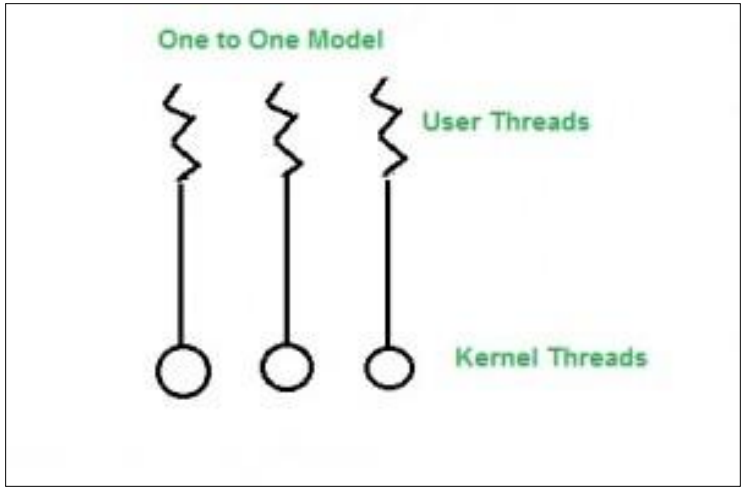
Multithreading allows the execution of multiple parts of a program at the same time. These parts are known as threads and are lightweight processes available within the process. Therefore, multithreading leads to maximum utilization of the CPU by multitasking.

Many operating systems support kernel thread and user thread in a combined way. Example of such system is Solaris. Multi threading model are of three types.

- **Many to many model.**
- **Many to one model.**
- **one to one model**

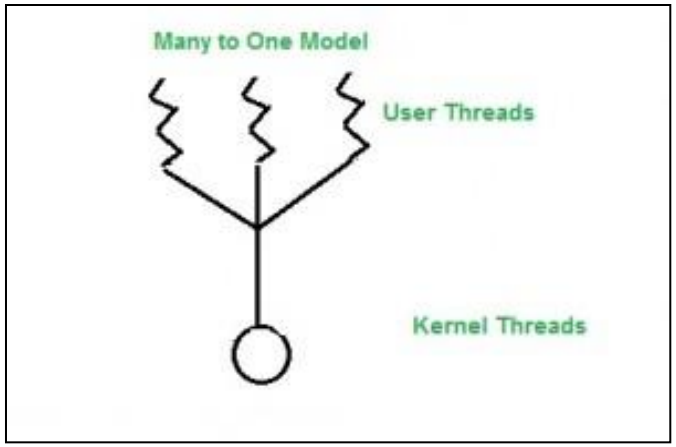
One to One: Multithreading Model

- In this model, one to one relationship between kernel and user thread. In this model multiple thread can run on multiple processor. Problem with this model is that creating a user thread requires the corresponding 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.
- The overhead of creating kernel threads can burden the performance of an application.
- Linux, along with the family of Windows operating systems, implement the one-to-one model.



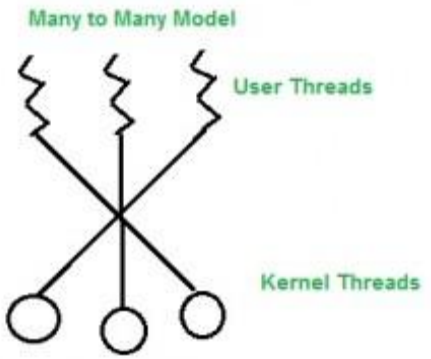
Many to One: Multithreading Model

- In this model, we have multiple 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.
- The thread management is done on the user level so it is more efficient.
- Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. Green threads— a thread library available for Solaris systems and adopted in early versions of Java — used the many-to-one model.



Many to Many: Multithreading Model

- In this model, we have multiple user threads multiplex to same or lesser number of kernel level threads. Number of kernel level threads are specific to the machine, 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.
- The many-to-one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time. The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application. The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.



Linux Thread Management

Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.

1. **Pthreads,** the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.
2. The **Windows thread library** is a kernel-level library available on Windows systems.
3. **The Java thread API** allows threads to be created and managed directly in Java programs.

- **POSIX Threads**, usually referred to as **pthreads**, is an execution model allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a *thread*, and creation and control over these flows is achieved by making calls to the POSIX Threads API. Pthreads defines a set of C programming language types, functions and constants. It is implemented with a pthread.h header and a thread library.

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

Linux Process Creation

- A running process may issue system calls to create new processes . For example, in UNIX: *fork* system call.
- The way we create a new process in Linux is using the

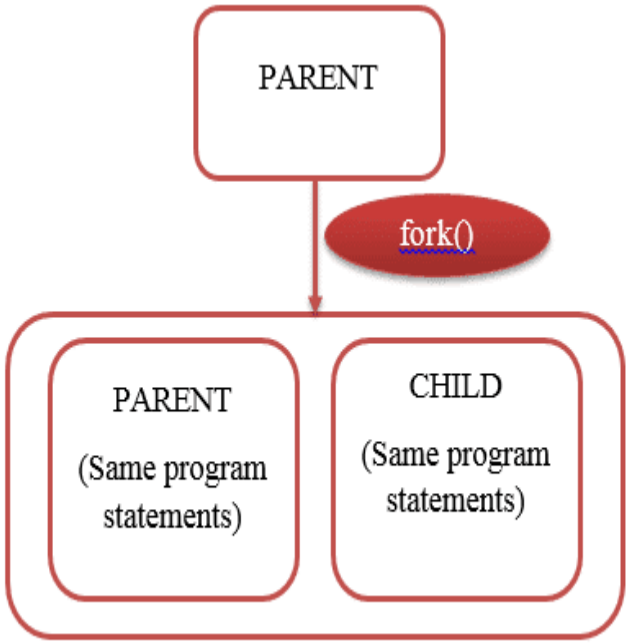
  Fork () command

- Fork creates a child process that is a duplicate of the parent process. Every process in Linux has an PID (process id) which will help us differentiate between one another.

- It takes no parameters and returns an integer value.
- Below are different values returned by fork().
- ***Negative Value***: creation of a child process was unsuccessful.
- ***Zero***: Returned to the newly created child process.
- ***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process.

Fork()

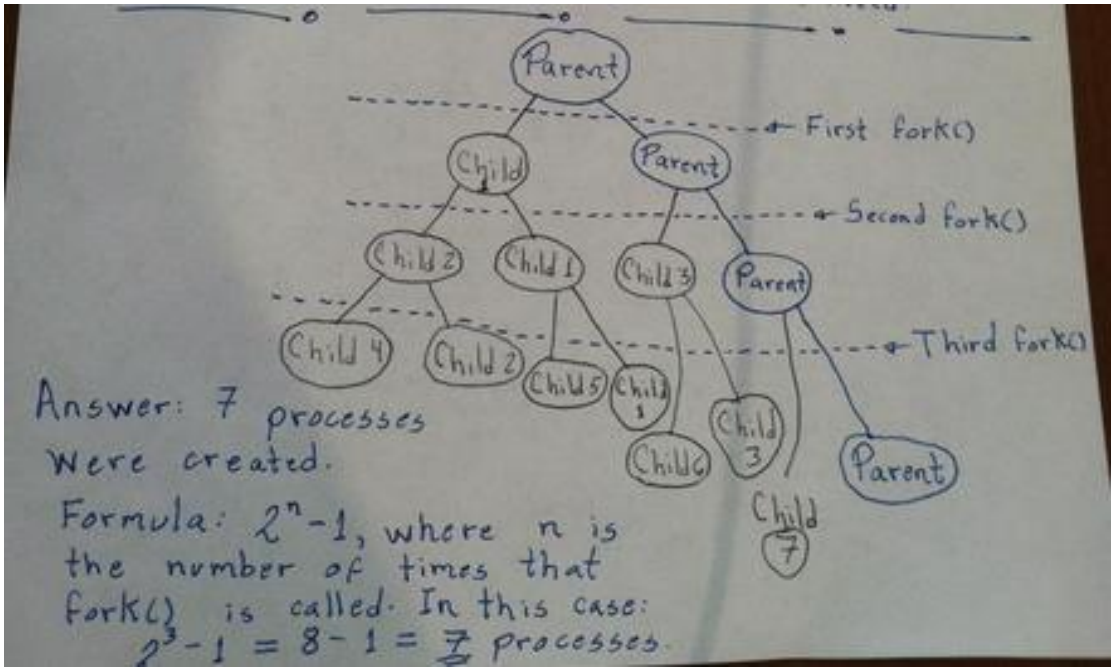
A diagram that demonstrates the fork() command is given as follows:



- **fork() vs exec()**
- The fork system call creates a new process. The new process created by fork() is a copy of the current process except for the returned value.
- The exec() system call is used to replace the current process image with the new process image. It loads the program into the current space, and runs it from the entry point.

Example: Fork()

```
#include <stdio.h>
   #include <unistd.h>
   int main()
{
for(int i=0;i<3;i++)
{
        fork();
}
}
```



Example: Fork()

How many Hello's are printed.
(Assume fork does not fail.)

```
int main()
{
 int i;
for(i = 0; i < 2; i++)
{
 fork();
printf("Hello\n");
 }
 return 0;
 }
```

Options

a. 2

b. 4

c. 6

d. 8

Example: Fork()

How many Hello's are printed.
(Assume fork does not fail.)

```
int main()
{
 int i;
for(i = 0; i < 2; i++)
{
 fork();
printf("Hello\n");
 }
 return 0;
 }
```

Options

a. 2

b. 4

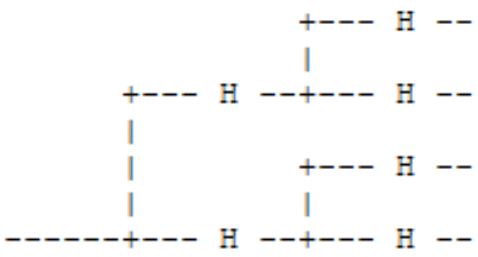
c. 6

d. 8

Solution Remarks: This code is equivalent to:

```
fork();
printf("Hello\n");
fork();
printf("Hello\n");
```

Now draw the diagram (where H is for the print statement):



References

- *Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2012. Operating System Concepts (9th. ed.). Wiley Publishing.*
- https://www.includehelp.com/c-programming-questions/compiling-program-with-pthread-library-linux.aspx
- https://www.studytonight.com/operating-system/multithreading
- https://computing.llnl.gov/tutorials/pthreads/
- https://www.gatevidyalay.com/process-synchronization-practice-problems/
- https://www.studytonight.com/operating-system/process-synchronization
- https://www.geeksforgeeks.org/introduction-of-process-synchronization/

Thank You

For Any Query Contact:

Er Inderjeet Singh

Email: Inderjeet.e8822@cumail.in

M: 8699100160