

Operating Systems – 1BCS304

Module -II

Course Coordinator:
Mrs. Pallavi A R
Assistant professor
Dept. of CSE-DS
ATMECE, Mysuru

Process Management

- Process Concept

- A process is a program under execution.
 - Its current activity is indicated by PC (Program Counter) and the contents of the processor's registers.
-
- Process memory is divided into four sections as shown in the figure:
 - The **stack** is used to store temporary data such as local variables, function parameters, function return values, return address etc.
 - The **heap** which is memory that is dynamically allocated during process run time
 - The **data** section stores global variables.
 - The **text** section comprises the compiled program code.
 - Note that, there is a free space between the stack and the heap. When the stack is full, it grows downwards and when the heap is full, it grows upwards.

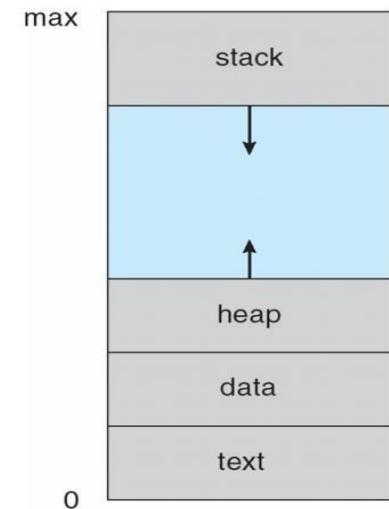


Fig. Process in memory

Process Management

Process State

- A Process has 5 states. Each process may be in one of the following states –
 1. **New** - The process is in the stage of being created.
 2. **Ready** - The process has all the resources it needs to run. It is waiting to be assigned to the processor.
 3. **Running** – Instructions are being executed.
 4. **Waiting** - The process is waiting for some event to occur. For example, the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
 5. **Terminated** - The process has completed its execution.

Process Management

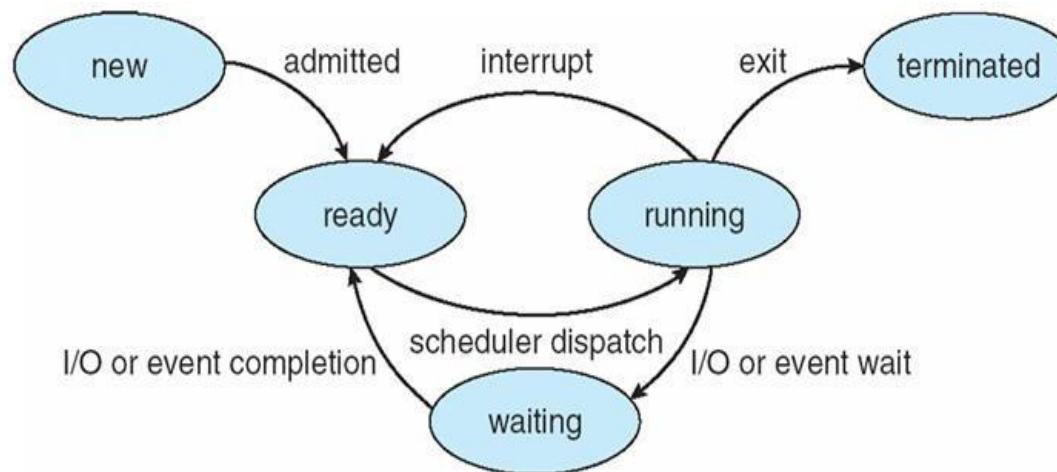


Fig. Diagram of process state

Process Management

Process Control Block

- For each process there is a Process Control Block (PCB), which stores the process-specific information as shown below –
 - **Process State** – The state of the process may be new, ready, running, waiting, 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. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

Process Management

Process Control Block

- **CPU scheduling information-** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** – This includes information such as the value of the base and limit registers, the page tables, or the segment tables.
- **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 Management

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



Fig. Process control block (PCB)

CPU Switch from Process to Process

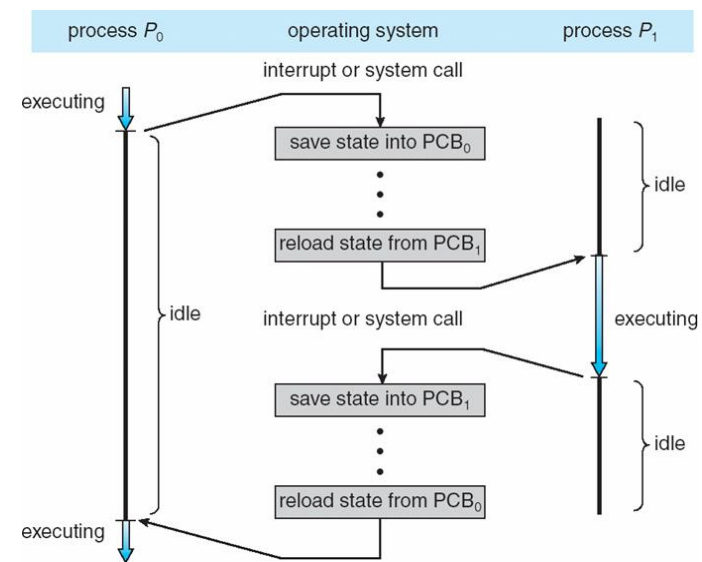


Fig. Diagram showing CPU switch from process to process

Process Management

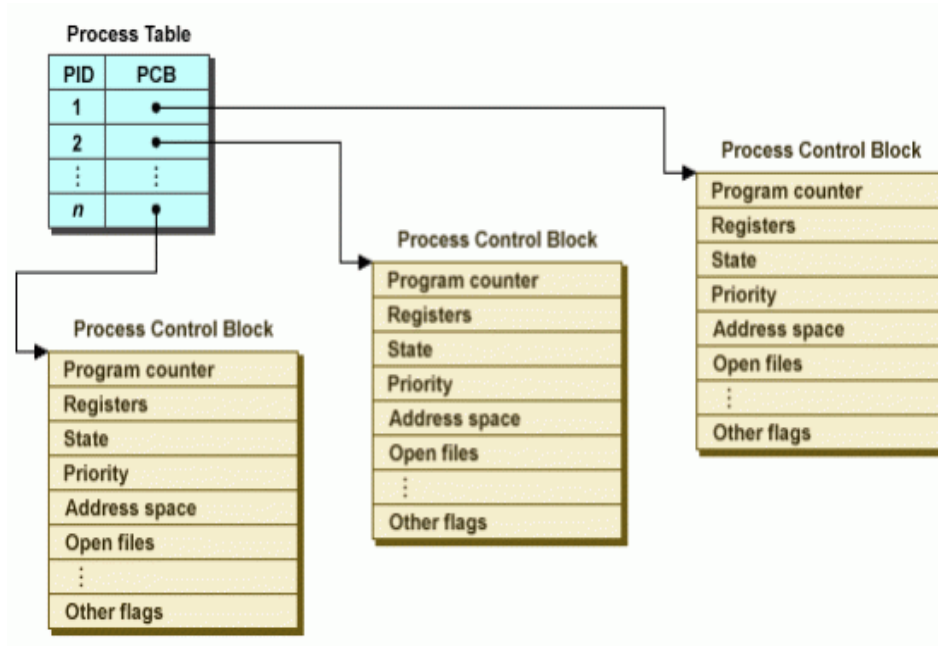


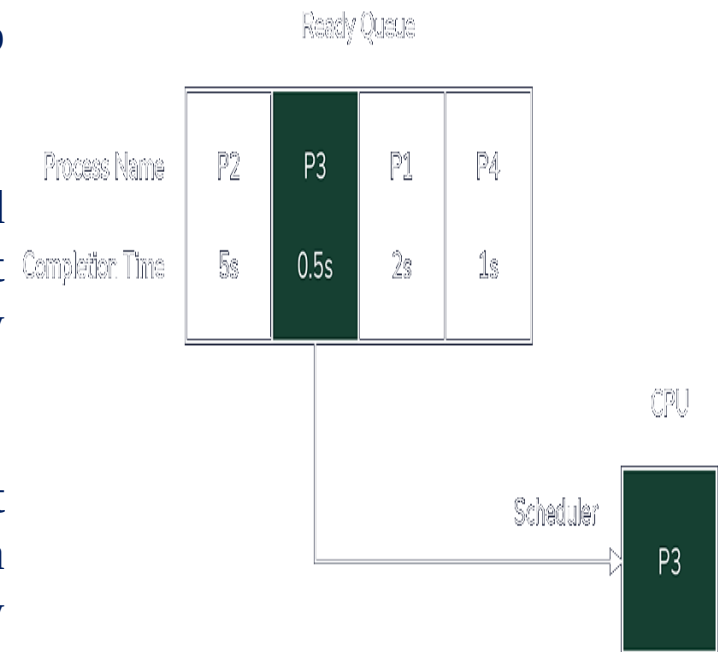
Fig. Process Control Block of multiple process

Process Scheduling

Process Scheduling

Scheduling Queues

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This 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.



Process Scheduling

Ready Queue and Various I/O Device Queues

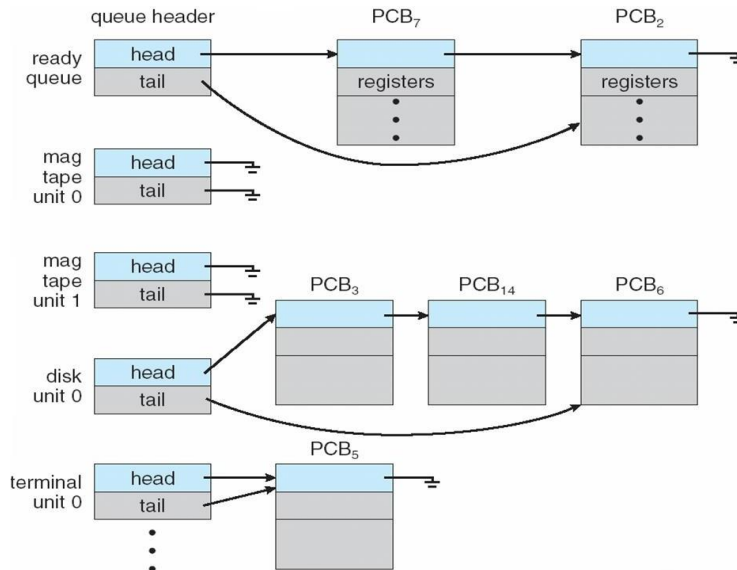
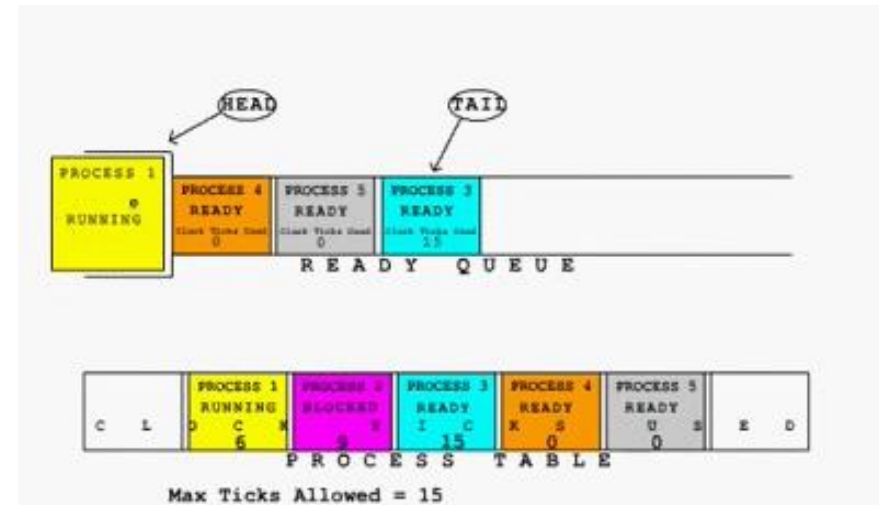


Fig. The ready queue and various I/O device queues



Process Scheduling

- A common representation of process scheduling is a queuing diagram. Each rectangular box in the diagram 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, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution and is given the CPU. 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 subprocess and wait for its termination.
 3. The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Process Scheduling

- 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.

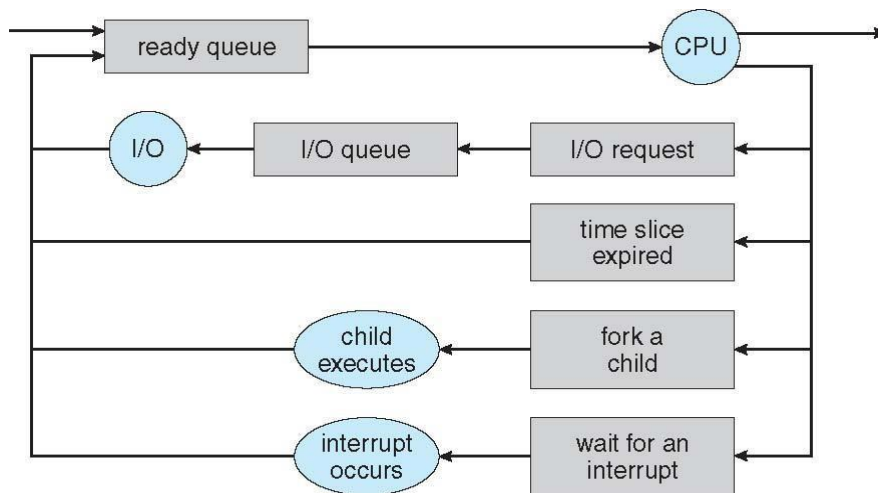
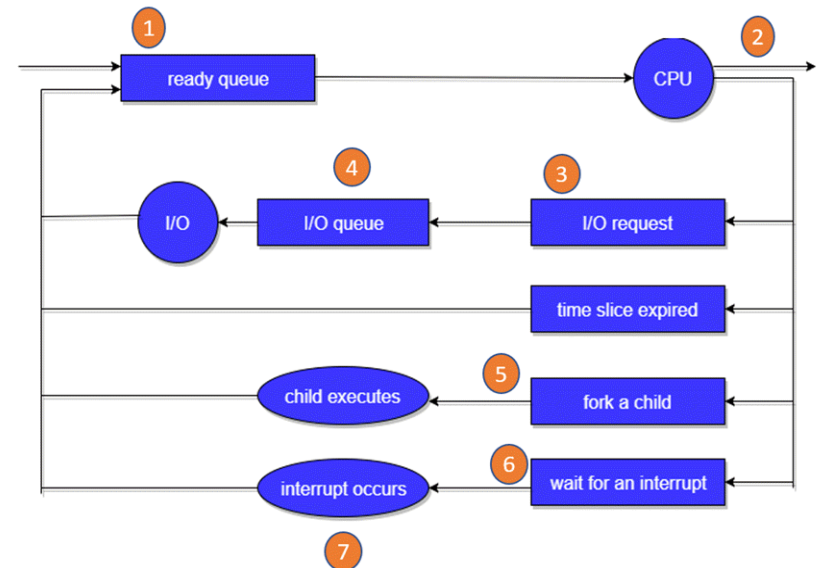


Fig. Queueing-diagram representation of process scheduling

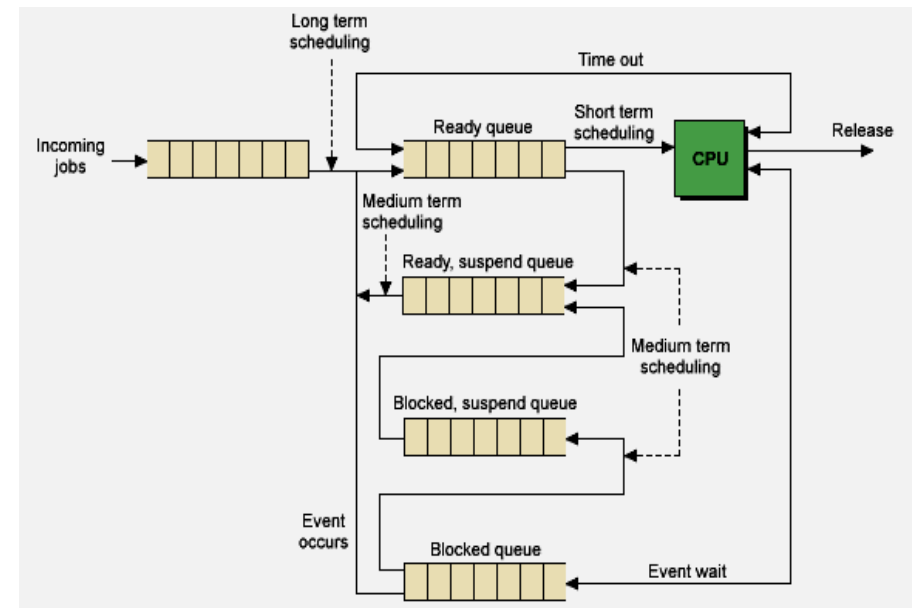
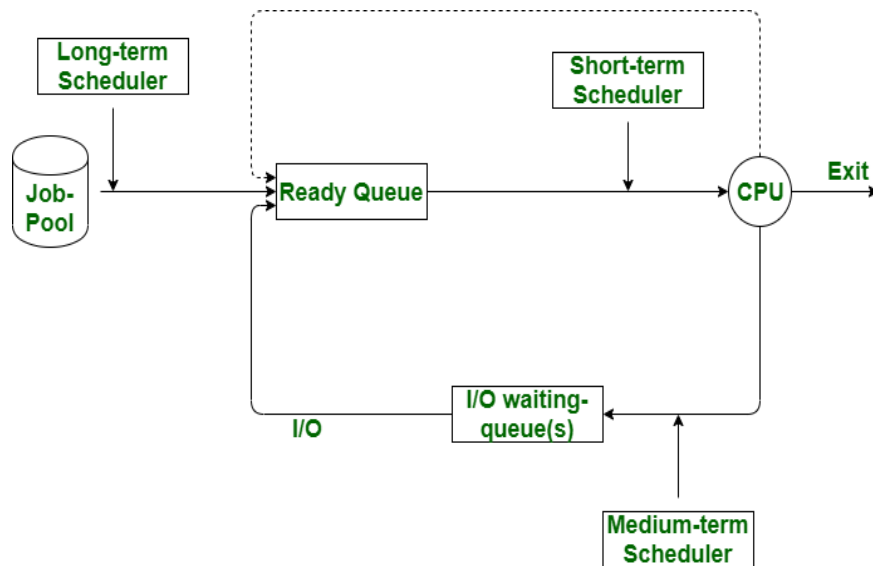


Process Scheduling

- **Schedulers**

- Schedulers are software which selects an available program to be assigned to CPU.
- A **long-term scheduler or Job scheduler** – selects jobs from the job pool (of secondary memory, disk) and loads them into the memory.
- If more processes are submitted, than that can be executed immediately, such processes will be in secondary memory. It runs infrequently, and can take time to select the next process.
- The **short-term scheduler, or CPU Scheduler** – selects job from memory and assigns the CPU to it. It must select the new process for CPU frequently.
- The **medium-term scheduler** - selects the process in ready queue and reintroduced into the memory.

Process Scheduling



Process Scheduling

- Processes can be described as either:
 - I/O-bound process** – spends more time doing I/O than computations,
 - CPU-bound process** – spends more time doing computations and few I/O operations.
- An efficient scheduling system will select a good mix of CPU-bound processes and I/O bound processes.
 - If the scheduler selects **more I/O bound process**, then I/O queue will be full and ready queue will be empty.
 - If the scheduler selects **more CPU bound process**, then ready queue will be full and I/O queue will be empty.
- Time sharing systems** employ a **medium-term scheduler**. It swaps out the process from ready queue and swap in the process to ready queue. When system loads get high, this scheduler will swap one or more processes out of the ready queue for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.

Process Scheduling

- Advantages of **medium-term scheduler** –
 - To remove process from memory and thus reduce the degree of multiprogramming (number of processes in memory).
 - To make a proper mix of processes (CPU bound and I/O bound).

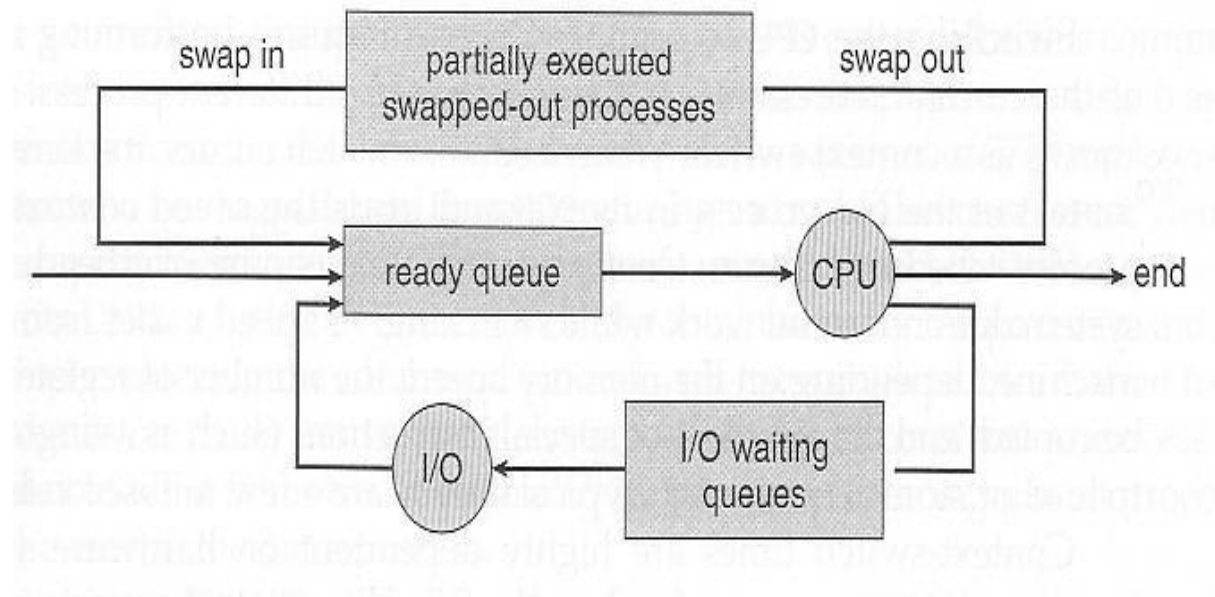


Fig. Addition of medium term scheduling to the queuing diagram

Process Scheduling

- Context switching

- The task of switching a CPU from one process to another process is called context switching. Context-switch times are highly dependent on hardware support (Number of CPU registers).
- Whenever an interrupt occurs (hardware or software interrupt), the state of the currently running process is saved into the PCB and the state of another process is restored from the PCB to the CPU.
- Context switch time is an overhead, as the system does not do useful work while switching.

CONTEXT SWITCHING



Operations on Processes

- A process may create several new processes. 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. Every process has a unique process ID.
- On typical Solaris systems, the process at the top of the tree is the '**sched**' process with PID of 0. The '**sched**' process creates several children processes – **init**, **pageout** and **fsflush**.
- Pageout and fsflush are responsible for managing memory and file systems. The init process with a PID of 1, serves as a parent process for all user processes.

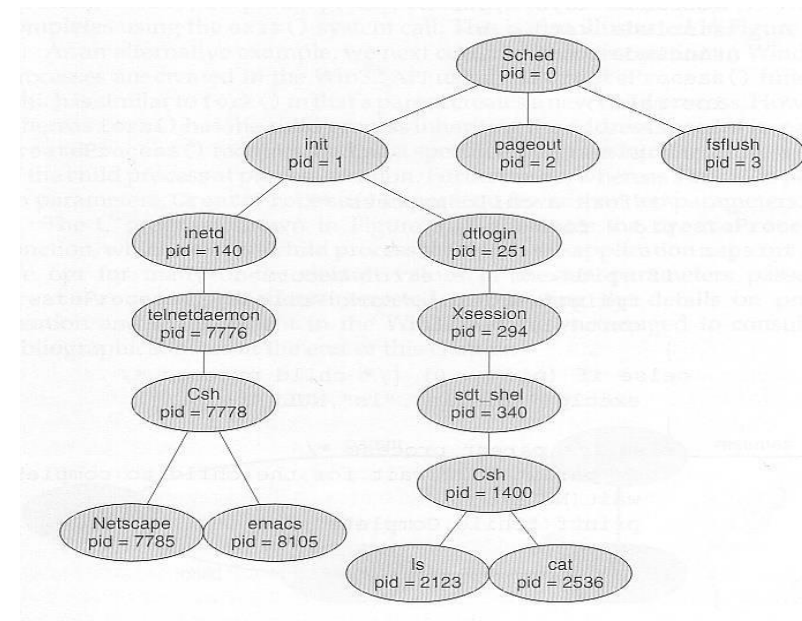


Fig. A tree of processes on a typical Solaris system

Operations on Processes

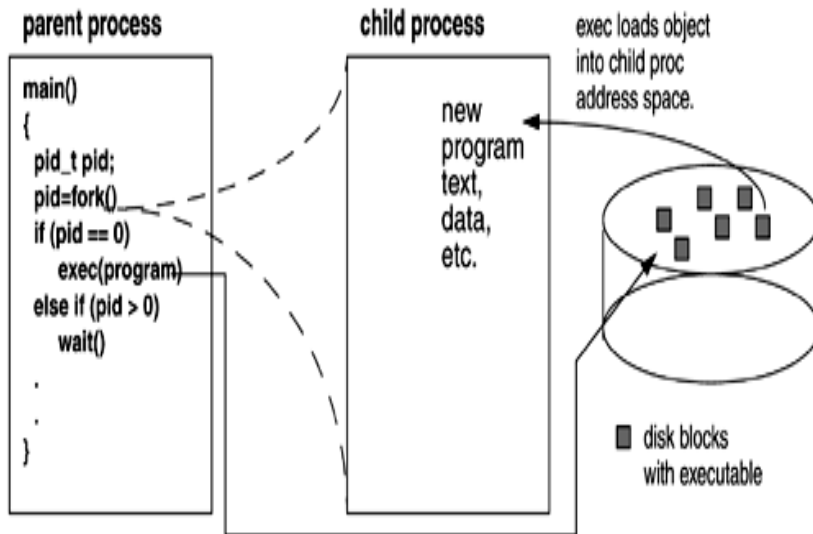


Fig. Process Creation using fork system call

```
iex(1)> execute_fun =
fn ->
  IO.puts "Hi! 🌟 I'm the process #{inspect(self)}."
end
#Function<45.65746770/0 in :erl_eval.expr/5>

iex(2)> pid = spawn(execute_fun)
Hi! 🌟 I'm the process #PID<0.234.0>.
#PID<0.234

iex(2)> Process.alive?(pid)
false
```



Fig. Process Creation using spawn system call

Operations on Processes

- A process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- When a process creates a subprocess, the subprocess may be able to obtain its resources in two ways:
 1. Directly from the operating system
 2. Subprocess may take the resources of the parent process.

The resource can be taken from parent in two ways –

 - The parent may have to partition its resources among its children
 - Share the resources among several children.

Operations on Processes

- **There are two options for the parent process after creating the child:**
 - Wait for the child process to terminate and then continue execution. The parent makes a wait() system call.
 - Run concurrently with the child, continuing to execute without waiting.
- **Two possibilities for the address space of the child relative to the parent:**
 - The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behaviour of the fork system call in UNIX.
 - The child process may have a new program loaded into its address space, with all new code and data segments. This is the behaviour of the spawn system calls in Windows.

Operations on Processes

- In UNIX OS, a child process can be created by fork() system call. The fork system call, if successful, returns the PID of the child process to its parents and returns a zero to the child process. If failure, it returns -1 to the parent. Process IDs of current process or its direct parent can be accessed using the getpid() and getppid() system calls respectively.
- The parent waits for the child process to complete with the wait() system call. When the child process completes, the parent process resumes and completes its execution.

```
#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");
        exit (-1) ;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
    exit (0) ;
}
```

Fig. C program forking a separate process

Operations on Processes

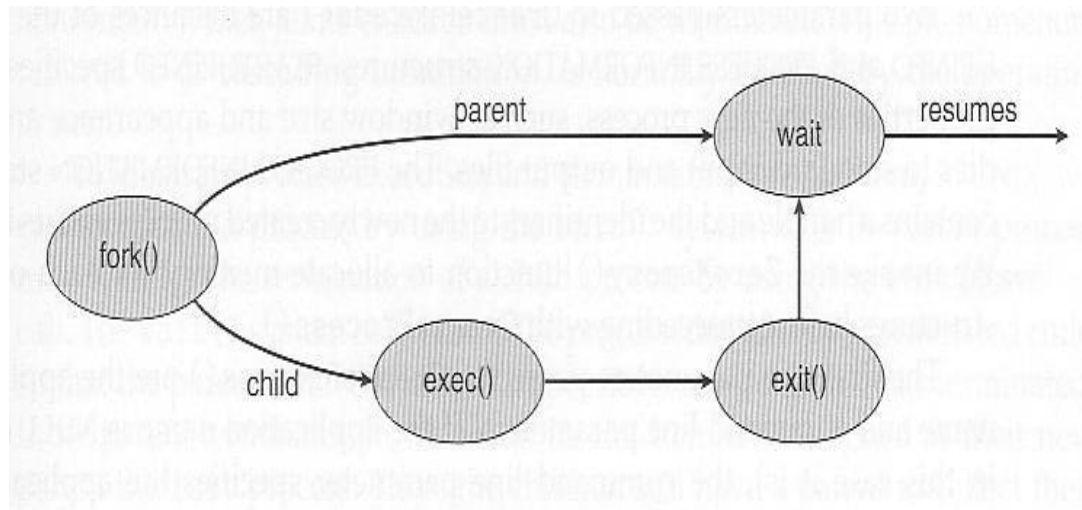


Fig. Process creation

- In windows the child process is created using the function `createprocess( )`. The `createprocess( )` returns 1, if the child is created and returns 0, if the child is not created.

Operations on Processes

- **Process Termination**

- A process terminates when it finishes executing its last statement and asks the operating system to delete it, by using the `exit()` system call. All of the resources assigned to the process like memory, open files, and I/O buffers, are deallocated by the operating system.
- A process can cause the termination of another process by using appropriate system call. The parent process can terminate its child processes by knowing of the PID of the child.
- A parent may terminate the execution of children for a variety of reasons, such as:
 - The child has exceeded its usage of the resources, it has been allocated.
 - The task assigned to the child is no longer required.
 - The parent is exiting, and the operating system terminates all the children. This is called cascading termination.

Interprocess Communication

- **Interprocess Communication-** Processes executing may be either co-operative or independent processes.
- **Independent Processes** – processes that cannot affect other processes or be affected by other processes executing in the system.
- **Cooperating Processes** – processes that can affect other processes or be affected by other processes executing in the system.

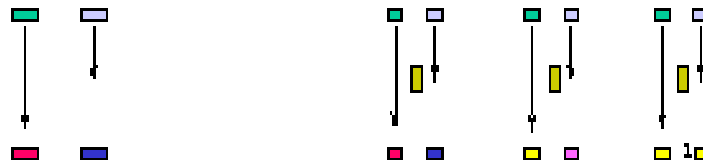


Fig. Independent v/s Cooperating processes

Interprocess Communication

- Co-operation among processes are allowed for following reasons –
 - **Information Sharing** - There may be several processes which need to access the same file. So the information must be accessible at the same time to all users.
 - **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks, which are solved simultaneously (particularly when multiple processors are involved).
 - **Modularity** - A system can be divided into cooperating modules and executed by sending information among one another.
 - **Convenience** - Even a single user can work on multiple tasks by information sharing.

Interprocess Communication

- Cooperating processes require some type of inter-process communication. This is allowed by two models:
 - Shared Memory systems
 - Message passing systems

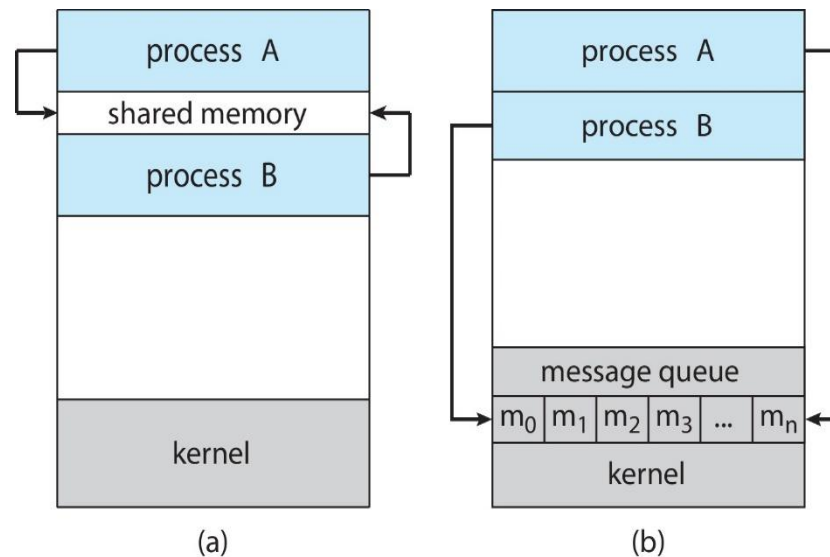


Fig. Communication models (a) Shared memory (b) Message passing

Interprocess Communication

Sl. No.	Shared Memory	Message passing
1.	A region of memory is shared by communicating processes, into which the information is written and read.	Message exchange is done among the processes by using objects.
2.	Useful for sending large block of data.	Useful for sending small data.
3.	System call is used only to create shared memory.	System call is used during every read and write operation.
4.	Message is sent faster, as there are no system calls.	Message is communicated slowly.
5.	No kernel intervention.	It involves kernel intervention.
6.	Example- Data from a client process may need to be transferred to a server process for modification before being returned to the client.	Example- Web browsers, Web Servers, Chat program on WWW (World Wide Web)

Shared-Memory Systems

- A region of shared-memory is created within the address space of a process, which needs to communicate. Other process that needs to communicate uses this shared memory.
- The form of data and position of creating shared memory area is decided by the process. Generally, a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.
- The process should take care that the two processes will not write the data to the shared memory at the same time.

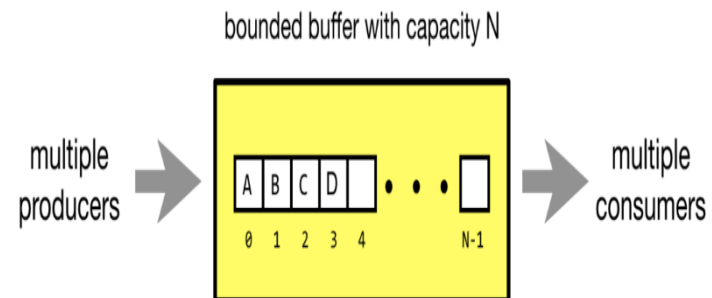
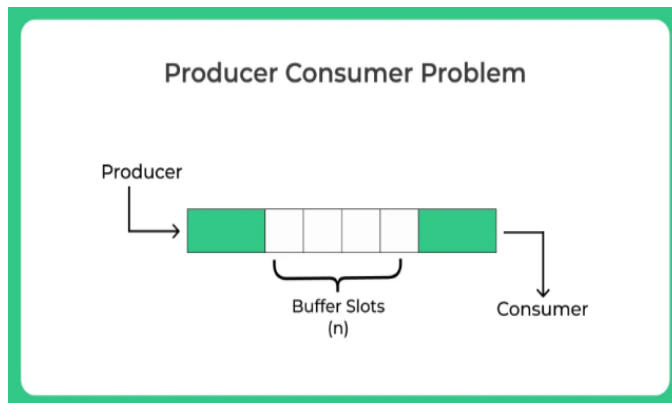
Shared-Memory Systems

- **Producer-Consumer Example Using Shared Memory**

- This is a classic example, in which one process is producing data and another process is consuming the data.
- The data is passed via an intermediary buffer (shared memory).
- The producer puts the data to the buffer and the consumer takes out the data from the buffer.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- In this situation, the consumer must wait until an item is produced.
- There are two types of buffers into which information can be put –
 - Unbounded buffer
 - Bounded buffer

Shared-Memory Systems

- **Unbounded-buffer** places no practical limit on the size of the buffer:
 - Producer never waits
 - Consumer waits if there is no buffer to consume
- **Bounded-buffer** assumes that there is a fixed buffer size:
 - Producer must wait if all buffers are full
 - Consumer waits if there is no buffer to consume



Shared-Memory Systems

- This example uses shared memory as a **circular queue**. The in and out are two pointers to the array. Note in the code below that only the producer changes "in", and only the consumer changes "out".
- First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

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

Shared-Memory Systems

- The producer process –
Note that the buffer is full
when
 $[(in+1) \% BUFFER_SIZE == out]$

```
item nextProduced;

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

Figure The producer process.

- The consumer process –
Note that the buffer is
empty when
 $[in == out]$

```
item nextConsumed;

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

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

Figure The consumer process.

Message-Passing Systems

- A mechanism to allow process communication without sharing address space. It is used in distributed systems.
- Message passing systems uses system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three methods of creating the link between the sender and the receiver-
 1. Direct or indirect communication (naming)
 2. Synchronous or asynchronous communication (Synchronization)
 3. Automatic or explicit buffering.

Message-Passing Systems

Naming

- Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.
- Direct communication:** the sender and receiver must explicitly know each other's name. The syntax for send() and receive() functions are as follows-
 - send (P, message)** – send a message to process P
 - receive(Q, message)** – receive a message from process Q
- Properties of communication link:**
 - A link is established automatically between every pair of processes that wants to communicate. The processes need to know only each other's identity to communicate.
 - A link is associated with exactly one pair of communicating processes
 - Between each pair, there exists exactly one link.

Message-Passing Systems

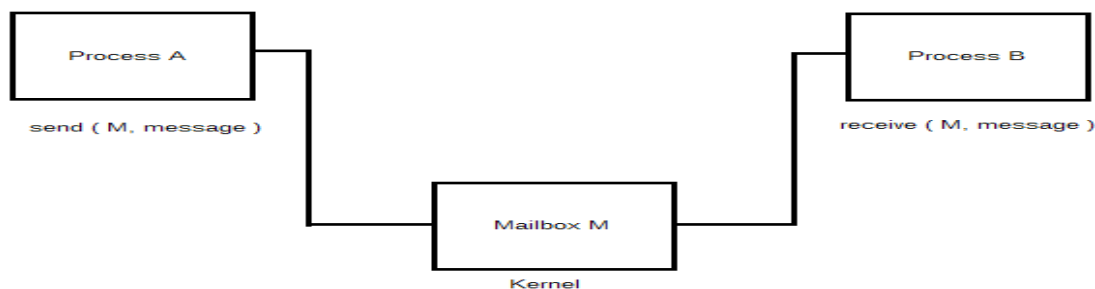
Types of addressing in direct communication –

- 1. Symmetric addressing** – the above-described communication is symmetric communication. Here both the sender and the receiver processes have to name each other to communicate.
- 2. Asymmetric addressing** – Here only the sender's name is mentioned, but the receiving data can be from any system.
 - send (P, message)** - Send a message to process P
 - receive (id, message)** - Receive a message from any process

Disadvantages of direct communication – any changes in the identifier of a process, may have to change the identifier in the whole system (sender and receiver), where the messages are sent and received.

Message-Passing Systems

- **Indirect communication** uses shared mailboxes, or ports.
 - A mailbox or port is used to send and receive messages.
 - Mailbox is an object into which messages can be sent and received.
 - It has a unique ID. Using this identifier, messages are sent and received.
- Two processes can communicate only if they have a shared mailbox.
- The send and receive functions are –
 - **send (A, message)** – send a message to mailbox A
 - **receive (A, message)** – receive a message from mailbox A



Message-Passing Systems

- **Indirect communication**
- **Properties of communication link:**
 - A link is established between a pair of processes only if they have a shared mailbox.
 - A link may be associated with more than two processes
 - Between each pair of communicating processes, there may be any number of links, each link is associated with one mailbox.
- A mail box can be owned by the operating system. It must take steps to –
 1. create a new mailbox
 2. send and receive messages from mailbox
 3. delete mailboxes

Message-Passing Systems

Synchronization

The send and receive messages can be implemented as either blocking or non-blocking.

- **Blocking (synchronous) send** - sending process is blocked (waits) until the message is received by receiving process or the mailbox.
- **Non-blocking (asynchronous) send** - sends the message and continues (does not wait)
- **Blocking (synchronous) receive** - The receiving process is blocked until a message is available
- **Non-blocking (asynchronous) receive** - receives the message without block. The received message may be a valid message or null.

Message-Passing Systems

Buffering

When messages are passed, a temporary queue is created. Such queue can be of three capacities:

- 1. Zero capacity** – The buffer size is zero (buffer does not exist). Messages are not stored in the queue. The senders must block until receivers accept the messages.
- 2. Bounded capacity** - The queue is of fixed size(n). Senders must block if the queue is full. After sending 'n' bytes the sender is blocked.
- 3. Unbounded capacity** - The queue is of infinite capacity. The sender never blocks.

Message Passing Systems

- The producer process –

```
message next_produced;  
while (true) {  
    /* produce an item in next_produced */  
    send(next_produced) ;  
}
```

- The consumer process –

```
message next_consumed;  
while (true) {  
    receive(next_consumed) ;  
    /* consume the item in next_consumed */  
}
```

Multi-threaded Programming

Introduction

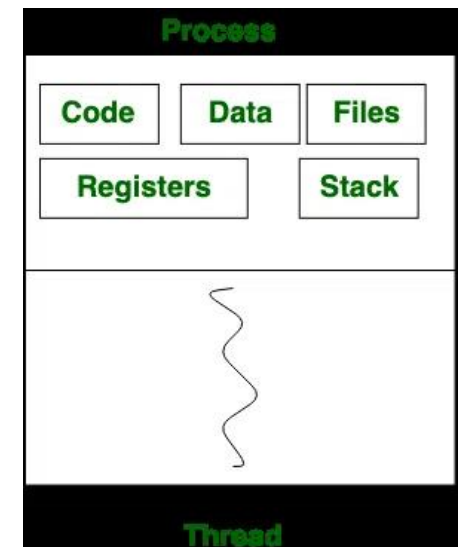
- Within a program, a Thread is a separate execution path. It is a lightweight process that the operating system can schedule and run concurrently with other threads.
- The operating system creates and manages threads, and they share the same memory and resources as the program that created them. This enables multiple threads to collaborate and work efficiently within a single program.
- A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes.
- Each thread belongs to exactly one process. In an operating system that supports multithreading, the process can consist of many threads.

Introduction

- The process can be split down into so many threads. For example, in a browser, many tabs can be viewed as threads. MS Word uses many threads - formatting text from one thread, processing input from another thread, etc.

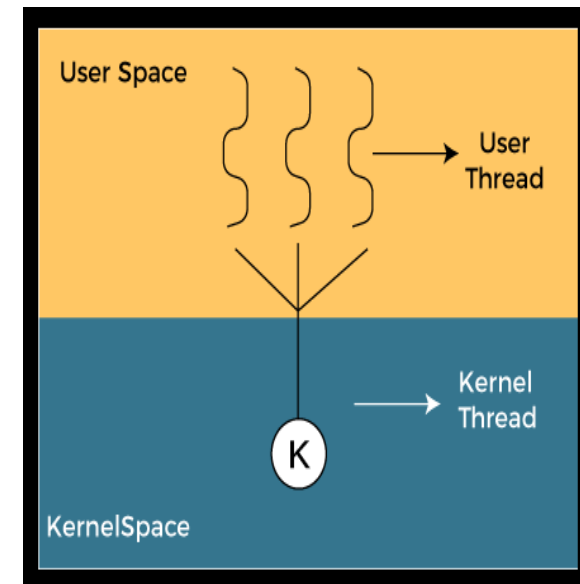
Need of Thread:

- It takes far less time to create a new thread in an existing process than to create a new process.
- Threads can share the common data, they do not need to use Inter- Process communication.
- Context switching is faster when working with threads.
- It takes less time to terminate a thread than a process.



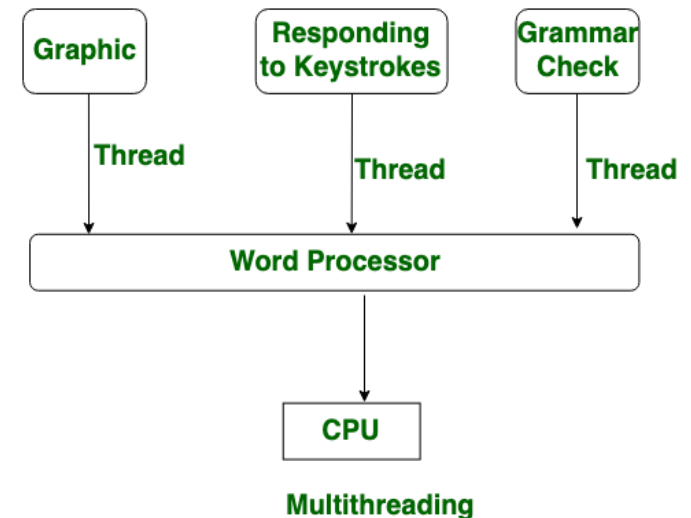
Types of Threads

- **User Level thread (ULT)** – Is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause interrupt to Kernel. Kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.
- **Kernel Level Thread (KLT)** – Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.

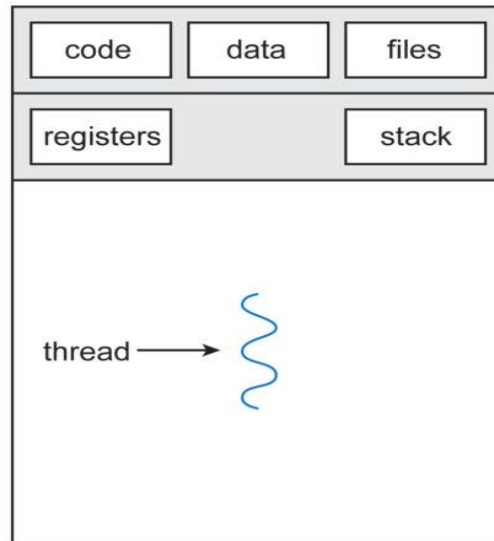


Multithreading

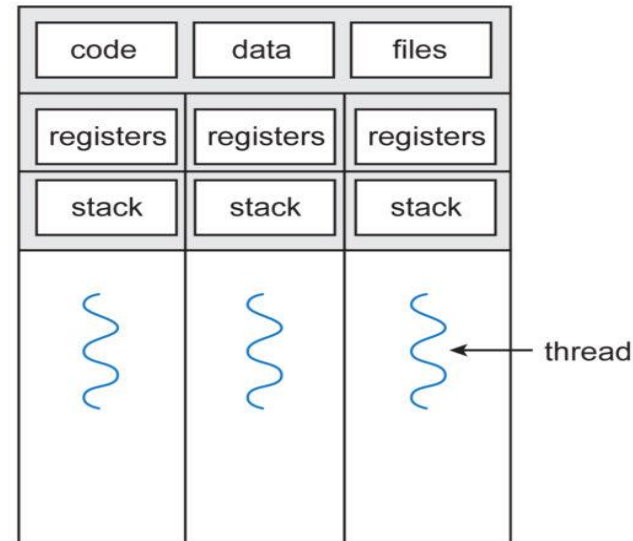
- **Multithreading** is a technique used in operating systems to improve the performance and responsiveness of computer systems.
- Multithreading allows multiple threads (i.e., lightweight processes) to share the same resources of a single process, such as the CPU, memory, and I/O devices.



Single & Multithreaded Processes



single-threaded process



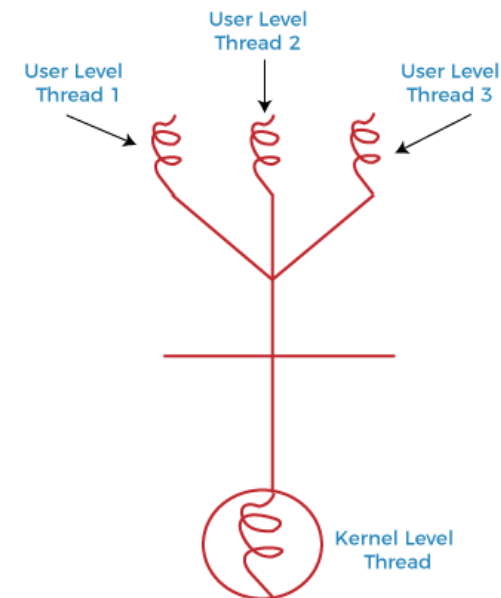
multithreaded process

Multithreading Models

- Multithreading allows the application to divide its task into individual threads. In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading. With the use of multithreading, multitasking can be achieved.
- The main drawback of single threading systems is that only one task can be performed at a time, so to overcome the drawback of this single threading, there is multithreading that allows multiple tasks to be performed.
- There exists three established multithreading models classifying these relationships are:
 - Many to one multithreading model
 - One to one multithreading model
 - Many to Many multithreading model

Many to one multithreading model

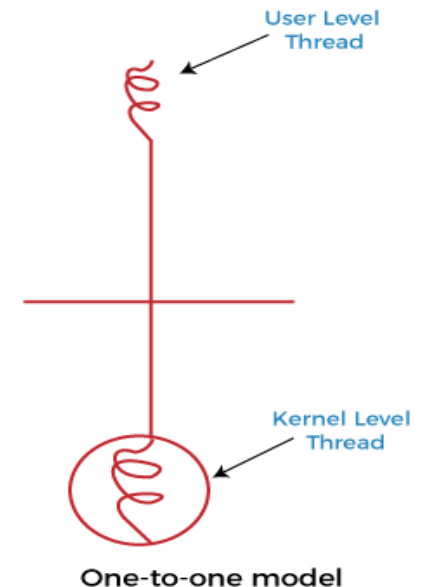
- The many to one model maps many user levels threads to one kernel thread. This type of relationship facilitates an effective context-switching environment, easily implemented even on the simple kernel with no thread support.
- The disadvantage of this model is that since there is only one kernel-level thread schedule at any given time, this model cannot take advantage of the hardware acceleration offered by multithreaded processes or multi-processor systems. In this, all the thread management is done in the userspace. If blocking comes, this model blocks the whole system.
- In the figure, the many to one model associates all user-level threads to single kernel-level threads.



Many-to-one model

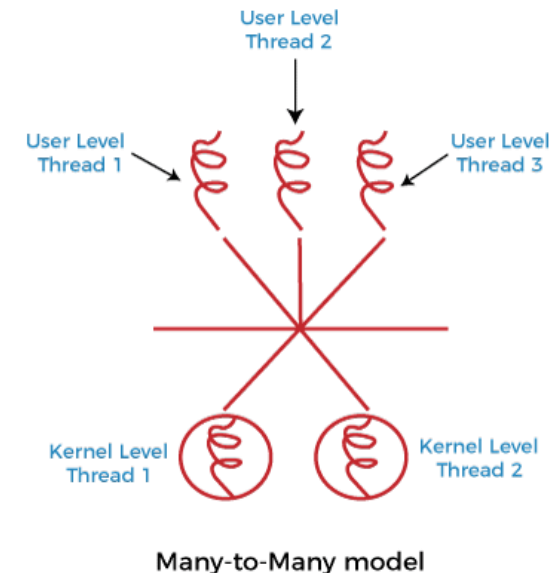
One to one multithreading model

- The one-to-one model maps a single user-level thread to a single kernel-level thread. This type of relationship facilitates the running of multiple threads in parallel. However, this benefit comes with its drawback. The generation of every new user thread must include creating a corresponding kernel thread causing an overhead, which can hinder the performance of the parent process. Windows series and Linux operating systems try to tackle this problem by limiting the growth of the thread count.
- In the figure, one model associates that one user-level thread to a single kernel-level thread.



Many to Many multithreading model

- In this type of model, there are several user-level threads and several kernel-level threads. The number of kernel threads created depends upon a particular application. The developer can create as many threads at both levels but may not be the same. The many to many model is a compromise between the other two models. In this model, if any thread makes a blocking system call, the kernel can schedule another thread for execution. Also, with the introduction of multiple threads, complexity is not present as in the previous models. Though this model allows the creation of multiple kernel threads, true concurrency cannot be achieved by this model. This is because the kernel can schedule only one process at a time.
- Many to many versions of the multithreading model associate several user-level threads to the same or much less variety of kernel-level threads in the above figure.



Thread Libraries

- A thread library provides the programmer with an Application program interface for creating and managing thread. Thread libraries can be the user level library or kernel level library.

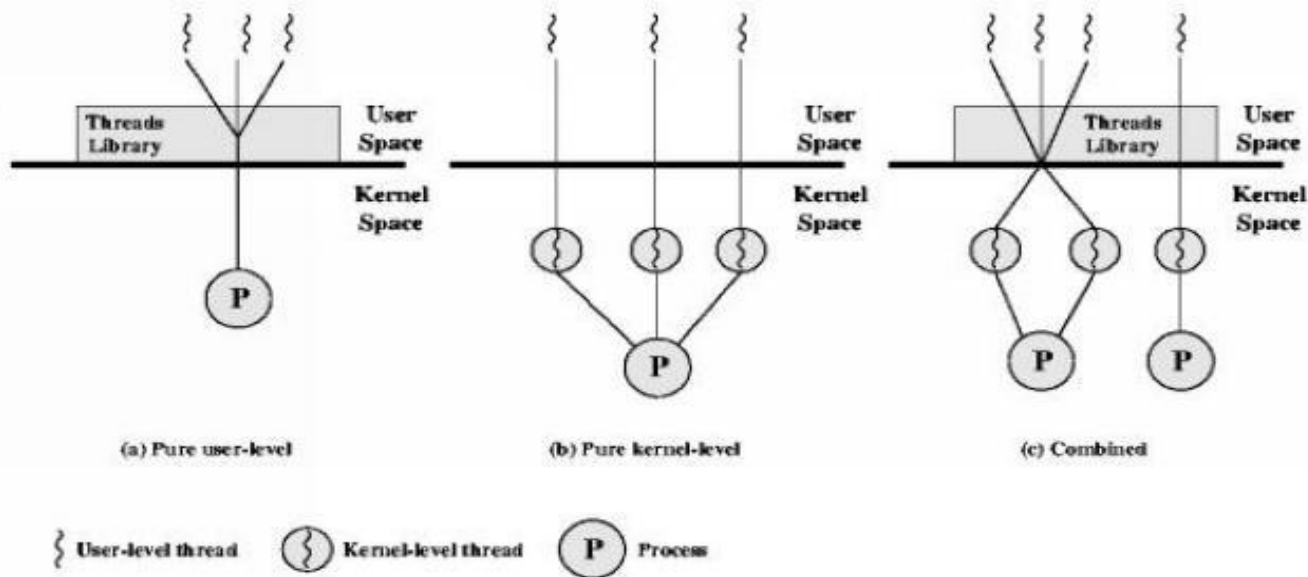
Ways of implementing thread library:

- There are two primary ways of implementing thread library, which are as follows –
 1. The first approach is to provide a library entirely in user space with kernel support. All code and data structures for the library exist in a local function call in user space and not in a system call.
 2. The second approach is to implement a kernel level library supported directly by the operating system. In this case the code and data structures for the library exist in kernel space.
- Invoking a function in the application program interface for the library typically results in a system call to the kernel.

Thread Libraries

- The main thread libraries which are used are given below –
 1. **POSIX threads** – Pthreads, the threads extension of the POSIX standard, may be provided as either a user level or a kernel level library.
 2. **WIN32 thread** – The windows thread library is a kernel level library available on windows systems.
 3. **JAVA thread** – The JAVA thread API allows threads to be created and managed directly as JAVA programs.

Thread Libraries

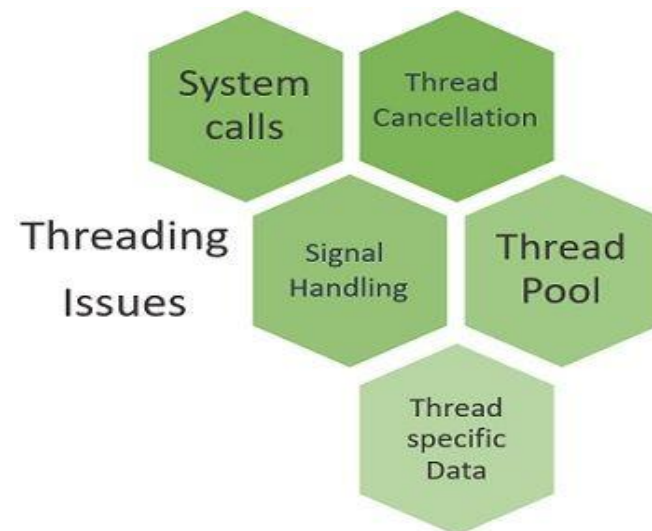


Threading Issues

- There are several threading issues when we are in a multithreading environment

Threading Issues in OS

1. System calls
2. Cancellation of thread
3. Signal handling
4. Thread pool
5. Thread-specific data



Threading Issues

System calls (The fork() and exec() system calls)

- The fork() is used to create a duplicate process. The meaning of the fork() and exec() system calls change in a multithreaded program.
- If one thread in a program which calls fork(), does the new process duplicate all threads, or is the new process single-threaded? If we take, some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
- If a thread calls the exec() system call, the program specified in the parameter to exec() will replace the entire process which includes all threads.

Threading Issues

Signal Handling

- Generally, signal is used in UNIX systems to notify a process that a particular event has occurred. A signal received either synchronously or asynchronously, based on the source of and the reason for the event being signalled.
- All signals, whether synchronous or asynchronous, follow the same pattern as given below –
 - A signal is generated by the occurrence of a particular event.
 - The signal is delivered to a process.
 - Once delivered, the signal must be handled.

Threading Issues

Cancellation

- Thread cancellation is the task of terminating a thread before it has completed.
- For example – If multiple database threads are concurrently searching through a database and one thread returns the result the remaining threads might be cancelled.
- A target thread is a thread that is to be cancelled, cancellation of target thread may occur in two different scenarios –
 - **Asynchronous cancellation** – One thread immediately terminates the target thread.
 - **Deferred cancellation** – The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an ordinary fashion.

Threading Issues

Thread pools

- Multithreading in a web server, whenever the server receives a request it creates a separate thread to service the request.
- Some of the problems that arise in creating a thread are as follows –
 - The amount of time required to create the thread prior to serving the request together with the fact that this thread will be discarded once it has completed its work.
 - If all concurrent requests are allowed to be serviced in a new thread, there is no bound on the number of threads concurrently active in the system.
 - Unlimited thread could exhaust system resources like CPU time or memory.
- A thread pool is to create a number of threads at process start-up and place them into a pool, where they sit and wait for work.

Threading Issues

Thread Specific data

- We all are aware of the fact that the threads belonging to the same process share the data of that process.
- Here the issue is what if each particular thread of the process needs its own copy of data. So the specific data associated with the specific thread is referred to as thread-specific data.
- Consider a transaction processing system, here we can process each transaction in a different thread. To determine each transaction uniquely we will associate a unique identifier with it, which will help the system to identify each transaction uniquely.

Process Scheduling

- Process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.
- The objective of multiprogramming is to have some process running at all times to maximize CPU utilization. The objective of time-sharing system is to switch the CPU among processes so frequently that users can interact with each program while it is running. For a uni-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.

CPU Scheduling – Basic Concept

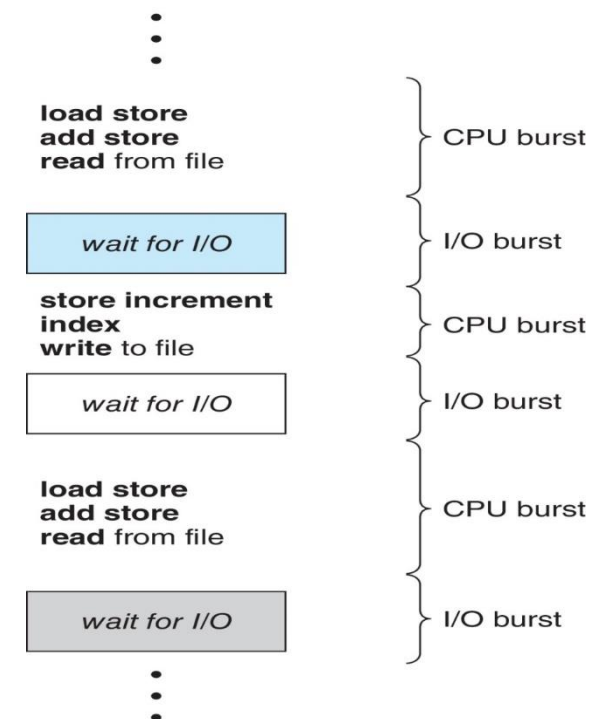
- In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively.

CPU Scheduling – Basic Concept

- Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.
- Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use.
- The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

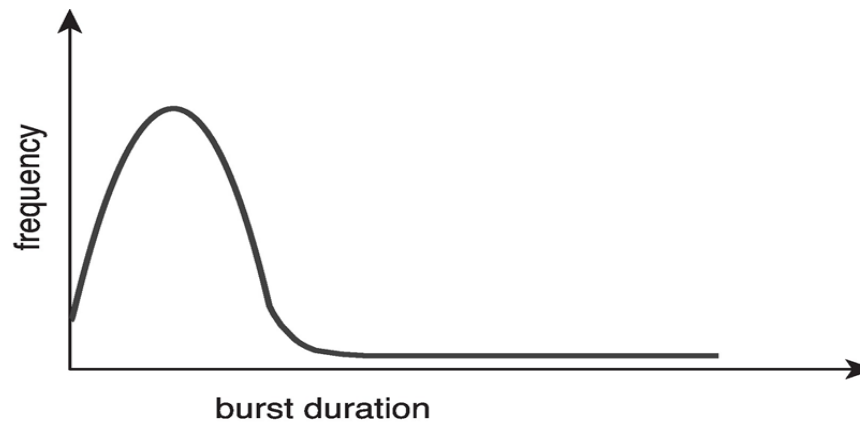
CPU- I/O Burst Cycle

- The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/O wait.
- Processes alternate between these two states.
- Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution



CPU- I/O Burst Cycle

- The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in figure.



CPU- I/O Burst Cycle

- The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts.
- An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

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).
- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. For situations 2 and 3, however, there is a choice.

CPU Scheduler

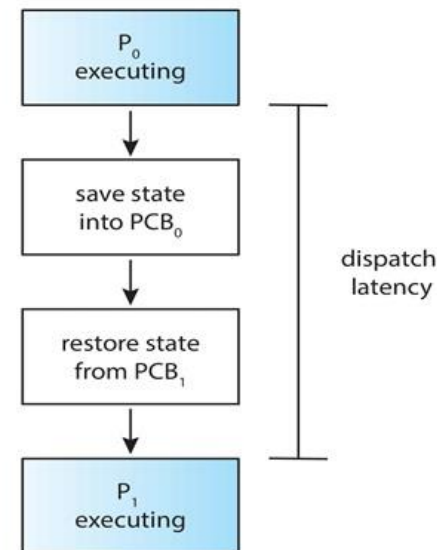
- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**; otherwise, it is **preemptive**.
- Under **nonpreemptive scheduling**, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.
- Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

CPU Scheduler

- Unfortunately, **preemptive scheduling** incurs a cost associated with access to shared data.
- Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations, we need new mechanisms to coordinate & access to shared data. (Race condition).
- Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues).

Dispatcher

- Another component involved in the CPU-scheduling function is the **dispatcher**.
- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:
 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 running is known as the dispatch latency.



Scheduling Criteria

- Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.
- The criteria include the following:
 1. CPU utilization
 2. Throughput
 3. Turnaround time
 4. Waiting time
 5. Response time

Scheduling Criteria

- **CPU utilization**: We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput**: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time**: From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Scheduling Criteria

- **Waiting time**: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**: In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

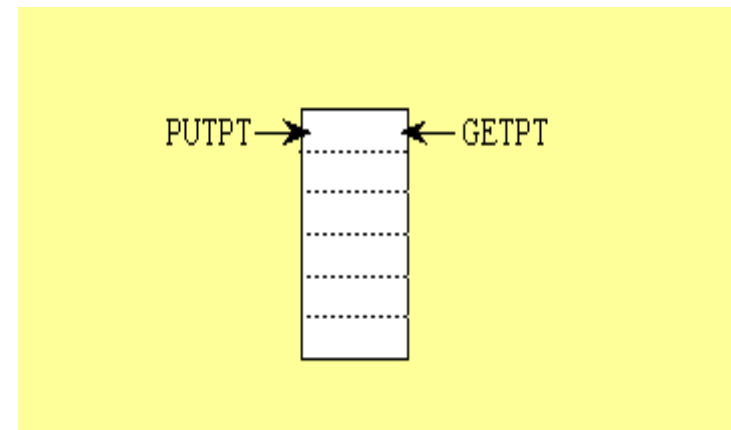
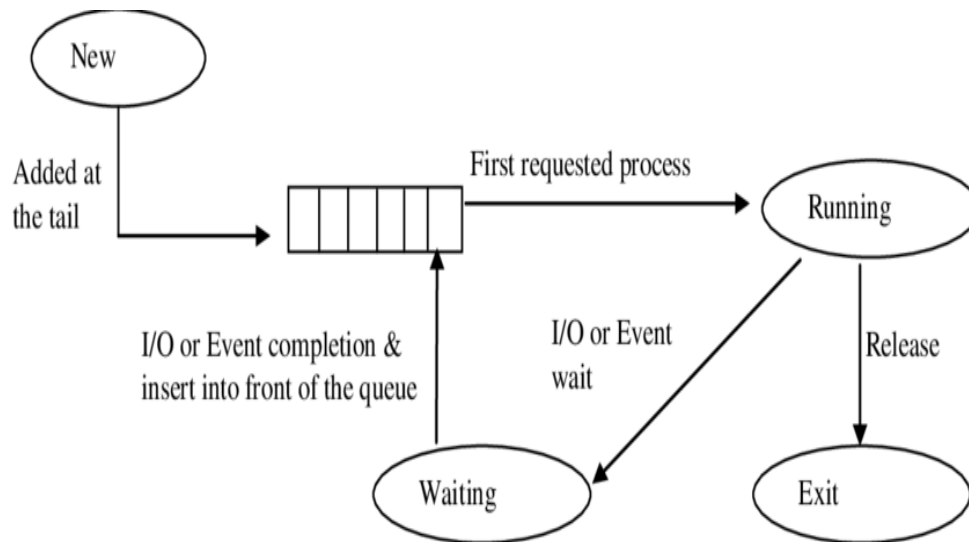
Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.
- There are many different CPU-scheduling algorithms.
 - First-Come, First-Served Scheduling
 - Shortest-Job-First Scheduling
 - Priority Scheduling
 - Round-Robin Scheduling
 - Multilevel Queue Scheduling
 - Multilevel Feedback Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

- The first-come, first-served (FCFS) scheduling algorithm is the simplest CPU-scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue. 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. The code for FCFS scheduling is simple to write and understand.
- On the negative side, the average waiting time under the FCFS policy is often quite long.

First-Come, First-Served (FCFS) Scheduling



First-Come, First-Served (FCFS) Scheduling

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

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

It is the total time that a process requires for its overall execution.

- If the processes arrive in the order P_1 , P_2 , P_3 , and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



First-Come, First-Served (FCFS) Scheduling

- The waiting time is 0 milliseconds for process P₁, 24 milliseconds for process P₂ and 27 milliseconds for process P₃. Thus, the average waiting time is $(0 + 24 + 27)/3 = 17$ **milliseconds**.
- If the processes arrive in the order P₂, P₃, P₁, however, the results will be as shown in the following Gantt chart:



- The average waiting time is now $(6 + 0 + 3)/3 = 3$ **milliseconds**. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

First-Come, First-Served (FCFS) Scheduling

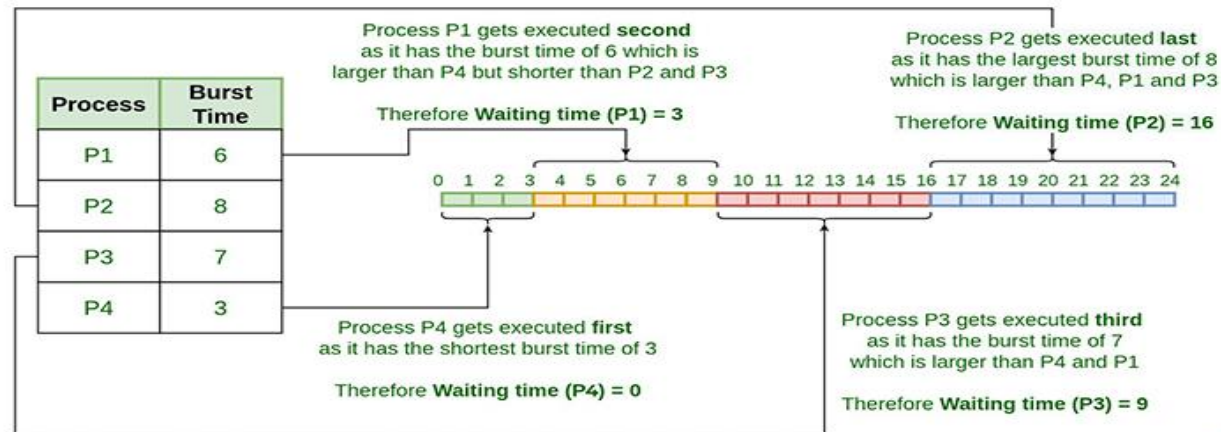
- There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
- The FCFS scheduling algorithm is **nonpreemptive**. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS algorithm is particularly **troublesome for time-sharing systems**, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

Shortest-Job-First (SJF) Scheduling

- A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst.
- 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 to break the tie.
- A more appropriate term for this scheduling method would be the **shortest-next-CPU-burst algorithm**, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

Shortest-Job-First (SJF) Scheduling

Shortest Job First (SJF) Scheduling Algorithm

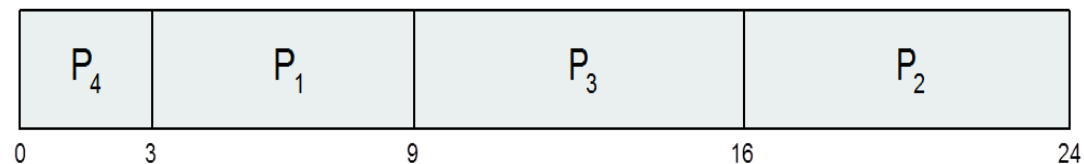


Shortest-Job-First (SJF) Scheduling

- Consider the following set of processes, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

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

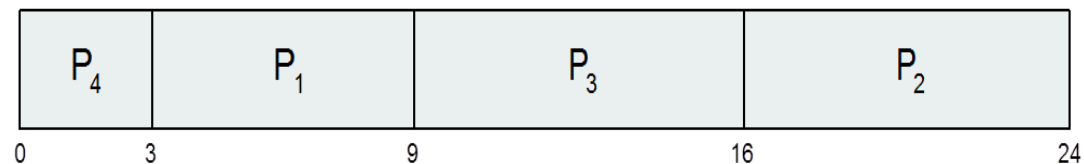


Shortest-Job-First (SJF) Scheduling

- Consider the following set of processes, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

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



Shortest-Job-First (SJF) Scheduling

- The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4 . Thus, the average waiting time is $(3 + 16 + 9 + 0) / 4 = \mathbf{7 \text{ milliseconds}}$. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be **10.25 milliseconds**.
- The SJF scheduling algorithm is provably **optimal**, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. *Consequently, the average waiting time decreases.*
- The real difficulty with the SJF algorithm is knowing the **length of the next CPU request**. Hence SJF scheduling is used frequently in long-term scheduling.

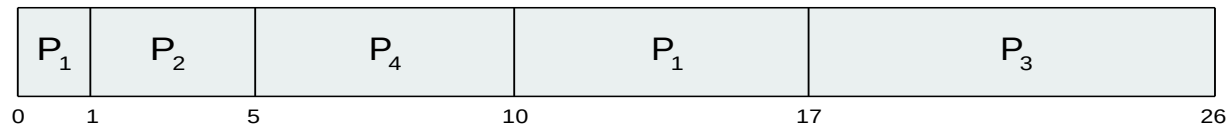
Shortest-Job-First (SJF) Scheduling

- A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling**.
- Consider the following four processes, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Arrival Time T</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Shortest-Job-First (SJF) Scheduling

- 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:



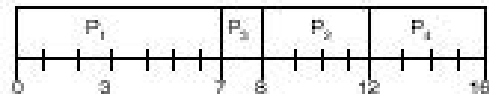
- Process P₁ is started at time 0, since it is the only process in the queue. Process P₂ arrives at time 1. The remaining time for process P₁ (7 milliseconds) is larger than the time required by process P₂ (4 milliseconds), so process P₁ is preempted, and process P₂ is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)] / 4 = 26 / 4 = \mathbf{6.5 \text{ milliseconds}}$.
- Nonpreemptive SJF scheduling would result in an average waiting time of **7.75 milliseconds**.

Shortest-Job-First (SJF) Scheduling

Example of Non-Preemptive SJF

Process Arrival Time Burst Time

P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



$$\text{average waiting time} = (0 + 6 + 3 + 7)/4 = 4$$

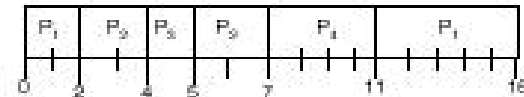
COMP3300

13

Example of Preemptive SJF

Process Arrival Time Burst Time

P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



$$\text{average waiting time} = (9 + 1 + 0 + 2)/4 = 3$$

COMP3300

14

Priority Scheduling

- The SJF algorithm is a special case of the general priority scheduling algorithm.
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- The implementation of the FCFS policy is easily managed with a FIFO queue. 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. The code for FCFS scheduling is simple to write and understand.
- On the negative side, the average waiting time under the FCFS policy is often quite long.

Priority Scheduling

- A priority is associated with each process.
- The CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- Priorities can be defined either internally or externally.
 - Internally-defined priorities.
- Use some measurable quantity to compute the priority of a process.
- For example: time limits, memory requirements, no. of open files.
 - Externally-defined priorities.
- Set by criteria that are external to the OS. For example: importance of the process, political factors

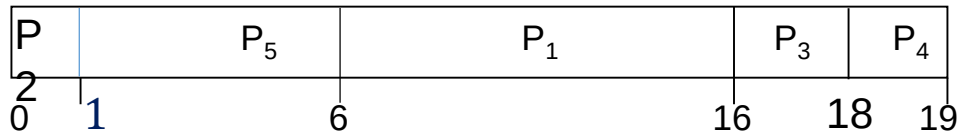
Priority Scheduling

- Priority scheduling can be either preemptive or non-preemptive.
- Preemptive:
 - The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process.
- Non Preemptive:
 - The new process is put at the head of the ready-queue
- Advantage:
- Higher priority processes can be executed first.

Priority Scheduling

Process	Burst Time	Priority
• P1	10	3
• P2	1	1
• P3	2	4
• P4	1	5
• P5	5	2

- The Gantt chart for the schedule is as follows:



- The average waiting time is
- $(6+0+16+18+1)/5 = 8.2$ milliseconds.

Round Robin (RR) Scheduling

- Designed especially for time sharing systems.
- It is similar to FCFS scheduling, but with preemption.
- A small unit of time is called a time quantum(or time slice).
- Time quantum is ranges from 10 to 100ms.
- The ready-queue is treated as a circular queue.
- The CPU scheduler goes around the ready-queue and allocates the CPU to each process for a time interval of up to 1 time quantum.
- To implement: The ready-queue is kept as a FIFO queue of processes

Round Robin (RR) Scheduling

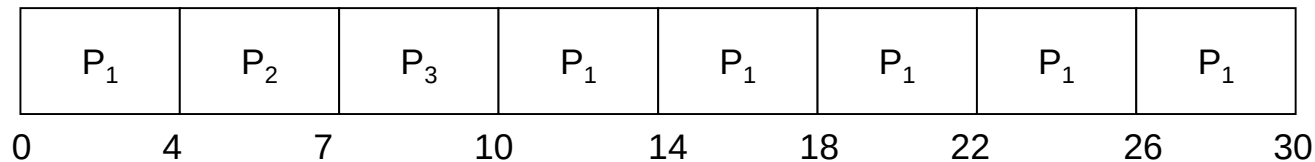
CPU scheduler

- Picks the first process from the ready-queue.
- Sets a timer to interrupt after 1 time quantum and
- Dispatches the process.
- One of two things will then happen.
 1. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily.
 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 OS. The process will be put at the tail of the ready-queue.

Round Robin (RR) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

The Gantt chart for the schedule is as follows:



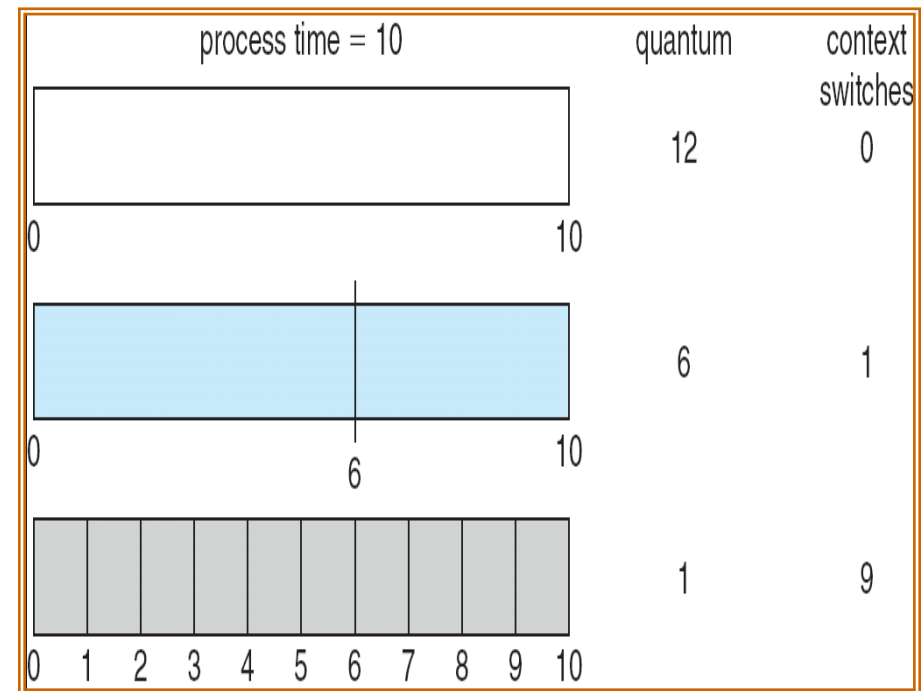
The average waiting time is $17/3 = 5.66$ milliseconds.

Round Robin (RR) Scheduling

- The RR scheduling algorithm is preemptive.
- No process is allocated the CPU for more than 1 time quantum in a row.
- If a process' CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready- queue.
- The performance of algorithm depends heavily on the size of the time quantum.
 1. If time quantum=very large, RR policy is the same as the FCFS policy
 2. If time quantum=very small, RR approach appears to the users as though each of n processes has its own processor running at $1/n$ the speed of the real processor.

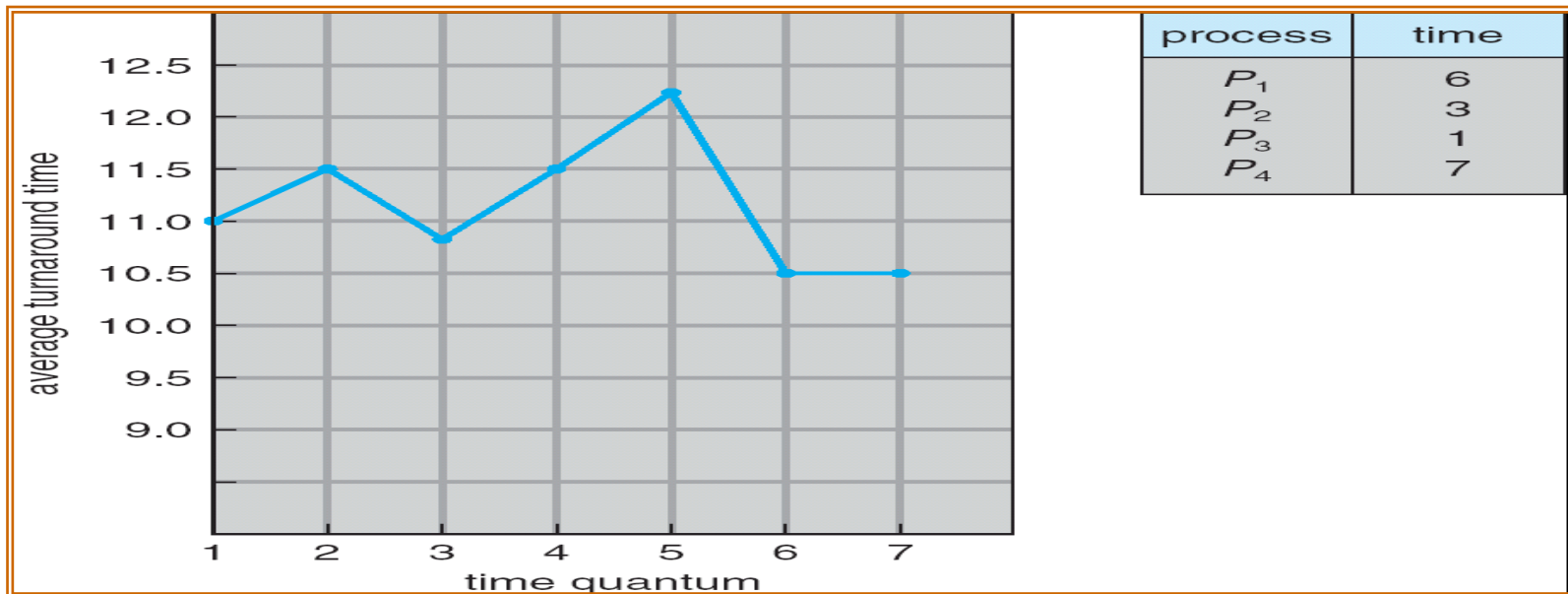
Round Robin (RR) Scheduling

- In software, we need to consider the effect of context switching on the
- performance of RR scheduling
- 1. Larger the time quantum for a specific process time, less time is spend on
- context switching.
- 2. The smaller the time quantum, more overhead is added for the purpose of
- context- switching.



Round Robin (RR) Scheduling

- Turnaround Time Varies With The Time Quantum

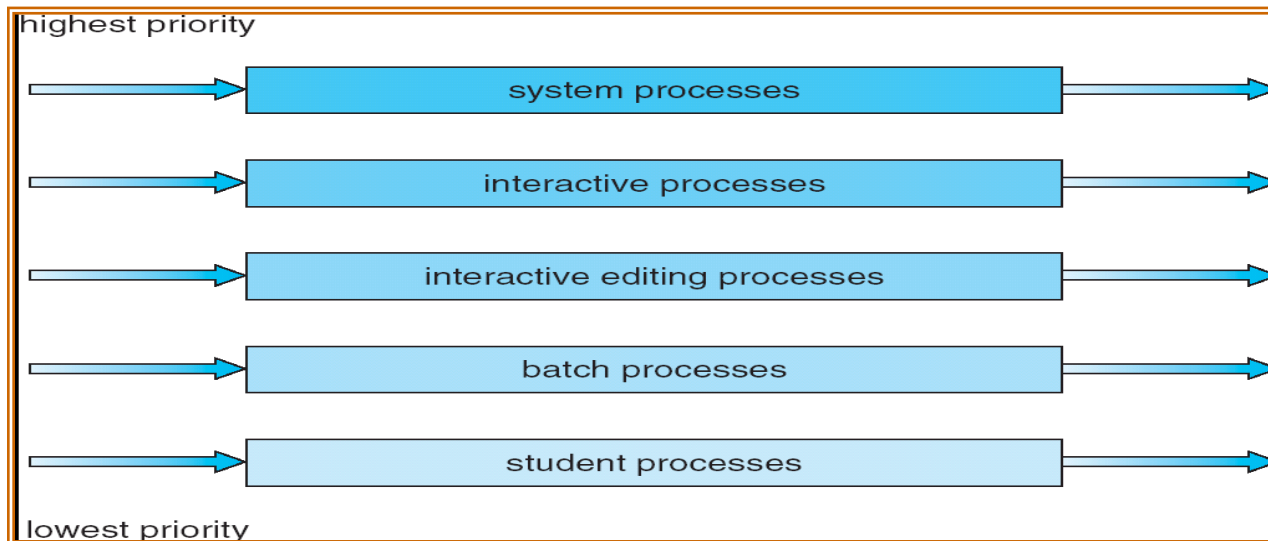


Multilevel Queue

- Useful for situations in which processes are easily classified into different groups. For example, a common division is made between
 - foreground (or interactive) processes and
 - background (or batch) processes.
- The ready-queue is partitioned into several separate queues (Figure 2.19).
- The processes are permanently assigned to one queue based on some property like
 - memory size
 - process priority or
 - process type.

Multilevel Queue

- Each queue has its own scheduling algorithm.
- For example, separate queues might be used for foreground and background processes.



Multilevel Queue

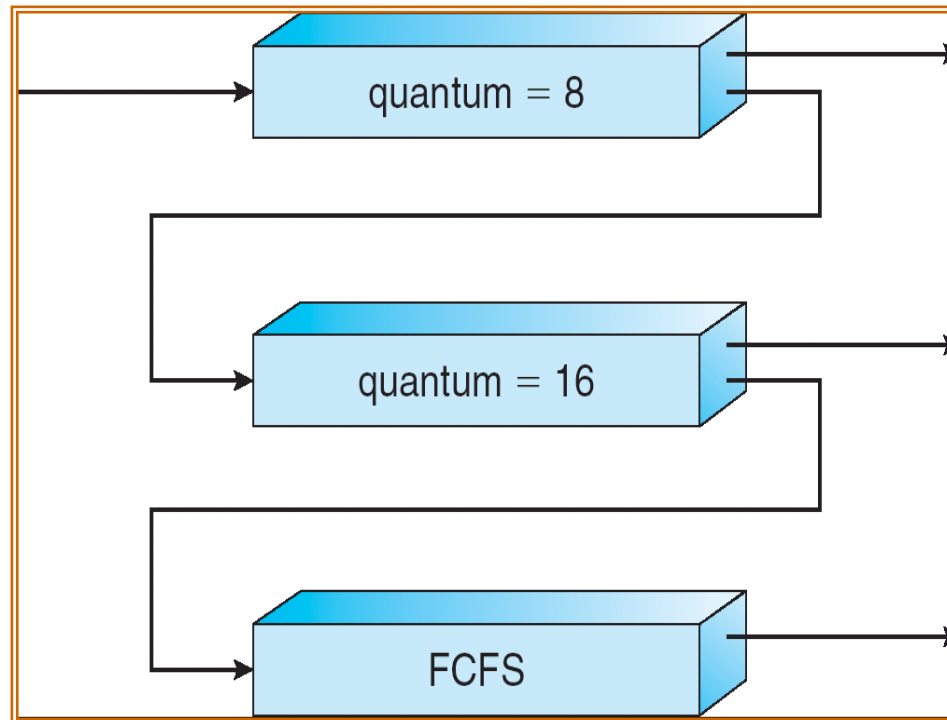
- There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
- For example, the foreground queue may have absolute priority over the background queue.
- Time slice: each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR 20% to background in FCFS

Multilevel Feedback Queue

- A process may move between queues
- The basic idea: Separate processes according to the features of their CPU bursts.
For example
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue This form of aging prevents starvation.

Example of Multilevel Feedback Queue

- Three queues:
 - Q0 – RR with time quantum 8 milliseconds
 - Q1 – RR time quantum 16 milliseconds
 - Q2 – FCFS
- Scheduling
 - A new job enters queue Q0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q1.
 - At Q1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q2.



Thread Scheduling

- On operating systems that support them, it is kernel-level threads-not processes-that are being scheduled by the operating system
- User-level threads are managed by a thread library, and the kernel is unaware of them.
- **Contention Scope**
- One distinction between user-level and kernel-level threads lies in how they are scheduled.
- On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP, a scheme known as **process-contention scope (PCS)**
- To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**.
- Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model, such as Windows XP, Solaris, and Linux, schedule threads using only SCS.

Question Bank

From Model Question Paper

1	With a neat diagram, explain the states of a process with a transition diagram and process control block. 6 marks
2	What is inter-process communication? Discuss message passing and the shared memory concept of IPC. 6 marks

From Previous year question papers

1	Define process. Explain different states of a process with state diagram. Dec'23/Jan'24 ,8 marks
2	What is IPC? Explain direct and indirect communication with respect to message passing. Dec'23/Jan'24, 8 marks
3	Explain context switching. Dec'23/Jan'24, 4marks

From Previous year question papers

4	Define IPC. What are the different methods used for logical implementation of a message passing system? Explain any one. Jan/Feb'2021 , 6 marks
5	Explain the functions of various schedulers with a block diagram. Aug/Sep'2020, 6 marks
6	Explain process with state diagram. What is the need for context switching? Explain the contents of PCB. Dec'19/Jan'20,6 marks
7	Describe the implementation of IPC using shared memory and message passing. Dec'17/Jan'18, 7 marks
8	Why it is important for scheduler to distinguish I/O bound process from CPU bound? Jun/July'17, 2 marks
9	With a neat diagram explain the components of PCB. Jun/July'14, 5marks

Question Bank

From Model Question Paper

- | | |
|---|--|
| 1 | Discuss in detail the multithreading model, its advantages and disadvantages with suitable illustration.
7 marks |
|---|--|

From Previous year question papers

- | | |
|---|---|
| 1 | What is multithreaded process? Explain the four benefits of multithreaded programming.
Dec'23/Jan'24 6 marks
A multithreaded process is a program execution model that allows multiple threads to run concurrently within a single process. |
| 2 | Explain the various multithreading model and benefits of multithreaded programs.
Jan/Feb'21 6 marks |
| 3 | Explain the three multithreading model.
Aug/Sept'20 6 marks |
| 4 | Why thread is called LWP? Describe any one threading model and threading issue.
Aug/Sept'20 4 marks
Threads are called lightweight processes (LWPs) because they require fewer resources to create than processes, and they share resources with other threads and the process they're part of. |

Process Scheduling