OPERATING SYSTEM (BISOS304)

MODULE-2

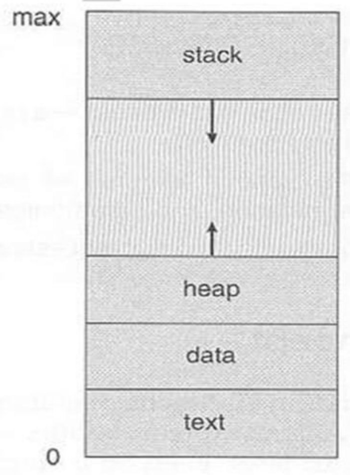
PROCESSES CONCEPT

- A process is a program under execution.
- Its current activity is indicated by PC(Program Counter) and CPU registers.

The Process

Process memory is divided into four sections as shown in the figure below:

- The stack is used to store local variables, function parameters, function return values, return address etc.
- The heap is used for dynamic memory allocation.
- The data section stores global and static 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.



ıure 3.1 Process in memory.

Process State

A Process has 5 states. Each process may be in one of the following states –

- **New -** The process is in the stage of being created.
- **Ready -** The process has all the resources it needs to run. It is waiting to be assigned to the processor.
- **Running** – Instructions are being executed..
- **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.
- **Terminated -** The process has completed its execution.

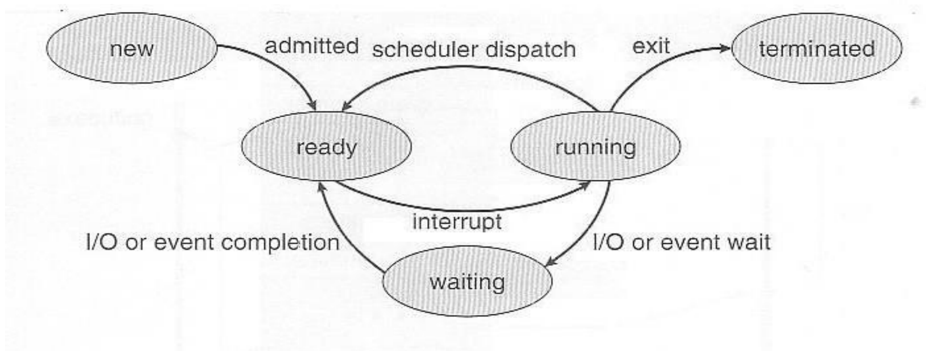
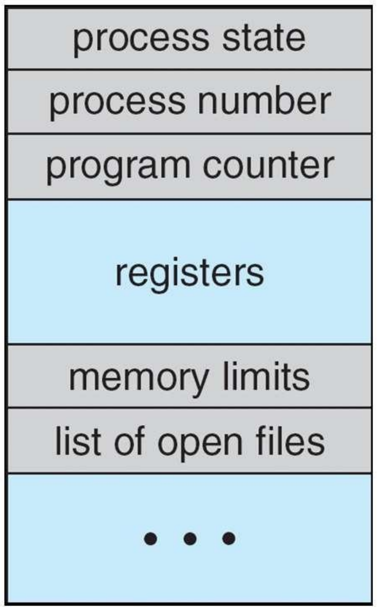


Figure 3.2 Diagram of process state.

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 theprogram counter, this state information must be saved when an interrupt occurs, to allow the process to becontinued correctly afterward.

CPU scheduling information- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory-management information – This include information such as the value of the base and limitregisters, 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.

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

Process Scheduling

Process Scheduler selects an available process for program execution on the CPU. In a multiprocessor system - one process will be under execution and the rest of the processes haveto wait until the CPU is free and can be rescheduled.

The main objective of process scheduling is to keep the CPU busy at all times.

Scheduling Queues

- All processes admitted to the system are stored in the **job queue.**
- Processes in main memory and ready to execute are placed in the **ready queue.**
- Processes waiting for a device to become available are placed in **device queues**. There is generally a separate device queue for each device.

These queues are generally stored as a linked list of PCBs. A queue header will contain two pointers - the **head pointer** pointing to the first PCB and the **tail pointer** pointing to the last PCB in the list. Each PCB has a pointer field that points to the next process in the queue.

When a process is allocated to the **CPU,** it executes for a while and eventually quits, interrupted, or waits for the completion of an I/O request. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk in the device queue.

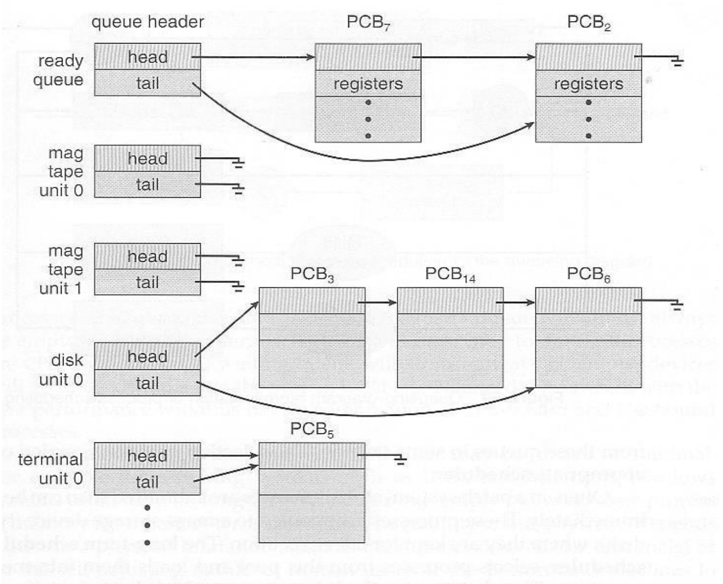


Figure 3.6 The ready queue and various I/O device queues.

A common representation of process scheduling is a ***queueing 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:

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

In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues.

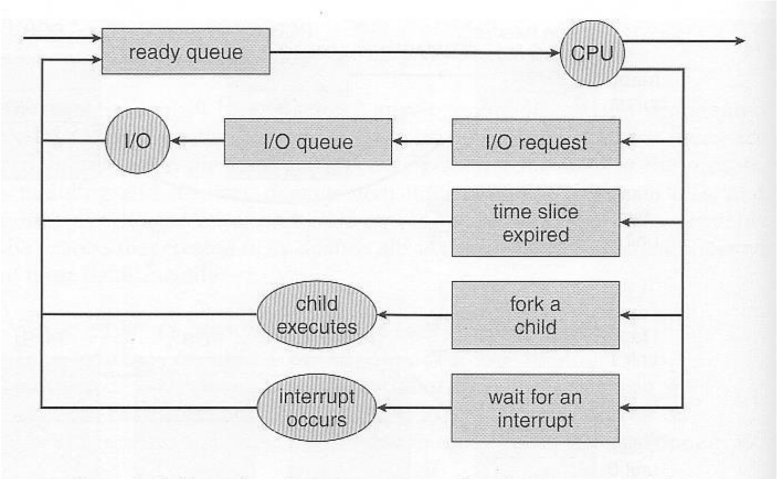


Figure 3.7 Queueing-diagram representation of process scheduling.

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 processeswill 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.

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.

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)

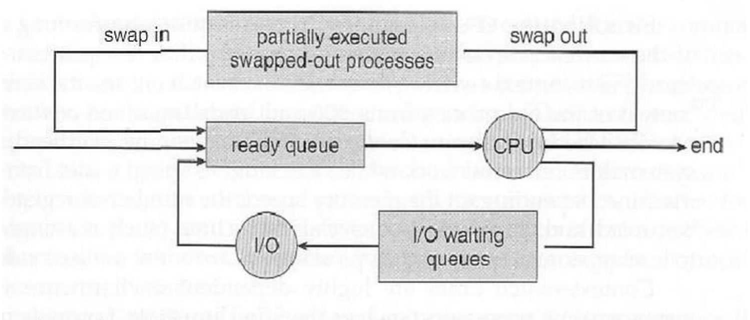
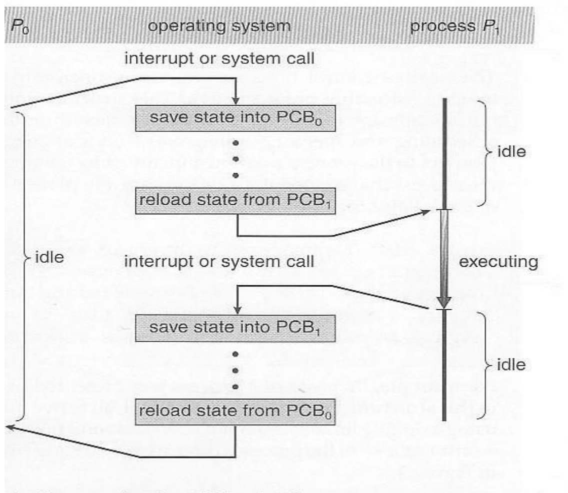


Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

Context Switch



.4 Diagram showing CPU switch from process to process.

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(No of CPU registers)

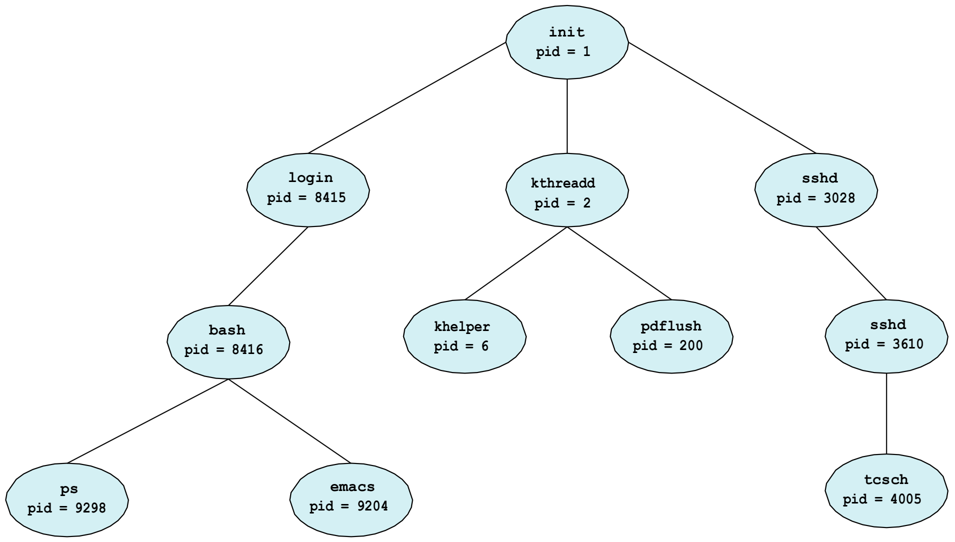
Whenever an interrupt occurs (hardware or software interrupt), the state of the currently running process is **saved** into the PCB and the state of another process is **restored** from the PCB to the CPU. Context switch time is an overhead, as the system does not do useful work while switching.

Operations on Processes

Process Creation

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 aparent process for all user 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 :

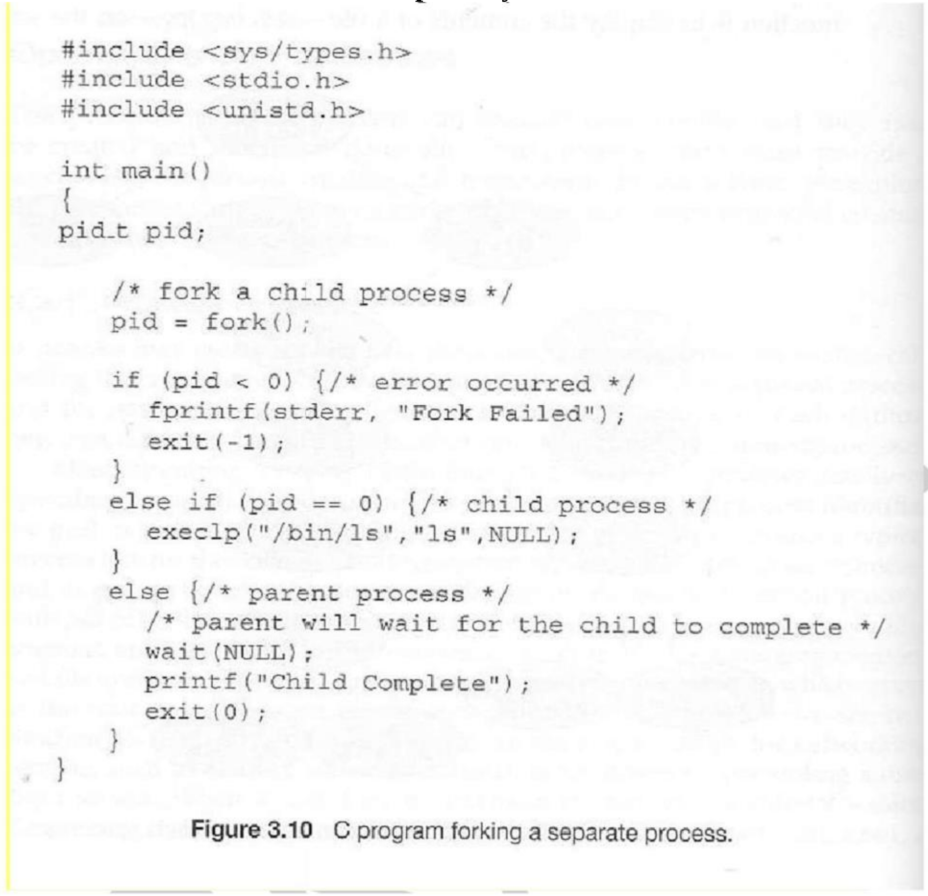
- directly from the operating system
- Subprocess may take the resources of the parent process.
  The resource can be taken from parent in two ways –
    - The parent may have to partition its resources among its children
    - Share the resources among several children.

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.



```
#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);
    }
}
```

Figure 3.10 C program forking a separate process.

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.

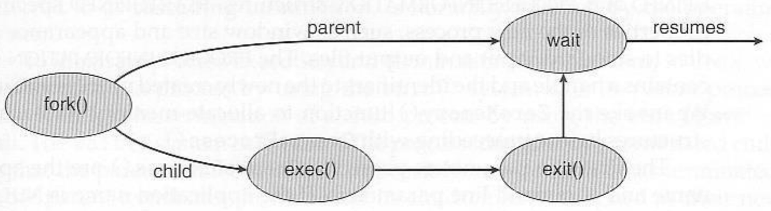


Figure 3.11 Process creation.

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.

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.

Note : Processes which are trying to terminate but which cannot because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off. (Modern UNIXshells do not produce as many orphans and zombies as older systems used to.

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.

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 task by information sharing.

Cooperating processes require some type of inter-process communication. This is allowed by two models : 1) Shared Memory systems 2)Message Passing systems.

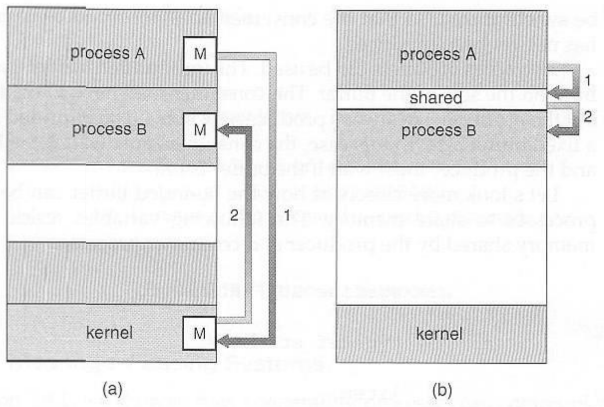
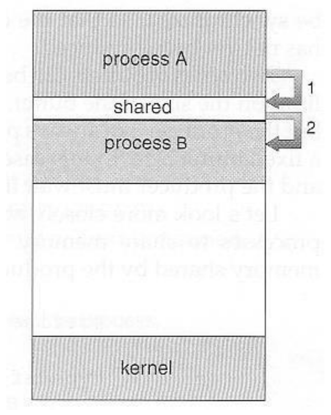


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

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.

- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small.

Shared-Memory Systems



A region of shared memory is created within the address space of a process, which needs to communicate. Other processes that need to communicate uses this shared memory. An area of shared memory segment needs to be created among the processes that wish to communicate. The communication is under the control of the users processes not the operating system. 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 cooperatingprocesses first to set up and coordinate the shared memory access.

Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory
The process should take care that the two processes will not write the data to the shared memoryat the same time.

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

With Unbounded buffer, there is no limit on the size of the buffer, and so on the data produced by producer. But the consumer may have to wait for new items.

With bounded-buffer – As the buffer size is fixed. The producer has to wait if the buffer is full and the consumer has to wait if the buffer is empty.

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              //buffer size
Typedef struct {
    . . .
} item;
item buffer[ BUFFER_SIZE ];
int in = 0;
int out = 0;
```

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

```
item nextProduced;
while( true )
{
        /* Produce an item and store it in nextProduced */
        nextProduced = makeNewItem( . . . );

        /* Wait for space to become available */
        while( ( ( in + 1 ) % BUFFER_SIZE ) == out )  //full
            ; /* Do nothing */

        /* And then, if not full store the item */
        buffer[ in ] = nextProduced;
        in = ( in + 1 ) % BUFFER_SIZE;
}
```

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

```
item nextConsumed;
while( true )
{
        /* Wait for an item to become available */
        while( in == out )      // buffer empty
            ; /* Do nothing */

        /* Get the next available item */
        nextConsumed = buffer[ out ];
        out = ( out + 1 ) % BUFFER_SIZE;
}
```

Message-Passing Systems

A mechanism to allow process communication without sharing addressspace. 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-
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication (Synchronization)
  - Automatic or explicit buffering.

a) Naming

The processes that wants to communicate should have a way to refer eachother. (using someidentity)

Direct communication the sender and receiver must explicitly know eachothers 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.

Types of addressing in direct communication –

- Symmetric addressing – the above described communication is symmetric communication. Here both the sender and the receiver processes have to name each other to communicate.
- Asymmetric addressing – Here only the sender 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.

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

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 –

- create a new mailbox
- send and receive messages from mailbox
- delete mailboxes.

b) 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 (doesnot 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.

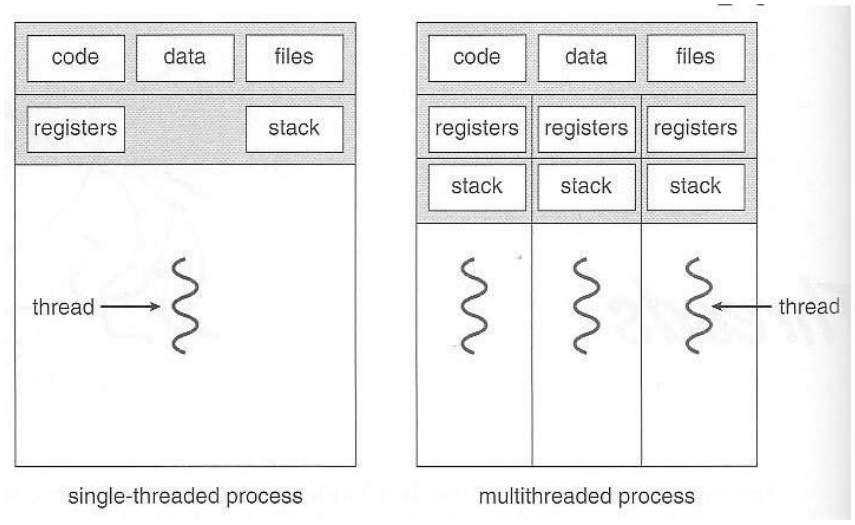
c) Buffering

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

- **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.
- **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.
- **Unbounded capacity -** The queue is of infinite capacity. The sender never blocks.

Multi threaded

- A ***thread*** is a basic unit of CPU utilization. It consists of a thread ID, program counter, a stack, and a set of registers.
- Traditional processes have a single thread of control. It is also called as **heavyweight process**. There is one program counter, and one sequence ofinstructions that can be carried out at any given time.
- A multi-threaded application have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files. Such process are called as **lightweight process**.



Motivation

- Threads are very useful in modern whenever a process hasmultiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired toallow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- In a web server - Multiple threads allow for multiple requests to be served simultaneously. A thread is created to service each request; meanwhile another thread listens for more client request.
- In a web browser – one thread is used to display the images and another threadis used to retrieve data from the network.

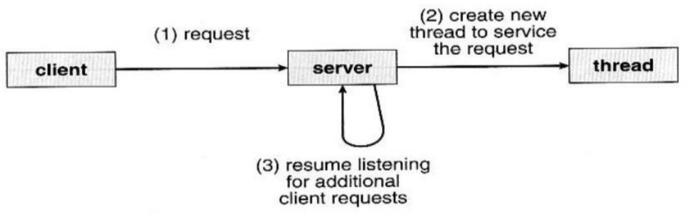

client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

Figure 4.2 Multithreaded server architecture.

Benefits

The four major benefits of multi-threading are:

1. **Responsiveness -** One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
   Multi threading allows a program to continue running even if part of itis blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
2. **Resource sharing -** By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. **Economy -** Creating and managing threads is much faster than performing the same tasks for processes. Context switching between threads takes less time.
4. **Scalability**, i.e. **Utilization of multiprocessor architectures** – Multithreading can be greatly utilized in a multiprocessor architecture. A single threaded process can make use of only one CPU, whereas the execution of a multi- threaded application may be split among the available processors. Multithreading on a multi-CPU machine increases concurrency. In a single processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time.

Multithreading Models

- There are two types of threads to be managed in a modern system: Userthreads and kernel threads.
- User threads are the threads that application programmers would put into theirprograms. They are supported above the kernel, without kernel support.
- Kernel threads are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple taskssimultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following models.

a) Many-To-One Model

In the many-to-one model, many user-level threads are all mapped onto asingle kernel thread.

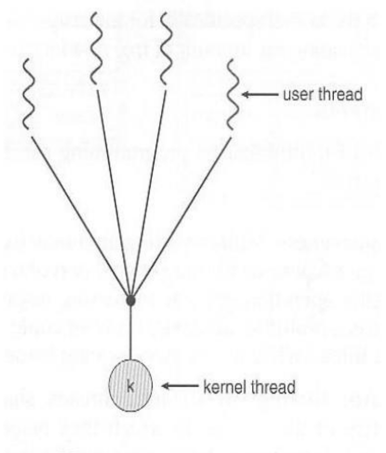


Figure 4.2 Many-to-one model.

- Thread management is handled by the thread library in user space, which is very efficient.
- If a blocking system call is made by one of the threads, then the entire process blocks. Thus blocking the other user threads from continuing the execution.
- Only one user thread can access the kernel at a time, as there is only one kernel thread. Thus the threads are unable to run in parallel on multiprocessors.
- Green threads of Solaris and GNU Portable Threads implement the many-to-one model.

b) One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- This model places a limit on the number of threads created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

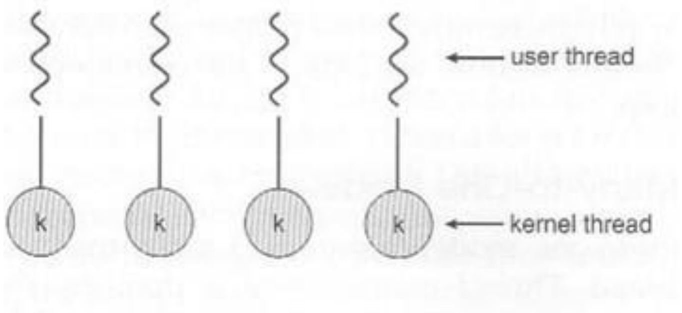


Figure 4.3 One-to-one model.

Many-To-Many Model

The many-to-many model multiplexes any number of user threads onto anequal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.

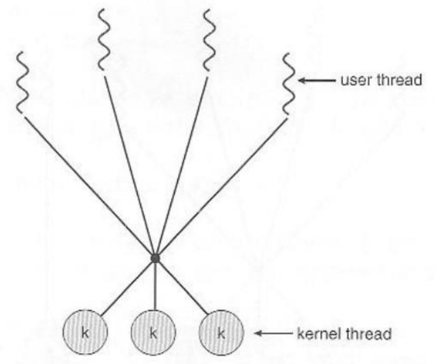


Figure 4.4 Many-to-many model.

- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.
- This model is also called as two-tier model.
- It is supported by operating system such as IRIX, HP-UX, and Tru64 UNIX.

Threading Issues

a) The fork( ) and exec( ) System Calls

The fork( ) system call is used to create a separate, duplicate process.

When a thread program calls fork( ),

- The new process can be a copy of the parent, with all the threads

- The new process is a copy of the single thread only (that invoked theprocess)

If the thread invokes the exec( ) system call, the program specified in the parameter to exec( ) will be executed by the thread created.

b) Signal Handling

A signal is used to notify a process that a particular event has occurred.
All signals follow same path-

1) A signal is generated by the occurrence of a particular event.
2) A generated signal is delivered to a process.
3) Once delivered, the signal must be handled.

A signal can be invoked in 2 ways : synchronous or asynchronous.
Synchronous signal – signal delivered to the same program. Eg – illegal memoryaccess, divide by zero error.
Asynchronous signal – signal is sent to another program. Eg – Ctrl C

In a single-threaded program, the signal is sent to the same thread. But, in multi- threaded environment, the signal is delivered in variety of ways, depending on the type of signal –

- Deliver the signal to the thread, to which the signal applies.
- Deliver the signal to every threads in the process.
- Deliver the signal to certain threads in the process.
- Deliver the signal to specific thread, which receive all the signals.

A signal can be handled by one of the two ways
–Default signal handler - signal is handled by OS.
-User-defined signal handler - User overwrites the OS handler.

c) Cancellation

Terminating the thread before it has completed its task is called thread cancellation. The thread to be cancelled is called **target thread**.
Example : Multiple threads required in loading a webpage is suddenlycancelled, if the browser window is closed.
Threads that are no longer needed may be cancelled in one of two ways:

1. **Asynchronous Cancellation -** cancels the thread immediately.
2. **Deferred Cancellation** – the target thread periodically check whetherit has to terminate, thus gives an opportunity to the thread, to terminate itself in an orderly fashion.

In this method, the operating system will reclaim all the resources before cancellation.

d) Thread-Local Storage

- Threads belonging to a process share the data of the process. In some circumstances, each thread might need its own copy of certain data. We will call such data **thread-local storage** (or **TLS**.)
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned

a unique identifier. To associate each thread with its unique identifier, we could use thread-local storage.

- Local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations. In some ways, TLS is similar to static data. The difference is that TLS data are unique to each thread.

e) Scheduler Activations

Scheduler Activation is the technique **used** for communication between the user-thread library and the kernel.

It works as follows:

— the kernel must inform an application about certain events. This procedureis known as an **upcall.**

— Upcalls are handled by the thread library with an **upcall handler,** andupcall handlers must run on a virtual processor.

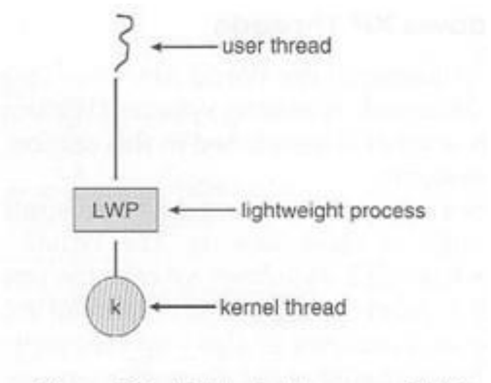


Figure 4.9 Lightweight process (LWP.)

<u>CPU SCHEDULING</u>

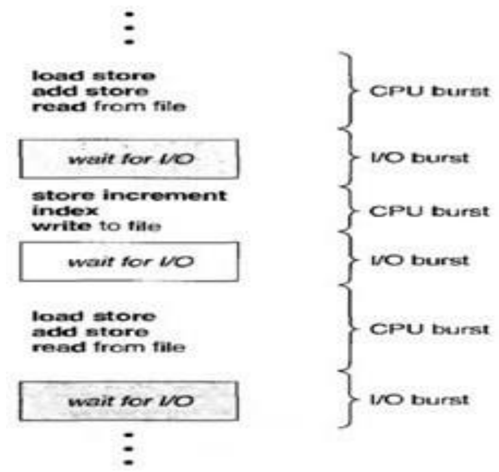
3.1 BASIC CONCEPTS

In a single-processor system, only one process can run at a time; other processes must wait until the CPU is free. The objective of multiprogramming is to have some process running at all times in processor, to maximize CPU utilization.

In multiprogramming, 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 one of the primary computer resources. Thus, its scheduling is central to operating-system design.

CPU-I/O Burst Cycle

Process execution consists of a cycle of CPU execution and I/O wait. The state of processunder execution is called **CPU burst** and the state of process under I/O request & its handling is called **I/O burst**. 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 requestto terminate execution as shown in the following figure:



CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes from 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 memorythat are ready to execute and allocates the CPU to that process.

A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply anunordered linked list. All the processes in the ready queue are lined up waiting for a chance torun on the CPU. The records in the queues are generally process control blocks (PCBs) of theprocesses.

Non - Preemptive 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.

Preemptive Scheduling – The process under execution, may be released from the CPU, in the middle of execution due to some inconsistent state of the process.

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:

- Switching context
- Switching to user mode
- 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

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU utilization** - The CPU must be kept as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 to 90percent .
- **Throughput** - If the CPU is busy executing processes, then work is done fast. One measure of work is the number of processes that are completed per time unit, called throughput.
- **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.
  Time spent waiting (to get into memory + ready queue + execution + I/O)
- **Waiting time -** The total amount of time the process spends waiting in the ready queue.
- **Response time -** The time taken from the submission of a request until the first responseis produced is called the response time. It is the time taken to start responding. In interactive system, response time is given criterion.

It is desirable to **maximize** CPU utilization and throughput and to **minimize** turnaroundtime, waiting time, and response time.

SCHEDULING ALGORITHMS

CPU scheduling deals with the problem of deciding which of the processes inthe ready queue is to be allocated the CPU.

1. **First-Come, First-Served Scheduling** Other names of this algorithm are:
   - First-In-First-Out (FIFO) •Run-to-Completion or Run-Until-Done

First-Come-First-Served algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. This algorithm is always nonpreemptive, once a process is assigned to CPU, it runs to completion.

Advantages :

- more predictable than other schemes since it offers time
- code for FCFS scheduling is simple to write and understand

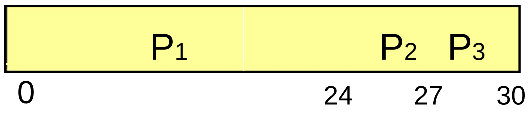
Disadvantages:

- Short jobs(process) may have to wait for long time
- Important jobs (with higher priority) have to wait
- cannot guarantee good response time
- average waiting time and turn around time is often quite long
- lower CPU and device utilization.

Example:-

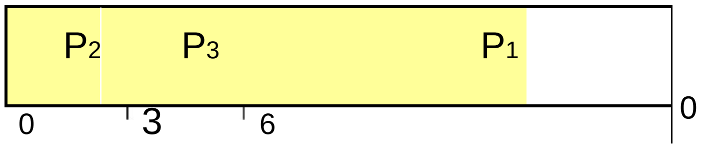
Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: *P1*, *P2* , *P3*
The Gantt Chart for the schedule is:



Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order *P2*,*P3*,*P1*
The Gantt chart for the schedule is:



Waiting time for *P1* = 6*; P2* = 0*; P3* = 3
Average waiting time: (6 + 0 + 3)/3 = 3 Much
better than previous case

Here, there is a ***Convoy effect***, as all the short processes wait for the completion of one big process. Resulting in lower CPU and device utilization.

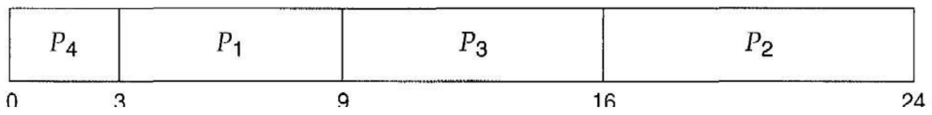
3.3.2 Shortest-Job-First Scheduling

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.

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

Process	Burst Time
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:



The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2,9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 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.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the nth CPU burst, and let t_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

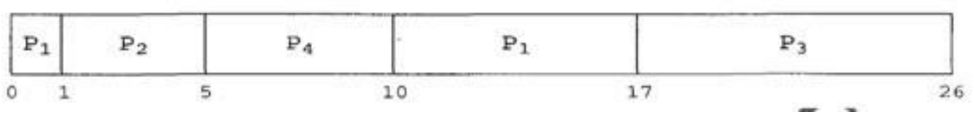
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than whatis left of the currently executing process. A preemptive SJF algorithm will preemptthe currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

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:

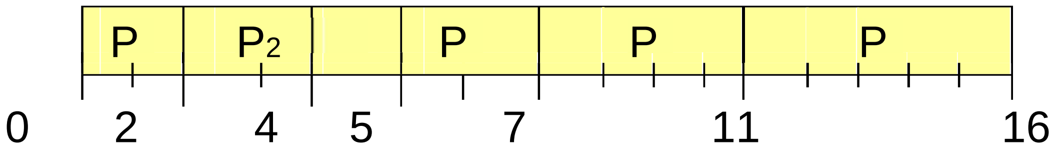


Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is ((10 -1) + (1-1) + (17 -2) + (5- 3))/4 = 26/4 = 6.5 milliseconds.

Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

->SJF (preemptive)

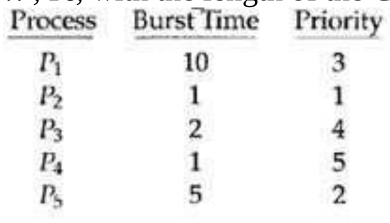


->Average waiting time = (9 + 1 + 0 +2)/4 = 3

3.3.3 Priority Scheduling

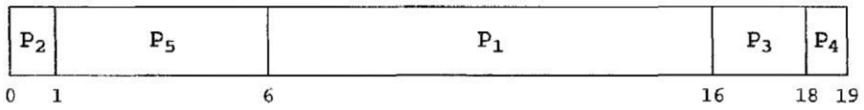
The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

As an example, consider the following set of processes, assumed to have arrivedat time 0, in the order P_1, P_2, … , P_5, 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.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low- priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher- priority processes can prevent a low-priority process from ever getting the CPU.

A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

3.3.4 Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated asa circular queue. The CPU scheduler goes around the ready queue, allocating the CPUto each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes.

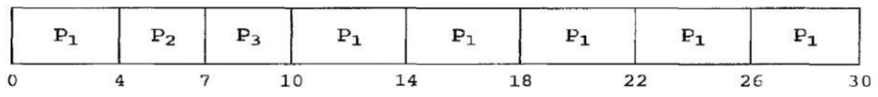
New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

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

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst givenin milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2.Since process P_2 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is



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

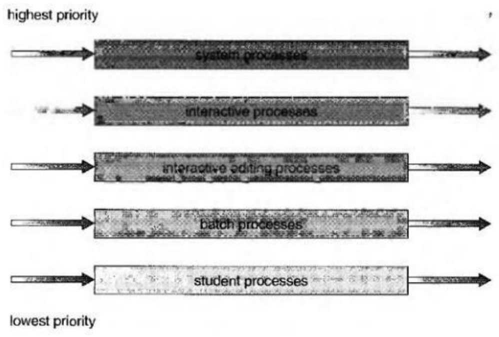
In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row . If a process's CPU burst exceeds 1 time quantum, that process is preempted arid is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n-1) x q time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

3.3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common divisionis made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.



In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foregroundqueue may have absolute priority over the background queue.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

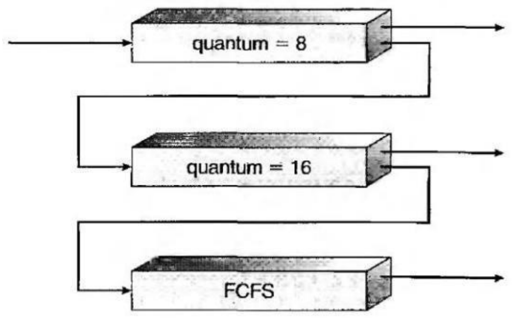
Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give toits processes on an FCFS basis.

3.3.6 Multilevel Feedback-Queue Scheduling

The multilevel feedback-queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. 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. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 (Figure). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.



A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burstof 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback-queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priorityqueue
- The method used to determine when to demote a process to a lower-priorityqueue
- The method used to determine which queue a process will enter when thatprocess needs service.

Example - The kernel triggers an upcall occurs when an application thread is about to block. The kernel makes an upcall to the thread library informing that a thread is about to block and also informs the specific ID of the thread.

The upcall handler handles this thread, by saving the state of the blocking threadand relinquishes the virtual processor on which the blocking thread is running.

The upcall handler then schedules another thread that is eligible to run on the virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. Thus assigns the thread to the available virtual processor.