MODULE 2

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.

The Process

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

- 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.

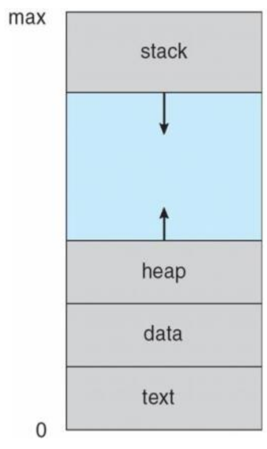


Figure: Process in memory.

PROCESS STATE

Q) Illustrate with a neat sketch, the process states and process control block.

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.

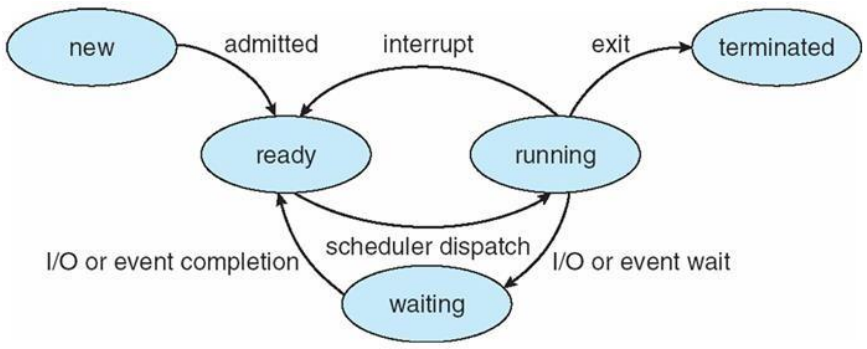


Figure: Diagram of process state

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 the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **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.

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

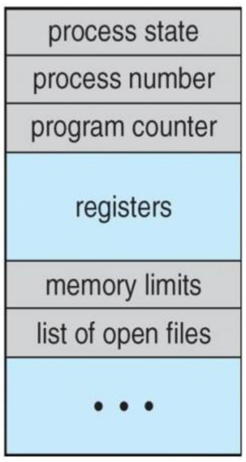


Figure: Process control block (PCB)

CPU Switch from Process to Process

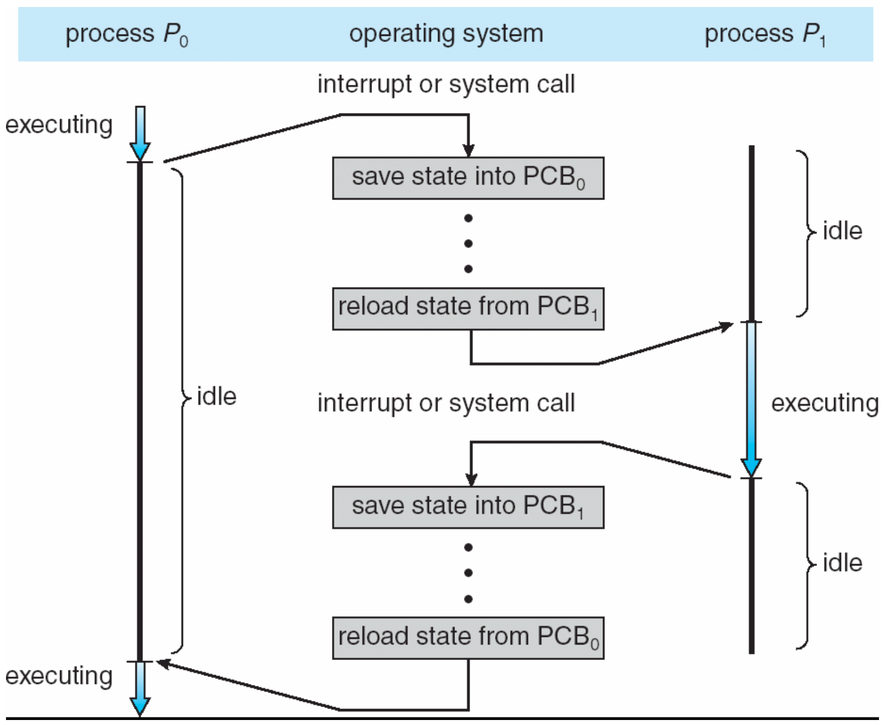


Figure: Diagram showing CPU switch from process to process.

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.

Ready Queue and Various I/O Device Queues

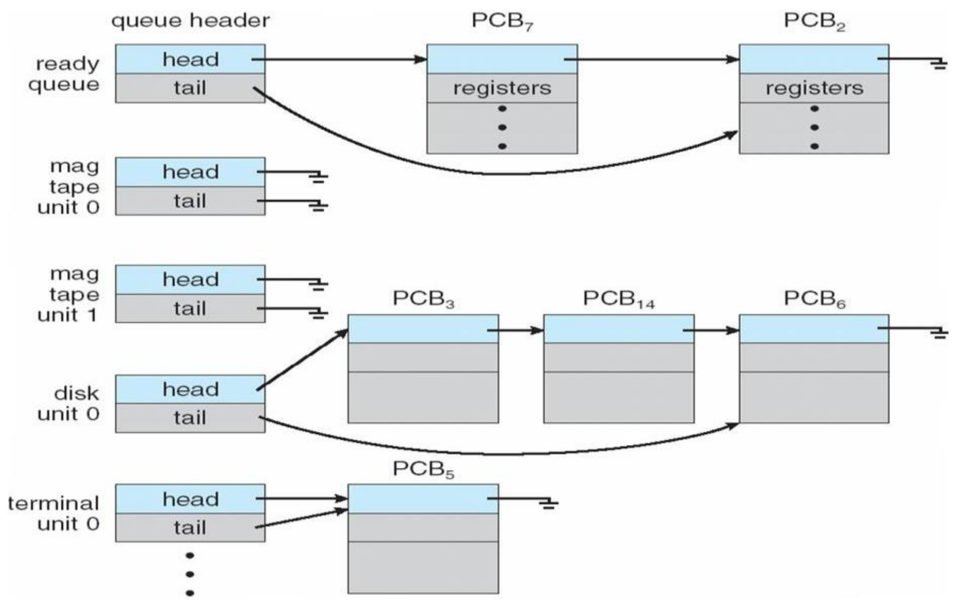


Figure: The ready queue and various I/O device queues

- 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.

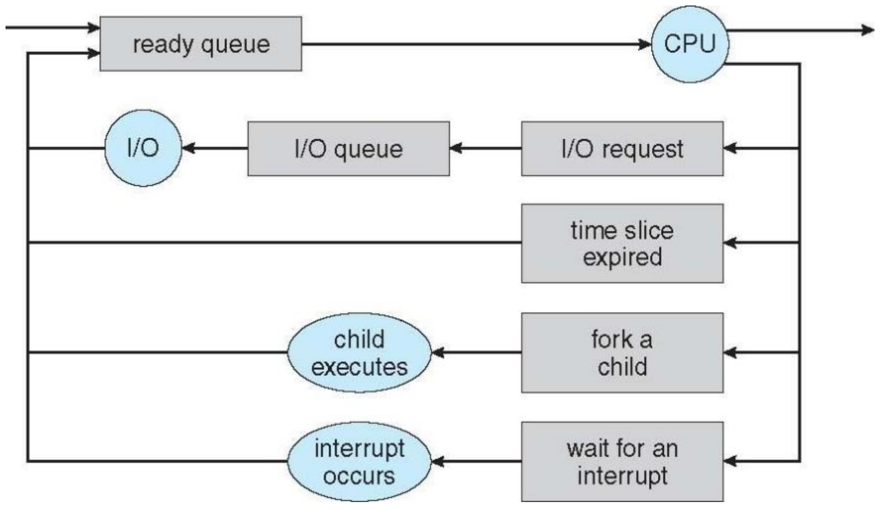


Figure: Queueing-diagram representation of process scheduling.

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 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.

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)

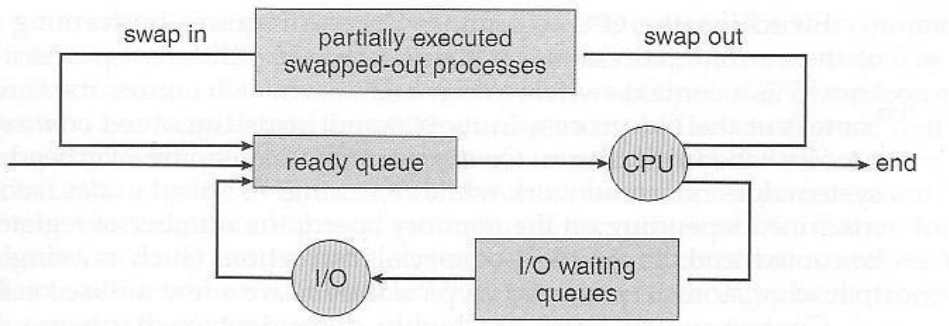


Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

Context 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.

OPERATIONS ON PROCESSES

Q) Demonstrate the operations of process creation and process termination in UNIX

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 a parent process for all user processes.

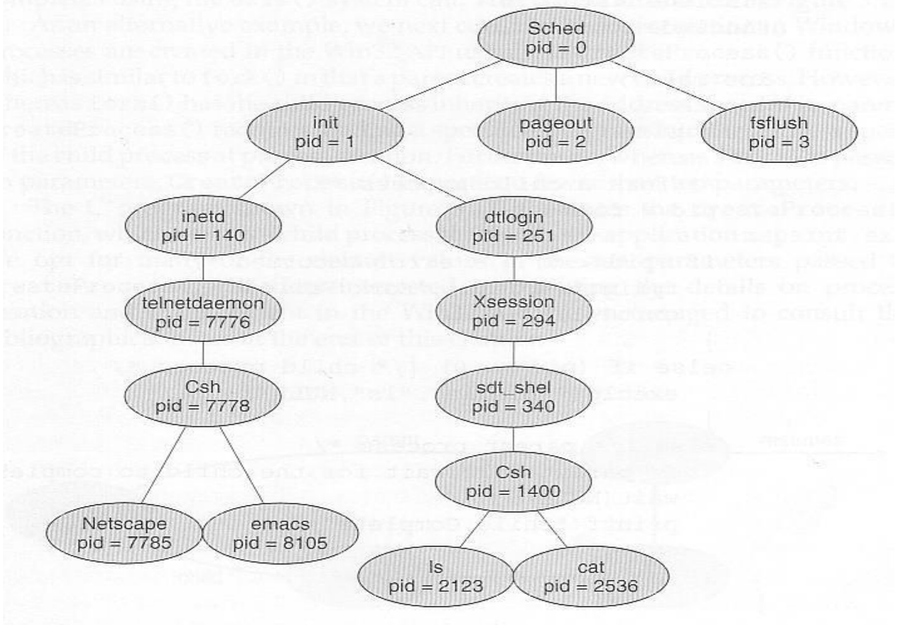


Figure 3.9 A tree of processes on a typical Solaris system.

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 behavior 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.

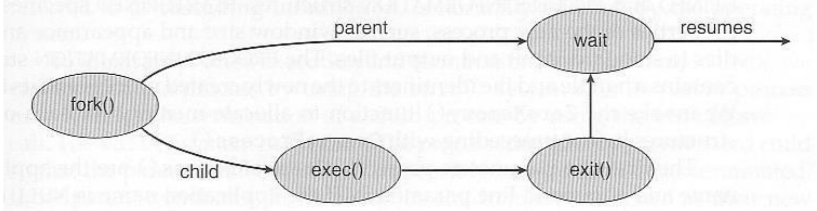


Figure 3.11 Process creation.

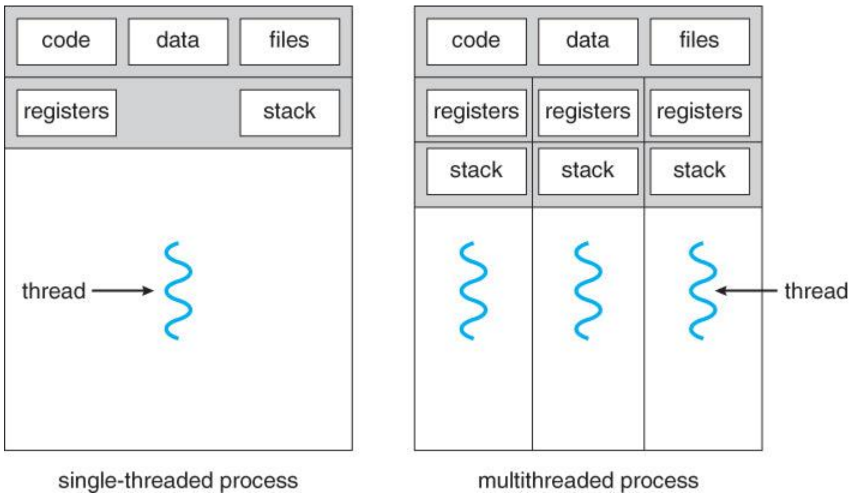
In windows the child process is created using the function **createprocess( )**. The createprocess( ) returns 1, if the child is created and returns 0, if the child is not created.

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**.

THREADS

A thread is a basic unit of CPU utilization, it comprises a thread ID, a program counter, a register set and a stack. It shares with other threads belonging to the same process its code section, data section and other operating system resources such as open files and signals.
A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.



Example 1: An application is implemented as a separate process with several threads of control. A web browser might have one thread display images or text, while another thread retrieves data from the

network.

Example 1: word processor may have a thread for displaying graphics, another thread responding to key strokes from the user, and third thread for performing spelling and grammar checking in background.

Benefits

- Responsiveness – it is an interactive application which allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- Resource sharing – processes may only share resources through techniques such as shared memory or message passing.
- Economy – allocating memory and resources for process creation is costly, because threads share the resources of the process to which they belong.
- Scalability – in multithreaded architecture, where threads may be running in parallel on different processors.

THREAD SCHEDULING

- On OSs, it is kernel-level threads but not processes that are being scheduled by the OS.
- User-level threads are managed by a thread library, and the kernel is unaware of them.
- To run on a CPU, user-level threads must be mapped to an associated kernel- level thread.

Contention Scope

- Two approaches:
  1. **Process-Contention scope**
     - 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.
     - Competition for the CPU takes place among threads belonging to the same process.
  2. **System-Contention scope**
     - The process of deciding which kernel thread to schedule on the CPU.
     - Competition for the CPU takes place among all threads in the system.
     - Systems using the one-to-one model schedule threads using only SCS.

Pthread Scheduling

- Pthread API that allows specifying either PCS or SCS during thread creation.
- Pthreads identifies the following contention scope values:
  1. PTHREAD_SCOPEJPROCESS schedules threads using PCS scheduling.
  2. PTHREAD-SCOPE_SYSTEM schedules threads using SCS scheduling.
- Pthread IPC provides following two functions for getting and setting the contention scope policy:
  1. pthread_attr_setscope(pthread_attr_t *attr, intscope)
  2. pthread_attr_getscope(pthread_attr_t *attr, int*scop)

INTERPROCESS COMMUNICATION

Q) What is interprocess communication? Explain types of IPC.

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 tasks 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.

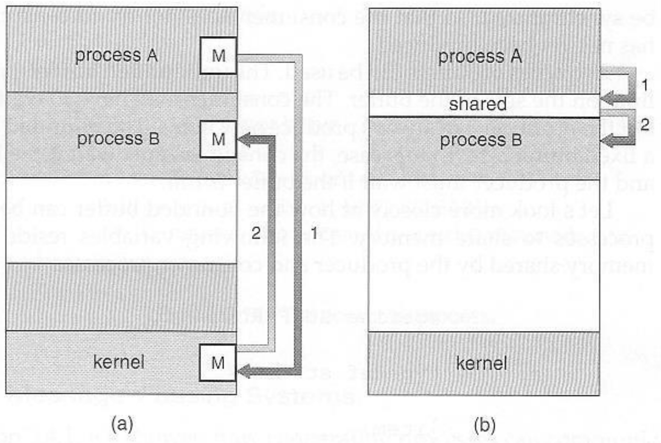


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

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 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.

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

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 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-
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication (Synchronization)
  - Automatic or explicit buffering.

1. Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

a) 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.

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'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.

b) 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.

2. 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.

3. 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.**

PROCESS SCHEDULING

Basic Concepts

- In a single-processor system,
  - Only one process may run at a time.
  - Other processes must wait until the CPU is rescheduled.
- Objective of multiprogramming:
  - To have some process running at all times, in order to maximize CPU utilization.

CPU-I/0 Burst Cycle

- Process execution consists of a cycle of
  - CPU execution and
  - I/O wait
- Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…
- Finally, a CPU burst ends with a request to terminate execution.
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.

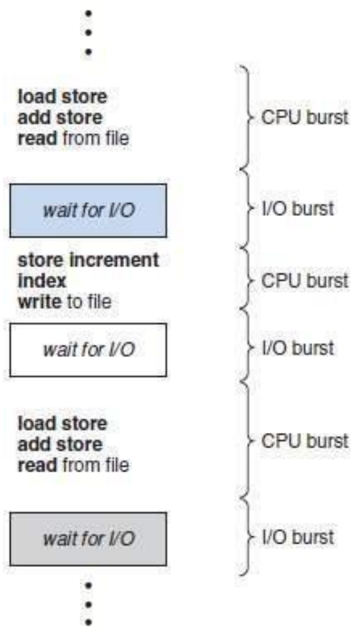


Fig Alternating sequence of CPU and I/O bursts

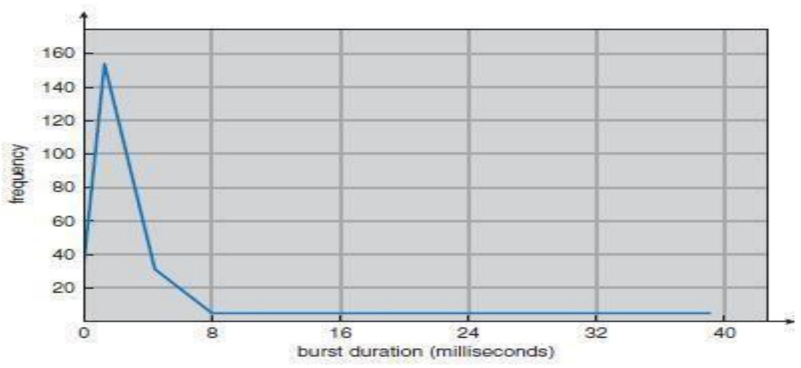


Fig: Histogram of CPU-burst durations

CPU Scheduler

- This scheduler
  - selects a waiting-process from the ready-queue and
  - allocates CPU to the waiting-process.
- The ready-queue could be a FIFO, priority queue, tree and list.
- The records in the queues are generally process control blocks (PCBs) of the processes.

CPU Scheduling

- Four situations under which CPU scheduling decisions take place:
  1. When a process switches from the running state to the waiting state. For ex; I/O request.
  2. When a process switches from the running state to the ready state. For ex: when an interrupt occurs.
  3. When a process switches from the waiting state to the ready state. For ex: completion of I/O.
  4. When a process terminates.
- Scheduling under 1 and 4 is non- preemptive. Scheduling under 2 and 3 is preemptive.

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

- This is driven by the idea of prioritized computation.
- Processes that are runnable may be temporarily suspended
- Disadvantages:
  1. Incurs a cost associated with access to shared-data.

2. Affects the design of the OS kernel.

<u>Dispatcher</u>

- It gives control of the CPU to the process selected by the short-term scheduler.
- The function involves:
    1. Switching context
    2. Switching to user mode&
    3. Jumping to the proper location in the user program to restart that program
- It should be as fast as possible, since it is invoked during every process switch.
- ***Dispatch latency*** means the time taken by the dispatcher to
    - stop one process and
    - start another running.

SCHEDULING CRITERIA

In choosing which algorithm to use in a particular situation, depends upon the properties of the various algorithms. Many criteria have been suggested for comparing CPU- scheduling algorithms. The criteria include the following:

1. **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).
2. **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.

3. **Turnaround time.** This is the important criterion which tells 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/0.
4. **Waiting time**: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0, 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.
5. **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.

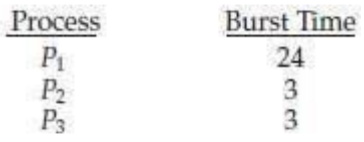
SCHEDULING ALGORITHMS

- CPU scheduling deals with the problem of deciding which of the processes in the ready-queue is to be allocated the CPU.
- Following are some scheduling algorithms:
  1. FCFS scheduling (First Come First Served)
  2. Round Robin scheduling
  3. SJF scheduling (Shortest Job First)
  4. SRT scheduling
  5. Priority scheduling
  6. Multilevel Queue scheduling and
  7. Multilevel Feedback Queue scheduling

FCFS Scheduling

- The process that requests the CPU first is allocated the CPU first.
- The implementation is easily done using a FIFO queue.
- Procedure:
  1. When a process enters the ready-queue, its PCB is linked onto the tail of the queue.
  2. When the CPU is free, the CPU is allocated to the process at the queue's head.
  3. The running process is then removed from the queue.

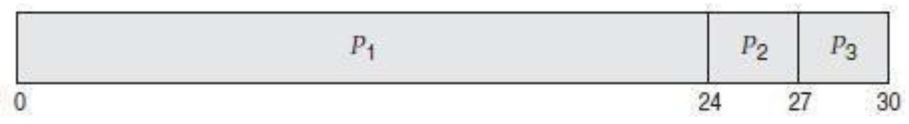
- Advantage:
  1. Code is simple to write & understand.
- Disadvantages:
  1. **Convoy effect:** All other processes wait for one big process to get off the CPU.
  2. Non-preemptive (a process keeps the CPU until it releases it).
  3. Not good for time-sharing systems.
  4. The average waiting time is generally not minimal.



Process	Burst Time
P_1	24
P_2	3
P_3	3

- Example: Suppose that the processes arrive in the order P1, P2,P3.

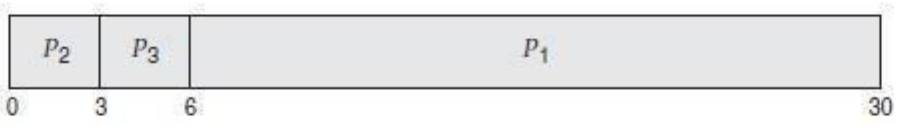
- The Gantt Chart for the schedule is as follows:



- Waiting time for *P1* = 0; *P2* = 24; *P3* =27

  Average waiting time: (0 + 24 + 27)/3 = 17ms

- Suppose that the processes arrive in the order P2, P3,P1.
- The Gantt chart for the schedule is as follows:



- Waiting time for P1 = 6;P2 = 0; P3 =3
- Average waiting time: (6 + 0 + 3)/3 = 3ms

SJF Scheduling

- The CPU is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.
- For long-term scheduling in a batch system, we can use the process time limit specified by the user, as the 'length'
- SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst
- Advantage:
  1. The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.
- Disadvantage:
  1. Determining the length of the next CPU burst.

- SJF algorithm may be either 1) non-preemptive or 2)preemptive.

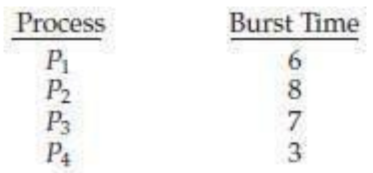
1. Non preemptiveSJF

The current process is allowed to finish its CPU burst.

2. PreemptiveSJF

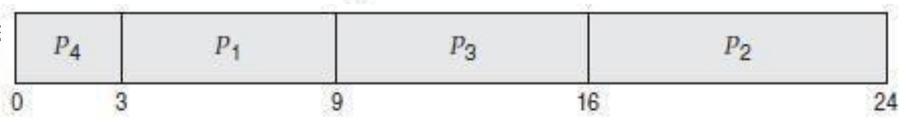
If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted. It is also known as **SRTF** scheduling (Shortest-Remaining-Time-First).

- Example (for non-preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.



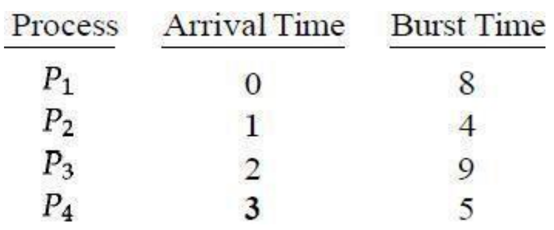
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- For non-pre



- Waiting time for P1 = 3; P2 = 16; P3 = 9; 4=0 Average waiting time: (3 + 16 + 9 + 0)/4= 7

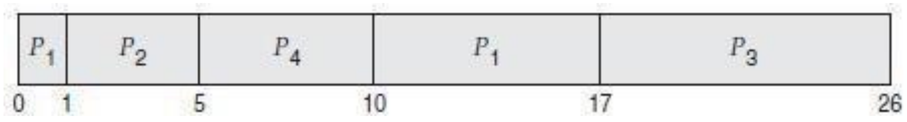
preemptive SJF/SRTF: Consider the following set of processes, with the length



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

of the CPU- burst time given in milliseconds.

- For preemptive SJF, the Gantt Chart is as follows:



- The average waiting time is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 =6.5.

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.
  1. ***Internally-defined*** priorities.
     - Use some measurable quantity to compute the priority of a process.
     - For example: time limits, memory requirements, no. f open files.
  2. ***Externally-defined*** priorities.
     - Set by criteria that are external to the OS For example:
     - importance of the process, political factors
- Priority scheduling can be either preemptive or non-preemptive.

  1. Preemptive

  The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process.

  2. Non Preemptive

  The new process is put at the head of the ready-queue

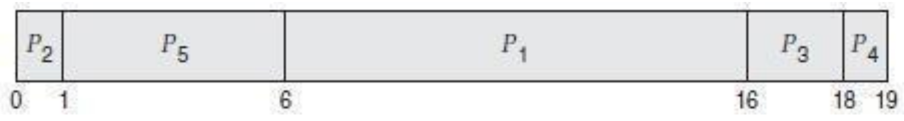
- Advantage:
  - Higher priority processes can be executed first.
- Disadvantage:
  - Indefinite blocking, where low-priority processes are left waiting indefinitely for CPU. Solution: ***Aging*** is a technique of increasing

- priority of processes that wait in system for a long time.

• Example: Consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., P5, with the length of the CPU-burst time 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

• The Gantt chart for the schedule is as follows:



• The average waiting time is 8.2milliseconds.

Round Robin 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

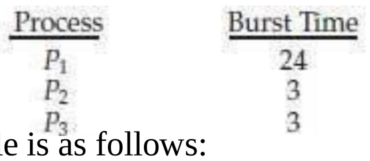
• CPU scheduler
  1. Picks the first process from the ready-queue.
  2. Sets a timer to interrupt after 1 time quantum and
  3. Dispatches the process.

• 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.

• Advantage:
  - Higher average turnaround than SJF.

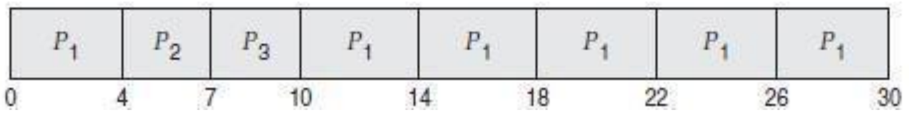
• Disadvantage:
  - Better response time than SJF.

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



Process	Burst Time
P_1	24
P_2	3
P_3	3

• The Gantt chart for the schedule is as follows:



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

• *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 l/n the speed of the real processor.

• 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.

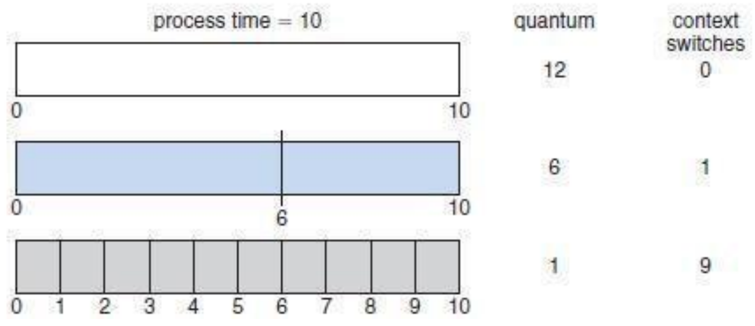
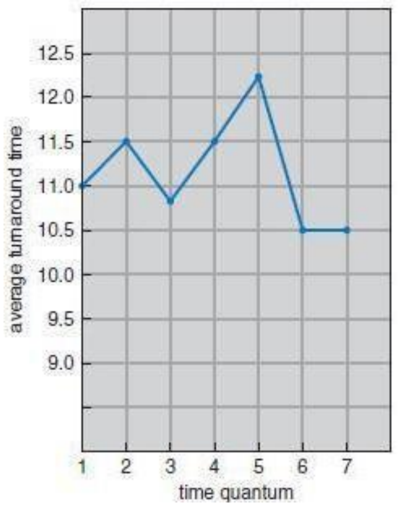


Fig: How a smaller time quantum increases context switches



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Fig: How turnaround time varies with the time quantum

Multilevel Queue Scheduling

- 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 (Figure2.19).
- The processes are permanently assigned to one queue based on some property like
    - 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.

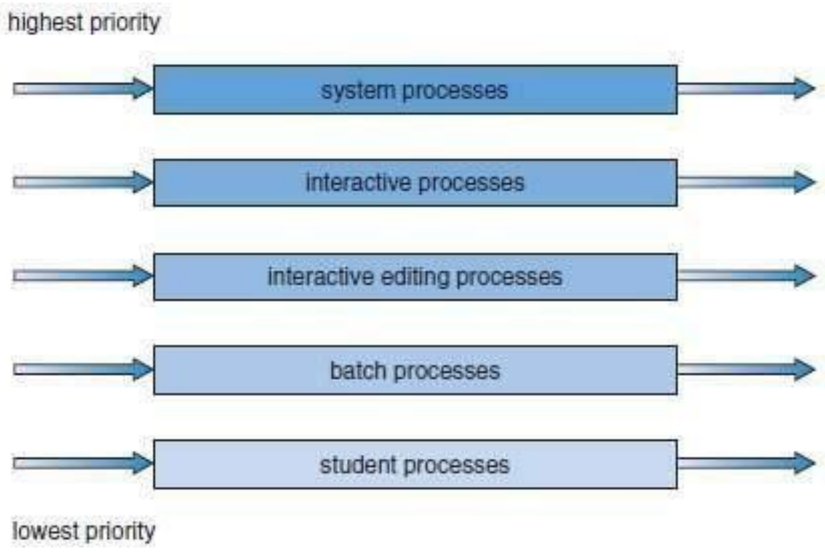


Fig Multilevel queue scheduling

- 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 Scheduling

- A process may move between queues
- The basic idea: Separate processes according to the features of their CPU bursts. For example
  1. 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.
  2. 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.

.

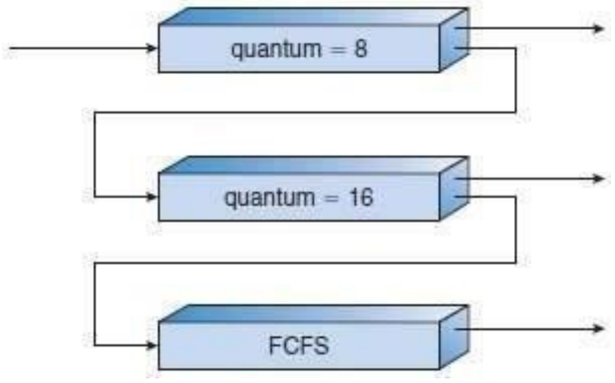


Figure 2.20 Multilevel feedback queues

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

1. The number of queues.
2. The scheduling algorithm for each queue.
3. The method used to determine when to upgrade a process to a higher priority queue.
4. The method used to determine when to demote a process to a lower priority queue.
5. The method used to determine which queue a process will enter when that process needs service

SYNCHRONIZATION

To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.

To present both software and hardware solutions of the critical-section problem.

To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity.

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
             while (count == BUFFER_SIZE)
                   ; // do nothing
               buffer [in] = nextProduced;
               in = (in + 1) % BUFFER_SIZE;
               count++;
}
```

Consumer

```
   while (true)  {
                while (count == 0)
                 ; // do nothing
                 nextConsumed =  buffer[out];
                  out = (out + 1) % BUFFER_SIZE;
                        count--;
                    /*  consume the item in nextConsumed
        }
```

Race condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with “count = 5” initially:

```
    S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }
S5: consumer execute count = register2   {count = 4}
```

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**

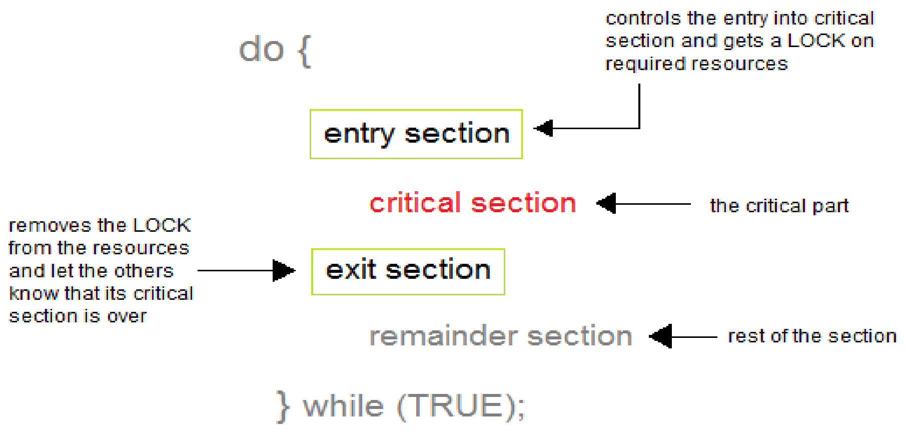
THE CRITICAL SECTION PROBLEM

Consider a system consisting of n processes {P_0, P_1, P_2……P_{N-1}}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating table, writing a file and so on.

The important feature of the system, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

- The critical section problem is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section

The general structure of process P_i

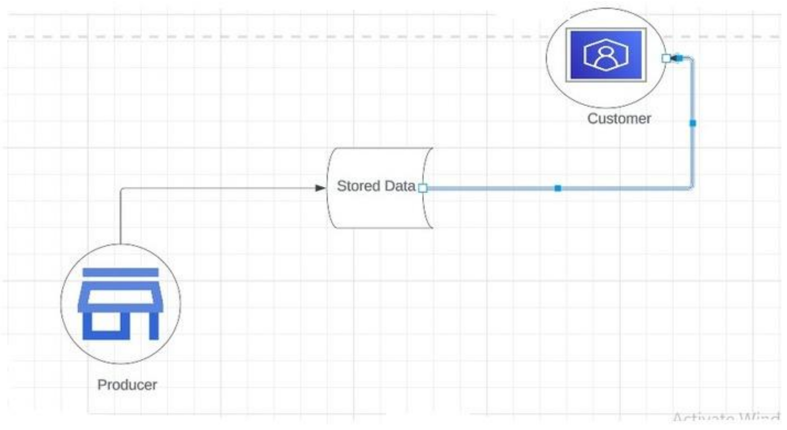

do {
entry section
controls the entry into critical section and gets a LOCK on required resources
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

A solution to the critical section problem must satisfy the following requirements

1. Mutual exclusive: if process p_i is executing in its critical section, then no other processes can be executing in their section.
2. Progress: if no process is executing in its critical section and some processes wish to enter their sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next and this selection cannot be postponed indefinitely.
3. Bounded waiting: there exist a bound or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

MUTEX LOCKS

In multitasking programming, mutex locks, also referred to as mutual exclusion locks, are synchronization basic functions used to prevent simultaneous possession of resources that are shared by numerous threads or procedures. The word "mutex" means "mutual exclusion."



A mutex lock makes it possible to implement mutual exclusion by limiting the number of threads or processes that can simultaneously acquire the lock. A single thread or procedure has to first try to obtain the mutex lock for something that is shared before it can access it. The seeking string or procedure gets halted and placed in a state of waiting as long as the encryption key turns into accessible if it is being organized by a different thread or process. The thread or process is able to use the resource that has been shared after acquiring the lock. When finished, it introduces the lock so that different threads or processes can take possession of it.

Components of Mutex Locks

Mutex Variable − A mutex variable is used to represent the lock. It is a data structure that maintains the state of the lock and allows threads or processes to acquire and release it.

Lock Acquisition − Threads or processes can attempt to acquire the lock by requesting it. If the lock is available, the requesting thread or process gains ownership of the lock. Otherwise, it enters a waiting state until the lock becomes available.

Lock Release − Once a thread or process has finished using the shared resource, it releases the lock, allowing other threads or processes to acquire it.

SEMAPHORES

A semaphore S is an integer variable that apart from initialization, is accessed only through two standard atomic operations wait() and signal().

The wait() operation was originally termed P to test. Signal() operation was originally termed V to increment.

```
Wait(S) {
        While S<=0
        S--;
        }
```

For signal

```
Signal(S) {
          S++;
            }
```

- The value of a counting semaphore can range over an unrestricted domain.
- The value of a binary semaphore can range only between 0 and 1.
- Binary semaphore is also known as mutex locks.

Implementation

The main disadvantage of the semaphore definition given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

```
do {
  wait (mutex);
   //critical section
  Signal(mutex
    //remainder section
    }while(true);
```

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
- Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
    - value (of type integer)
    - pointer to next record in the list
- Two operations:
    - block – place the process invoking the operation on the appropriate waiting queue.
    - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

<u>Implementation of wait:</u>

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

<u>Implementation of signal:</u>

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

CLASSIC PROBLEMS OF SYNCHRONIZATION

THE BOUNDED BUFFER PROBLEM

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized the value as 1. The semaphore empty is initialized to value n; the semaphore full is initialized to the value 0.

The structure of the producer process

```
do {
        //   produce an item in next p
        wait (empty);
        wait (mutex);
        // add the item to the buffer
        signal (mutex);
        signal (full);
    } while (TRUE);
```

The structure of the consumer process

```
do {
         wait (full);
        wait (mutex);
         // remove an item from buffer to next c
        signal (mutex);
        signal (empty);

  // consume the item in next c
        } while (TRUE);
```

THE READERS- WRITERS PROBLEM

A data set is shared among a number of concurrent processes

- Readers – only read the data set; they do not perform any updates
- Writers – can both read and write

Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time

This problem has various variation:

- It requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for others readers to finish simply because a writer is waiting.
- It requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In first case, writer may starve and in second case reader may starve.

Shared Data

- Data set
- Semaphore mutex initialized to 1

- Semaphore writes initialized to 1
- Integer read count initialized to 0

<u>The structure of a writer process</u>

```
 do {
        wait (wrt) ;

             //    writing is performed
        signal (wrt) ;
} while (TRUE);
```

<u>The structure of a reader process</u>

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                        wait (wrt) ;
        signal (mutex)

              // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
        signal (wrt) ;
        signal (mutex) ;
 } while (TRUE);
```

It provides reader- writer lock on some systems. Acquiring reader writer lock it requires specifying the mode of lock: either read or write access.

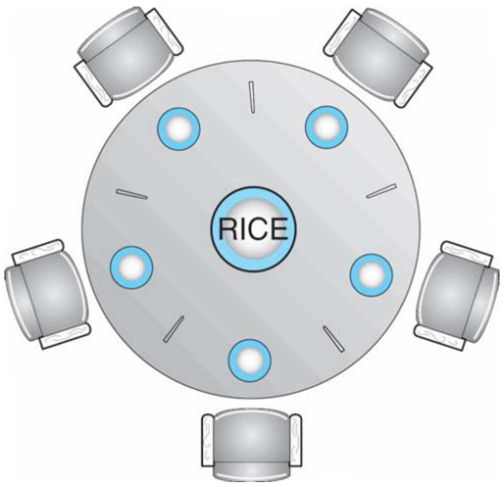
- When a process wishes only to read shared data, it requests the reader-writer lock in read mode.
- When a process wishes to modify the shared data must request the lock in write mode.

THE DINING- PHILOSOPHERS PROBLEM

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosophers. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

- when a philosopher thinks, he does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the chopsticks that are closest to him.
- A philosopher may pick up only one chopstick at a time.
- He cannot pick the stick which is already in the hand of their neighbor.
- When a hungry philosopher has both her chopsticks at the same time, he eats without releasing her chopsticks.
- When he finishes eating, he puts down both of the chopstick and starts thinking again.

- This problem is large class of concurrency control problem.
- It is simplest need to allocate several resources among several processes in a deadlock free and starvation free manner.



Shared data

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

Simple solution is to represent each chopstick with a semaphore.

- A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore.
- He releases his chopstick by executing the signal () operation on the appropriate semaphore.

```
Semaphore chopstick [5];
```

Where, all the elements of chopstick are initialized to 1.

<u>The structure of Philosopher *i*:</u>

```
do  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

          //  eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

         //  think
        } while (TRUE);
```

Several possible remedies to deadlock problem:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
- Use an asymmetric solution, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.