UNIT II: Process Management

CHAPTER 2: Process Management

Syllabus

Processes - Process Concept - Process Scheduling - Operations on Processes - Inter-process Communication; CPU Scheduling - Scheduling criteria - Scheduling algorithms: Threads - Multithread Models - Threading issues; Process Synchronization - The Critical-Section problem - Synchronization hardware - Semaphores - Mutex - Classical problems of synchronization - Monitors; Deadlock - Methods for handling deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from deadlock.

Process Concept

• Process term is used in MULTICS system in the 1960. Process is an asynchronous activity.

• Process is an active entity that requires a set of resources, including a processor, program counter, registers to perform its function. Multiple processes may be associated with one program.

• Process execution must progress in sequential order. It also called task. Task is a single instance of an executable program. Fig. 2.1.1 shows memory layout for a process.

• Each process has its own address space. Address space is divided into two regions : 1. Text region 2. Data region 3. Stack region

• **Text region:** It stores the code that that the processor executes.

• **Data region:** It stores variables and dynamically allocated memory that the process uses during execution.

• **Stack region:** It stores instructions and local variables for active procedure calls.

• Process is a dynamic entity that executes a program on a Higher particular set of data using resources allocated by the operating system.

• A process can run to completion only when all requested hardware and software resources have been allocated to the process.

• Program is a passive entity. Program becomes process when executable file loaded intomemory. A process may be independent of other processes in the system.

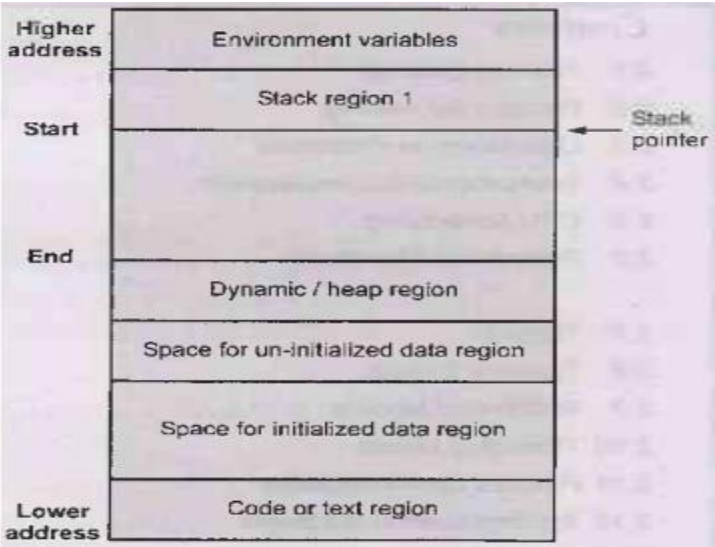


Fig. 2.1.1 Memory layout for a process

• After booting operating system, it creates foreground processes and background processes.

• Foreground processes interact with user and background processes is used by system. Background processes are related e-mail, web pages, news and printing.

• A process is used as a fundamental unit for resource allocation in operating system. A process normally has its own private memory area in which it runs.

• Text region contains executable instructions. It is be placed below the heap or Bold forines stack.

• An initialized data region contains the static and global variables that are initialized by the user.

• Un-initialized data region contains all static and global variables that are initialized by kernel to zero.

• Heap is used for dynamic memory allocation and stack contains program counter.

Difference between Process and Program

Sr. No.	Process	Program
1.	Process is active entity.	Program is passive entity.
2.	Process is a sequence of instruction executions.	Program contains the instructions.
3.	Process exists in a limited span of time.	A program exists at single place and continues to exist.
4.	Process is a dynamic entity.	Program is a static entity.

Process Control Block

• Operating system keeps an internal data structure to describe each process it manages.

• When OS creates process, it creates this process descriptor. In some operating system, it calls Process Control Block (PCB).

• Fig. 2.1.2 shows process control block.

• Process control block will change according to the operating system. PCB is also called task control block.

• The PCB is identified by an integer Process ID (PID). When a process is running, its hardware state is inside the CPU.

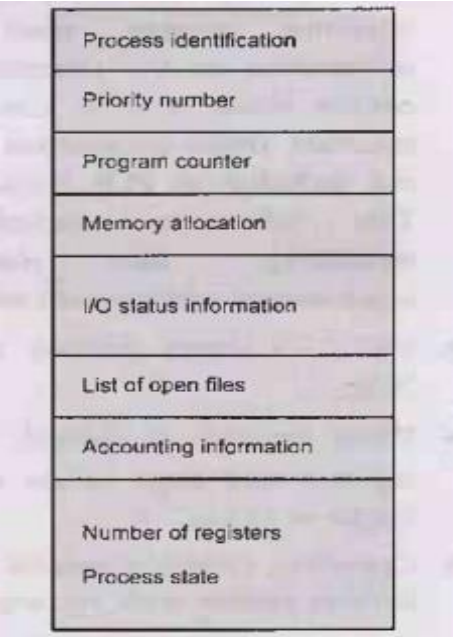


Fig. 2.1.2 Process control block

• When the OS stops running a process, it saves the register's values in the PCB.

• When a process is created by operating system, it allocates a PCB for it. OS initializes PCB and puts PCB on the correct queue. Following information is stored in process control block.

1. Process identification: Each process is uniquely identified by the user's identification and a pointer connecting it to its descriptor.

2. Priority number: Operating system allocates the priority number to each process. According to the priority number it allocates the resources.

3. Program counter: The PC indicates the address of the next instruction to be executed for this current process.

4. Memory allocation: It contains the value of the base registers, limit registers and the page tables depending on the memory system used by the operating system.

5. I/O status information: It maintains information about the open files, list of I/O devices allocated to the process etc.

6. List of open files: Process uses number of files for operation. Operating system keeps track of all opened file by this process.

7. Process state: Process may be in any one of the state: new, ready, running, and waiting, terminate.

• When process changes the state, the operating system must update information in the process's process control block. Process control block maintain other information which is not included in PCB block diagram. es This information includes CPU190 scheduling, file management, or input-output management information.

• Fig. 2.1.3 shows process table with PCB.

• When process is created, hardware registers and flags values are set by loader or linker.

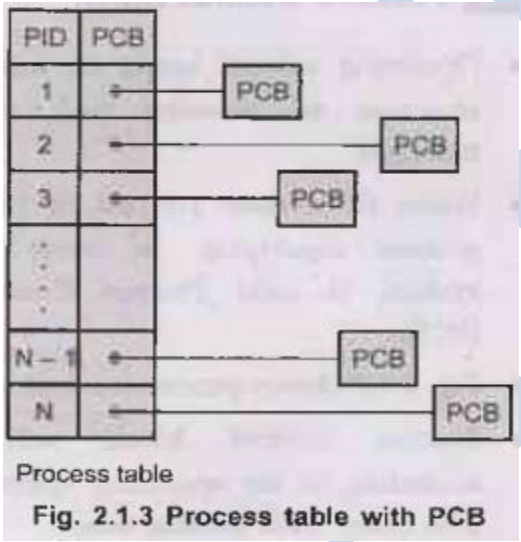


Fig. 2.1.3 Process table with PCB

• Operating system maintains pointers to each process's PCB in a per user process table or system wide process table. This information is used to access PCB quickly.

Process States

• Each process has an execution state which indicates what process is currently doing. The process descriptor is the basic data structure used to represent the specific state for each process.

• Fig. 2.1.4 shows a process state diagram. A state diagram is composed of a set of states and transitions between states.

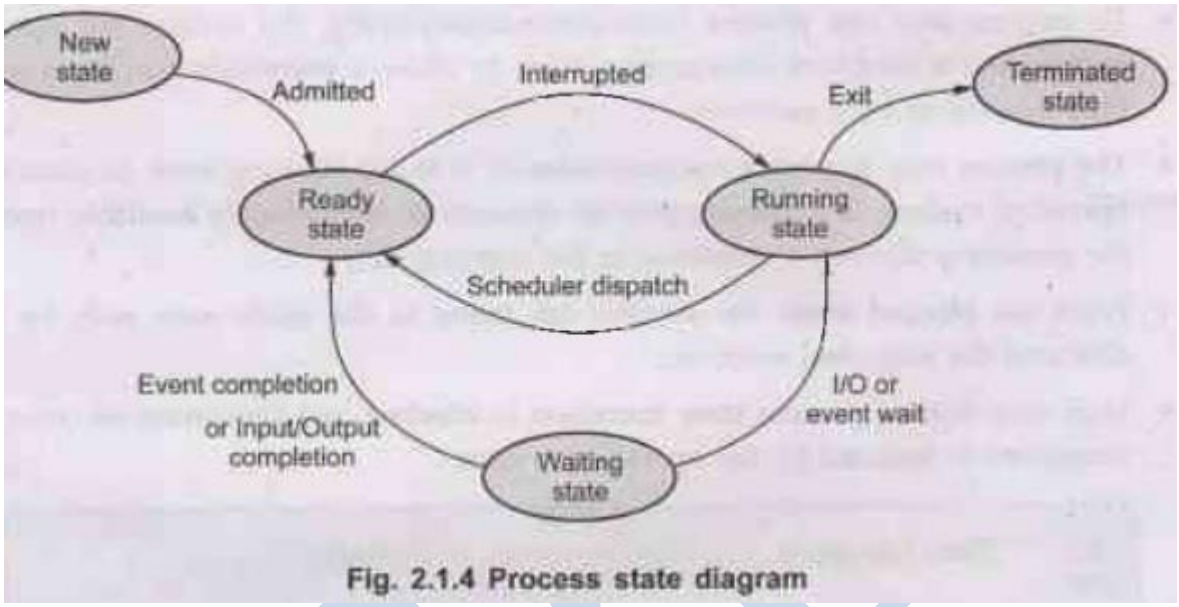


Fig. 2.1.4 Process state diagram

• State diagram is used by process manager to determine the type of service to provide to the process.

• The process states are as follows:

1. New 2. Ready 3. Running 4. Waiting 5. Terminated

• New: Operating system creates new process by using fork() system call. These process are newly created process and resources are not allocated.

• Ready: The process is competing for the CPU. Process reaches to the head of the list (queue).

• Running The process that is currently being executed. Operating system allocates all the hardware and software resources to the process for execution.

• Waiting A process is waiting until some event occurs such as the completion of an input-output operation.

• Terminated: A process completes its operations and releases all resources.

• Operating system maintains a ready list of ready process and a blocked list of blocked processes. The ready list is maintained in priority order and blocked list is typically unordered.

• The act of assigning a processor to the first process on the ready list is called dispatching and is performed by a system entity called the dispatcher.

• When process is in new state, the program remains in the secondary storage. Here process itself is not in the main memory and space is not allocated.

• The process exists a system because of two reasons :

1. Process is terminated when it completes its operation.

2. Process aborts due to an unrecoverable error.

• To prevent any one process from continuously using the system, the operating system sets a hardware interrupting clock to allow a process to run for a specific time interval or time quantum.

• The process may request a resource when it is in the running state. In most of the operating system, if a running process requests an immediately available resources the process is allowed to continue in the running state.

• From the blocked state, the process can move to the ready state only by being allocated the requested resource.

• User only initiate process state transition is blocked and remaining all other state transitions is initiated by the operating system.

Sr. No.	State transition	Remarks
1.	Ready to running	Process is dispatched.
2.	Running to ready	Process time slice expires.
3.	Running to blocked	When a process blocks.
4.	Blocked to ready	When the event for which it has been waiting occurs.
5.	Ready to exit	This is possible when parent process may terminate child process at any time.
6.	Running to exit	When currently running process completes its operation then OS terminates the process.

Suspended Processes

• Each process to be executed must be loaded into main memory. In uniprocessor system, processor remains idle because of execution speed mismatch of processor I/O devices. An I/O device is slower than processor.

• Main memory accommodates multiple processes and the processor select next I process when one process is blocked. So even with multiprogramming operating system, a processor could be idle most of the time. Main memory size is increased for accommodating more processes but cost will increase.

• **Suspended state:** when all of the processes in main memory are in the blocked state, the operating system can suspend one process by putting it in the suspend state and transferring it to disk. The space that is freed in main memory can then goo be used to bring in another process.

• Characteristics of a suspended process :

1. The process cannot execute immediately.

2. The process may or may not be waiting for an event.

3. The process was placed in a suspended state by a parent process, or itself or by operating system.

4. The process may not be removed from suspended state until its turn comes.

APEC

Process Scheduling

• CPU utilization is maximizing by using multiprogramming concept. Processor is not idle, it is executing a process. Processor scheduler selects one process for execution from the ready queue.

Scheduling Queue

• Scheduling queue is processes or input-output devices. When the user process enters into the system, they put into the job queue. Job queue consists of all processes of the system.

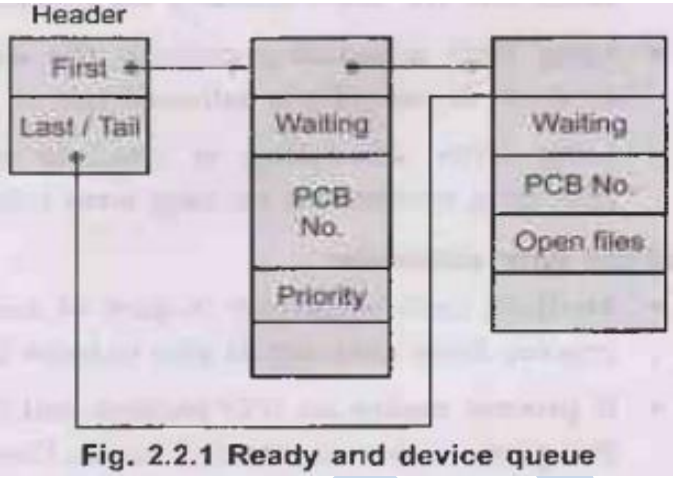


Fig. 2.2.1 Ready and device queue

• Operating system maintains different types of queues for different purposes. The queue are ready queue, device queue etc. shows ready queue and device.

• **Ready queue:** The processes which are ready and waiting to execute are kept in the ready queue. Ready queue is stored in main memory. Linked list is used for representing ready queue. Pointer field of PCB is used for this.

• **Device queue:** Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has maintained its own device queue.

• Process scheduling is represented by queueing diagram. Newly created process is kept into the ready queue and waits for processor. When CPU select the process for executing, following things happens :

1. Process may require I/O device to perform operation. So it is put into the I/O queue.

2. Process may create child process and wait for child process termination.

3. Because of interrupt, process pre-empted from CPU and put into the ready queue.

Schedulers

• Schedulers are used to handles process scheduling. It is one type of system software and selects the jobs from the system and decide which process to run. Schedulers are of three types -

1. Long term scheduler 2. Short term scheduler 3. Medium term scheduler

• Fig. 2.2.2 shows queueing diagram for process scheduling. (See Fig. 2.2.2 on next page.)

Long term scheduler

• Long term scheduler is also called job scheduler. It determines which process are admitted to the system for processing. Processes are selected from the queue and loads into the main memory for execution.

• Long term scheduling controls the degree of multiprogramming in multitasking systems. It provides a balanced mix of jobs, such as I/O bound and CPU bound. •Long term scheduling is used in real time operating system. Time sharing operating system has no long term scheduler.

Medium term scheduler

•Medium term scheduler is part of swapping function. Sometimes it removes the process from memory. It also reduces the degree of multiprogramming.

• If process makes an I/O request and it is in memory then operating system takes this process into suspended states. Once the process becomes suspended, it cannot make any progress towards completion.

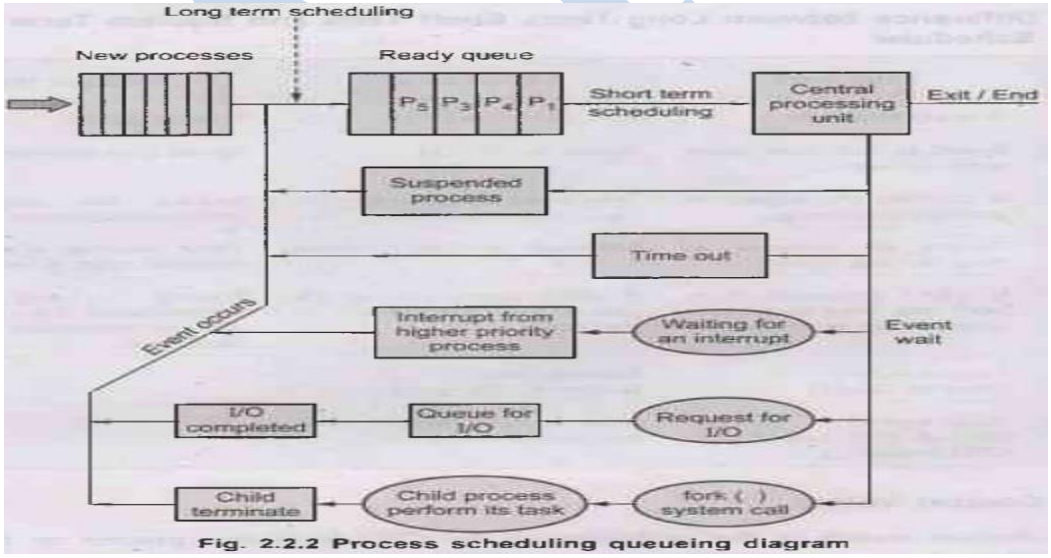


Fig. 2.2.2 Process scheduling queueing diagram

• In this situation, the process is removed from memory and makes free space for other process.

• The suspended process is stored in the secondary storage device i.e. hard disk. This process is called swapping.

Short term scheduler

• Short term scheduler is also called CPU scheduler. It selects the process from queue which are ready to execute and allocate the CPU for execution.

• Short term scheduler is faster that long term scheduler. This scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers.

• A scheduling decision will at a minimum have to be made after every time slice, and these are very short.

• It is also known as dispatcher.

Difference between Long Term, Short Term and Medium Term Scheduler

Sr. No.	Long term	Short term	Medium term
1.	It is job scheduler.	It is CPU scheduler.	It is swapping.
2.	Speed is less than short term scheduler.	Speed is very fast.	Speed is in between both.
3.	It controls the degree of multiprogramming.	Less control over degree of multiprogramming.	Reduce the degree of multiprogramming.
4.	Absent or minimal in time sharing system.	Minimal in time sharing system.	Time sharing system use medium term scheduler.
5.	It select processes from pool and load them into memory for execution.	It select from among the processes that are ready to execute.	Process can be reintroduced into memory and its execution can be continued.
6.	Process state is (New to Ready).	Process state is (Ready to Running)	–
7.	Select a good process, mix of I/O bound and CPU bound.	Select a new process for a CPU quite frequently.	–

Context Switch

• A context switch is the switching of the CPU from one process or thread to another. A context is the contents of a CPU's registers and program counter at any point in time.

• Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.

• A context switch can mean a register context switch, a task context switch, a mot thread context switch or a process context switch.

• A register is a small amount of very fast memory inside of a CPU that is used to speed the execution of computer programs by providing quick access to commonly used values.

• A program counter is a specialized register that indicates the position of the CPU in its instruction sequence and which holds either the address of the instruction being executed or the address of the next instruction to be executed, depending on the specific system.

• Context switching can be described in more detail as the Kernel performing the following activities with regard to processes (including threads) on the CPU:

1. Suspending the progression of one process and storing the CPU's state (i.e., the context) for that process somewhere in memory.

2. Retrieving the context of the next process from memory and restoring it in the CPU's registers and

3. Returning to the location indicated by the program counter in order to resume to the process.

• Context switches can occur only in Kernel mode (system mode). Kernel mode is a privileged mode of the CPU in which only the Kernel runs and which provides access to all memory locations and all other system resources.

• Other programs, including applications, initially operate in user mode, but they can run portions of the Kernel code via system calls. Software context switching an can be used on all CPUs and can be used to save and reload only the state that needs to be changed.

• To use the hardware context switch you need to tell the CPU where to save the existing CPU state and where to load the new CPU state from. The CPU state is always stored in a special data structure called a TSS (Task State Segment).

• Context switch times are highly dependent on hardware support. Context switching represents a substantial cost to the system in terms of CPU time and it can be the most costly operation on an operating system.

• There are three situations where a context switch needs to occur. They are multitasking, interrupt handling, user and Kernel mode switching.

Operations on Processes

• Following operations are performed on the process :

1. Process creation

2. Process termination

• A mechanism for process creation and termination via system calls. Programmers usually access these system calls via application interface (API) / library.

Process Creation

• Operating system creates the process in following situations:

1. Starting of new batch job.

2. User request for creating new process.

3. To provide new services by OS.

4. System call from currently running process.

• Operating system creates a new process with the specified or default attributes and identifier. A process may create several new sub-process.

• Parent process is creating process and the new processes are called the children of the process. When operating system creates process, it builds the data structure for managing process and allocates address space in primary main memory.

• Operating system creates foreground and background process. Process is identified cor by unique process identifier (PID) in UNIX and windows operating system. PID value is an integer number.

• All processes in UNIX are created using the fork() system call. The forking process is called the parent process. The new process is called the child process.

• Both the parent and child process have their own and private memory. Open files are shared between parent and child.

• If the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

• When a process is created, OS assigns some attributes. These are priority, privilege level, requirement of memory, access right, memory protection, PID etc. To perform operation, process needs software and hardware resources. It includes CPU time, files, memory, I/O device.

• Relation between parent process and child process is as follows:

1. Parent process continues to execute concurrently with its child process.

2. Parent process waits until some or all of its children have terminated.

```
void main()
{
printf("Operating System\n");
fork();
printf("Technical Publications\n");
return 0;
}
```

• In above program Operating System is printed only once and Technical Publications is printed two times.

Process Termination

• When process finishes its normal execution then that process is terminate. Operating system delete that process using exit () system call. After deleting process, memory space becomes free.

• OS passes the child's exit status to the parent process and then discards the process. At the same time, it de-allocate all the resources hold by this process. Following are the various reasons for process termination :

1. Normal completion of operation

2. Memory is not available

3. Time slice expired

4. Parent termination

5. Failure of I/O

6. Request from parent process

7. Misuse of access rights

Inter-process Communication

• Exchange of data between two or more separate, independent processes/threads is possible using IPC. Operating systems provide facilities/resources for Inter-Process Communications (IPC), such as message queues, semaphores, and shared memory.

• A complex programming environment often uses multiple cooperating processes to perform related operations. These processes must communicate with each other and share resources and information. The Kernel must provide mechanisms that make this possible. These mechanisms are collectively referred to as interprocess communication.

• Distributed computing systems make use of these facilities/resources to provide Application Programming Interface (API) which allows IPC to be programmed at a higher level of abstraction. (e.g., send and receive).

• Five types of inter-process communication are as follows:

1. Shared memory permits processes to communicate by simply reading and writing to a specified memory location.

2. Mapped memory is similar to shared memory, except that it is associated with a file in the file system.

3. Pipes permit sequential communication from one process to a related process.

4. FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the file system.

5. Sockets support communication between unrelated processes even on different computers.

Name	Description	Scope	Use
File	• Data is written to and read from a typical UNIX file. • Any number of processes can interoperate.	Local	Sharing large data sets.
Pipe	• Data is transferred between two processes using dedicated file descriptors. • Communication occurs only between a parent and child process.	Local	Simple data sharing, such as producer and consumer.
Named pipe	• Data is exchanged between processes via dedicated file descriptors. • Communication can occur between any two peer processes on the same host.	Local	Producer and consumer, or command-and-control, as demonstrated with MySQL server and its command-line query utility.
Signal	• An interrupt alerts the application to a specific condition.	Local	Cannot transfer data in a signal, so mostly useful for process management.
Shared memory	• Information is shared by reading and writing from a common segment of memory.	Local	Cooperative work of any kind, especially if security is required.
Socket	• After special setup, data is transferred using common input/output operations.	Local or remote	Network services such as FTP, ssh and the Apache web server.

Table 2.4.1 Interprocess communication in UNIX

• Purposes of IPC

1. Data transfer: One process may wish to send data to another process.

2. Sharing data: Multiple processes may wish to operate on shared data, such that if a process modifies the data, that change will be immediately visible to other processes sharing it.

3. Event modification: A process may wish to notify another process or set of processes that some event has occurred.

4. Resource sharing: The Kernel provides default semantics for resource allocation; they are not suitable for all application.

5. Process control : A process such as debugger may wish to assume complete control over the execution of another process.

• IPC has two forms: IPC on same host and IPC on different hosts

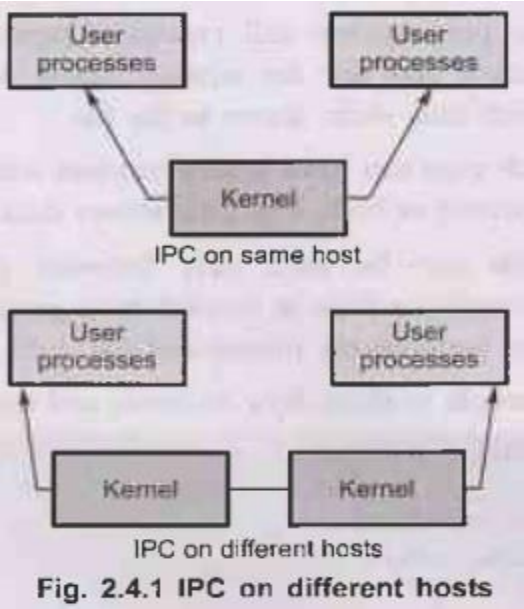


Fig. 2.4.1 IPC on different hosts

IPC is used for 2 functions:

1. Synchronization: Used to coordinate access to resources among processes and also to coordinate the execution of these processes. They are record locking, semaphores, mutexes and condition variables.

2. Message passing: Used when processes wish to exchange information. Message passing takes several forms such as: Pipes, FIFOs, message queues and shared memory.

Pipes

• A pipe is a unidirectional, first-in first-out, unstructured data stream of fixed maximum size. Writers add data to the end of the pipe; readers retrieve data from the front of the pipe.

• Once read, the data is removed from the pipe and is unavailable to other readers. A pipe provides a simple flow control mechanism.

• A process attempting to read from empty pipe blocks until more data is written to the pipe. A process trying to write to a full pipe lock until another process reads data from pipe.

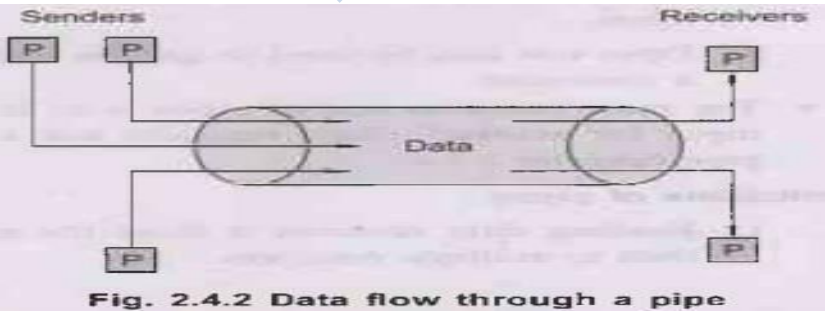


Fig. 2.4.2 Data flow through a pipe

• The pipe system call creates a pipe and returns two file descriptors: one for reading and one for writing. These descriptors are inherited by child processes, which thus share access to the file.

• Each pipe can have several readers and writers. A given process may be a reader or writer or both. Fig. 2.4.2 shows data flow through a pipe.

• Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork and the pipe is used between the parent and the child.

• Example to show how to create and use a pipe :

```
main()
{
int pipefd[2], n;
char buff[100];
if (pipe(pipefd) < 0)
err_sys("pipe error");
printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
if (write(pipefd[1], "hello world\n", 12) != 12)
err_sys("write error");
if ( (n=read(pipefd[0], buff, sizeof(buff))) <=0) err_sys("read error");
write (1, buff, n); /*fd=1= stdout*/
}
```

Properties of pipe :

1. Pipes do not have a name. For this reason, the processes must share a parent process. This is the main drawback to pipes. However, pipes are treated as file descriptors, so the pipes remain open even after fork and exec.

2. Pipes do not distinguish between messages; they just read a fixed number of bytes.

3. Pipes can also be used to get the output of a command or to provide input to a command

• The most common use of pipes is to let the output of one program become the input for another. Users typically join two programs by a pipe using the shell's pipe operator (1).

Limitations of pipes:

1. Reading data removes it from the pipe, a pipe cannot be used to broadcast data to multiple receivers.

2. Data in a pipe is treated as a byte stream and has no knowledge of message boundaries.

3. If there are multiple readers on a pipe, a writer cannot direct data to a specific reader.

• Program for sending data from parent process to child process over a pipe

```
#include <stdio.h>
int main(void)
{
int n; int fd[2]; pid_tpid;
char line[MAXLINE];
if (pipe(fd)<0) err_sys("pipe error");
if ((pid = fork()) < 0)
{
err_sys("fork error");
} else if (pid> 0)
{ /* parent */ close(fd[0]);
write(fd[1], "hello world\n", 12);
} else { /* child */
close(fd[1]);
n = read(fd[0], line, MAXLINE);
write(STDOUT_FILENO, line, n);
}
exit(0);
 }
```

• Pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe (int filedes[2]);
```

• Two file descriptor are returned through the filedes argument :filedes[0] is open for reading and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

• Fig. 2.4.3 shows two ways to view the UNIX pipe.

• The fstat function returns a file type of FIFO for the file descriptor of either end of a pipe. A pipe in a single process is not useful.

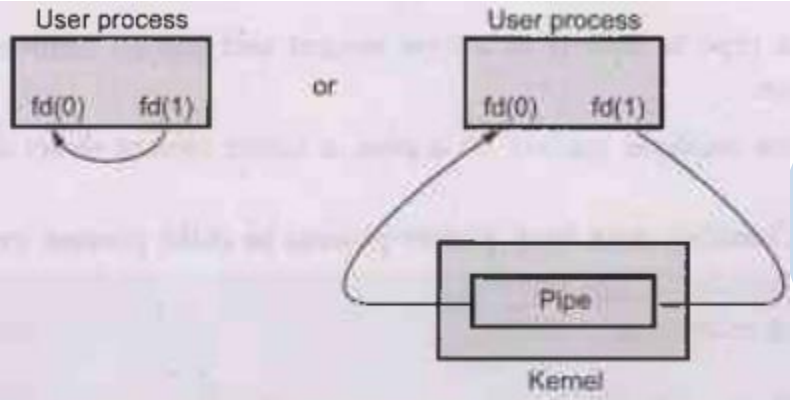


Fig. 2.4.3 Two ways to view the UNIX pipe

• Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa.

• Fig. 2.4.4 shows half-duplex pipe after a fork.

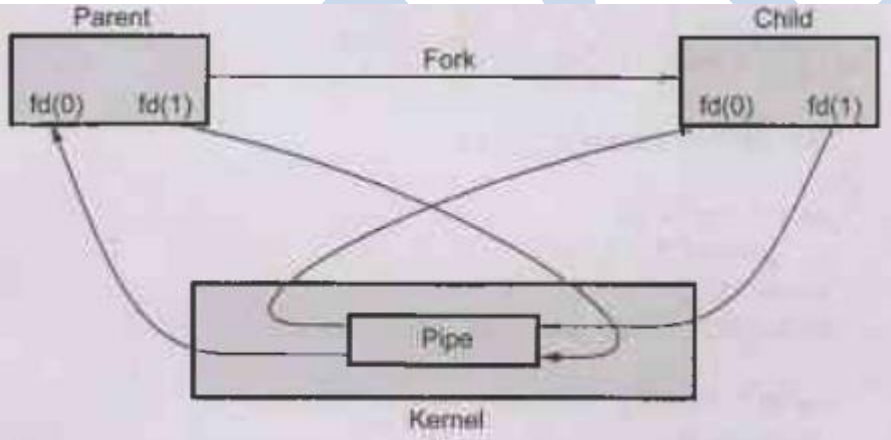


Fig. 2.4.4 Half-duplex pipe after a fork

• Fig. 2.4.5 shows pipe from parent to child. For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]).

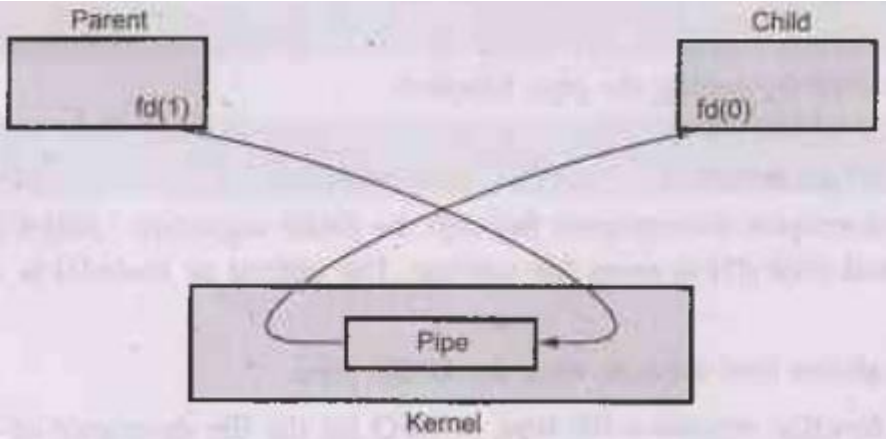


Fig. 2.4.5 Pipe from parent to child

• For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]. When one end of a pipe is closed, the following two rules apply.

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.

2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns 1 with errno set to EPIPE.

Features of Message Passing

1. Simplicity : Message passing system should be simple and easy to use. It should be possible to communicate with old and new applications.

2. Uniform semantics: Message passing is used for two types of IPC.

a. Local communication: Communicating processes are on the same node.

b. Remote communication: Communicating processes are on the different nodes.

3. Efficiency : IPC become so expansive if message passing system is not effective. Users try avoiding to IPC for their applications. Message passing system will become more efficient if we try to avoid more message exchanges during communication process. For examples :

a. Avoiding the costs of establishing and terminating connection.

b. Minimizing the costs of maintaining the connections.

c. Piggybacking of acknowledgement.

4. Reliability: Distributed systems are prone to different catastrophic events such as node crashes or physical link failures. Loss of message because of

communication link fails. To handle the loss messages, we required acknowledgement and retransmission policy. Duplicate message is one of the major problems. This happens because of timeouts or events of failures.

5. Correctness: Correctness is a feature related to IPC protocols for group communication. Issues related to correctness are as follows:

i. Atomicity : Every message sent to a group of receivers will be delivered to either all of them or none of them.

ii. Ordered delivery: Messages arrive to all receivers in an order acceptable to the application.

iii. Survivability: Messages will be correctly delivered despite partial failures of processes, machines, or communication links.

6. Security: Message passing system must provide a secure end to end communication.

7. Portability: Message passing system should itself be portable.

IPC Message Format

• Message passing system requires the synchronization and communication between the two processes. Message passing used as a method of communication in microkernels. Message passing systems come in many forms. Messages sent by a process can be either fixed or variable size. The actual function of message passing is normally provided in the form of a pair of primitives.

a) Send (destination_name, message)

b) Receive (source_name, message).

• Send primitive is used for sending a message to destination. Process sends information in the form of a message to another process designated by a in destination. A process receives information by executing the receive primitive, which indicates the source of the sending process and the message.

• Design characteristics of message system for IPC.

1. Synchronization between the process
2. Addressing
3. Format of the message
4. Queueing discipline

Issues in IPC by Message Passing

• Message is a block of information.

• A message is a meaningful formatted block of information sent by the sender be process to the receiver process.

• The message block consists of a fixed length header followed by a variable size collection of typed data objects.

• The header block of a message may have the following elements :

1. Address: A set of characters that uniquely identify both the sender and receiver.

2. Sequence number: It is the message identifier to identify duplicate and lost messages in case of system failures.

3. Structural information: It has two parts. The type part that specifies whether the data to be sent to the receiver is included within the message or the message only contains a pointer to the data. The second part specifies length of the variable-size message.

• Fig. 2.4.6 shows the typical message format.

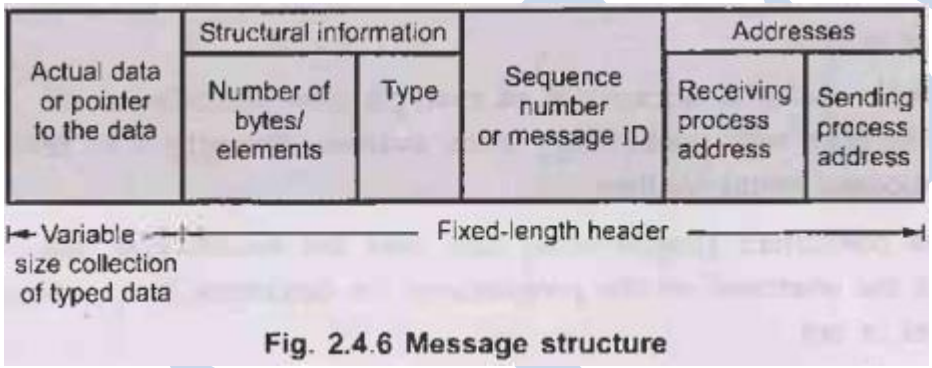


Fig. 2.4.6 Message structure

• Some important issues to be considered for the design of an IPC protocol based

message passing system :

i. The sender's identity

ii. The receiver's identity

iii. Number of receivers

iv. Guaranteed acceptance of sent messages by the receiver

v. Acknowledgment by the sender

vi. Handling system crashes or link failures

vii. Handling of buffers

viii. Order of delivery of messages

IPC Synchronization

• Send operation can be synchronous or asynchronous. Receive operation can be blocking or nonblocking.

• Sender and receiver process can be blocking mode or nonblocking mode. Different possibility of sender and receivers are as follows:

1. Blocking send, blocking receive

2. Nonblocking send, blocking receive

3. Nonblocking send, Nonblocking receive

Blocking send, blocking receive

• Blocking send must wait for the receiver to receive the message. Synchronous communication is an example of blocking send. Both processes (sender and receiver) are blocked until the message is delivered.

Rendezvous: sending a message to another process leaving the sender process suspended until the message is received and processed.

Nonblocking send, blocking receive

• Sender is free to send the messages but receiver is blocked until the requested message arrives.

• Asynchronous communication is an example of nonblocking send. Asynchronous communication with nonblocking sends increases throughput by reducing the time that processes spend waiting.

• Natural concurrent programming task uses the nonblocking send. Nonblocking send is the overhead on the programmer for determine that a message has been received or not.

Shared Memory

• A region of memory that is shared by co-operating processes is established. Processes can then exchange information by reading and writing data to the shared region.

• Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of Kernel intervention.

• In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the Kernel is required. • Fig. 2.4.7 shows client/server with shared memory.

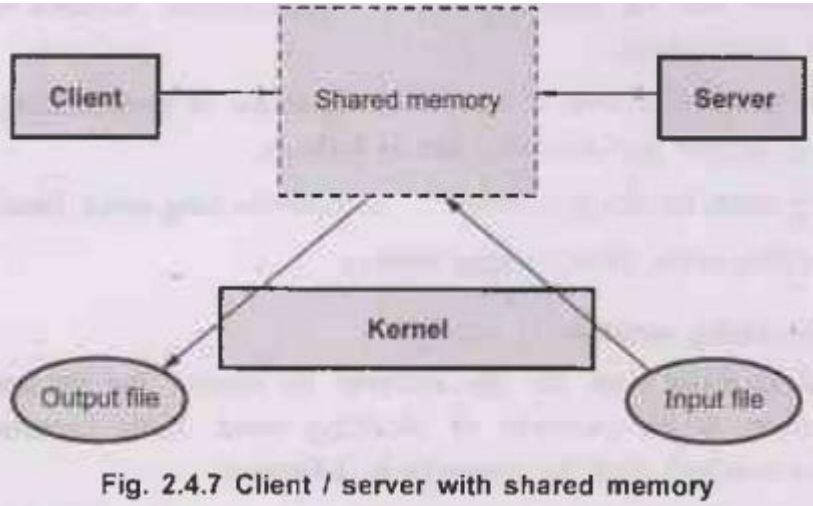


Fig. 2.4.7 Client / server with shared memory

Advantages:

1. Good for sharing large amount of data.

2. Very fast.

Limitations

1. No synchronization provided - applications must create their own.

2. Alternative to shared memory is mmap system call, which maps file into the address space of the caller.

CPU Scheduling

• In a multiprogramming environment, usually more programs to be executed than could possibly be run time at one time. In CPU scheduling it switches from one process to another process. CPU resource management is commonly known as scheduling.

• Objective of the multiprogramming is to increases the CPU utilization.

• User program contains combination of CPU burst cycle and I/O burst cycles. Process starts with CPU burst cycle then I/O burst cycle again CPU burst cycle again I/O burst cycle. In this way the process completes its executions. Process will terminates after final CPU burst cycle completions.

Program execution sequence: CPU burst cycle → I/O burst cycle → CPU burst cycle → I/O burst cycle → CPU burst cycle → Program terminates.

• A CPU bound process tends to use the processor time that the system allocates for it. An I/O bound process tends to use the processor only briefly before generating an I/O request. The CPU bound processes spend most of their time using the processor.

CPU Scheduler

• CPU scheduler is also called as short scheduler. System programmers use parameters like program size, resources required for program and any other special devices are needed to determine scheduling policies.

• Scheduling mechanism is the part of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

• Scheduler is responsible for multiplexing processes on the CPU:

Preemptive and Non-preemptive Scheduling

• The scheduling policy determines when it is time for process to be removed from the CPU and which ready process should be allocated the CPU next. The scheduling mechanism is composed of several different parts, depending on exactly how it is implemented in any particular operating system.

• CPU scheduling is divided into two types: Preemptive scheduling and non-preemptive scheduling.

• **Preemptive scheduling:** A scheduling method that interrupts the processing of a process and transfers the CPU to another process is called a preemptive CPU scheduling. The process switches from running state to the ready state and waiting the ready state state to the ready state.

• **Non-preemptive scheduling:**Non-preemptive operation usually proceeds towards completion uninterrupted. Once the system has assigned a processor to a process, the system cannot remove that processor from the process. The process switches from running state to the waiting state and termination of process.

• Preemptive scheduling increases the cost and it has higher overheads. This scheduling method is useful only in the high priority processes require rapid response.

• Non-preemptive scheduling is simple to implement. It requires some hardware platform. This method is attractive because of its simplicity. Windows 3.1 and Apple Macintosh operating system uses this scheduling method.

Difference between Preemptive and Non-preemptive Scheduling

Sr. No.	Preemptive scheduling	Non-preemptive scheduling
1.	Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process.	Non-preemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.
2.	Preemptive scheduling incurs a cost associated with access shared data.	Non-preemptive scheduling does not increase the cost.
3.	It also affects the design of the operating system Kernel.	It does not affects the design of OS Kernel.
4.	Preemptive scheduling is more complex.	Simple, but very inefficient.
5.	Example : Round robin method.	Example : First come first serve method.

CPU Scheduling Criteria

- Depending on the system, CPU scheduling criteria will change.

1. Throughput: CPU scheduling should attempt to service the maximum number of processes per unit time. The higher is the number, the more work is done by the system.

2. Waiting time: The average period of time a process spends waiting. Process is normally in the ready queue in waiting time.

3. Turnaround time: Turnaround time start from process submission to completion of process.

Turnaround time = Burst time + Waiting time

4. Response time: It is the time from the submission of a request until the first response is produced.

5.CPU utilization: It is average function of time during which the processor is busy.

6. Fairness: Avoid the process from the starvation. All the processes mu given equal opportunity to execute.

7. Priority: If the operating system assigns priorities to processes, the scheduling mechanism should favor the higher priority processes.

8. Predictability: A given process always should run in about the same amount of time under similar system loads.

• Depending upon the nature of operations the scheduling policy may differ. The CPU utilization and throughput are the system centered parameters. Fairness is snit affect by user and system.

Dispatcher

• Using dispatcher, CPU selects the process from the short term scheduler. Functions of the dispatcher is as follows :

1. Switching context
2. Switching to user mode
3. Jump to proper location in the user program for restarting that program.

• Dispatcher is a small program that switches the processor from one process to another.

• **Dispatch Latency:** It is time taken by dispatcher to stop running one process and start other process running is called as dispatch latency.

Scheduling Algorithms

• CPU scheduling algorithms are as follows:

1. First In First Out (FIFO) scheduling
2. Shortest job first scheduling
3. Priority scheduling
4. Round robin scheduling

First Come First Serve Scheduling

• This method of scheduling means that the first process to request the processor gets it until it finished execution. With this algorithm, processes are assigned the CPU in the order they request it.

• Normally there is a single queue of ready processes. When the first process enters the system from the outside, it is started immediately and allowed to run as long as it wants it. At the same time, other process enters into the system; they are put onto the end of the queue.

• New process enters the tail of the queue and the schedule selects the the head of the queue.

• When new process enters into the system, its process control block is linked to the end of the ready queue and it is removed from the front of the queue.

• When a process waits or blocks, it is removed from the queue and it queues up Anton again in FCFS queue when it gets ready.

• FCFS is non-preemptive CPU scheduling algorithm. It is also called First In First Out method (FIFO).

• FCFS is simple to implement because it uses a FIFO queue. This algorithm is fine for most of the batch operating systems.

• FCFS is not useful in scheduling interactive processes because it cannot guarantee short response time. This algorithm performs much better for long processes than and it short ones.

• Turnaround time is unpredictable with the FCFS algorithm. Average waiting time of the FCFS is often quite long.

• Real life analogy is buying tickets.

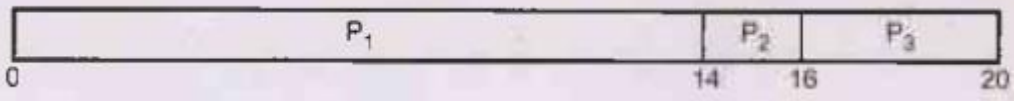
Example 2.6.1 Consider the following set of process that arrives at time 0, with the length of the CPU burst given in milliseconds

Calculate the average waiting time.

Process	Burst time
P_1	14
P_2	2
P_3	4

Solution :

Gantt chart :



Such a diagram is called "Gantt charts", showing when each CPU burst uses the CPU. Here, each CPU burst comes from a different thread.

Waiting Time :

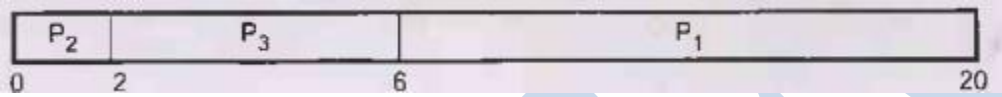
Process	Waiting time
P_1	0
P_2	14
P_3	16

Average waiting time = $\frac{\text{Sum of } P_1, P_2 \text{ and } P_3 \text{ waiting time}}{3}$

$= \frac{0+14+16}{3} = 10$

• If the processes change their order of arrive i.e. P2, P3, P1, then the results will be different than first one and it is shown below:

Gantt chart:



Waiting Time :

Process	Waiting time
P_1	6
P_2	0
P_3	2

Average waiting time = $\frac{\text{Sum of } P_1, P_2 \text{ and } P_3 \text{ waiting time}}{3}$

$= \frac{6+0+2}{3} = 2.66$

Convey effect

• To reduce I/O device utilization, all I/O bound processes will be waiting excessively long for processor bound ones. This is called as convey effect.

• If one process monopolizes the system, the extent of its overall effect on system performance depends on the scheduling policy and whether the process is processor bound or I/O bound.

Advantages

1. Simple to implement

2. Fair

Disadvantages

1. Waiting time depends on arrival order

2. Convoy effect: short process stuck waiting for long process

3. Also called head of the line blocking

Example 2.6.2 Consider the following set of process that arrive at time 0, with the length of CPU burst given in milliseconds. Calculate the average waiting time when the processes arrive in the following order:

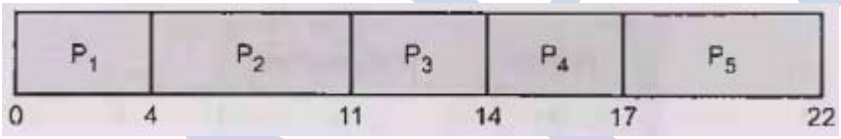
a. P1, P2, P3 P4 P5

Provide the Gantt chart for the same.

Process	*Burst time*
P_1	4
P_2	7
P_3	3
P_4	3
P_5	5

Solution :

Gantt chart :



Such diagram is called Gantt charts, showing when each process burst uses the CPU.

Waiting time:

Process	Waiting time
P_1	0 – 0 = 0
P_2	4 – 0 = 4
P_3	11 – 0 = 11
P_4	14 – 0 = 14
P_5	17 – 0 = 17

Average waiting time $= \frac{0+4+11+14+17}{5} = \frac{46}{5} = 9.2$

Turnaround time: Turnaround time = Waiting time + Burst time

Process	Turnaround time
P_1	0 + 4 = 4
P_2ws	4 + 7 = 11
P_3	11 + 3 = 14
P_4	14 + 3 = 17
P_5	17 + 5 = 22

Average turnaround time $= \frac{4+11+14+17+22}{5} = \frac{68}{5} = 13.60$

Shortest Job First Scheduling

• Shortest job first scheduling algorithm is also known as Shortest Job Next (SJN) scheduling algorithm. It handles the process based on length of their CPU cycle time. It reduces average waiting time over FIFO algorithm.

• SJF is a non-preemptive CPU scheduling algorithm.

• It does not work in interactive system because users do not estimate in advance the CPU time required to run their processes.

• SJF scheduling algorithm is used frequently in long term scheduling.

• When a process request a CPU, it must inform the system for how long it wants to use the CPU.

• When CPU become available, the system allocates into processes with the least expected execution time.

• To break ties, it follows the FCFS algorithm.

• Preemptive version of SJF is called as Shortest Remaining Time Next (SRTN)

• Preemptive SJF algorithm will preempt the currently exceuting process, whereas a non preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

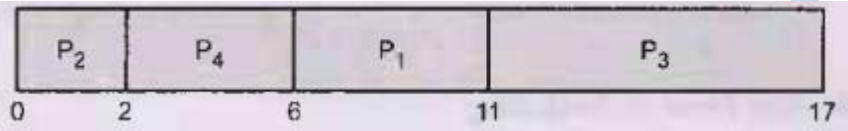
• SIF selects processes for service in a manner ensuring the next one will complete and leave the system as soon as possible.

Example 2.6.3 Calculate the average waiting time and average turnaround time. Provide the Gantt chart for the same.

Process	Burst time
P_1	5
P_2	2
P_3	6
P_4	4

Solution :

Gantt chart:



Waiting time :

Process	Waiting time
P_1	6 – 0 = 6
P_2	0 – 0 = 0
P_3	11 – 0 = 11
P_4	2 – 0 = 2

Average waiting time $= \frac{6+0+11+2}{4} = \frac{19}{4} = 4.75$

Turnaround time: Turnaround time = Waiting time + Burst time

Process	Turnaround time
P_1	6 + 5 = 11
P_2	0 + 2 = 2
P_3	11 + 6 = 17
P_4	2 + 4 = 6

Average turnaround time $= \frac{11+2+17+6}{4} = \frac{36}{4} = 9$

• The SJF scheduling algorithm is optimal only when all of the processes are available at the same time and the CPU estimates are available.

• SJF scheduling algorithm may be preemptive or non-preemptive.

Priority Scheduling

• Priority CPU scheduling algorithm is preemptive and non-preemptive algorithm. It is one of the most common scheduling algorithm in batch system. Priority can be assigned by a system admin using characteristics of the process.

• In this scheduling algorithm, CPU select higher priority process first. If the priority of two process is same then FCFS scheduling algorithm is applied for solving the problem.

• The priority of a process determines how quickly its request for a CPU will be granted if other processes make competing requests.

• Each process is assigned a priority number for the purpose of CPU scheduling.

• The priority number is normally a non negative integer number.

• Non preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

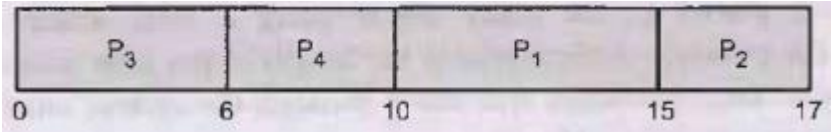
• Preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

Example 2.6.4 Consider the following set of process that arrive at time 0, with the length of CPU burst given in milliseconds. Calculate the average waiting time and average turnaround time. Provide the Gantt chart for the same. (Time slice = 2)

Process	*Burst time*	*Priority*
P_1	5	3
P_2	2	4
P_3	6	1
P_4	4	2

Solution: For non-preemptive priority scheduling

Gantt chart :



Waiting time:

Process	Waiting time
P_1	10 – 0 = 10
P_2	15 – 0 = 15
P_3	0 – 0 = 0
P_4	6 – 0 = 6

Average waiting time $= \frac{10+15+0+6}{4} = \frac{31}{4} = 7.75$

Turnaround time: Turnaround time = Waiting time + Burst time

Process	Turnaround time
P_1	10 + 5 = 15
P_2	15 + 2 = 17
P_3	0 + 6 = 6
P_4	6 + 4 = 10

Average turnaround time $= \frac{15+17+6+10}{4} = \frac{48}{4} = 12$

Problem with Priority Scheduling

• Waiting time is more for lower priority process even if there required CPU burst time is less. Priority scheduling algorithm faces the starvation problem. Starvation problem can be solved by using aging techniques.

• In aging, the priority of the process will increase which is waiting for long time in the ready queue.

Round Robin Scheduling

• Round robin is a preemptive scheduling algorithm. It is used in interactive system. Here process are given a limited amount of time of processor time called a time slice or time quantum. If a process does not complete before its quantum expires, the system preempts it and gives the processor to the next waiting process. The system then places the preempted process at the back of the ready queue.

• Processes are placed in the ready queue using a FIFO scheme. With the RR algorithm, the principle design issue is the length of the time quantum to be

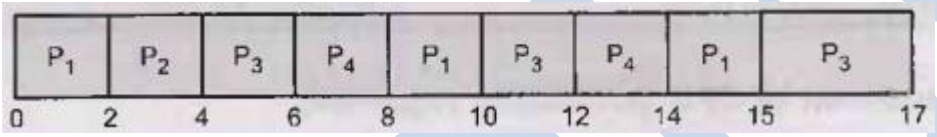
used. For short time slice, processes will move through the system relatively quickly. It increases the processing overheads.

Example 2.6.5 Consider the following set of process that arrive at time 0, with the length of CPU burst given in milliseconds. Calculate the average waiting time and average turnaround time. Provide the Gantt chart for the same. (Time slice = 2)

Solution:

Process	*Burst time*
P_1	5
P_2	2
P_3	6
P_4	4

Gantt chart:



Waiting time :

Process	Waiting time
P_1	0 – 0 + 8 – 2 + 14 – 10 = 10
P_2	2 – 0 = 2
P_3	4 – 0 + 10 – 6 + 15 – 12 = 11
P_4	6 – 0 + 12 – 8 = 10

Average waiting time $= \frac{10+2+11+10}{4} = \frac{33}{4} = 8.25$

Turnaround time: Turnaround time = Waiting time + Burst time

Process	Turnaround time
P_1	10 + 5 = 15
P_2	2 + 2 = 4
P_3	11 + 6 = 17
P_4	10 + 4 = 14

Average turnaround time $= \frac{15+4+17+14}{4} = \frac{50}{4} = 12.5$

Comparison between FCFS and RR

Comparison of CPU Scheduling Algorithm

Sr. No.	FCFS	Round robin
1.	FCFS decision made is non-preemptive.	RR decision made is preemptive.
2.	It has minimum overhead.	It has low overhead.
3.	Response time may be high.	Provides good response time for short processes.
4.	It is troublesome for time sharing system.	It is mainly designed for time sharing system.
5.	The workload is simply processed in the order of arrival.	It is similar like FCFS but uses time quantum.
6.	No starvation in FCFS.	No starvation in RR.

Algorithm	Policy type	Used in	Advantages	Disadvantages
FCFS	Non-preemptive	Batch	1. Easy to implement. 2. Minimum overhead.	1. Unpredictable turn around time. 2. Average waiting is more.
RR	Preemptive	Interactive	1. Provides fair CPU allocation. 2. Provides reasonable response times to interactive users.	1. Requires selection of good time slice.
Priority	Non-preemptive	Batch	1. Ensures fast completion of important jobs.	1. Indefinite postponement of some jobs. 2. Faces starvation problem.
SJF	Non-preemptive	Batch	1. Minimizes average waiting time. 2. SJF algorithm is optimal.	1. Indefinite postponement of some jobs. 2. Cannot be implemented at the level of short term scheduling. 3. Difficulty is knowing the length of the next CPU request.
Multilevel queues	Preemptive or Non-preemptive	Batch/Interactive	1. Flexible. 2. Gives fair treatment to CPUbound jobs.	1. Overhead incurred by monitoring of queues.

Example 2.6.6 Consider the following process, with the CPU burst time given in milliseconds.

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

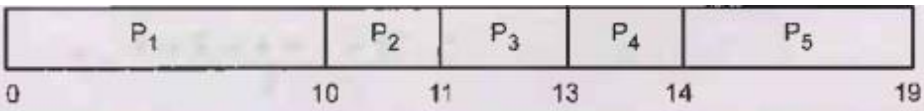
Processes are arrived in P1, P2, P3, P4 P5 order at time 0.

(i) Draw four Gantt charts to show execution using FIFO, SJF, non-preemptive priority and Round Robin (Quantum = 1) scheduling.

(ii) Also calculate waiting time and turnaround time for each scheduling algorithm. **AU May-18, Marks 15**

Solution :

1) FIFO: Gantt chart:



Process	Waiting time for FIFO	Turnaroud time for FIFO
P_1	0	0 + 10 = 10
P_2	10	10 + 1 = 11
P_3	11	11 + 2 = 13
P_4	13	13 + 1 = 14
P_5	14	14 + 5 = 19

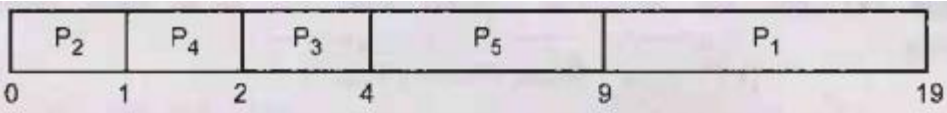
$$\text{Average waiting time} = \frac{\text{Sum of all processes waiting time}}{5}$$

$$= \frac{0 + 10 + 11 + 13 + 14}{5} = 9.6$$

$$\text{Average turnaround time} = \frac{\text{Sum of all processes turnaround time}}{5}$$

$$= \frac{10 + 11 + 13 + 14 + 19}{5} = 13.4$$

2) SJF: Gantt chart:



Process	Waiting time for SJF	Turnaroud time for SJF
P_1	9	9 + 10 = 19
P_2	0	0 + 1 = 1
P_3	2	2 + 2 = 4
P_4	1	1 + 1 = 2
P_5	4	4 + 5 = 9

$$\text{Average waiting time} = \frac{\text{Sum of all processes waiting time}}{5}$$

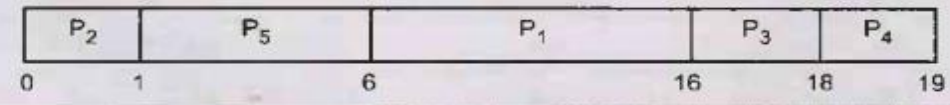
$$= \frac{9 + 0 + 2 + 1 + 4}{5} = 3.2$$

$$\text{Average turnaround time} = \frac{\text{Sum of all processes turnaround time}}{5}$$

$$= \frac{19 + 1 + 4 + 2 + 9}{5} = 7$$

3) Non-Preemptive Priority

Gantt chart: (1 is highest priority and 4 is lowest priority)



Process	Waiting time for Non-Preemptive Priority	Turnaroud time for Non-Preemptive Priority
P_1	6	6 + 10 = 16
P_2	0	0 + 1 = 1
P_3	16	16 + 2 = 18
P_4	18	18 + 1 = 19
P_5	1	1 + 5 = 6

$$\text{Average waiting time} = \frac{\text{Sum of all processes waiting time}}{5}$$

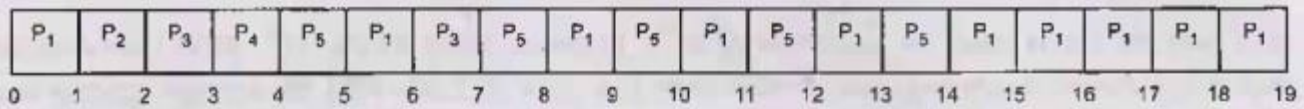
$$= \frac{6 + 0 + 16 + 18 + 1}{5} = 8.2$$

$$\text{Average turnaround time} = \frac{\text{Sum of all processes turnaround time}}{5}$$

$$= \frac{16 + 1 + 18 + 19 + 6}{5} = 13.2$$

4) Round Robin (Quantum = 1)

Gantt chart:



Process	Waiting time for Round Robin	Turnaroud time for Round Robin
P_1	9	9 + 10 = 19
P_2	1	1 + 1 = 2
P_3	3	3 + 2 = 5
P_4	3	3 + 1 = 4
P_5	9	9 + 5 = 14

$$\text{Average waiting time} = \frac{\text{Sum of all processes waiting time}}{5}$$

$$= \frac{9 + 1 + 3 + 3 + 9}{5} = 5$$

$$\text{Average turnaround time} = \frac{\text{Sum of all processes turnaround time}}{5}$$

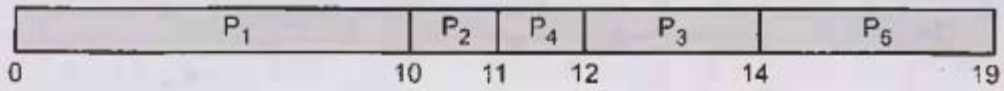
$$= \frac{19 + 2 + 5 + 4 + 14}{5} = 8.8$$

Example 2.6.7 Consider the following set of processes with the length of CPU - burst time given in milliseconds.

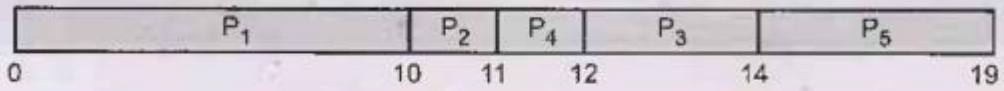
Process	Burst time	Priority	Arrival time
P_1	10	3	0
P_2	1	1	1
P_3	2	3	2
P_4	1	4	1

Draw the Gantt chart for execution of these processes using FCFS, SJF, SRTS, pre-emptive and non-pre-emptive priority, round robin with time slice of 2 ms. Find the average waiting and turnaround time using each of the methods.

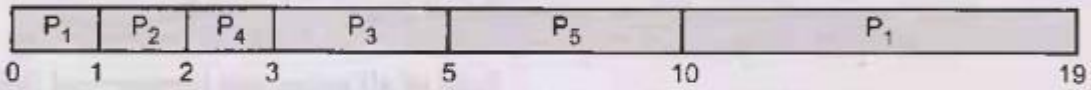
i) FCFS (Gantt chart)



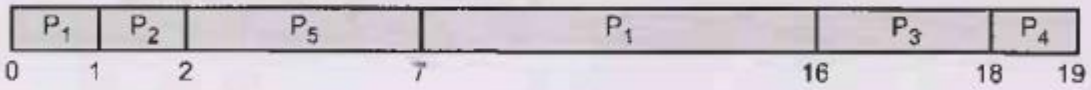
ii) SJF



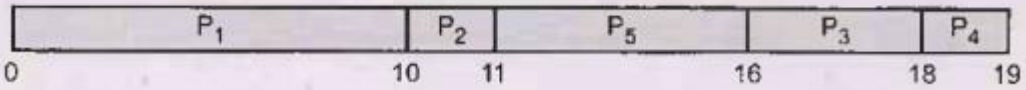
iii) SRTF (Gantt chart)



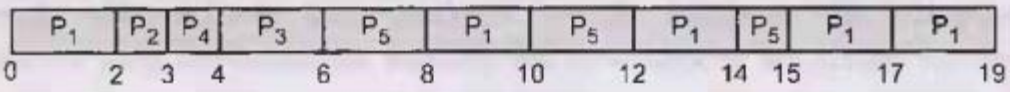
iv) Preemptive priority



v) Non-preemptive priority



vi) Round robin (time slice = 2)



Waiting time:

Process	FCFS	SJF	SRFT	Preemptive Priority	Non-preemptive Priority	Round Robin
P_1	0 – 0 = 0	0 – 0 = 0	0 – 0 + 10 – 1 = 9	0 – 0 + 7 – 1 = 6	0 – 0 = 0	0 – 0 + 8 – 2 + 12 – 1 0 + 15 – 14 = 9
P_2	10 – 1 = 9	10 – 1 = 9	1 – 1 = 0	1 – 1 = 0	10 – 1 = 9	2 – 1 = 1
P_3	12 – 2 = 10	12 – 2 = 10	3 – 2 = 1	16 – 2 = 14	16 – 2 = 14	4 – 2 = 2
P_4	11 – 1 = 10	11 – 1 = 10	2 – 1 = 1	18 – 1 = 17	18 – 1 = 17	3 – 1 = 2
P_5	14 – 2 = 12	14 – 2 = 12	5 – 2 = 3	2 – 2 = 0	11 – 2 = 9	6 – 2 + 10 – 8 + 14 – 12 = 8

Turnaround time

	FCFS	SJF	SRTF	Preemptive Priority	Non-preemptive Priority	Round Robin
P_1	10 + 0 = 10	10 + 0 = 10	10 + 9 = 19	10 + 6 = 16	10 + 0 = 10	10 + 9 = 19
P_2	1 + 9 = 10	1 + 9 = 10	1 + 0 = 1	1 + 0 = 1	1 + 9 = 10	1 + 1 = 2
P_3	2 + 10 = 12	2 + 10 = 12	2 + 1 = 3	2 + 14 = 16	2 + 14 = 16	2 + 2 = 4
P_4	1 + 10 = 11	1 + 10 = 11	1 + 1 = 2	1 + 17 = 18	1 + 17 = 18	1 + 2 = 3
P_5	5 + 12 = 17	5 + 12 = 17	5 + 3 = 8	5 + 0 = 5	5 + 9 = 14	5 + 8 = 13

Average waiting time and turnaround time

	Averege Waiting time	Average Turnaround time
FCFS	(0 + 9 + 10 + 10 + 12) / 5 = 8.2	(10 + 10 + 12 + 11 + 17) / 5 = 12
SJF	(0 + 9 + 10 + 10+ 12) / 5 = 8.2	(10 + 10 + 12 + 11 + 17) / 5 = 12
SRTF	(9 + 0 + 1 + 1 + 3) / 5 = 2.8	(19 + 1 + 3 + 2 + 8) / 5 = 7
Preemptive priority	(6 + 0 + 14 + 17 + 0) / 5 = 7.4	(16 + 1 + 16 + 18 + 5) / 5 = 11.2
Non-Preemptive priority	(0 + 9 + 14 + 17 + 9) / 5 = 9.8	(10 + 10 + 16 + 18 + 14) / 5 = 13.6
Round Robin	(9 + 1 + 2 + 2 + 8) / 5 = 4.4	(19 + 2 + 4 + 3 + 13) / 5 = 8.2

Shortest Remaining Time Next

• Preemptive SJF is called shortest remaining time first. In this algorithm, the scheduler selects the process with the smallest estimated run time to completion.

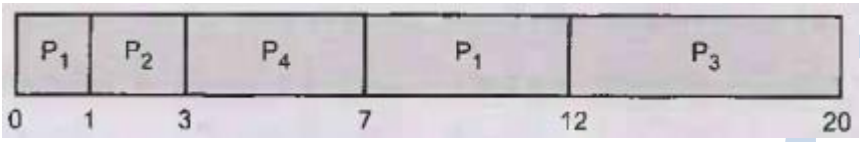
• This scheduler gives minimum wait times in theory but in certain situation due to preemption overhead, shortest process first might performance better.

• An advantageous of this method is that, the short processes are handled very quickly. The system requires very little overhead since it only makes a decision when a process completes or a new process is added. When a new process is admitted the SRTN algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

• Consider the four processes with their arrival and burst time.

Process	Burst Time	Arrival time
P1	6	0
P2	2	1
P3	8	2
P4	4	3

• The execution of the process is shown by the Gantt chart:



• Now we calculate waiting time and turnaround time for all processes.

Process	Waiting Time	Turnaround time
P1	(0 - 0) + (7 -1) = 6	6 + 6 = 12
P2	1 - 1 = 0	2 + 0 = 2
P3	12 - 2 = 10	8 + 10 = 18
P4	3 - 3 = 0	4 + 0 = 4

$$\text{Average waiting time} = \frac{6+0+10+0}{4} = \frac{16}{4} = 4$$

$$\text{Average waiting time} = \frac{12+2+18+4}{4} = \frac{36}{4} = 9$$

Advantages:

1. Very short processes get very good service.

2. The penalty ratios are small; this algorithm works extremely well in most cases

3. This algorithm pravably gives the highest throughput of all scheduling algorithms if the estimates are exactly correct.

Multilevel Queue Scheduling

• Multilevel queue is not separate scheduling algorithm but it works in conjunction with several CPU scheduling algorithms. The processes can be grouped according to common characteristics.

• One type of multilevel queue is based on the priority based system with different queues for each priority level.

• System consists of two types of processes or jobs: Interactive and batch. Interactive jobs are shorter and batch jobs are longer. Fig. 2.6.1 shows multilevel queue scheduling.

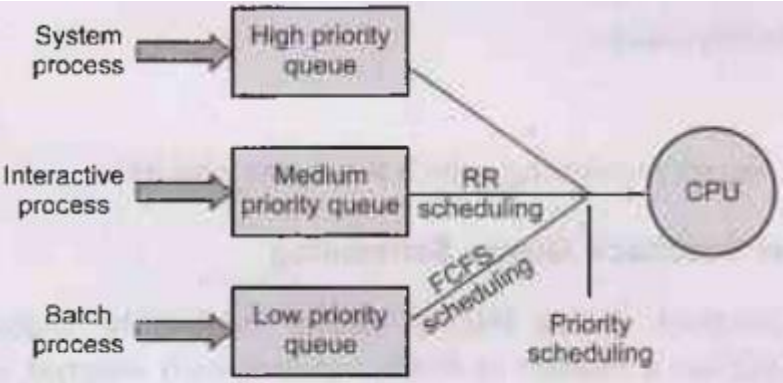


Fig. 2.6.1 Multilevel queue scheduling

• To keep minimum response time for user, interactive jobs are scheduled before the batch jobs. So ready queue is partitioned into two separate queue: interactive (foreground) and batch (background). Each queue uses their own scheduling algorithm.

• Once the jobs assigns to the queue, it will remain in that queue. Jobs never change the queue. Here we assume that FCFS is applied to one queue and round robin method is applied to another queue.

•The batch jobs are put in one queue called the background queue while the interactive jobs are put in a foreground queue. Batch jobs are scheduled by first come first serve method and interactive jobs by round robin method.

• The scheduler now has to decide which queue to run. The methods are as follows:

1. Higher priority queues can be processed until they are empty before the lower priority queues are executed.

2. Each queue can be given a certain amount of the CPU. Maybe, the interactive queue could be assigned 75 % of the CPU, with the batch queue being given 25 %.

• It should also be noted that there can be many other queues. Many systems will have a system queue. This queue will contain processes that are important to keep the system running. For this reason these processes normally have the highest priority.

• It should be also need to specify which queue a process will be put to when it arrives to the system and/or when it starts a new CPU burst.

Advantages:

1. It can be preemptive or non-preemptive.
2. Short CPU bound processes are executed first.
3. Simple to implement.

Disadvantage :

1. Queues require monitoring, which is a costly activity.

Multilevel Feedback Queue Scheduling

• Multilevel Feedback Queue (MLFQ) allows moving the processes between the queues. MLFQ has a number of distinct queues; each assigned a different priority level. At any given time, a process that is ready to run is on a single queue. MLFQ uses priorities to decide which process should run at a given time. A process with higher priority is selected for execution.

• Consider processes with different CPU burst characteristics. If a process uses too much of the CPU it will be moved to a lower priority queue. This will leave I/O bound and interactive processes in the higher priority queue.

• Fig. 2.6.2 shows multilevel feedback queue scheduling. (See Fig. 2.6.2 on next page.)

• MLFQ also uses concept of aging to prevent starvation. The processes with lower priority will move towards the higher priority queue in aging concept. The key to MLFQ scheduling lies in how the scheduler sets priorities. Rather than giving a fixed priority to each process, MLFQ varies the priority of a process based on its observed behavior.

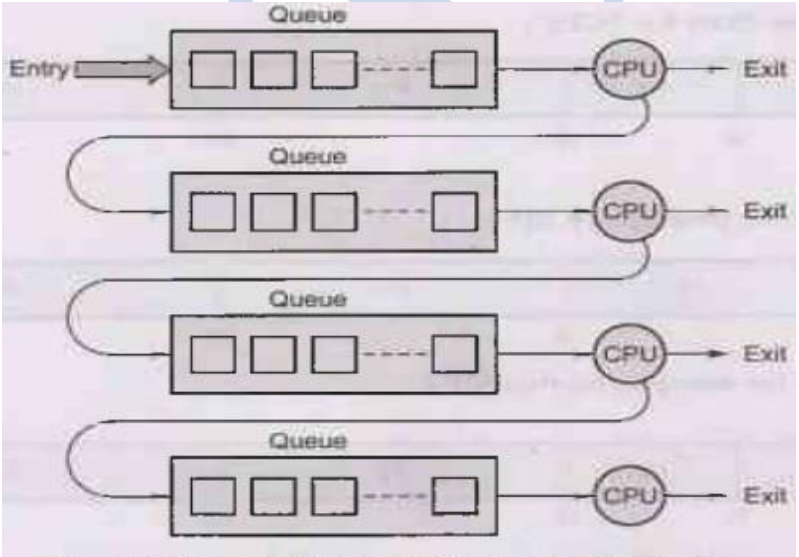


Fig. 2.6.2 Multilevel feedback queue scheduling

• Assume we have three queues (Q0, Q1 and Q2). Q0 is the lowest priority queue and Q2 is the highest priority queue. The process enters at the highest priority ESA (Q2). After a single time-slice of 10 ms, the scheduler reduces the process's priority by one, and thus the job is on Q1. After running at Q1 for a time slice, the job is finally lowered to the lowest priority in the system (Q0), where it remains.

• Following parameters are defined by scheduler for implementing MLFQ:

1. Number of queue required.

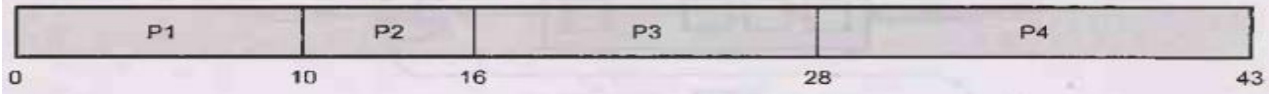
2. Scheduling algorithm for each queue.

3. Method is required for finding to increase and/or decrease priority of processes.

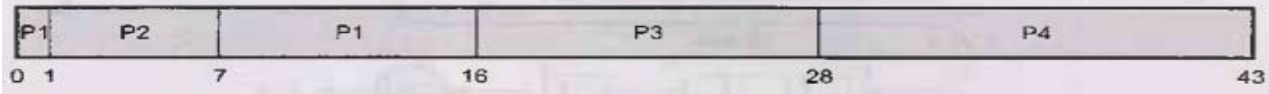
Example 2.6.8 Explain the FCFS, preemptive and non preemptive versions of shortest-job-first and roudn robin (time slice = 2) schedulign algorithms with Gantt chart for the four processes given. Compare their average turn around and waiting time.

Process	Arrival time	Burst time
P1	0	10
P2	1	6
P3	2	12
P4	3	15

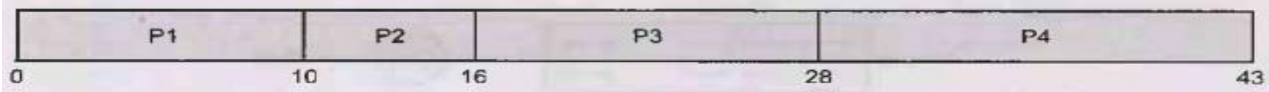
Solution : a) Gantt chart for FCFS :



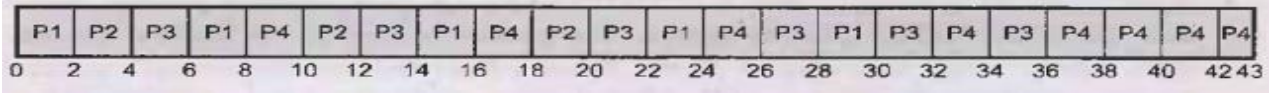
b) Gantt chart for preemptive SJF :



c) Gantt chart for non-preemptive SJF :



d) Gantt chart for round robin :



Waiting time

Process	FCFS	Preemptive SJF	Nonpreemptive SJF	RR
P1	0	6	0	20
P2	9	0	9	13
P3	14	14	14	22
P4	25	25	25	25

AverageWaiting time

$$\text{FCFS} = \frac{0+9+14+25}{4} = \frac{48}{4} = 12$$

$$\text{Preemptive SJF} = \frac{6+0+14+15}{4} = \frac{35}{4} = 8.75$$

$$\text{Nonpreemptive SJF} = \frac{0+9+14+15}{4} = \frac{48}{4} = 12$$

$$\text{RR} = \frac{20+13+22+25}{4} = \frac{80}{4} = 20$$

Turnaround time

Process	FCFS	Preemptive SJF	Nonpreemptive SJF	RR
P1	10	16	10	30
P2	15	6	15	19
P3	26	26	26	34
P4	40	40	40	40

Average Turnaround time

$$\text{FCFS} = \frac{10+15+26+40}{4} = \frac{91}{4} = 22.75$$

$$\text{Preemptive SJF} = \frac{16+6+26+40}{4} = \frac{88}{4} = 22$$

$$\text{Nonpreemptive SJF} = \frac{10+15+26+40}{4} = \frac{91}{4} = 22.75$$

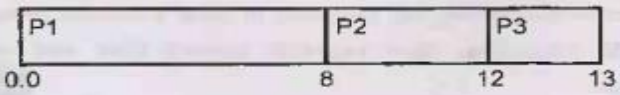
$$\text{RR} = \frac{30+19+34+40}{4} = \frac{123}{4} = 30.75$$

Example 2.6.9 What is the average turnaround time for the following process using a) FCFS b) SJF non-prrmptive c) Preemptive SJF.

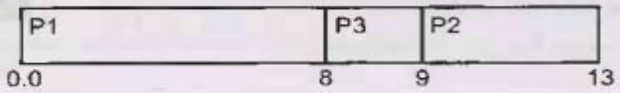
Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4

Solution: Gantt chart:

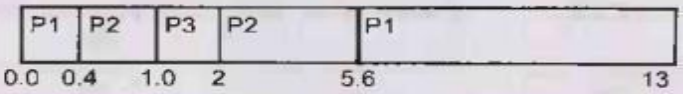
a) FCFS



b) SJF non-preemptive



c) Preemptive SJF



Process	Waiting Time		
	FCFS	SJF Non-preemptive	Preemptive SJF
P1	0	0	5.2
P2	7.6	8.6	1
P3	11	7	0

Process	Turnaround Time		
	FCFS	SJF Non-preemptive	Preemptive SJF
P1	8	8	13.2
P2	11.6	12.6	5
P3	12	8	1
Average turnaround time	7.2	9.53	6.4

University Questions

1.Consider the following set of processes, with the length of the CPU-burst time in given ms:

Process	*Burst Time*	*Arrival Time*
P1	*8*	*0.00*
P2	*4*	*1.001*
P3	*9*	*2.001*
P4	*5*	*3.001*
P5	*3*	*4.001*

Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, priority and RR (quantum = 2) scheduling. Also calculate waiting time and turnaround time for each scheduling algorithms. **AU May-17, Marks 13**

2. Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short process. :

i) RR ii) Multilevel feedback queues. **AU May-18, Marks 13**

3. Consider the following set of processes, 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	3
P_4	1	4
P_5	5	2

The process are assumed to have arrived in the order P1, P2, P3, P4, P all at time 0.

i) Draw Gantt chart that illustrate the execution of these processes using the scheduling algorithms FCFS (smaller priority number implies higher priority) and RR (quantum = 1).[10]

ii) What is the waiting time of each process for each of the scheduling algorithms. [5]

4. Explain - multi level queue and multi level feedback queue scheduling with suitable examples.

Threads

• Thread is a dispatchable unit of work. It consists of thread ID, program counter, stack and register set. Thread is also called a Light Weight Process (LWP). Because they take fewer resources then a process. A thread is easy to create and destroy.

• It shares many attributes of a process. Threads are scheduled on a processor. Each thread can execute a set of instructions independent of other threads and processes.

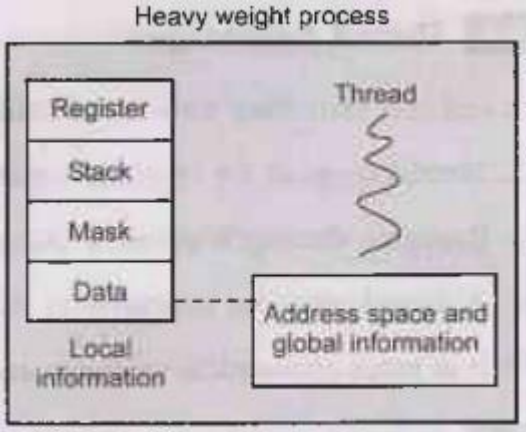


Fig. 2.7.1 Thread

• Every program has at least one thread. Programs without multithreading executes sequentially. That is, after executing one instruction the next instruction in sequence is executed.

• Thread is a basic processing unit to which an operating system allocates processor time and more than one thread can be executing code inside a process.

• When user double click on an icon by using mouse, the' operating system creates a process and that process has one thread that runs the icon's code.

• Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.

• Normally, an operating system will have a separate thread for each different activity. The OS will have a separate thread for each process and that thread will perform OS activities on behalf of the process. In this condition, each user process is backed by a kernel thread. When process issues a system call to read a file, the process's thread will take over, find out which disk accesses to generate, and issue the low level instructions required to start the transfer. It then suspends until the disk finishes reading in the data.

• When process starts up a remote TCP connection, its thread handles the low-level details of sending out network packets.

• In Remote Procedure Call (RPC), servers are multithreaded. Server receives the messages using a separate thread.

• Different operating system uses threads in different ways.

1. Free memory is managed by kernel thread in LINUX OS.

2. Solaries uses a kernel thread for interrupt handling.

Thread Advantages

1. Context switching time is minimized.

2. Thread support for efficient communication.

3. Resource sharing is possible using threads.

4. A thread provides concurrency within a process.

5. It is more economical to create and context switch threads.

Difference between Thread and Process

Sr. No.	Thread	Process
1.	Thread is also called lightweight process.	Process is also called heavyweight process.
2.	Operating system is not required for thread switching.	Operating system interface is required for process switching.
3.	One thread can read, write or even completely clean another threads stack.	Each process operates independently of the other process.
4.	All threads can share same set of open files and child processes.	In multiple processing, each process executes the same code but has its own memory and file resources.
5.	If one thread is blocked and waiting then second thread in the same task can run.	If one server process is blocked then other server process cannot execute until the first process is unblocked.
6.	Uses fewer resources.	Uses more resources.

Thread Lifecycle

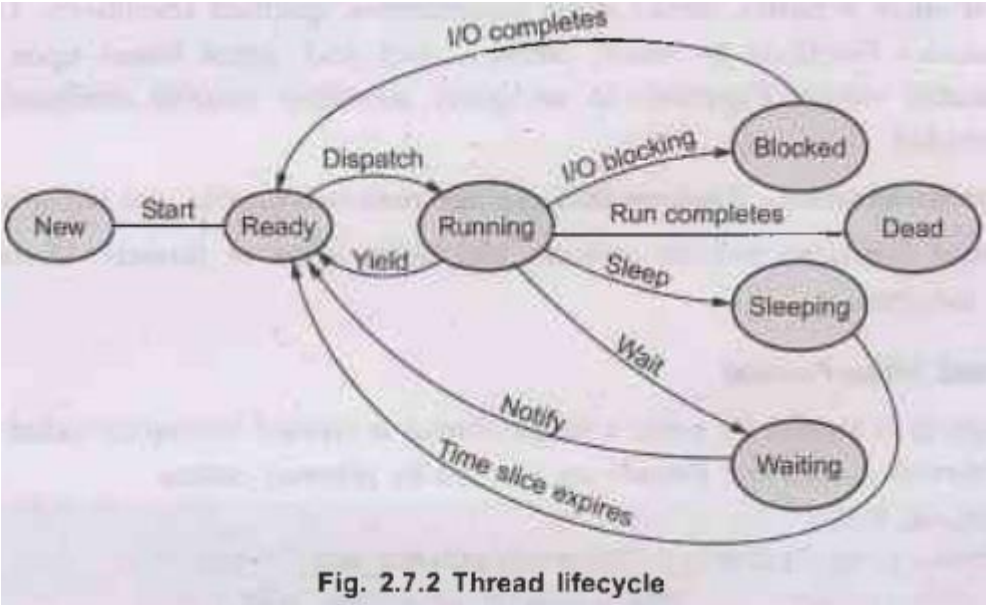


Fig. 2.7.2 Thread lifecycle

1. **New:** Thread is just created.
2. **Ready:** Thread's start() method invoked and now it is executing. OS put thread into Ready queue.

3. **Running**: Highest priority ready thread enters the running state. Thread is assigned a processor and now is running.

4. **Blocked**:This is the state when a thread is waiting for a lock to access an org object. il

5. **Waiting**: Here thread is waiting indefinitely for another thread to perform an action.

6. **Sleeping**: Thread sleep for a specified time of period. When sleeping time expires, it enters to ready state. CPU is not used by sleeping thread.

7. **Dead**: When thread completes task or operation.

Thread Programming and Libraries

• The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

1. **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)

2. **Mutexes**: Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes.

3. **Condition variables**: Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

4. **Synchronization:** Routines that manage read/write locks and barriers.

Basic thread functions include creation and termination of threads. There are five such thread function.

pthread_create Function

• A program is started by exec, a single thread is created known as initial thread or main thread. Additional threads are created by pthread_create

#include <pthread.h>

int pthread_create (pthread_t *tid, const pthread_attr_t *attr,

void *(*func) (void*), void* arg);

• Each thread is identified by a thread ID, of which data type is pthread_t. Each thread has numerous attributes, its priority, its initial stack size. The thread starts by calling this function and then terminates explicitely or implicitly. The address of function is specified as func argument which is called with a single pointer argument, arg.

pthread_join Function

• A thread can be terminated by calling pthread_join. In Unix process pthread_create is similar to fork and pthread_join is similar to wait pid.

#include<pthread.h>

int pthread_ join (pthread_ttid, void **status);

The tid is specified as the wait. But there is no way to wait for any of the threads.

pthread_self Function

• Each thread has an ID which identifies it within a given process. This thread ID is returned by pthread_create and is used by pthread_join.

• A thread fetches this value for itself by using ptheread_self.

#include <pthread.h>

pthread_tpthread_self (void);

pthread_detach Function

• A thread is joinable or detachable. When joinable thread terminates, its thread ID and exit status are retained until another thread calls pthread_join.

• A detached thread is like a daemon process i.e. when it is terminated all its resources are released. The pthread_detach function changes the specified thread so that it is detached.

#include<pthread.h>

int pthread_detach (pthread_ttid);

pthread_exit Function

A way for a thread to terminate is to call pthread_exit.

#include <pthread.h>

*void pthread_exit (void *status);*

• But if thread is not detached, its thread ID and exit status are retained for a later pthread_join by any other thread in calling processes.

pthread_sigmask, pthread_kill

The prototype of the pthread_sigmask and pthread_kill functions are:

#include <signal.h>

#include <pthread.h>

*int pthread_sigmask (int mode, sigset_t *sigsetp, sigset_t *oldstp);*

int pthread_kill (pthread_ttid, int signum);

The pthread_sigmask function sets the signal mask of a calling thread. The sigsetpargument contains one or more of the signal numbers to be applied to the calling thread.

• The mode argument specifies how the signal specified in the sigsetp argument is to boxile be used. Possible value of mode argument is declared in <signal.h> header and their meanings are:

Mode value	Meaning
SIG_BLOCK	Adds signals contained in the *sigsetp* argument to the thread signal mask.
SIG_UNBLOCK	Removes signals contained in the *sigsetp* argument from the thread signal mask.
SIG_SETMASK	Replace the thread signal mask with the signal specified in the *sigsetp* argument.

• *The pthread_kill function sends a signal, as specified in the signum argument, to aafbesthread whose ID is given by the tid argument. The sending and receiving threads must be in the same process.*

• If the default action for a signal is to terminate the process, then sending the bas signal to a thread will still kill the entire process.

sched_yield

The function prototype of the sched_yield API is :

#include <pthread.h>

int sched yield (void);

• *The sched_yield function is called by a thread to yield its execution to other threads with the same priority. This function returns 0 on success and -1 when fails.*

Thread Scheduling

• An application can be implemented as a set of threads, which cooperate and execute concurrently in the same address space. Four general approach for multiprocessor thread scheduling are as follows:

1. Load sharing
2. Gang scheduling
3. Dedicated processor scheduling
4. Dynamic scheduling

1. Load sharing

• Threads are not assigned to a particular processor. When a processor is idle, it selects a thread from a global queue serving all processors.

b. In this method, load is evenly distributed among processors. No centralized scheduler is required.

• This policy is implemented in different ways: FCFS and priority method is used. These are depends upon queue structure.

Three different versions of load sharing :

a.First-come-first-served (FCFS): On arrival of the thread, it is placed at the end of the shared queue. Next ready thread is selected when processor is idle. Processor executes the thread until its completion or blocking.

b. Smallest number of threads first: Shared ready queue is organized according to the priority. Processor selects highest priority thread with the smallest number of unscheduled threads. If the priority of the two thread is same, then FCFS method is applied.

C.Preemptive smallest number of threads first: Highest priority is given to jobs with the smallest number of unscheduled threads.

Advantages of Load sharing

1. No centralized scheduler is required.
2. Load distribution is equal.
3. Global queue is maintained by the system.

Disadvantages of Load sharing

1. It enforces mutual exclusion because of memory.
2. High degree of coordination is required between the threads of a program.
3. Caching becomes less efficient.

2. Gang scheduling

• With gang scheduling, a set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis.

• Synchronization blocking may be reduced or increased.

• If closely related processes executes in parallel, synchronization blocking may be reduced.

• Set of related threads is scheduled to run on a set of processors.

• Gang scheduling has three parts.

1. Group of related threads are scheduled as a unit, a gang.
2. All members of a gang run simultaneously on different time shared CPUs.
3. All gang members start and end their time slices together.

• The trick that makes gang scheduling work is that all CPU are scheduled synchronously. This means that time is divided into discrete quanta.

• An example of how gang scheduling works is given in the Fig. 2.7.1. Here we have a multiprocessor with six CPU being used by five processes, A through E, with a total of 24 ready threads.

		CPU					
		0	1	2	3	4	5
Time slot	0	A_0	A_1	A_2	A_3	A_4	A_5
	1	B_0	B_1	B_2	C_0	C_1	C_2
	2	D_0	D_1	D_2	D_3	D_4	E_0
	3	E_1	E_2	E_3	E_4	E_5	E_6
	4	A_0	A_1	A_2	A_3	A_4	A_5
	5	B_0	B_1	B_2	C_0	C_1	C_2
	6	D_0	D_1	D_2	D_3	D_4	E_0
	7	E_1	E_2	E_3	E_4	E_5	E_6

Fig. 2.7.3 Gang scheduling

• During time slot 0, threads Ao through A5 are scheduled and run.

To cops

• During time slot 1, threads Bo, B1, B2, Co, C1, C2 are scheduled and run.

• During time slot 2, D's five threads and Eo get to run.

• The remaining six threads belonging to process E run in the time slot 3. Then the cycle repeats, with slot 4 being the same as slot 0 and so on.

• Gang scheduling is useful for applications where performance severly degrades to toe when any part of the application is not running.

3. Dedicated processor assignment

• When application is scheduled, its threads are assigned to a processor.

• Some processor may be idle and no multiprogramming of processors.

• Provides implicit scheduling defined by assignment of threads to processors. For the duration of program executation, each program is allocated a set of processors equal in number to the number of threads in the program. Processors are chosen from the available pool.

4. Dynamic scheduling

• Number of threads in a process are altered dynamically by the application.

• Operating system and the application are involved in making scheduling decisions. The OS is responsible for partitioning the processors among the jobs.

Operating system adjusts load to improve the use

1. Assign idle processors.

lave of Process Management

2. New arrivals may be assigned to a processor that is used by a job currently using more than one processor.

3. Hold request until processor is available.

4. New arrivals will be given a processor before executing running applications.

Types of Thread

• Threads are of two types:

1. User level thread

2. Kernel level thread

• All modern operating system support threading model. Implementation of thread will change according to the operating system.

User Level Thread

• User level thread uses user space for thread scheduling. These threads are transparent to the operating system. User level threads are created by runtime libraries that cannot execute privileged instructions.

User-level threads have low overhead but it can achieve high performance in computation. User-level threads are managed entirely by the run-time system.

User-level threads are small and faster. A thread is simply represented by registers, stack and small thread control block.Fig. 2.8.1 shows User level thread.

APEC

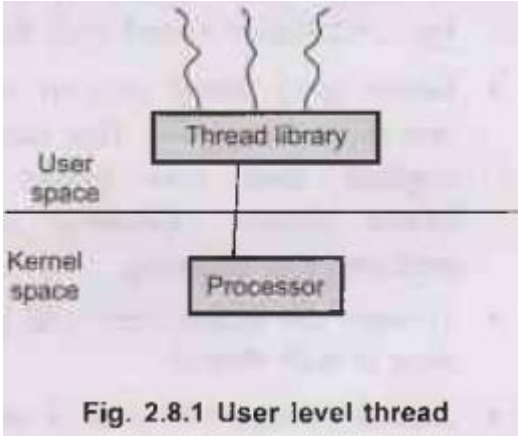


Fig. 2.8.1 User level thread

• The code for creation and destroying thread, message passing and data transfer, thread scheduling is included into thread library. Kernel is unaware of user level thread.

• User level threads do not invoke the Kernel for scheduling decision.

• User level thread are also called many to one mapping thread because the operating system maps all threads in a multithreaded process to a single execution context. The operating system considers as each multithreaded processes as a single execution unit.

• Example: POSIX Pthreads and Mach C-threads.

Advantages:

1. Kernel mode privilege does not require for thread switching.
2. These threads are fast to create and manage.
3. User level thread works even if the OS does not support threads.
4. User level threads are more portable.
5. Threading library controls flow of thread.

Disadvantages:

1. If thread blocks, the Kernel may block the all threads.
2. Not suitable for multiprocessor system.
3. User level threads also do not support system wide scheduling priority.

Kernel Level Thread

• In Kernel level thread, thread management is done by Kernel. Operating systems support the Kernel level thread. Since Kernel managing threads,

• Kernel can schedule another thread if a given thread blocks rather than blocking the entire processes. Fig. 2.8.2 shows Kernel level thread.

• Kernel level thread support one to one thread mapping. This mapping requires each user thread with kernel thread. Operating system performs this mapping.

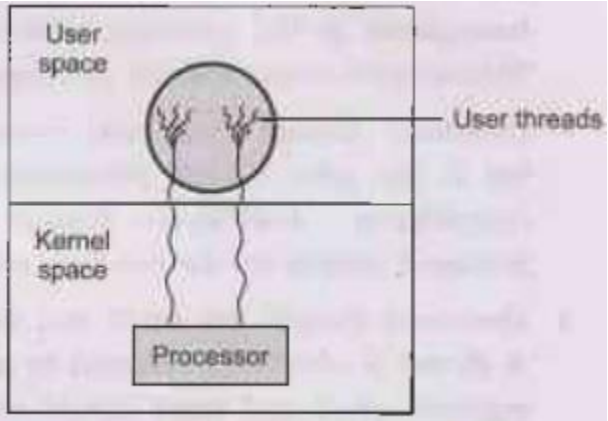


Fig. 2.8.2 Kernel level thread

• Threads are constructed and controlled by system calls. The system knows the state of each thread.

• Thread management code is not included in the application code. It is only API to the Kernel thread. Windows operating system uses this facility.

• Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

• Kernel performs scheduling on a thread basis. The Kernel support for scheduling and management, thread creation only in Kernel space.

• Kernel level threads are slower than user level threads.

Example: Windows 95/98/NT, Sun Solaris and Digital UNIX.

Advantages :

1. Each thread can be treated separately.

2. A thread blocking in the Kernel does not block all other threads in the same process.

3. Kernel routines itself as multithreaded.

Disadvantages:

1. Slower than the user level thread.

2. There will be overhead and increased in Kernel complexity.

Difference between User Level Thread and Kernel Level Thread

Sr. No.	User level threads	Kernel level threads
1.	User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
2.	Implemented by a thread library at the user level.	Operating system support directly to Kernel threads.
3.	User level thread can run on any operating system.	Kernel level threads are specific to the operating system.
4.	Support provided at the user level called user level thread.	Support may be provided by Kernel is called Kernel level threads.
5.	Multithread application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.
6.	Example : POSIX Pthreads and Mach C-threads	Example : Windows 95/98/NT, Sun Solaris and Digital UNIX
7.	User level threads are also called many to one mapping thread.	Kernel level thread support one to one thread mapping.

Multithread Models

• Operating system uses user level thread and kernel level thread. User level threads are managed without kernel support. Operating system support and manage the kernel level threads.

• Modern operating system support kernel level threads. Kernel performs multiple simultaneous tasks in operating system. In most of the application, user threads are mapped with kernel level threads.

• Different methods of mapping is used in operating system are as follows:

1. One to one
2. Many to one
3. Many to many

1. One to one

• In this model, one user level thread maps with one kernel level thread. If there are three user threads then system creates three separate kernel thread. This model provides more concurrency because of separate thread. Fig. 2.9.1 shows one to one mapping.

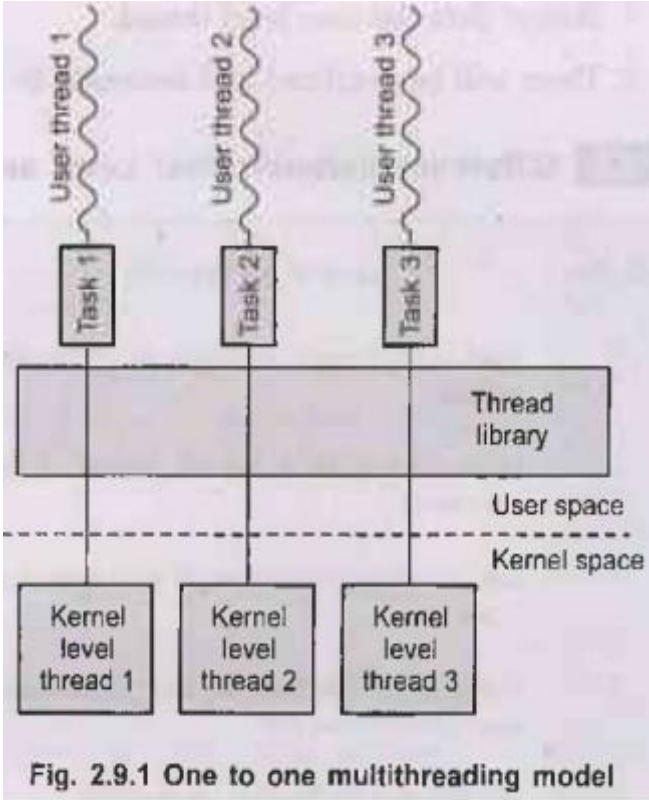


Fig. 2.9.1 One to one multithreading model

• In this method, the operating system allocates data structure that represents kernel threads. Here multiple threads are run in parallel on multiprocessor system.

• Number of threads in the system increases, the amount of memory required is also increases.

• Windows 95/XP and Linux operating system uses this one to one thread mapping.

• Only overhead in this method is creation of kernel level thread for user level thread. Because of this overhead, system performance is slowdown.

2. Many to one

• Many to one mapping means many user thread maps with one kernel thread. Fig. 2.9.2 shows many to one mapping.

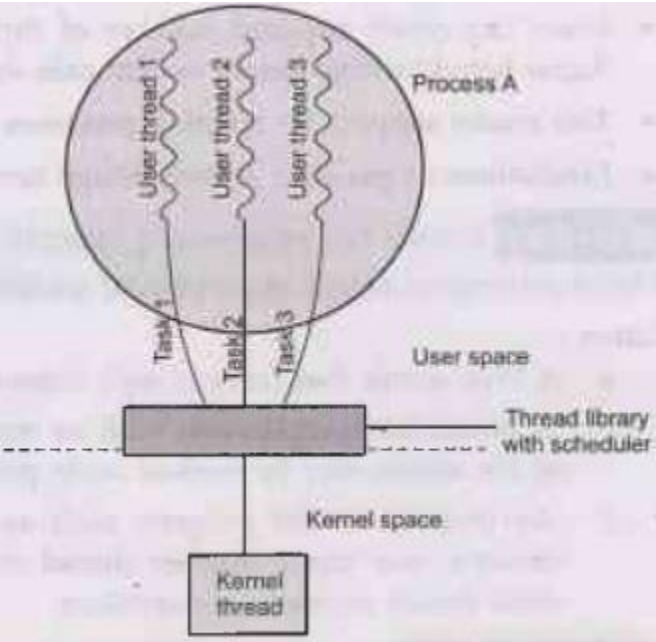


Fig. 2.9.2 Many to one multithreading model

• Operating system blocks the entire multi-threaded process when a single thread blocks because the entire multi-threaded process is a single thread of control. So when operating system re receive any a blocking I/O request, it blocks the entire

• Thread library handle thread management in user space. The many-to-one model does not allow individual processes to be split across multiple CPUs.

•This type of relationship provides an effective context-switching environment, easily implementable even on simple kernels with no thread support.

• Green threads for Solaris and GNU portable threads implement the many-to-one model in the past, but few systems continue to do so today.

• Advantage: System performance is improved by customizing the thread algorithm.

3. Many to many

• In the many to many mapping, many user-level threads maps with equal number of kernel threads. Fig. 2.9.3 shows many to many thread mapping.

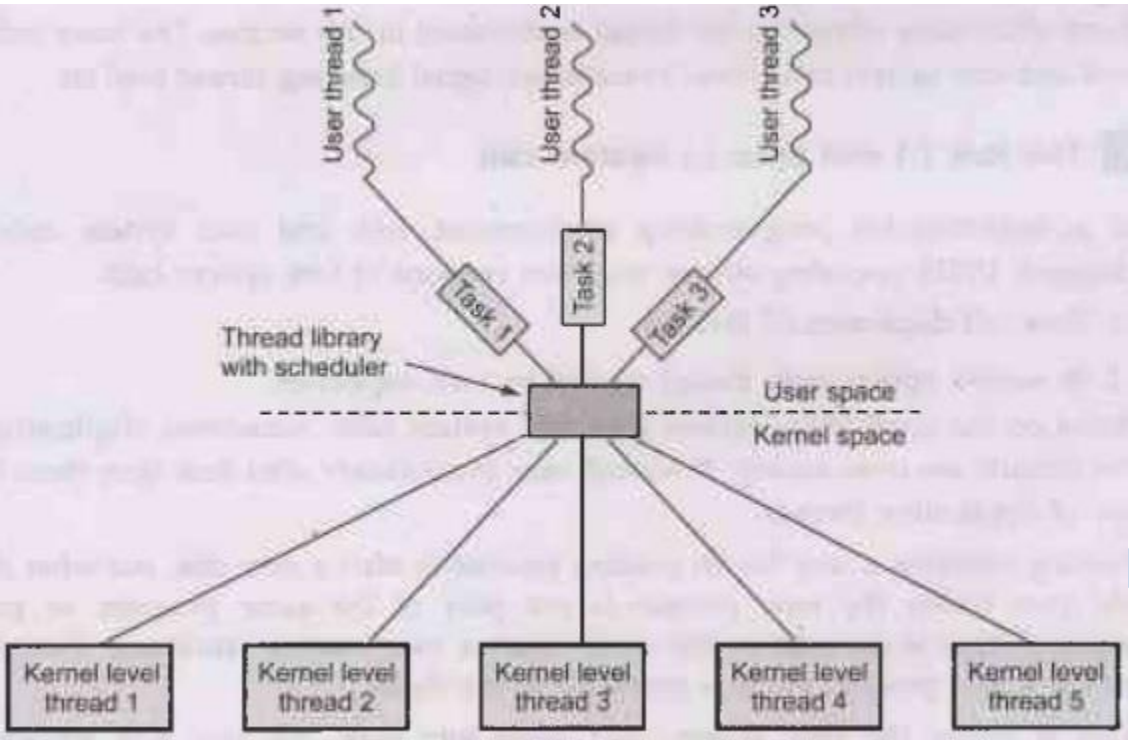


Fig. 2.9.3 Many to many multithreading model

• Many to many mapping is also called M-to-N. thread mapping. Thread pooling is used to implement this method.

• Users can create required number of thread and there is no limitation for user. Again here blocking Kernel system calls do not block the entire process.

• This model support for splitting processes across multiple processors.

Limitations: Operating system design becomes complicated.

Example 2.9.1: Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

Solution:

a. A Web server that services each request in a separate thread.

b. A parallelized application such as matrix multiplication where different parts of the matrix may be worked on in parallel.

C. An interactive GUI program such as a debugger where a thread is used to monitor user input, another thread represents the running application, and a third thread monitors performance.

Process Synchronization

• Logical control flows are concurrent if they overlap in time. This general phenomenon is known as concurrency. Concurrency refers to any form of interaction among processes or threads. The familiar examples are hardware exception handlers, processes and UNIX signal handlers.

• Modern operating systems can handle more than one process at a time. System scheduler manages processes and their competition for the CPU. The role of memory manager is to manage sharing of main memory between active processes in the system. Look at how processes coexist and communicate with each other on modern computer.

• Concurrency is good for users because working on the same problem, simultaneous execution of programs, background execution.

• Concurrency creates problems also because of access to shared data structures, deadlock due to resource contention and enabling process interaction.

• Applications that use application-level concurrency are known as concurrent Bish programs. Modern operating systems provide three basic approaches for building concurrent programs: Processes, I/O multiplexing and threads.

• Concurrency means that two or more calculations happen within the same time oil frame and there is usually some sort of dependency between them. Parallelism means that two or more calculations happen simultaneously.

• But concurrency describes a problem, while parallelism describes a solution. Parallelism is one way to implement concurrency, but it is not the only one. Another popular solution is interleaved processing.

Principle of Concurrency

• Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of Scooperating processes.ug or ai asges

•Concurrency arises in the same way at different levels of execution streams. Following are the example of concurrency in different types of operating systems:

1. Concurrency in multiprogramming: An interaction between multiple processes running on one CPU.

2. Concurrency in multithreading: An interaction between multiple threads running in one process.

3. Concurrency in multiprocessors: An interaction between multiple CPUs running multiple processes or threads.

4. Concurrency in multi-computers: An interaction between multiple computers running distributed processes or threads.

• Java is a concurrent programming language.

• Process synchronization is required in uni-processor system, multiprocessor system and network. If more than one thread exists in a system at the same time, then the threads are said to be concurrent. Synchronization problems can occur whenever two or more concurrent processes use any shared resource.

• Cooperating process share the logical address space.

Is concurrency is possible without parallelism?

• Yes, concurrency is possible without parallelism.

Concurrency means that more than one process or thread is progressing at the same time. However, it does not imply that the processes are running simultaneously. The scheduling of tasks allows for concurrency, but parallelism is supported only on systems with more than one processing core.

Race Condition

• Race condition occurs when two or more operations occur in an undefined manner. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when,are called race condition.

• There is a "race condition" if the outcome depends on the order of the execution. The outcome depends on the CPU scheduling or "interleaving" of the threads.

• Race condition should be avoided because they can cause fine errors in applications and are difficult to debug.

• In banking system balance is a shared variable. After depositing the amount, it update by bank employee balance = balance + amount. At the same time, some amount is transfer then it becomes balance balance amount. These two tasks gain are in a race to write variable balance. In this the process that updates last determines the final value of balance.

• The concurrent execution of two processes is not guaranteed to be determinate, since different executions of the same program on the same data may not produce the same result.

Operating system concerns

• Design and management issue for concurrency are as follows:

1. Track of various processes is kept by operating system.

2. Operating systems allocates and deallocate software and hardware resources to active process.

3. Operating system must protect user data and physical resources from un-authorized process.

4. Process execution speed do not depends upon other process

Critical Section Problem

• A critical section is a block of code that only one process at a time can execute; so, when one process is in its critical section, no other process may be in its critical section. The critical section problem is to ensure that only one process at a time is allowed to be operating in its critical section.

• Critical section means, process may change some common variable, writing files, updating memory location, updating a process table etc. When is accessing shared modifiable data, it is said to be in a critical section.

• Each process takes permission from operating system to enter into the critical section. Structure of process is as follows:

1. Entry section

2. Remainder section

3. Exit section

• **Entry section:** It is block of code executed in preparation for entering critical section.

• **Exit section:** The code executed upon leaving the critical section.

• **Remainder section:** Rest of the code is remainder section.

• Each process cycles through remainder, entry, critical, exit sections in this order.

• Consider the following example:

• **Problem:** Design a protocol to solve this problem.

Process 1	Process 2
Count = count + 1	Count = count – 1 Output: count = ?

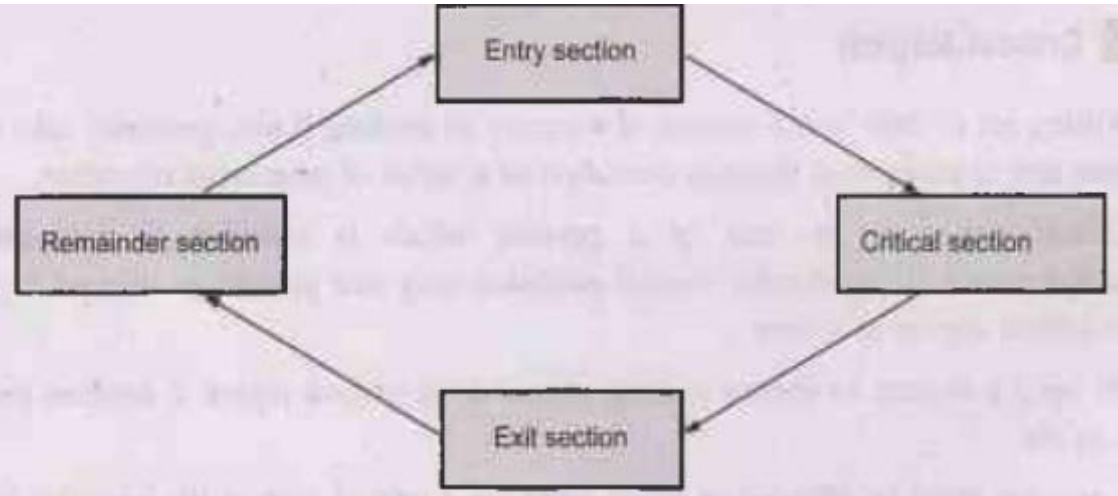


Fig. 2.11.1 Critical section

• **General framework:** Every solution will have the following layout:

• To prevent critical section problem, the system on should ensure that only one process at a time can execute the instruction in its critical section for a particular resource. If one process attempts to enter its critical section while another is in its own critical section, the first process should wait until the executing process exits the critical section.

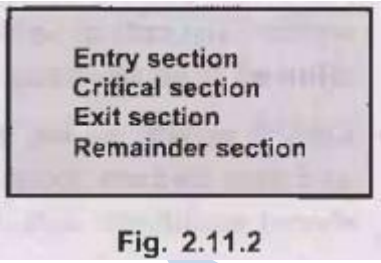


Fig. 2.11.2

• If a process in a critical section terminates by any reasons, then the operating system must release mutual exclusion so that other process may enter their critical sections.

Solution of critical section

• Solution to critical section problem must satisfy the mutual exclusion, progress and bounded waiting parameters.

• Software and hardware solution to critical section problem is available.

• Process using Kernel is active in the system at any given time. Race condition is with implementing a Kernel code. Kernel data structure maintains a list of all open files in the system. System always modifies this list depending upon the process open files and closed files.

• Operating system handles critical section problem by using Kernel. Kernel is classified as preemptive Kernel and non-preemptive Kernel.

Preemptive Kernel: If process running in kernel mode then also it is preempted.

• **Non-preemptive Kernel:** It does not allow process to preempt.

Critical Region

• Writing set of data into a section of memory or reading it out, generally take some time and is performed through execution of a series of primitives operation.

• Critical region is an area of a process which is sensitive to inter-process complications. To guarantee mutual exclusion only one process is allowed to enter its critical region at a time.

• OS need a system to ensure process cannot enter critical region if another process is in it's.

• A process must be allowed to finish work on a critical part of the program before other processes can have access to it. It is called a critical region because it is a

critical section and its execution must be handled as a unit. Other processes must wait before accessing critical region resources.

• Each conditional critical region is a "resource" that has a name and a set of variables that can only be accessed in that region. All variables can only be accessed within a conditional critical region.

Mutual Exclusion

• The need for mutual exclusion comes with concurrency. There are several kinds of In concurrent execution:

1. Interrupt handlers
2. Interleaved preemptively scheduled processes/threads
3. Multiprocessor clusters, with shared memory
4. Distributed systems

• Mutual exclusion methods are used in concurrent programming to avoid the ob simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections

• Requirement of mutual exclusion is that, when process P1 is accessing a shared resource R1 then on other process should be able to access resource R1 until process P1 has finished its operation with resource R1.

• Examples of such resources include files, I/O devices such as printers and shared data structures.

Approaches to implementing mutual exclusion:

1. Software method: Leave the responsibility with the processes themselves. These methods are usually highly error-prone and carry high overheads.

2. Hardware method: Special-purpose machine instructions are used for accessing shared resources. This method is faster but cannot provide complete solution. Hardware solution cannot give guarantee the absence of deadlock and starvation

3. Programming language method: Provide support through the operating system or through the programming language.

Requirements of mutual exclusion

1. At any time, only one process is allowed to enter in its critical section.

2. Solution is implemented purely in software on a machine.

3. A process remains inside its critical section for a bounded time only.

4. No assumption can be made about relative speeds of asynchronous concurrent processes.

5. A process cannot prevent any other process for entering into critical section.

6. A process must not be indefinitely postponed from entering its critical section.

Lock

• Lock is a shared variable. Software mutual exclusion required that a process read a variable to determine that no other process is executing a critical section,then set a variable known as lock. It indicates that the process is executing its critical section.

• Initially lock variable is initialized with zero. The process set it to 1 and enters its critical section.

Lock = 0 no process in the critical section

Lock = 1 process in the critical section

• Using simple lock variable, process synchronization problem is not solved. To avoid this, spinlock is used. A lock that uses busy waiting is called a spin lock.

Mutex

Mutex is used to ensure only one thread at a time can access the resource Bets protected by the mutex. The process that locks the mutex must be the one that unlocks it. Mutex is good only for managing mutual exclusion to some shared resource.

• Mutex is easy and efficient for implement.

• Mutex can be in one of two states: locked or unlocked

• Mutex is represented by one bit. Zero (0) means unlock andsteig represents locked. It uses two procedures.

• When a process needs access to a critical section, it checks the condition of mutex_locks. If the mutex is currently unlocked then calling process enters into critical section.

• If the mutex is locked, the calling process is enters into blocked state and wait until the process in the critical section finishes its execution.

• Mutex variables have only two states so they are simple to implement. Their use is limited to guarding entries to critical regions.

• A segment of code in which a thread may be accessing some shared variable is called a critical region.

• Mutex variable is like a binary semaphore. But both are not same.

Synchronization Hardware

Test and Set Operations

• Test and set is a single individual machine instruction. It is simply known as TS. It was introduced by IBM for its multiprocessing system.

• In a one machine cycle, it tests to see if the key is available and, if it is, sets it to unavailable. TS are a hardware solution for critical section problem.

• TS instruction eliminates the possibility of preemption occurring during the of interval. The instruction :

testAndset (p, q)

• The instruction reads the value of q, which may be either true or false. Then the value is copied into "p" and the instruction sets the value of "q" to true.

• A process P1 would test the condition code using TS instruction before entering a critical section. If no other process was in this critical section, then process P1 would be allowed to proceed and the condition code would be changed from zero to one.

• When process P1 exist the critical section, the condition code is reset to zero so another process can enter into critical section.

• If process P1 finds a busy condition code then it is placed in a waiting loop where it continues to test the condition code and waits until it is free.

• The "TestandSet" instruction is a tool that programmers use to simplify software solution to mutual exclusion but the instruction itself does not enforce mutual exclusion.

Advantages:

1. Simple to implement.
2. It works well for a small number of processes.
3. It can be used to support multiple critical sections..

Drawbacks:

1. It suffers from starvation.
2. There is possibility of busy waiting.
3. There may be deadlock.

Semaphores

• Semaphore is described by Dijkstra. Semaphore is a nonnegative integer variable that is used as a flag. Semaphore is an operating system abstract data type. It takes only integer value. Its used to solve critical section problem.

• Dijkstra introduced two operations (P and V) to operate on semaphore to solve process synchronization problem. A process calls the P operation when it wants to enter its critical section and calls V operation when it wants to exit its critical section. The P operation is called as wait operation and V operation is called as signal operation.

• A wait operation on a semaphore decease its value by one.

Waits: while S < 0

do loops;

S:= S-1;

• A signal operation increments its value:

signal:

S:= S+ 1;

• A proper semaphore implementation requires that P and V be indivisible operations. A semaphore operation is atomic. This may be possible by taking hardware support. The operation P and V are executed by the operating system in response to calls issued by any one process naming a semaphore as parameter.

• There is no guarantee that no two processes can execute wait and signal operations on the same semaphore at the same time.

Properties of semaphore:

1. Semaphores are machine independent.
2. Semaphores are simple to implement.
3. Correctness is easy to determine.
4. Semaphore acquire many resources simultaneously.
5. Can have many different critical sections with different semaphores.

Binary Semaphore

• Binary semaphore is also known as mutex locks. It deals with the critical section for multiple processes.

• Binary semaphore value is only between 0 and 1.

Counting semaphore

• Used with that resource which has a finite number of instances.

• It works over unrestricted domain.

• Nonbinary semaphore often referred to as either a counting semaphore or a general semaphore.

• Process waiting for semaphore is stored in the queue for binary and counting semaphore. Queue uses first in first out policy.

• A semaphore that does not specify the order in which processes are removed from the queue is called as weak semaphore. The process that has been blocked the longest is released from the queue first is called a strong semaphore.

• Semaphore can be implemented in user applications and in the kernel. Both operations can be implemented in the kernel by blocking waiting process to avoid busy waiting.

Busy Waiting

• Busy waiting is a situation in which a process is blocked on a resource but does not yield the processor. A busy wait keeps the CPU busy in executing a process even as the process does nothing.

• Busy waiting is also called as spin waiting.

Busy waiting waste CPU cycles that some other process might be able to use productively.

• To avoid busy waiting, a process waiting for critical section, they put into the blocked state. When process allowed entering into critical section, then process is kept in ready state.

• To overcome the need for bust waiting:

In wait() operation :

a. When process reads semaphore value and it is not positive value then process blocked itself for some time.

b. Semaphore keeps all blocked process in the waiting queue so process is kept in this queue,

c. CPU scheduler selects another process for execution.

In signal () operation :

a. The wakup () operation is used for restating the blocked processes. When any other process executes signal () operation, then blocked process state changed from blocked state to ready state.

b. This process is kept in the ready queue.

Drawbacks of Semaphore

1. They are essentially shared global variables.

2. Access to semaphores can come from anywhere in a program.

3. There is no control or guarantee of proper usage.

4. There is no linguistic connection between the semaphore and the data to which most have the semaphore controls access.

5. They serve two purposes, mutual exclusion and scheduling constraints.

Semaphore Programming

1. Semget( )

• To create a semaphore or gain access to one that exists, the semget system call is used.

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semget (key t key,intnsems, int semflg);

• The semget system call takes three arguments :

a. The first argument, key, is used by the system to generate a unique semaphore identifier.

b. The second argument, nsems, is the number of semaphores in the set.

c. The third argument, semflg, is used to specify access permission and/or special creation conditions.

• If the semget system call fails, it returns a - 1 and sets the value stored in errno.

• When a semaphore is first created, the kernel assigns to it an associated :

a. Semaphore Control Block (SCB),

b. A unique ID,

c. A value (binary or a count), and

d. A task waiting list

• A Kernel can support many different types of semaphores, including binary, counting, and mutual-exclusion (mutex) semaphores.

2. semctl () function

• The semctl system call allows the user to perform a variety of generalized control

operations on the system semaphore structure, on the semaphores as a set, and on individual semaphores.

• The function prototype is as follows:

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semunarg);

• The semctl system call takes four arguments :

1. The first argument, semid is a valid semaphore identifier that was returned by a previous semget system call.

2. The second argument, semnum, is the number of semaphores in the semaphore set (array), 0 means this number (index) is not relevant.

3. The third argument to semctl, cmd, is an integer command value. The cmd value directs semctl to take one of several control actions. Each action requires specific access permissions to the semaphore control structure.

4. The fourth argument to semctl, arg, is a union of type semun. Given the action specified by the preceding cmd argument, the data in arg can be one of of any the following four values:

a. An integer already was set in the val member of sem_union that used with SETVAL to indicate a change of specific value for a particular semaphore within the semaphore set.

b. A reference to a semid_ds structure where information is returned when IPC_STAT or IPC_SET is specified.

c. A reference to an array of type unsigned short integers; the array is used either to initialize the semaphore set or as a return location when specifying GETALL.

d. A reference to a seminfo structure when IPC_INFO is requested.

• If semctl fails, it returns a value of 1 and sets errno to indicate the specific error.

3. semop() function

Additional operations on individual semaphores are accomplished by using the semop system call. Its syntax is :

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops); for.

• The semop system call takes three arguments

a .The first argument, semid, is a valid semaphore identifier that was returned by a previous successful semget system call.

b. The second argument, sops, is a reference to the base address of an array of semaphore operations that will be performed on the semaphore set associated with by the semid value.

c. The third argument, nsops, is the number of elements in the array of semaphore operations.

• If semop fails, it returns a value of - 1 and sets errno to indicate the specific error. • If a semop call must block after completing some of its component operations, the kernel rewinds the operation to the beginning to ensure atomicity of the entire call.

• When the sem_op value is negative, the process specifying the operation is attempting to decrement the semaphore. The decrement of the semaphore is used to record the acquisition of the resource affiliated with the semaphore.

• When a semaphore value is to be modified, the accessing process must have altered permission for the semaphore set.

• When the sem_op value is positive, the process is adding to the semaphore value. Os The addition is used to record the return (release) of the resource affiliated with the semaphore.

• Again, when a semaphore value is to be modified, the accessing process must have alter permission for the semaphore set.

• When the sem_op value is zero, the process is testing the semaphore to determine if it is at 0.

• When a semaphore is at 0, the testing process can assume that all the resources affiliated with the semaphore are currently allocated.

4. sem_post()

Syntax :

#include <semaphore.h>

int sem_post(sem_t *sem);

• The sem_post() function unlocks the specified semaphore by performing a semaphore unlock operation on that semaphore. When this operation results in a positive semaphore value, no threads were blocked waiting for the semaphore to be unlocked; the semaphore value is simply incremented.

• When this operation results in a semaphore value of zero, one of the threads waiting for the semaphore is allowed to return successfully from its invocation of the sem_wait() function. The sem_post() interface is reentrant with respect to signals and may be invoked from a signal-catching function.

5. sem_init()

Syntax :

#include <semaphore.h>

• int sem_init(sem_t *sem, int pshared, unsigned int value);

The sem_init() function is used to initialize the semaphore's value. The pshared argument must be 0 for semaphores local to a process. The value argument specifies the initial value for the semaphore.

• The pshared argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.

• If pshared has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads.

• If pshared is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory.

Classical Problems of Synchronization

Dinning Philosopher Problem

• Dinning philosophers problem is one of classical process synchronization problem. Here five philosophers are seated around a circular table. They spend their lives in thinking and eating. Five plates with five forks are kept on the circular table.

• While philosophers think, they ignore the eating and do not require fork. When a philosopher decides to eat, then he/she must obtain a two fork, one from left side fork and another from right side fork. Philosophers can pick up only a single fork at one time.

After consuming a food, the philosophers replace the fork and resumes thinking. A philosophers to the left or right of a ow dinning philosopher cannot eat while the dinning philosophers is eating, since forks are a shared resource.

Fig. 2.14.1 shows seating arrangement of five philosophers.

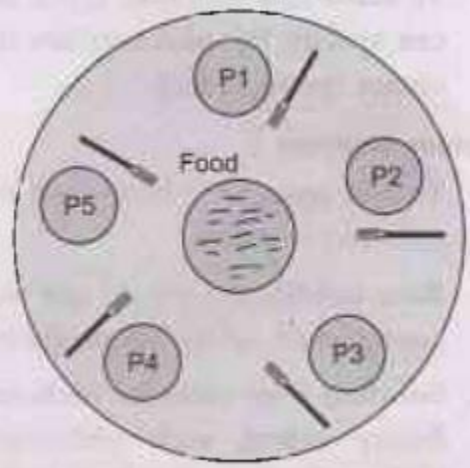


Fig. 2.14.1 Dinning philosophers

Consider the following code :

```
void dinning_Philosopher( )
while (true)
{
thinking ( );
eating ( );
}
void eating ()
{
takeleftfork();
takerightfork( );
eatfood(delay);
putrighttfork( );
putleftfork( );
```

• Here philosophers operates asynchronously and concurrently, it is possible for each philosophers to enter into eating mode. The procedure takeleftfork( ) waits until the specified fork is available and then seizes it. This is not possible because of the following reasons.

APEC

• Each philosopher will hold exactly one fork, and no forks will remain available on the table. The philosophers will all deadlock.

• To solve this problem, write some extra code in takerightfork () procedure. User can specify the philosophers to put down the left fork if that philosophers cannot obtain the right fork.

Problem analysis :

• Shared resource: Here each fork is shared by two philosophers so we called it is a shared resource.

• Race condition: we do not want a philosopher to pick up a fork that has already been picked up by his neighbor. This is a race condition.

• Solution: we consider each fork as a shared item and protected by a mutex lock. Before eating, each philosopher first lock left fork and then right fork. If philosopher acquires both locks successful, then philosophers have two locks (two fork) so he can eat a food. After finishes easting, this philosopher releases both fork, and thinks.

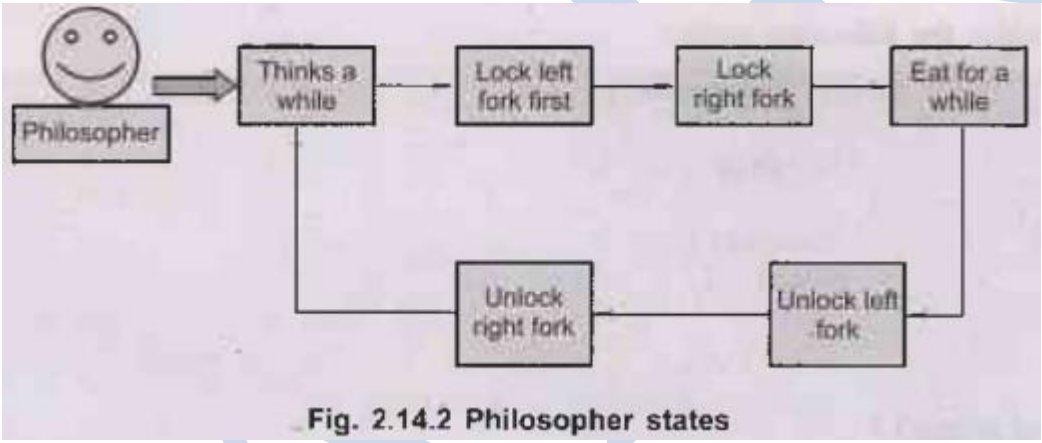


Fig. 2.14.2 Philosopher states

Some other alternatives :

1. Pick up the left fork, if the right fork is not available for a given time, put the left fork down, wait and try again. Even if each philosopher waits a different random time, an unlucky philosopher may starve.

2. Require all philosophers to acquire a binary semaphore before picking up any forks. This guarantees that no philosopher starves but limits parallelism dramatically.

• Because we need to lock and unlock a chopstick, each chopstick is associated with a mutex lock. Since we have five philosophers who think and eat simultaneously, we need to create five threads, one for each philosopher. Since

each philosopher must have access to the two mutex locks that are associated with its left and right to chopsticks, these mutex locks are global variables.

Reader-Writer Problem

• When two types of processes need to access a shared resource such as a file or database. They called these processes reader and writers.

• For example in railway reservation system, readers are those who want train ar schedule information. They are called reader because readers do not change the content of the database. Many readers may access the database at once. There is sb no need to enforce mutual exclusion among readers.

• The writers are those who making reservations on a particular train. Writer can 19 modify the database, so it must have exclusive access. When writer is active, no or other readers or writers may be active. Therefore, it must enforce mutual exclusion

• Reader writer problem is similar to one which a file is to be shared among a set of processes. If a process want only to read the file, then it may share the file with any other process that also wants to read the file. If the writer wants to append the file, then no other process should have access to the file when the writer has access to i

• If reader having higher priority than the writer, then there will be starvation with writers. For writer having higher priority than reader then starvation with readers.

The Producer Consumer Problem

• Producer - consumer problem is example classic problems of synchronization. Producer process produce data item that consumer process consumes later.

• Buffer is used between producer and consumer. Buffer size may be fixed or variable. The producer portion of the application generates data and stores it in a buffer, and the consumer reads data from. the buffer. Fig. 2.14.3 shows producer-consumer problem.

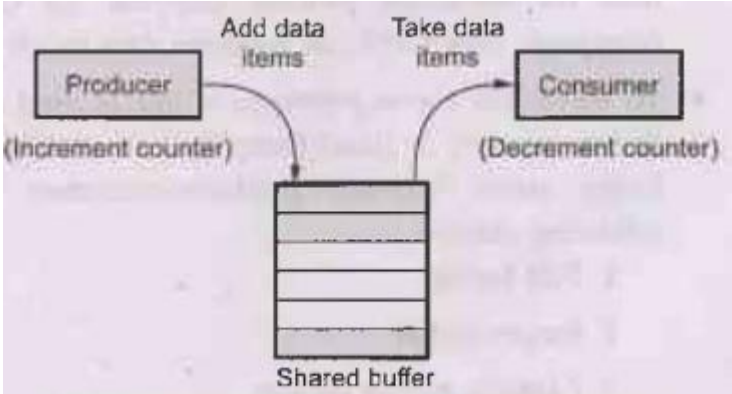


Fig. 2.14.3 Producer - consumer problem

• Producer-consumer is also called bounded buffer problem. There exist several producers and consumers.

The producer-consumer problem shows the need for synchronization in systems where many processes share a resource. In the problem, two processes share a fixed-size buffer. One process produces data items and deposits it in the buffer, while the other process consumes data items from the buffer.

• The producer cannot deposit its data if the buffer is full. Similarly, a consumer cannot retrieve any data if the buffer is empty. If the buffer is not full, a producer can deposit its data. The consumer should be allowed to retrieve a data item if buffer contains.

• The consumer retrieves a data item, so the data from buffer start decreasing. When buffer becomes empty, the producer should be allowed to deposit its data.

• Problem arises when the buffer is full and producer wants to add a new data item. Solution to this problem is that producer goes to sleep until consumer removes data item from the buffer. Similar condition exists with the consumer. Consumer wants to consume data item from the buffer, but buffer is empty then consumer go to sleep. It wakeup when producer add some data items into the buffer.

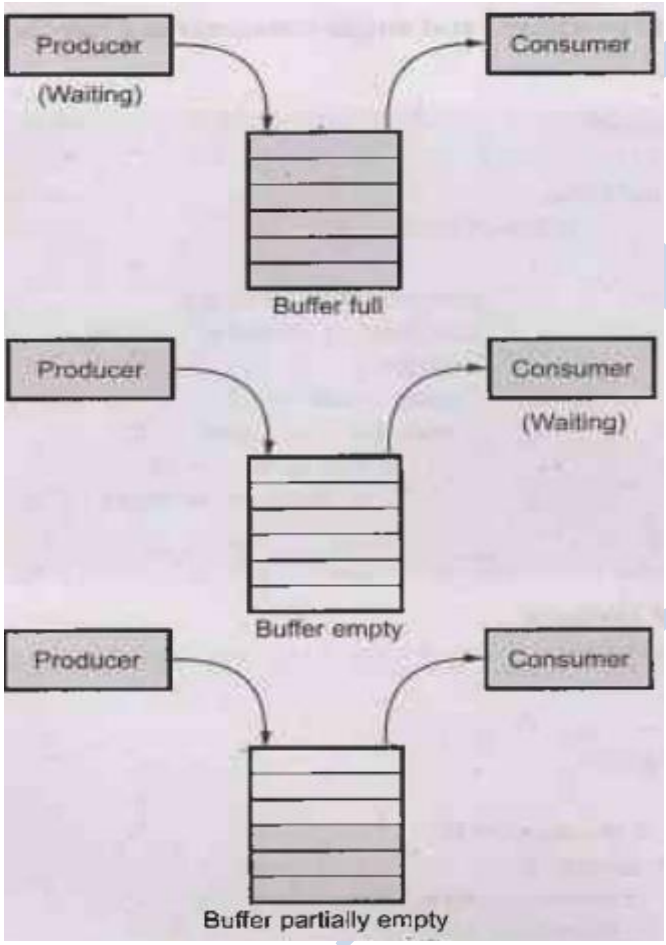
• In order to synchronize these processes, both procedure and consumer are blocked on some condition. The producer is blocked when the buffer is full, and the consumer is blocked when the buffer is empty.

• Example of procedure consumer problem:

1. Printing word file: When user gives the print command for printing file, a word processor spools data to a buffer and that data are subsequently consumed by the printer as it prints the document. Printer stops printing when buffer is empty.

2. A compiler may produce assembly code, which is consumed by an assembler. Speed of the procedure and consumer is different. The producer process is faster than the consumer process. Suppose the CPU is producer and line printer is consumer, then CPU can generate data much faster than a line printer.To solve this above problem, buffer is used in between procedure and consumer. (Buffer capacity is fixed (bounded) or variable (unbounded). Fig. 2.14.4 shows the buffer states between producer-consumer problem. Buffer is in any of the following states:

1. Full buffer
2. Empty buffer
3. Partially empty buffer



• A solution to the producer-consumer problem satisfy the following conditions:

1. When buffer is full, producer must wait. Producer do not produce data item.
2. When buffer is empty, consumer must wait.
3. Mutual exclusion is required for accessing the buffer.

• Producer-consumer problem is solved by using semaphore, mutex and monitor. Mutual exclusion must be enforced on the buffer itself.

• Each data item produce by producer to the shared buffer must be consumed exactly once by the consumer. Data item must be consumed in the same order (First in First Out Order) in which it is out into the buffer.

• If there is no synchronization between producer and consumer then data can be lost if the producer add new data item into the buffer before the consumer consumes the previous data item.

• We use variable counter to keep track of the number of data items in the buffer.

Solution to single producer and single consumer is given below:

1. Code for producer

```
producer (void)
{
int  item;
while (TRUE)
{
produce item(&item);
produce if (counter == N)
sleep();
enter item(item);
counter counter + 1;
if (counter == 1)
wakeup(consumer);
}
}
```

2. Code for consumer process

```
consumer(void)
{
int  item;
while (TRUE)
{
```

```
if (count == 0) sleep();
remove_item(&item);
counter counter - 1;
if (count == N-1)
wakeup(producer);
consume_item(item);
}
}
```

• Initially we assume that the buffer is empty. Consumer read the counter and context switch occurs. The producer process produce data item and increments the counter by one. After incrementing the counter, it sends a wakeup signal to the consumer. Here one more context switching occurs and consumer process runs.

• Solution to producer consumer problem by using binary semaphore :

```
void main()
{
count = 0;
int i;
binarysemaphore s = 1;
int producer ()
{
While (true)
{
produce_data_item( );
semwaitB(s);
append();
count count + 1;
if (count == 1)
semSignalB(delay);
semSignalB(s);
int consumer ()
```

```
{
int P;
semWaitB(delay);
while (true)
{
semWaitB(s);
consume();
count count - 1;
p = count; semSignalB(s); consumedata();
if (p == 0) semWaitB(delay);
of bewollsei
```

University Question

1. Consider a situation where we have a file shared between many people.

If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her. However if some person is reading the file, then others may read it at the same time.

a) What kind of situation is this?

b) Consider the following problem parameters to solve this situation.

Problem parameters:

1) One set of data is shared among a number of processes.

2) Once a writer is ready, it performs it write. Only one writer: may write t write at a time.

3) If a process is writing no other process can read it.

4) If at least one reader is reading, no other process can read it.

5) Readers may not write and only read.

Ans. It is reader and writer problem. Here we consider reader having higher priority than writer. Reader should not wait if the share is currently opened for reading.

• To implement this problem three variables are used; readcount, mutex and wrt.

Semaphore function: wait() and signal()

semophore mutex, wrt;

int readcount;

• For writer process:

1. Writer requests the entry to critical section.

2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.

3. It exits the critical section.

do {

// writer requests for critical section

wait(wrt);

// performs the write and leaves the critical section

signal(wrt);

} while(true);

•For reader process:

1. Reader requests the entry to critical section.

2.If allowed: It increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.

• It then, signals mutex as any other reader is allowed to enter while others are already reading.

• After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.

3. If not allowed, it keeps on waiting.

do {

// Reader wants to enter the critical section

wait(mutex);

// The number of readers has now increased by 1

readcount = readcount + 1;

// there is atleast one reader in the critical section, this ensure no writer

can enter if there is even one reader

// thus we give preference to readers here

HSM if (readcnt==1)

// other r neaders can enter while this current reader is inside

// the critical section

signal(mutex);

// current reader performs reading here

wait(mutex); // a reader wants to leave

readcount = readcount -1;

// that is, no reader is left in the critical section,

if (readcount == 0)

signal(wrt);

// writers can enter

signal(mutex); // reader leaves

} while(true);

Semaphore 'wrt' is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

Monitors

• Monitor is an object that contains both data and procedures needed to perform allocation of a shared resource. It is a programming language construct that support both data access synchronization and control synchronization.

• Fig. 2.15.1 shows structure of monitor. (See Fig. 2.15.1 on next page.)

• Monitor is implemented in programming languages like Pascal, java and C++. Java makes extensive use of monitors to implement mutual exclusion.

• Monitor is an abstract data type for which only one process may be executing a procedure at any given time. Monitor is a collection of procedure, variables and data structure. Data inside the monitor may be either global to all routines within the monitor or local to a specific routine.

• Monitor data is accessible only within the monitor. A shared data structure can be protected by placing it in a monitor. If the data in a monitor represents some

resource, then the monitor provides a mutual exclusion facility for accessing the resource. A procedure defined within a monitor can access only those variable declared locally within the monitor and its formal parameters.

• When a process calls a monitor procedure, the first few instruction of the procedure will check to see if any other process is currently active within the monitor. If process is active then calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

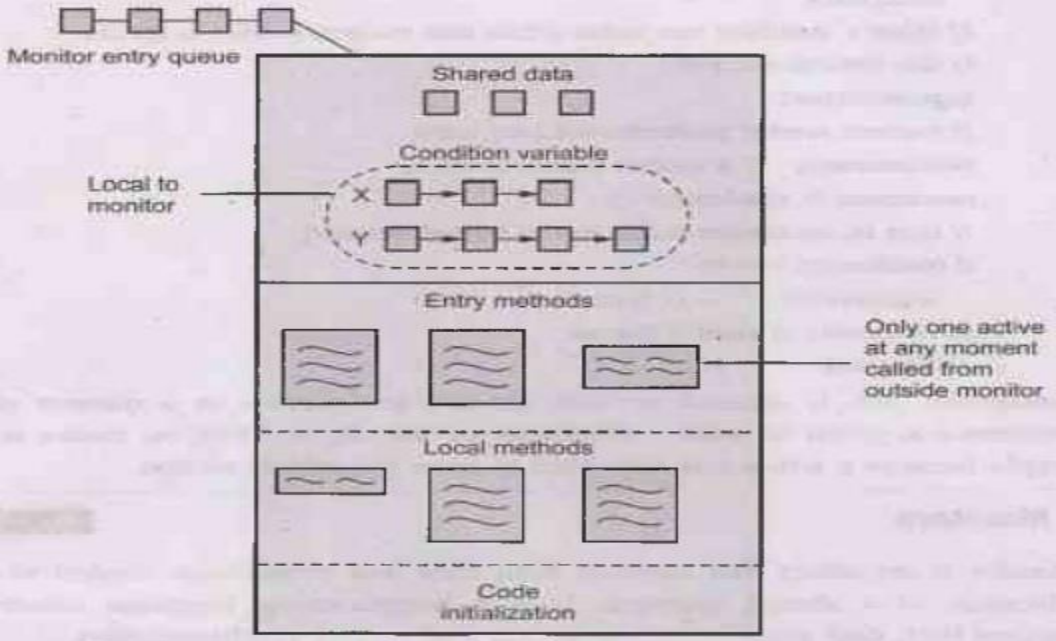


Fig. 2.15.1 Monitor structure

• Monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Every conditional variable has an associated queue. Condition variables operates on two

cwait(condition variable)

csignal (condition variable)

• cwait: It suspend execution of the calling process on condition.

• csignal: Resume execution of some process blocked after a cwait on the same condition.

• The cwait must come before the csignal.

• Fig. 2.15.2 shows monitor structure with condition variables.

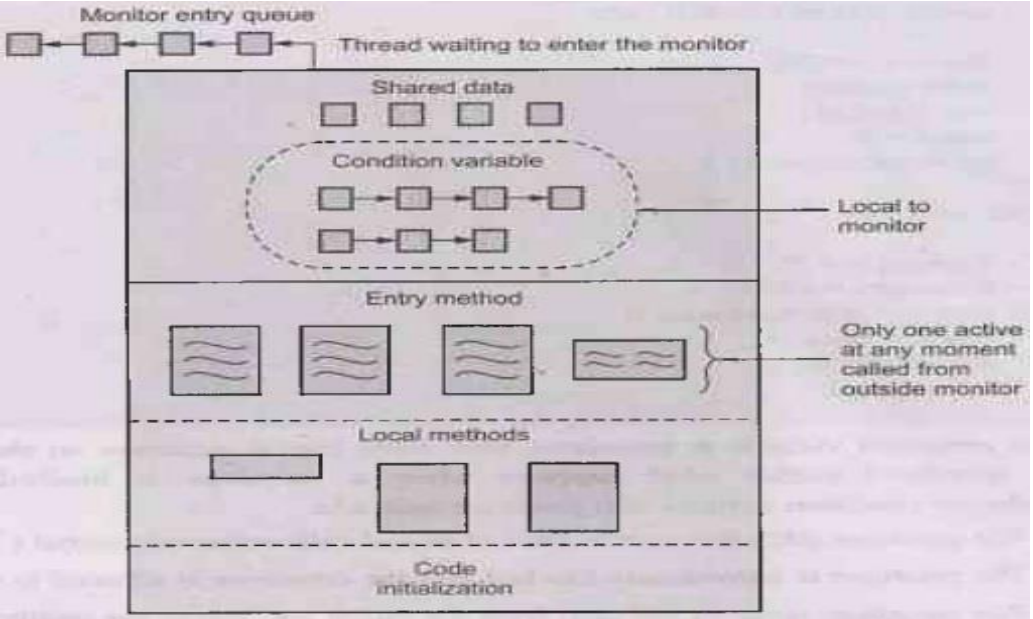


Fig. 2.15.2 Monitor structure with condition variable

• CPU is a resource that must be shared by all processes. The part of the kernel that apportions CPU time between processes is called the scheduler.

• A condition variable is like a semaphore, with two differences:

1. A semaphore counts the number of excess up operations, but a signal operation on a condition variable has no effect unless some process is waiting. A wait on a condition varibale always blocks the calling process.

2. A wait on a condition variable automatically does an up on the monitor mutex and blocks the caller.

Example: Solve producer consumer problem by using monitors

Bounded Buffer problem using monitors

```
monitor Bounded Buffer
{
private Buffer b = new Buffer (20);
private int count = 0;
private condition nonfull, nonempty;
public void add (object item)
}
if(count == 20)
nonfull.wait();
b.add (item);}
```

```
count ++;
nonempty. signal ();
}
public object remove ()
{
if (count == 0)
nonempty.wait();
item result = b.remove();
count count-1;
nonfull.signal();
return result;
```

• Each condition variable is associated with some logical condition on the state of the monitor. Consider what happens when a consumer is blocked on the nonempty condition variable and producer calls add.

1. The producer adds the item to the buffer and calls nonempty signal ( ).

2. The producer is immediately blocked and the consumer is allowed to continue. 3. The consumer removes the item from the buffer and leaves the monitor.

4. The producer wakes up and since the signal operation wait the last statement in add, leaves the monitor.

• Monitors are a higher level concept than P and V. They are easier and safer to use but less flexible. Many languages are not supported by the monitor. Java is making monitors much more popular and well known.

Example 2.15.1 Solve the reader-writer problem using monitor with readers priority.

Solution :

```
readers-writers: monitor;
begin
integer readercount;
condition okread, okwrite;
boolean busy;
procedure startread;
```

```
begin
if busy then ok read.wait;
readercount := readercount+1;
okread.signal;
end startread;
procedure endread;
begin
reader count:= readercount-1;
if      reader count = 0 then okwrite.signal;
end endread;
procedure startwrite;
begin if busy OR readercount ≠ 0 then okwrite.wait;
busy = true;
end startwrite;
procedure endwrite;
begin
busy := false;
if okread.queue then okread.signal
else okwrite.signal
end endwrite;
begin (*Initialisation*)
readercount :=0;
busy: =false;
end;
end readers-writers;
```

Example 2.15.2 Solve procedure-consumer problem with monitors.

Solution :

```
monitor procedure-consumer
condition full empty;
integer count;
```

```
Procedure insert (item: integer);
begin
if count = N then wait (full);
insert-item(item);
count: = count+1;
if count = 1 then signal (empty)
end;
function remove:integer;
begin
if count = 0 then wait (empty);
remove = remove_item;
count = count-1;
if count = N-1 then signal (full);
end;
count := 0;
end monitor;
procedure producer;
begin
while true do
begin
item = produce_item;
producer_consumer.insert(item)
end
end;
procedure consumer;
begin
while true do
begin
item procedure_consumer.remove;
consume_item(item) end end;
```

APEC

2.15.1 Drawbacks of Monitors

1. Major weakness of monitors is the absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time.

2. There is the possibility of deadlocks in the case of nested monitors calls.

3. Monitor concept is its lack of implementation most commonly used programming languages.

4. Monitors cannot easily be added if they are not natively supported by the language.

2.15.2 Difference between Monitors and Semaphore

• A semaphore is an operating system abstract data type whereas monitors are based on abstract data types.

• Semaphore-level synchronization primitives are difficult to use for complex synchronization situations. Monitors were derived to simplify the complexity of synchronization problems by abstructing away details.

• Semaphores provide a general-purpose mechanism for controlling access to critical sections. Their use does not guarantee that access will be mutually exclusive or deadlock will be avoided. A monitor is a programming language construct that guarantees appropriate access to critical sections.

Monitor uses condition variables. A condition variable is like a semaphore, with two differences

1) A semaphore counts the number of excess up operations, but a signal operation on a condition variable has no effect unless some process is waiting. A wait on a condition variable always blocks the calling process.

2) A wait on a condition variable automatically does an up on the monitor mutex and blocks the caller.

Example 2.15.3 Dining-philosopher critical section problem solution using monitor

.

Solution: Below solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. Here we consider, three state with datastructure.

enum {THINKING, HUNGRY, EATING} state[5];

• Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(1+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).

```
monitor DiningPhilosophers
enum {THINKING, HUNGRY, EATING} state[5];
condition self[5];
void pickup(int i)
{
state[i] = HUNGRY;
test(i);
if (state[i]!= EATING)
self[i].wait();
}
void putdown(int i)
state[i] THINKING;
test((i + 4) % 5);
test((i + 1) % 5);
void test(int i)
if ((state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) && (state[(i + 1)% 5] != EATING))
{
state[i] EATING;
self[i].signal();
}
}
initialization code()
{
for (int i = 0; i< 5; i++)
state[i]= THINKING;
}
}
```

APEC

Two Marks Questions with Answers

Q.1 Define process. What is the information maintained in a PCB ?

Ans. A process is simply a program in execution. i.e. an instance of a program. execution. PCB maintains pointer, state, process number, CPU register, PC, memory allocation etc.

Q.2 Define task control block.

Ans. TCB is also called PCB.

Q.3 What is PCB? Specify the information maintained in it. AU CSE/IT: Dec.-12

Ans. Each process is represented in the operating system by a process control block. PCB contains information like process state, program counter, CPU register, accounting information etc.

Q.4 What is independent process ?

Ans. Independent process cannot affect or be affected by the execution of another process.

Q.5 Name and draw five different process states with proper definition. AU: Dec.-17

Ans. Process states are new, running, waiting, ready and terminated. Fig. 2.16.1 shows process state diagram.

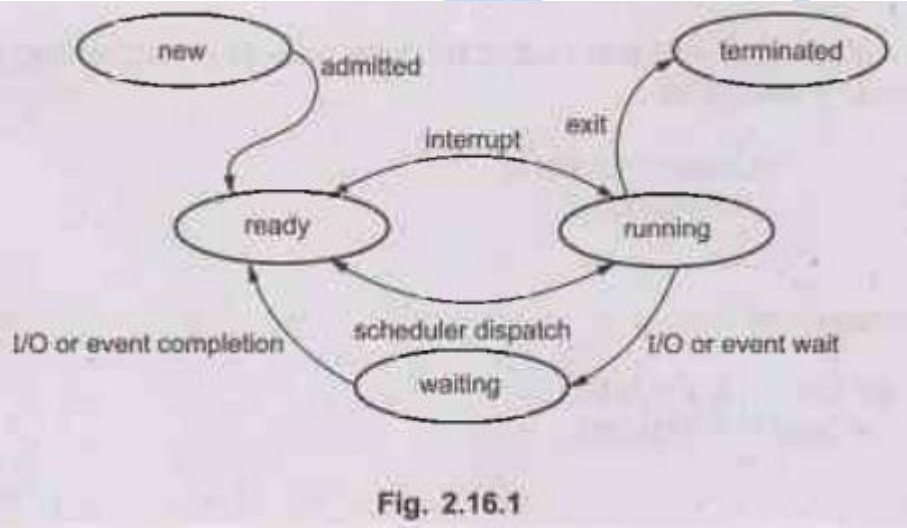


Fig. 2.16.1

Q.6 Define context switching.

Ans. Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as context switch.

Q.7 What are the reasons for terminating execution of child process ?

Ans. Parent may terminate execution of children processes via abort system call for a variety of reasons, such as:

1.Child has exceeded allocated resources.

2. Task assigned to child is no longer required.

3. Parent is exiting and the operating system does not allow a child to continue if its parent terminates.

Q.8 What is ready queue ?

Ans. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue.

Q.9 List out the data fields associated with process control blocks.

Ans. Data fields associated with process control block is CPU registers, PC, process state, memory management information, input-output status information etc.

Q.10 What are the properties of communication link ?

Ans. Properties of communication link

1. Links are established automatically.

2. A link is associated with exactly one pair of communicating processes.

3. Between each pair there exists exactly one link.

4. The link may be unidirectional, but is usually bidirectional.

Q.11 What is socket ?

Ans. A socket is defined as an endpoint for communication.

Q.12 What is non-preemptive scheduling ?

Ans. Non-preemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

Q.13 Differentiate preemptive and non-preemptive scheduling.

Ans. :

No.	Preemptive scheduling	Non-preemptive scheduling
1.	It allows a process to be interrupted in the middle of its execution, taking the CPU away and allocating it to another process.	It ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.
2.	Complex to implement.	Simple to implement.
3.	Costly.	Cost is less.
4.	Higher overhead.	Less overhead.
5.	Process switches from running state to ready state and waiting state to ready state.	Process switches from running state to waiting state and process terminates.

Q.14 What do you mean by short term scheduler ?

Ans. Short term scheduler, also known as a dispatcher executes most frequently, and makes the finest-grained decision of which process should execute next. This scheduler is invoked whenever an event occurs.

Q.15 Which are the criteria used for CPU scheduling ?

Ans. Criteria used for CPU scheduling are CPU utilization, throughput, turnaround time, waiting time, response time.

Q.16 Explain why two level scheduling is commonly used.

Ans. It provides the hybrid solution to the problem of providing good system utilization and good user service simultaneously. Code

Q.17 Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

Ans. I/O-bound programs have the property of performing only a small amount of computation before performing IO. Such programs typically do not use up their entire CPU quantum. CPU-bound programs, on the other hand, use their entire quantum without performing any blocking IO operations. Consequently, one could make better use of the computer's resources by giving higher priority to I/O-bound programs and allow them to execute ahead of the CPU-bound programs.

Q.18 What is response time?

Ans. Response time is the amount of time it takes from when a request was, submitted until the first response is produced, not output.

Q.19 Define waiting time.

Ans. Amount of time a process has been waiting in the ready queue.

Q.20 Define scheduling algorithm ?

Ans. In multiprogramming systems, whenever two simultaneously in the ready state, a choice has to be made which process to run next. The part of the OS that makes the choice is called the scheduler and the algorithm it uses is called the scheduling algorithm.

Q.21 Define the term dispatch latency.

Time it takes for the dispatcher to stop one process and start another running.

Q.22 What is preemptive priority method?

Ans. A preemptive priority will preempt the CPU if the newly arrived process is higher than the priority of the currently running process.

Q.23 What is medium term scheduling ?

Ans. Medium-term scheduling used especially with time-sharing systems as an intermediate scheduling level. A swapping scheme is implemented to remove partially run programs from memory and reinstate them later to continue where they left off.

Q.24 What is preemptive scheduling ?

Ans. Preemptive scheduling can preempt a process which is utilizing the CPU in between its execution and give the CPU to another process.

Q.25 What is the difference between long-term scheduling and short-term scheduling ?

Ans. Long term scheduling adds jobs to the ready queue from the job queue. Short term scheduling dispatches jobs from the ready queue to the running state.

Q.26 List out any four scheduling criteria.

Ans. Response time, throughput, waiting time and turn around time.

Q.27 Define the term 'Dispatch latency'.

Ans. Dispatch latency: Time it takes for the dispatcher to stop one process and start another running. It is the amount of time required for the scheduler to stop one process and start another.

Q.28 State what does a thread share with peer threads.

Ans. Thread share the memory and resource of the process.

Q.29 Distinguish between CPU-bounded and I/O bounded processes.

Ans.

CPU-bounded	I/O bounded processes
CPU Bound processes are ones that are implementing algorithms with a large number of calculations	Processes that are mostly waiting for the completion of input or output (I/O) are I/O Bound
Given a lower priority by the scheduler	Given high priority by the scheduler
CPU bound does too much computation to keep I/O busy	I/O bound does too much I/O to keep CPU busy
Example : matrix multiplication	Example : netscape

Q.30 Define priority inversion problem.

Ans. The higher priority process would be waiting for the low priority one to finish. This situation is known as priority inversion problem.

Q.31 What advantage is there in having different time-quantum sizes on different levels of a multilevel queuing system?

Ans. Processes that need more frequent servicing, for instance, interactive processes such as editors, can be in a queue with a small time quantum. Processes switch no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing, and thus making more efficient use of the computer.

Q.32 How does real-time scheduling differs from normal scheduling ?

Ans. Normal scheduling provides no guarantee on when a critical process will be scheduled; it guarantees only that the process will be given preference over non-critical processes. Real-time systems have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.

Q.33 What is Shortest-Remaining-Time-First (SRTF) ?

Ans. If a new process arrives with CPU burst length less than remaining time of process, current executing preempt. This scheme is known as the

Shortest-Remaining-Time-First.

Q.34 What is round robin CPU scheduling ?

Ans. Each process gets a small unit of CPU time (time quantum). After this time has elapsed, the process is preempted and added to the end of the ready queue.

Q.35 What is meant by starvation in operating system?

Ans. Starvation is a resource management problem where a process does not get the resources (CPU) it needs for a long time because the resources are being allocated to other processes.

Q.36 What is an aging?

Ans. Aging is a technique to avoid starvation in a scheduling system. It works by adding an aging factor to the priority of each request. The aging factor must increase the requests priority as time passes and must ensure that a request will eventually be the highest priority request

Q.37 How to solve starvation problem in priority CPU scheduling ?

Ans. Aging - as time progresses increase the priority of the process, so eventually the process will become the highest priority and will gain the CPU. i.e., the more time is spending a process in ready queue waiting, its priority becomes higher and higher.

Q.38 What is convoy effect?

Ans. A convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. Essentially it causes poor utilization of the other resources in the system.

Q.39 How can starvation / indefinite blocking of processes be avoided in priority scheduling ?

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

Q.40 "Priority inversion is a condition that occurs in real time systems where a low priority process is starved because higher priority processes have gained hold of the CPU" - Comment on this statement. AU CSE: May-17

Ans. A low priority thread always starts on a shadow version of the shared resource, the original resource remains unchanged. When a high-priority thread needs a resource engaged by a low -priority thread, the low priority thread is pre-empted, the original resource is restored and the high priority thread is allowed to use the original

Q.41 Provide two programming examples in which multithreading provides better. performance than a single-threaded solution.

Ans. A Web server that services each request in a separate thread.

A parallelized application such as matrix multiplication where different parts of the matrix may be worked on in parallel.

An interactive GUI program such as a debugger where a thread is used to monitor user input, another thread represents the running application, and a show third thread monitors performance.

Q.42 Define a thread. State the major advantage of threads.

Ans. A thread is a flow of execution through the process's code with its own program counter, system registers and stack.

2. Efficient communication.

Advantages: 1. Minimize context switching time.

Q.43 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single processor system ?

Ans. A multithreaded system comprising of multiple user-level threads cannot make use of the different processors in a multiprocessor system simultaneously. The operating system sees only a single process and will not schedule the different threads of the process on separate processors. Consequently, there is no performance benefit associated with executing multiple user-level threads on a multiprocessor system.

Q.44 What are the differences between user-level threads and kernel-level threads ? (Refer section 2.8.3)

Sr. No.	User level threads	Kernel level threads
1.	User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
2.	Implemented by a thread library at the user level.	Operating system support directly to Kernel threads.
3.	User level thread can run on any operating system.	Kernel level threads are specific to the operating system.
4.	Support provided at the user level called user level thread.	Support may be provided by Kernel is called Kernel level threads.
5.	Multithread application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.
6.	Example : POSIX Pthreads and Mach C-threads	Example : Windows 95/98/NT, Sun Solaris and Digital UNIX
7.	User level threads are also called many to one mapping thread.	Kernel level thread support one to one thread mapping.

Q.45 What are the benefits of multithreads?

Ans. Benefits of multithreading is responsiveness, resource sharing, economy and utilization of multiprocessor architecture.

Q.46 Why a thread is called as light weight process ?

Ans. Thread is light weight taking lesser resources than a process. It is called light weight process to emphasize the fact that a thread is like a process but is more efficient and uses fewer resources and they also share the address space.

Q.47 Name one situation where threaded programming is normally used ?

Ans. Threaded programming would be used when a program should satisfy multiple tasks at the same time. A good example for this would be a program running with

GUI.

Q.48 Describe the actions taken by a thread library to context switch between user-level threads.

Ans. Context switching between user threads is quite similar to switching between Kernel threads, although it is dependent on the threads library and how it maps user threads to kernel threads. In general, context switching between user threads involves taking a user thread of its LWP and replacing it with another thread. This act typically involves saving and restoring the state of the registers.

Q.49 What is a thread pool ?

Ans. A thread pool is a collection of worker threads that efficiently execute asynchronous call-backs on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads.

Q.50 What is deferred cancellation in thread ?

Ans. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion. With deferred cancellation, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not. This allows a thread to check whether it should be cancelled at a point when it can be cancelled safely.

Q.51 What is Pthread ?

Ans. Pthreads refers to the POSIX standard defining an API for thread creation and synchronization. This is a specification for thread behavior, not an implementation. Operating system designers may implement the specification in any way they wish.

Q.52 What is thread cancellation ?

Ans. Under normal circumstances, a thread terminates when it exits normally, either by returning from its thread function or by calling pthread_exit.However, it is possible for a thread to request that another thread terminate. This is called cancelling a thread.

Q.53 List the benefit of multithreading.

Ans.

Benefit of multithreading:

It takes less time to create thread than a new process.

It takes less time to terminate thread than process.

Q.54 Under what circumstances is user level threads is better than the kernel level threads?

Ans. User-level threads are much faster to switch between, as there is no context switch; further, a problem-domain-dependent algorithm can be used to schedule among them. CPU-bound tasks with interdependent computations, or a task that will switch among threads often, might best be handled by user-level threads

Q.55 What resources are required to create threads?

Ans. Thread is smaller than a process, so thread creation typically uses fewer resources than process creation.

Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

Q.56 Differentiate single threaded and multi-threaded processes.

Ans. Single-threading is the processing of one command at a time. When one thread is paused, the system waits until this thread is resumed. In Multithreaded processes, threads can be distributed over a series of processors to scale. When one thread is paused due to some reason, other threads run as normal.

Q.57 Give an programming example in which multithreading does not provide better performance than single threaded solution.

Ans. Any kind of sequential program is not a good candidate to be threaded. An example of this is a program that calculates an individual tax return.

Another example is a "shell" program such as the C-shell or Korn shell. Such a program must closely monitor its own working space such as open files, environment variables, and current working directory.

Q.58 Define mutual exclusion.

Ans. If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources.

Q.59 How the lock variable can be used to introduce mutual exclusion?

Ans. We consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first tests the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, a 0 means that no process in its critical section, and 1 means hold your horses - some process is in its critical section.

Q.60 What is the hardware feature provided in order to perform mutual exclusion operation indivisibly ?

Ans. Hardware features can make any programming task easier and improve system efficiency. It provide special hardware instructions that allow user to test and modify the content of a word.

Q.61 Discuss why implementing synchronization primitives by disabling interrupts is gnome not appropriate in a single processor system if the synchronization primitives be are to be used in user level programs.

Ans. If a user level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes execute.

Q.62 What is race condition ?

Ans. A race condition is a situation where two or more processes access shared data concurrently and final value of shared data depends on timing.

Q.63 Define entry section and exit section.

Ans. Each process must request permission to enter its critical section. The section of the code implementing this request is the entry section. The critical section is followed by an exit section. The remaining code is the remainder section.

Q.64 Elucidate mutex locks with its procedure.

Ans. Mutex lock is software tools to solve the critical-section problem. A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable

Q.65 Name any two file system objects that are neither files nor directories and what the advantage of doing so is.

Ans. Semaphore and monitors the system objects. Advantage is to avoid critical section problem.

Q.66 What is binary semaphore ?

Ans. Binary semaphore is a semaphore with an integer value that can range only between 0 and 1.

Q.67 What is semaphore? Mention its importance in operating systems.

Ans. Semaphore is an integer variable. It is a synchronization tool used to solve critical section problem. The various hardware based solutions to the critical section problem are complicated for application programmers to use.

Q.68 What is the meaning of the term busy waiting ?

Ans. Busy waiting means a process waits by executing a tight loop to check the status/value of a variable.

Q.69 What is bounding waiting ?

Ans. After a process made a request to enter its critical section and before it is granted the permission to enter, there exists a bound on the number of times that other processes are allowed to enter.

Q.70 Why can't you use a test and set instruction in place of a binary semaphore ?

Ans. A binary semaphore requires either a busy wait or a blocking wait, semantics not provided directly in the Test and Set. The advantage of a binary semaphore is that it does not require an arbitrary length queue of processes waiting on the semaphore.

Q.71 What is concept behind strong semaphore and spinlock ?

Ans. Semaphore can be implemented in user applications and in the kernel. The process that has been blocked the longest is released from the queue first is called a strong semaphore..

Using simple lock variable, process synchronization problem is not solved. To avoid this, spinlock is used. A lock that uses busy waiting is called a spin lock.

Q.72 What is bounded buffer problem ?

Ans. The bounded buffer producers and consumers assume that there are fixed buffer sizes i.e. a finite numbers of slots are available. To suspend the producers when the buffer is full, to suspend the consumers when the buffer is empty, and to make sure that only one process at a time manipulates a buffer so there are no race conditions or lost updates.

Q.73 State the assumption behind the bounded buffer producer consumer problem.

Ans. Assumption: It is 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 to the value 1.

Q.74 What is a critical section and what requirements must a solution to the critical section problem satisfy ?

Ans. Consider a system consisting of several processes, each having a segment of code called a critical section, in which the process may be changing common variables, updating tables, etc. The important feature of the system is that when one process is executing its critical section, no other process is to be allowed to execute its critical section. Execution of the critical section is mutually exclusive in time.

A solution to the critical section problem must satisfy the following three requirements: 1. Mutual exclusion 2. Progress 3. Bounded waiting

Q.75 Define 'monitor'. What does it consist of?

Ans. Monitor is a highly structured programming language construct. It consists of private variables and private procedures that can only be used within a monitor.

Q.76 Explain the use of monitors.

Ans. Use of monitors :a) It provides a mutual exclusion facility. b) A monitor support synchronization by the use of condition variables. c) Shared data structure can be protected by placing it in a monitor.

Q.77 Give the queuing diagram representation of process scheduling.

Ans.

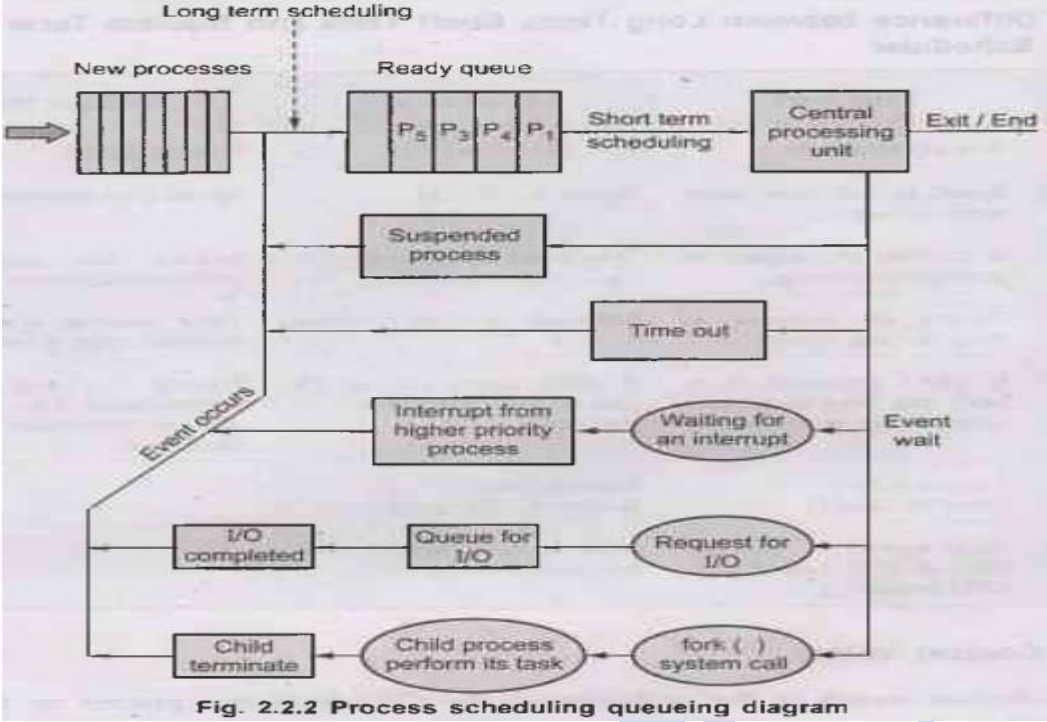


Fig. 2.2.2 Process scheduling queueing diagram

Q.78 What are kernel threads.

Ans. Kernel threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel.

University Question

1. *What is a process? Discuss components of process and various states of a process with the help of a process state transition diagram.*
2. *Describe the difference among short-term, medium-term and long-term scheduling with suitable example.*
3. Write the difference between user thread and kernel thread.
4. Explain with a neat diagram about various multi threading models.
5. What is a race condition? Explain how a critical section avoids this condition. What are the properties which a data item should possess to implement a critical section? Describe a solution to the Dining philosopher problem so that no races arise.
6. Describe critical section problem with a suitable example.
7. Write an algorithm using test-and-set() instruction that satisfy all the critical section requirements.
8. Explain the dining-philosopher critical section problem solution using monitor.