Unit III
Process and threads.

Contents:

- Process Concept,
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client- Server Systems
- Overview of threads, Multithreading Models

Early computer systems allowed only one program to be executed at a time. Current day computer systems allow multiple programs to be loaded into memory and executed concurrently.

* Process Concept –

a) The Process –

A program is a set of instructions to perform a specific task. A process is a program in execution. Also, an executable part of a program called as process. A program is dependent on user's logic. A process is dependent on a program. In multiprogramming environment, multiple processes are possible in main memory at same time, so process management is important task of operating system.

A program is a passive entity, whereas a process is an active entity. A program becomes a process when an executable file is loaded into memory.

b) Process States –

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states as shown in following figure,

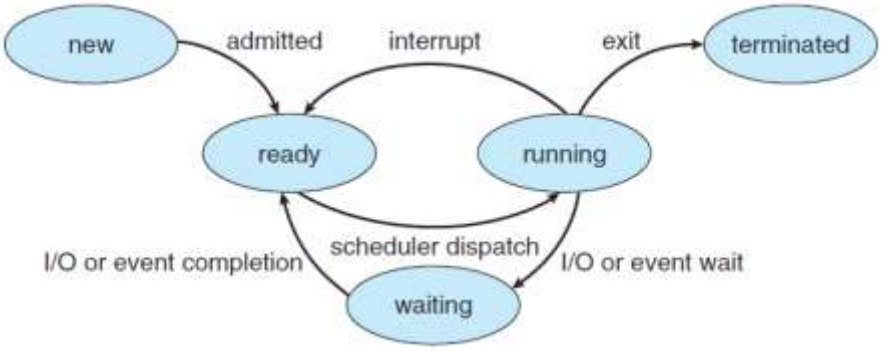


Figure – Process State Model

- New – The process is being created.
- Ready – The process is waiting to be assigned to a processor.

• Running – Instructions are being executed.
• Waiting – The process is waiting for some event to occur (such as an I/O completion).
• Terminated – The process has finished execution.

c) Process Control Block (PCB) –

Each process is represented in the operating system by a process control block (PCB). It is also called a task control block as shown in following figure,

Pointer
Process ID
Process State
Process Priority
Memory Management Information
Accounting Information
I/O Status Information

Figure – Process Control Block (PCB)

• Pointer – It points to the next PCB in the ready queue.
• Process ID – This is a number allocated by the operating system to the process on creation for unique identification.
• Process State – It denotes current state of the process. It may be new, ready, run, or wait.
• Process Priority – It determines importance of the process to the system.
• Memory Management Information – This information may include such information as the value of the base and limit address.
• Accounting Information – This information includes the amount of CPU time required.
• I/O Status Information – This information includes the list of I/O devices allocated to the process.

*Process Scheduling –

The following tasks are performed by process scheduler,
• Keep track of status of processes.
• Decide which process gets processor?
• Allocate processors to processes.
• Deallocate processors from processes.

a) Scheduling Queues –

As programs enter in the system, they are put into a job queue. The processes that are available in main memory and are ready and waiting to execute are kept on a list called the ready queue. Processes waiting for devices are kept in device queue or I/O queue. Each device has its own device queue. A common representation for a process scheduling is a queuing diagram, as shown in following figure,

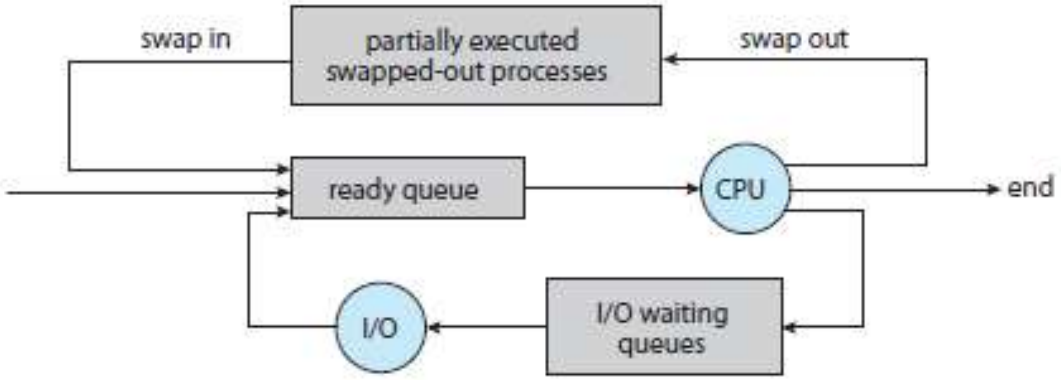


Figure – Queuing diagram representation of process scheduling

Each rectangular box represents a queue. Each circle represents the resource that serves the queue and the arrows indicate the flow of processes in the system. A new process is initially put in the ready queue. It waits there until it is selected for execution or is dispatched.

Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could finish
- The process could issue an I/O request and then be placed in an I/O queue.

b) Schedulers

In a system, more processes are submitted than can be executed immediately. These processes are spooled to a mass storage device (typically a disk), where they are kept for later execution.

The long term scheduler or job scheduler selects processes from this pool and loads them into memory for execution. The short term scheduler or CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them.

Most processes can be divided into either I/O bound or CPU bound. An I/O bound process is one that spends more of its time doing I/O computations. A CPU bound process uses more of its time doing computations.

c) Context Switch –

Multiprogramming is a concept of increasing utilization of CPU by always having something for CPU to execute. When CPU switches from one process to another process, the time which is required for switching called as, “context switching”.

While context switching, the contents of old process are stored in specific memory area called as, “register save area”.

Consider the following example,

```
main ()
{
 ….
 c = add (a, b);
 ….
 }
int add (x, y)
 {
 return (x + y);
 }
```

Consider, there are two processes are created by operating system from above program i.e. P1 for main () and P2 for add (). First, process P1 is allocated to a CPU for execution; CPU executes instruction one by one from process P1.

When it comes to the instruction c = add (a, b); then stops execution because the result of this instruction depends on result of another process i.e. P2. Thus, there is need of execution of

process P2 before process P1; so, CPU stores content of current process i.e. P1 in register save area and sends back to main memory.
Then CPU accepts another process i.e. P2, executes that process and sends result to suspended process i.e. P1. Again, CPU accepts suspended process i.e. P1 as well as current contents of process P1 from register save area and then resume execution of process P1 and completes its execution.
When CPU switches from process P1 to P2 or process P2 to P1, the time which is required for switching called as context switching.

* Operations on Processes

The processes in operating systems can created and deleted dynamically.

(a)Process Creation –

A process is part of a program running in a computer system. A process may create several new processes, during the execution. The creating process is called a parent process, and the new processes are called the children of that process.

Most operating systems identify processes according to a unique process identifier. In general, a process will need certain resources (CPU time, memory, files, and I/O devices) to accomplish its task.

When a process creates a sub-process, that sub-process may be able to obtain its resources directly from the operating system, or from the parent process.

In UNIX, a new process is created by the fork () system call and in Windows OS, by the CreateProcess ().

b) Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it. All the resources of the process are deallocated by the operating system. In UNIX, a process is deleted by the exit () system call and in Windows OS, by the TerminateProcess (). A parent may terminate the execution of one of its children for a variety of reasons, such as,

- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems, do not allow a child to exist if its parent has terminated. In such systems, if a process terminates, then all its children must also be terminated. This term is called cascading termination.

*Interprocess Communication –

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. A process is cooperating if it can affect or be affected by the other processes executing in the system.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

There are two fundamental models of interprocess communication:

(1) Shared memory systems **(2) Message passing systems**

1)Shared Memory Systems –

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.

Typically, a shared memory region resides in the address space of the process creating the shared memory segment. The following figure shows communication model using shared memory systems,

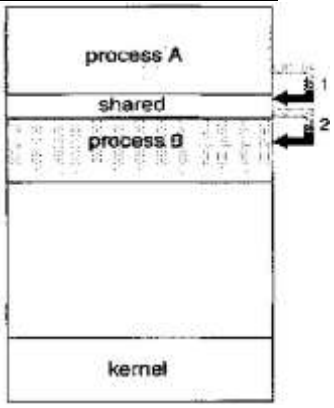


Figure – Communications model using shared memory system

Other processes that wish to communicate using this shared memory segment must attach it to their address space. Normally, the operating system tries to prevent one process from accessing another process's memory.

Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.

The form of the data and the location are determined by these processes and are not under the operating systems control.

2) Message Passing Systems –

A message passing facility provides two operations: send (message) and receive (message). If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.

This link can be implemented in a variety of ways,

a) Naming **b) Synchronization** **c) Buffering**

(a)Naming –

Processes that want to communicate with each other, they can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. This scheme is symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate.

In this scheme, the send () and receive () primitives are defined as:

• send (Q, message)—Send a message to process Q.

• receive (P, message)—Receive a message from process P.

With indirect communication, the messages are sent to and received from mailboxes. Each mailbox has a unique identification. Two processes can communicate only if the processes have a shared mailbox, however.

The send () and receive () primitives are defined as follows:

• send (A, message) — Send a message to mailbox A.

• receive (A, message) — Receive a message from mailbox A.

b) Synchronization –

Message passing may be blocking or non-blocking — also known as synchronous and asynchronous.

- Blocking Sender – The sending process is blocked until the message is received by the receiving process.
- Blocking Receiver – The receiver block until a message is available.
- Non-blocking Sender – The sending process sends the message and resumes operation.
- Non-blocking Receiver – The receiver retrieves message even if another message still available.

c) Buffering-

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- Zero capacity – The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- Bounded capacity – The queue has finite length n; thus, at most 'n' messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. If the queue is full, the sender must block until space is available in the queue.
- Unbounded capacity – The queues length is infinite; thus, any number of messages can wait in it. The sender never blocks.

Examples of IPC Systems

Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using methods like shared memory and message parsing.

Some of the ways in which inter process communication can take place are as follows:

Some of the examples of IPC systems are as follows:

Posix: Uses shared memory method.

Mach: Uses message passing.

Windows XP: Uses message passing using local procedural calls.

2. Various mechanisms for inter-process communication in a client/server architecture are as follows:

- Pipes
- Sockets
- Remote Procedural Calls (RPCs)

Ex Shared Memory Method : Producer-Consumer problem

- There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed.
- There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up

to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem.

- First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer.
- Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it.

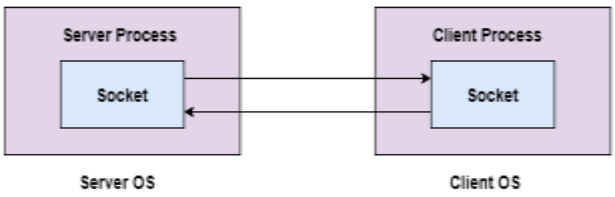
Communication in Client- Server Systems

- Client/Server communication involves two components, namely a client and a server. They are usually multiple clients in communication with a single server. The clients send requests to the server and the server responds to the client requests.
- There are three main methods to client/server communication.
- These are given as follows:

1) Sockets
2) Remote Procedure Calls
3) Pipes

1) **Sockets**

- Sockets facilitate communication between two processes on the same machine or different machines. They are used in a client/server framework and consist of the IP address and port number. Many application protocols use sockets for data connection and data transfer between a client and a server.

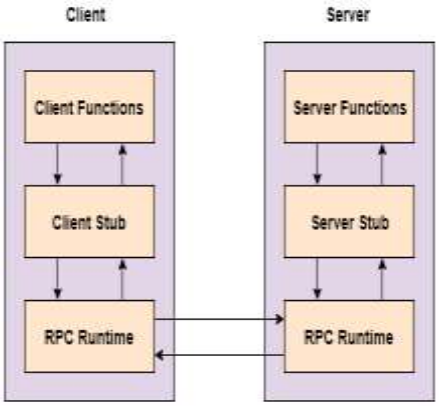


- Socket communication is quite low-level as sockets only transfer an unstructured byte stream across processes. The structure on the byte stream is imposed by the client and server applications.

A diagram that illustrates sockets is as follows:

Remote Procedure Calls

- These are interprocess communication techniques that are used for client-server based applications. A remote procedure call is also known as a subroutine call or a function call.
- A client has a request that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client.
- A diagram that illustrates remote procedure calls is given as follows:



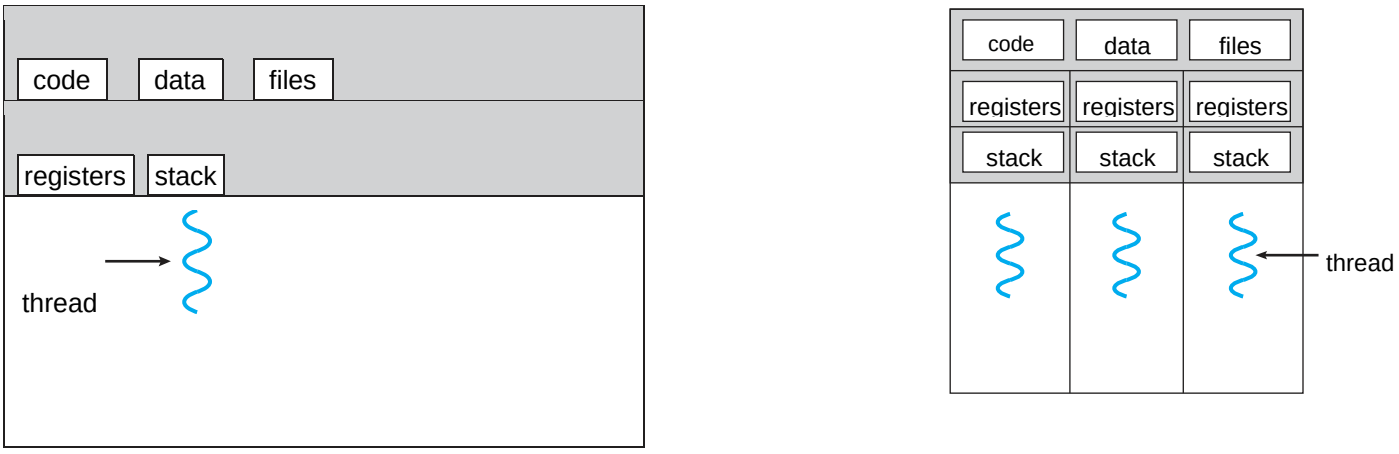
Pipes

- These are interprocess communication methods that contain two end points. Data is entered from one end of the pipe by a process and consumed from the other end by the other process.
- The two different types of pipes are ordinary pipes and named pipes. Ordinary pipes only allow one way communication. For two way communication, two pipes are required. Ordinary pipes have a parent child relationship between the processes as the pipes can only be accessed by processes that created or inherited them.
- Named pipes are more powerful than ordinary pipes and allow two way communication. These pipes exist even after the processes using them have terminated. They need to be explicitly deleted when not required anymore.

- A diagram that demonstrates pipes are given as follows:

Overview of threads

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



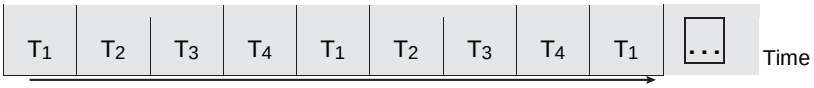
single-threaded process multithreaded process

Figure : Single-threaded and multithreaded processes

Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.
2. **Resource sharing.** Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is significantly
more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.



Multithreading Overview : –

A process is a program in execution, whereas a thread is a path of execution within a process. Sometimes, a thread is also called as lightweight process. A thread is a basic unit of CPU utilization. A traditional process has a single thread of control. If a process has multiple threads of control; it can perform more than one task at a time.

Many software packages that run on modern desktop PCs are multithreaded. A web browser might have one thread display images or text while another thread retrieves data from the network.

Also, a word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in background.

* Multithreading Models –

There are two categories of thread implementation,

1) User level threads

2) Kernel level threads

In user level thread, all thread management is done by the application by using a thread library. User level threads can be run on any operating system. User level threads are faster to create and manage.

In kernel level thread, all thread management is done by the kernel by using system calls. Kernel level threads are specific to the operating system. Kernel level threads are slower to create and manage.

There must be a relationship between user threads and kernel threads. There are three common ways of establishing this relationship,

a) Many-to-one
b) One-to-one
c) Many-to-many

a) Many-to-One Model –

It maps many user level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient.

Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

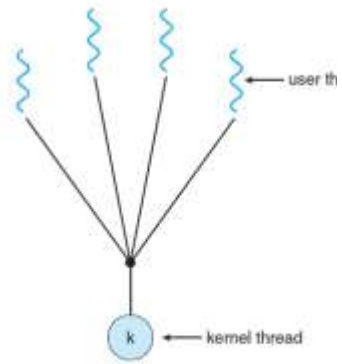


Figure – Many-to-one model

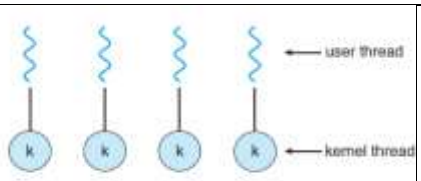


Figure - One-to-one model

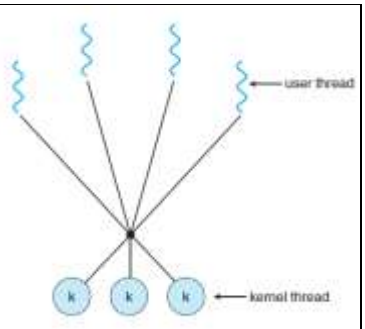


Figure – Many-to-many model

B) One-to-One Model –

It maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model. It allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Linux, along with the family of Windows operating systems – including Windows 95, 98, NT, 2000, and XP—implement the one-to-one model.

C) Many-to-Many Model –

It maps many user level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. In many-to-many model, developers can create as many user threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor.