Process in Operating System

A process is a program in execution which then forms the basis of all computation. The process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to the program which is considered to be a 'passive' entity. Attributes held by the process include hardware state, memory, CPU, etc.

Process memory is divided into four sections for efficient working :

- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The **Data section** is made up of the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.

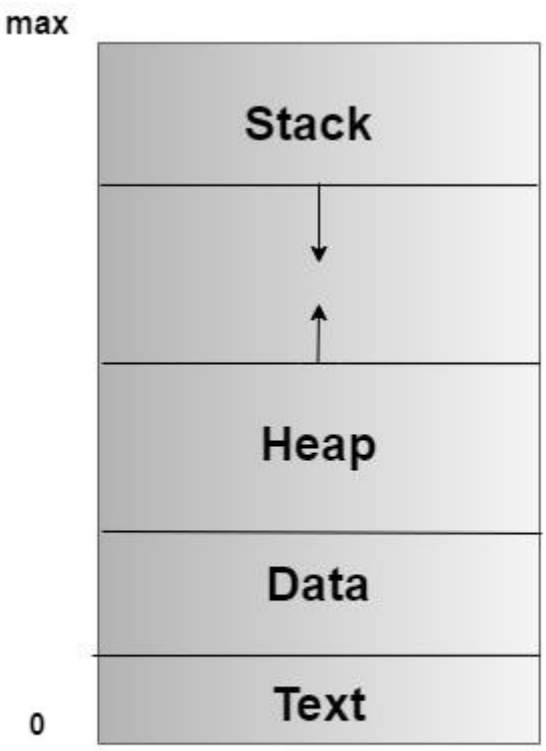
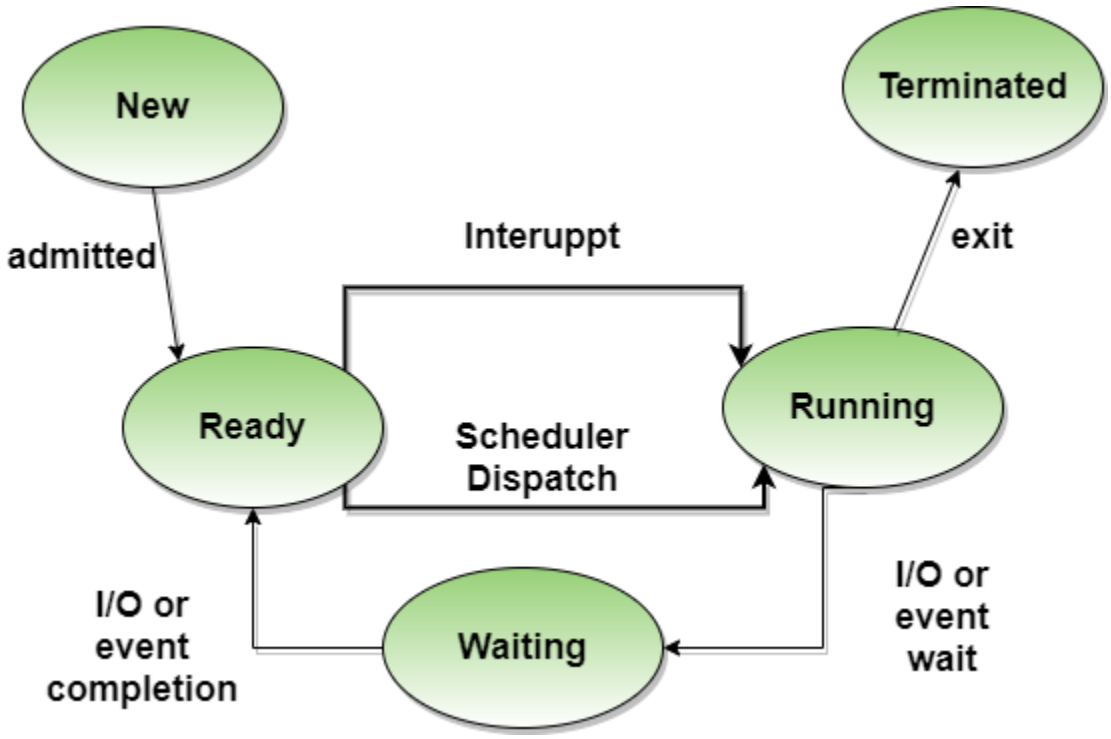


Figure: Process in the Memory

The different Process States

Processes in the operating system can be in any of the following states:

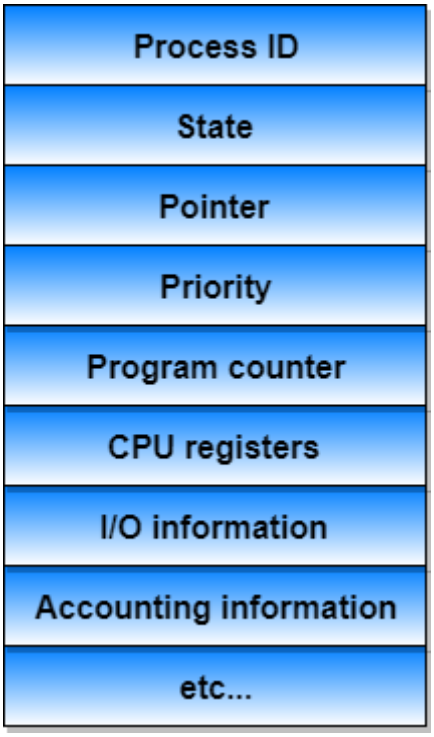
- NEW- The process is being created.
- READY- The process is waiting to be assigned to a processor.
- RUNNING- Instructions are being executed.
- WAITING- The process is waiting for some event to occur(such as an I/O completion or reception of a signal).
- TERMINATED- The process has finished execution.



Process Control Block

There is a Process Control Block for each process, enclosing all the information about the process. It is also known as the task control block. It is a data structure, which contains the following:

- **Process State**: It can be running, waiting, etc.
- **Process ID** and the **parent process ID**.
- CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
- **CPU Scheduling** information: Such as priority information and pointers to scheduling queues.
- **Memory Management information**: For example, page tables or segment tables.
- **Accounting information**: The User and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information**: Devices allocated, open file tables, etc.



Process vs Program

Let us take a look at the differences between Process and Program:

Process	**Program**
The process is basically an instance of the computer program that is being executed.	A Program is basically a collection of instructions that mainly performs a specific task when executed by the computer.
A process has a **shorter lifetime**.	A Program has a **longer lifetime**.

Process	Program
A Process requires resources such as memory, CPU, Input-Output devices.	A Program is stored by hard-disk and does not require any resources.
A process has a dynamic instance of code and data	A Program has static code and static data.
Basically, a process is the **running instance** of the code.	On the other hand, the program is the **executable code**.

Process Scheduling

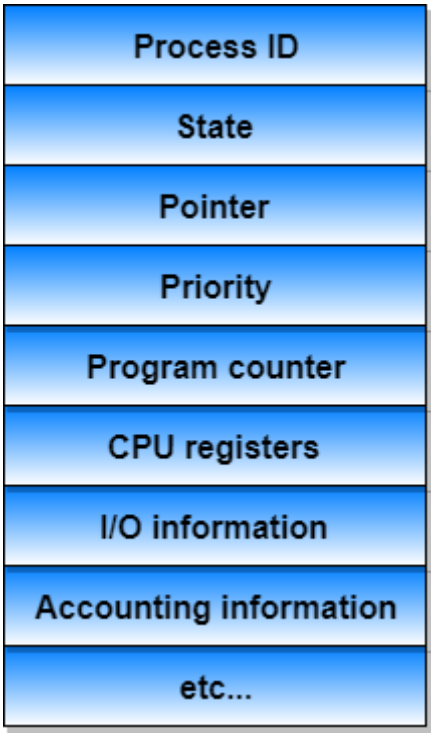
When there are two or more runnable processes then it is decided by the Operating system which one to run first then it is referred to as Process Scheduling.

A scheduler is used to make decisions by using some scheduling algorithm.

Given below are the properties of a **Good Scheduling Algorithm**:

- Response time should be minimum for the users.
- The number of jobs processed per hour should be maximum i.e Good scheduling algorithm should give maximum throughput.
- The utilization of the CPU should be 100%.

- Each process should get a fair share of the CPU.



What is Process Scheduling?

The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Scheduling fell into one of the two general categories:

- **Non Pre-emptive Scheduling:** When the currently executing process gives up the CPU voluntarily.

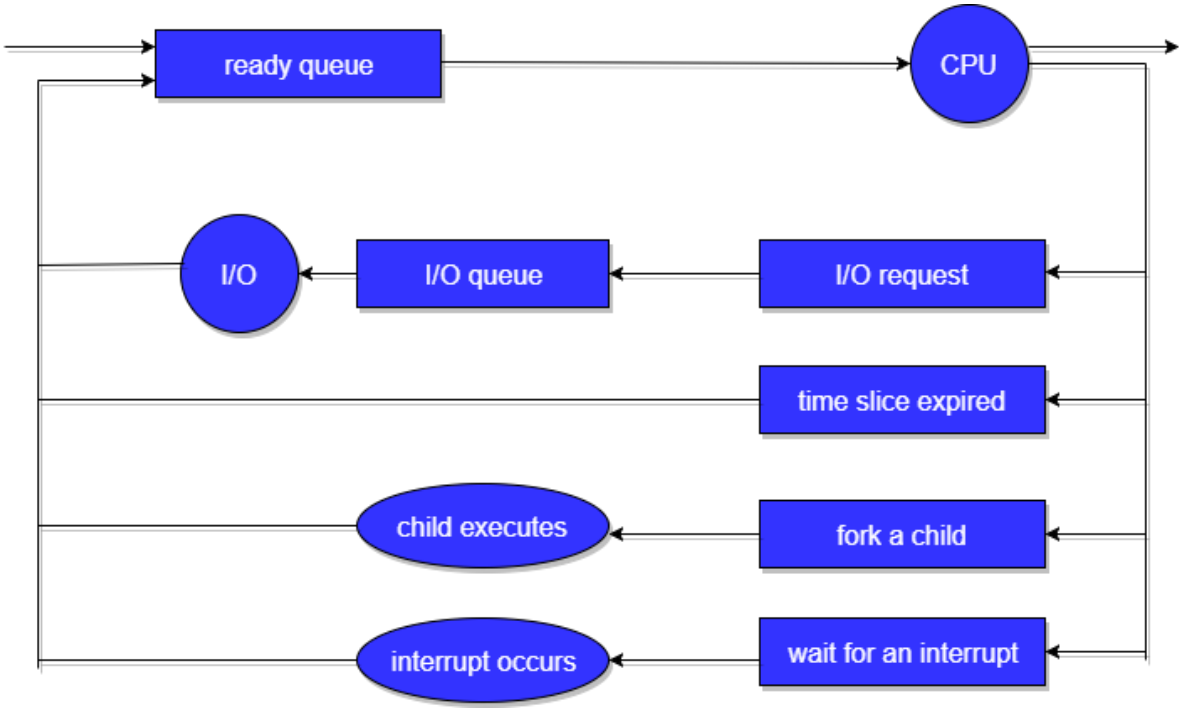
- **Pre-emptive Scheduling:** When the operating system decides to favour another process, pre-empting the currently executing process.

What are Scheduling Queues?

- All processes, upon entering into the system, are stored in the **Job Queue**.
- Processes in the `Ready` state are placed in the **Ready Queue**.
- Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.

A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution(or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

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



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

Types of Schedulers

There are three types of schedulers available:

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

Let's discuss about all the different types of Schedulers in detail:

Long Term Scheduler

Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution. Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

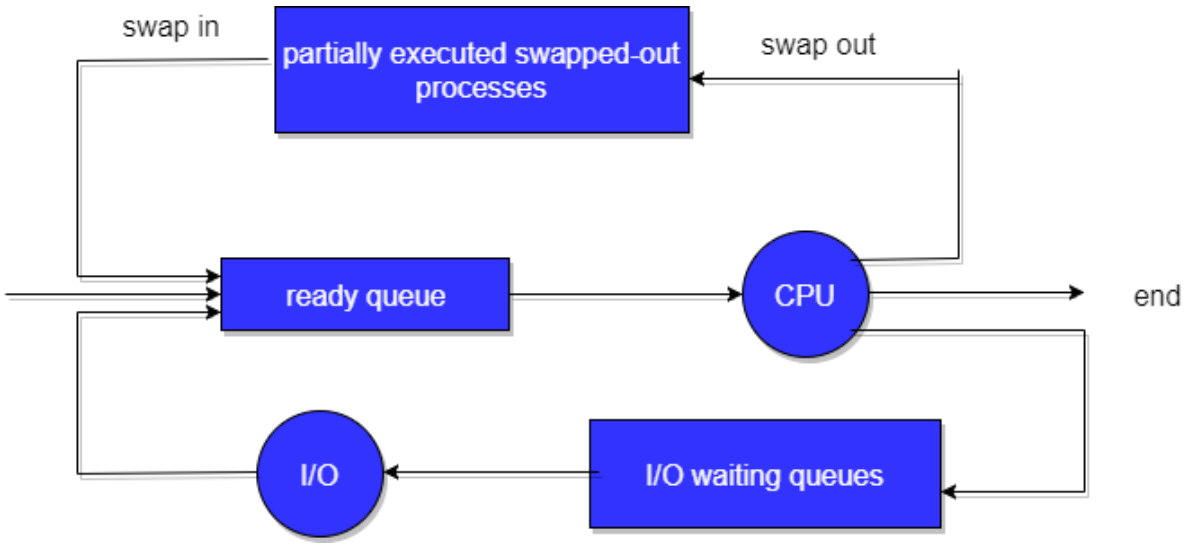
Short Term Scheduler

This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

Medium Term Scheduler

This scheduler removes the processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution van be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium term scheduler.

Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. This complete process is descripted in the below diagram:



Addition of Medium-term scheduling to the queueing diagram.

What is Context Switch?

1. 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 a **Context Switch**.
2. The **context** of a process is represented in the **Process Control Block(PCB)** of a process; it includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
3. Context switch time is **pure overhead**, because the **system does no useful work while switching**. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions(such as a single instruction to load or store all registers). Typical speeds range from 1 to 1000 microseconds.

4. Context Switching has become such a performance **bottleneck** that programmers are using new structures(threads) to avoid it whenever and wherever possible.

Operations on Process

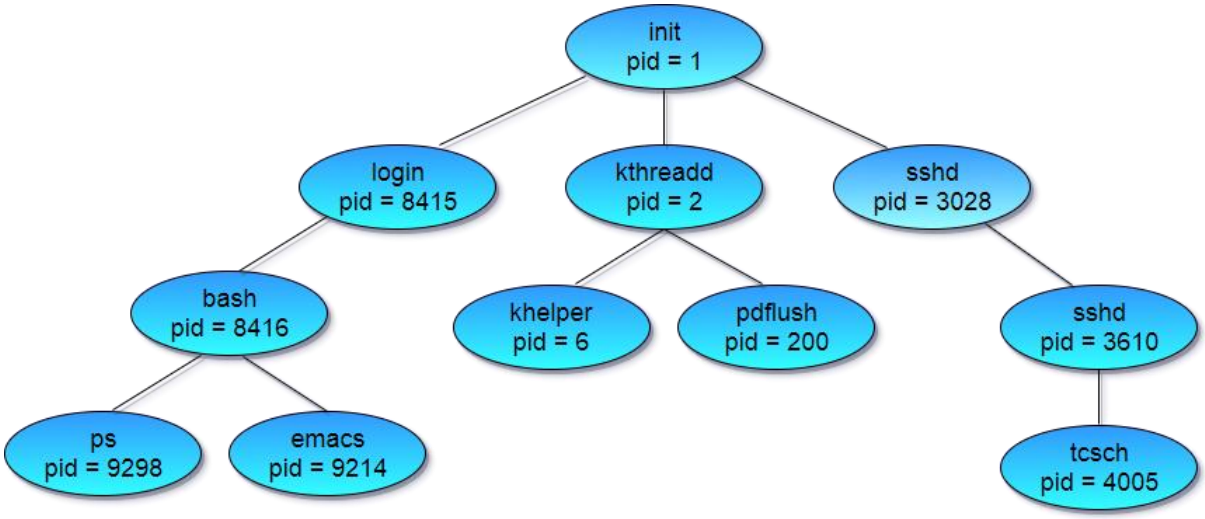
Below we have discussed the two major operation **Process Creation** and **Process Termination**.

Process Creation

Through appropriate system calls, such as fork or spawn, processes may create other processes. The process which creates other process, is termed the **parent** of the other process, while the created sub-process is termed its **child**.

Each process is given an integer identifier, termed as process identifier, or PID. The parent PID (PPID) is also stored for each process.

On a typical UNIX systems the process scheduler is termed as `sched`, and is given PID 0. The first thing done by it at system start-up time is to launch `init`, which gives that process PID 1. Further Init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes.



A child process may receive some amount of shared resources with its parent depending on system implementation. To prevent runaway children from consuming all of a certain system resource, child processes may or may not be limited to a subset of the resources originally allocated to the parent.

There are two options for the parent process after creating the child :

- Wait for the child process to terminate before proceeding. Parent process makes a `wait()` system call, for either a specific child process or for any particular child process, which causes the parent process to block until the `wait()` returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
- Run concurrently with the child, continuing to process without waiting. When a UNIX shell runs a process as a background task, this is the operation seen. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.

2. The child process has a program loaded into it.

To illustrate these different implementations, let us consider the **UNIX** operating system. In UNIX, each process is identified by its **process identifier**, which is a unique integer. A new process is created by the **fork** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork system call, with one difference: **The return code for the fork system call is zero for the new(child) process, whereas the(non zero) process identifier of the child is returned to the parent.**

Typically, the **execlp system call** is used after the fork system call by one of the two processes to replace the process memory space with a new program. The execlp system call loads a binary file into memory - destroying the memory image of the program containing the execlp system call – and starts its execution. In this manner the two processes are able to communicate, and then to go their separate ways.

Below is a **C program** to illustrate forking a separate process using UNIX(made using Ubuntu):

```
#include<stdio.h>



void main(int argc, char *argv[])

{

    int pid;
```

```
    /* Fork another process */
    pid = fork();

    if(pid < 0)
    {
        //Error occurred
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0)
    {
        //Child process
        execlp("/bin/ls","ls",NULL);
    }
    else
    {
        //Parent process
        //Parent will wait for the child to
complete
        wait(NULL);
```

```
            printf("Child complete");

            exit(0);

        }

}
```

Copy

GATE Numerical Tip: *If **fork** is called for n times, the number of child processes or new processes created will be: $2^n - 1$.*

Process Termination

By making the exit(system call), typically returning an int, processes may request their own termination. This int is passed along to the parent if it is doing a wait(), and is typically zero on successful completion and some non-zero code in the event of any problem.

Processes may also be terminated by the system for a variety of reasons, including :

- The inability of the system to deliver the necessary system resources.
- In response to a KILL command or other unhandled process interrupts.
- A parent may kill its children if the task assigned to them is no longer needed i.e. if the need of having a child terminates.
- If the parent exits, the system may or may not allow the child to continue without a parent (In UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them.)

When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.

The processes which are trying to terminate but cannot do so because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off.

CPU Scheduling in Operating System

CPU scheduling is a process that allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast, and fair.

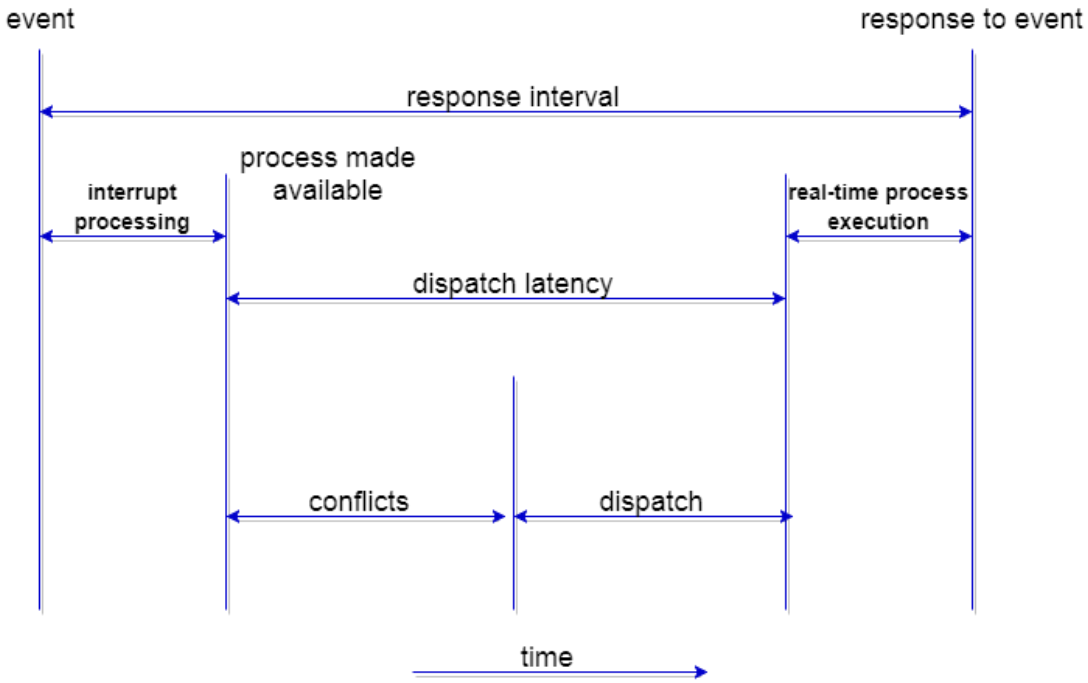
Whenever the CPU becomes idle, the operating system must select one of the processes in the **ready queue** to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them.

CPU Scheduling: Dispatcher

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

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program from where it left last time.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**. Dispatch Latency can be explained using the below figure:



Types of CPU Scheduling

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the **running** state to the **waiting** state(for I/O request or invocation of wait for the termination of one of the child processes).
2. When a process switches from the **running** state to the **ready** state (for example, when an interrupt occurs).

3. When a process switches from the **waiting** state to the **ready** state(for example, completion of I/O).
4. When a process **terminates**.

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process(if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3.

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **non-preemptive**; otherwise, the scheduling scheme is **preemptive**.

Non-Preemptive Scheduling

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

It is the only method that can be used on certain hardware platforms because It does not require the special hardware(for example a timer) needed for preemptive scheduling.

In non-preemptive scheduling, it does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then after that it can allocate the CPU to any other process.

Some Algorithms based on non-preemptive scheduling are: Shortest Job First (SJF basically non-preemptive) Scheduling and Priority (non- preemptive version) Scheduling, etc.

Process	Arrival time	CPU Burst Time (in millisecond
P0	2	8
P1	3	6
P2	0	9
P3	1	4

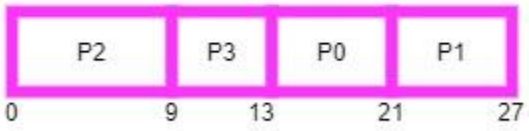


Figure: Non-Preemptive Scheduling

Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

Thus this type of scheduling is used mainly when a process switches either from running state to ready state or from waiting state to ready state. The resources (that is CPU cycles) are mainly allocated to the process for a limited amount of time and then are taken away, and after that, the process is again placed back in the ready queue in the case if that process still has a CPU burst time remaining. That process stays in the ready queue until it gets the next chance to execute.

Some Algorithms that are based on preemptive scheduling are Round Robin Scheduling (RR), Shortest Remaining Time First (SRTF), Priority (preemptive version) Scheduling, etc.

Process	Arrival time	CPU Burst Time (in millisecond
P0	2	3
P1	3	5
P2	0	6
P3	1	5

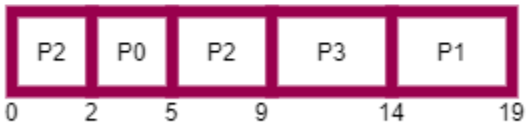


Figure: Preemptive Scheduling

CPU Scheduling: Scheduling Criteria

There are many different criteria to check when considering the **"best"** scheduling algorithm, they are:

CPU Utilization

To make out the best use of the CPU and not to waste any CPU cycle, the CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

It is the total number of processes completed per unit of time or rather says the total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

Turnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from the time of submission of the process to the time of completion of the process(Wall clock time).

Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

Response Time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling Algorithms

To decide which process to execute first and which process to execute last to achieve maximum CPU utilization, computer scientists have defined some algorithms, they are:

1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling

6. Multilevel Feedback Queue Scheduling
7. Shortest Remaining Time First (SRTF)
8. Longest Remaining Time First (LRTF)
9. Highest Response Ratio Next (HRRN)

What is Inter Process Communication?

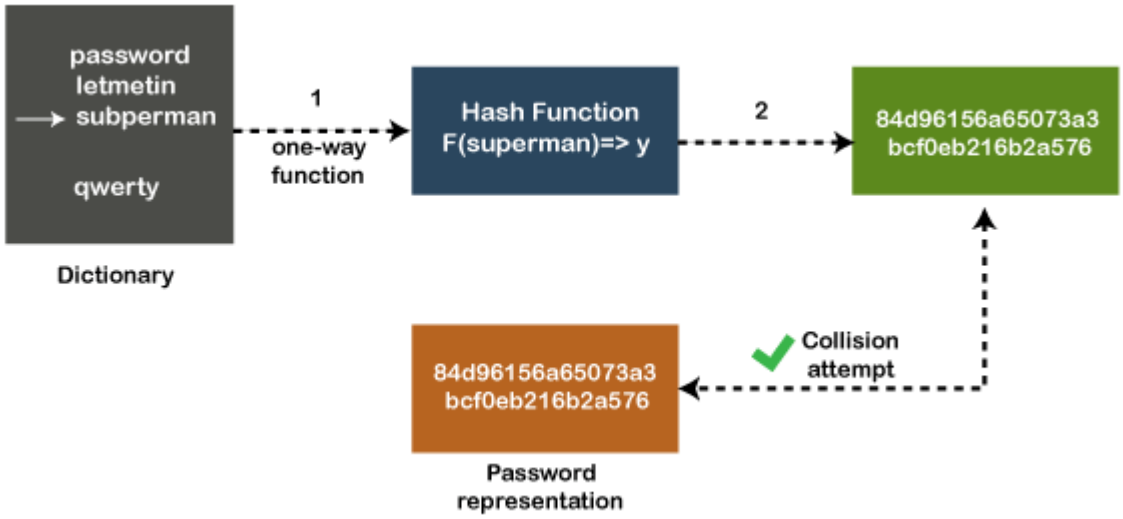
In general, Inter Process Communication is a type of mechanism usually provided by the operating system (or OS). The main aim or goal of this mechanism is to provide communications in between several processes. In short, the intercommunication allows a process letting another process know that some event has occurred.

Let us now look at the general definition of inter-process communication, which will explain the same thing that we have discussed above.

Definition

"Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)."

To understand inter process communication, you can consider the following given diagram that illustrates the importance of inter-process communication:



Role of Synchronization in Inter Process Communication

It is one of the essential parts of inter process communication. Typically, this is provided by interprocess communication control mechanisms, but sometimes it can also be controlled by communication processes.

These are the following methods that used to provide the synchronization:

1. **Mutual Exclusion**
2. **Semaphore**
3. **Barrier**
4. **Spinlock**

Mutual Exclusion:-

It is generally required that only one process thread can enter the critical section at a time. This also helps in synchronization and creates a stable state to avoid the race condition.

Semaphore:-

Semaphore is a type of variable that usually controls the access to the shared resources by several processes. Semaphore is further divided into two types which are as follows:

1. Binary Semaphore
2. Counting Semaphore

Barrier:-

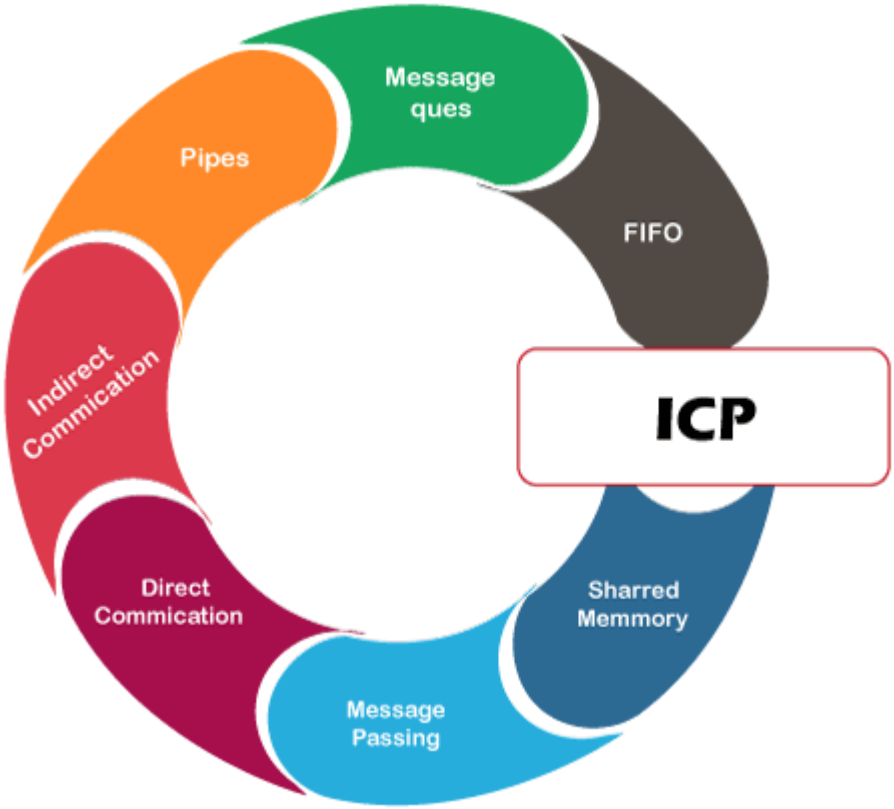
A barrier typically not allows an individual process to proceed unless all the processes does not reach it. It is used by many parallel languages, and collective routines impose barriers.

Spinlock:-

Spinlock is a type of lock as its name implies. The processes are trying to acquire the spinlock waits or stays in a loop while checking that the lock is available or not. It is known as busy waiting because even though the process active, the process does not perform any functional operation (or task).

Approaches to Interprocess Communication

We will now discuss some different approaches to inter-process communication which are as follows:



These are a few different approaches for Inter- Process Communication:

1. **Pipes**
2. **Shared Memory**
3. **Message Queue**
4. **Direct Communication**
5. **Indirect communication**
6. **Message Passing**
7. **FIFO**

To understand them in more detail, we will discuss each of them individually.

Pipe:-

The pipe is a type of data channel that is unidirectional in nature. It means that the data in this type of data channel can be moved in only a single direction at a time. Still, one can use two-channel of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output. These pipes are used in all types of POSIX systems and in different versions of window operating systems as well.

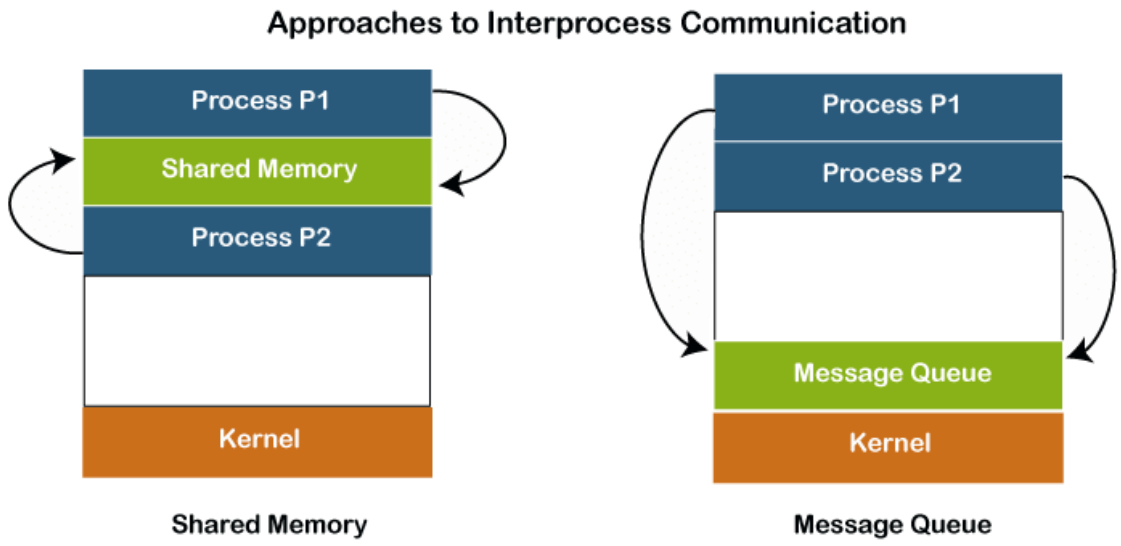
Shared Memory:-

It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously. It is primarily used so that the processes can communicate with each other. Therefore the shared memory is used by almost all POSIX and Windows operating systems as well.

Message Queue:-

In general, several different messages are allowed to read and write the data to the message queue. In the message queue, the messages are stored or stay in the queue unless their recipients retrieve them. In short, we can also say that the message queue is very helpful in inter-process communication and used by all operating systems.

To understand the concept of Message queue and Shared memory in more detail, let's take a look at its diagram given below:



Message Passing:-

It is a type of mechanism that allows processes to synchronize and communicate with each other. However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

Usually, the inter-process communication mechanism provides two operations that are as follows:

- send (message)
- received (message)

Note: The size of the message can be fixed or variable.

Direct Communication:-

In this type of communication process, usually, a link is created or established between two communicating processes. However, in every pair of communicating processes, only one link can exist.

Indirect Communication

Indirect communication can only exist or be established when processes share a common mailbox, and each pair of these processes shares multiple communication links. These shared links can be unidirectional or bi-directional.

FIFO:-

It is a type of general communication between two unrelated processes. It can also be considered as full-duplex, which means that one process can communicate with another process and vice versa.

Some other different approaches

- **Socket:-**

It acts as a type of endpoint for receiving or sending the data in a network. It is correct for data sent between processes on the same computer or data sent between different computers on the same network. Hence, it used by several types of operating systems.

- **File:-**

A file is a type of data record or a document stored on the disk and can be acquired on demand by the file server. Another most important thing is that several processes can access that file as required or needed.

- **Signal:-**

As its name implies, they are a type of signal used in inter process communication in a minimal way. Typically, they are the massages of systems that are sent by one process to another. Therefore, they are not used for sending data but for remote commands between multiple processes.

Usually, they are not used to send the data but to remote commands in between several processes.

Why we need interprocess communication?

There are numerous reasons to use inter-process communication for sharing the data. Here are some of the most important reasons that are given below:

- It helps to speedup modularity
- Computational
- Privilege separation
- Convenience
- Helps operating system to communicate with each other and synchronize their actions as well.

Communication in Client –Server Systems

3.6 Communication in Client-Server Systems

3.6.1 Sockets

- A **socket** is an endpoint for communication.
- Two processes communicating over a network often use a pair of connected sockets as a communication channel. Software that is designed for client-server operation may also use sockets for communication between two processes running on the same computer - For example the UI for a database program may communicate with the back-end database manager using sockets. (If the program were developed this way from the beginning, it makes it very easy to port it from a single-computer system to a networked application.)
- A socket is identified by an IP address concatenated with a port number, e.g. 200.100.50.5:80.

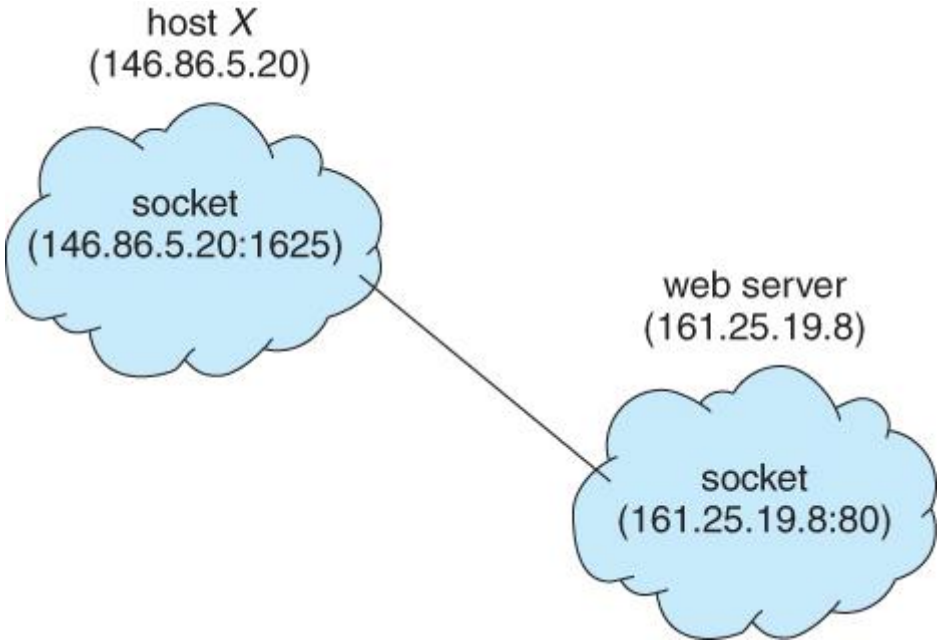


Figure 3.20 - Communication using sockets

- Port numbers below 1024 are considered to be *well-known*, and are generally reserved for common Internet services. For example, telnet servers listen to port 23, ftp servers to port 21, and web servers to port 80.
- General purpose user sockets are assigned unused ports over 1024 by the operating system in response to system calls such as socket( ) or soctkepair( ).

- Communication channels via sockets may be of one of two major forms:
    - **Connection-oriented (TCP, Transmission Control Protocol)** connections emulate a telephone connection. All packets sent down the connection are guaranteed to arrive in good condition at the other end, and to be delivered to the receiving process in the order in which they were sent. The TCP layer of the network protocol takes steps to verify all packets sent, re-send packets if necessary, and arrange the received packets in the proper order before delivering them to the receiving process. There is a certain amount of overhead involved in this procedure, and if one packet is missing or delayed, then any packets which follow will have to wait until the errant packet is delivered before they can continue their journey.
    - **Connectionless (UDP, User Datagram Protocol)** emulate individual telegrams. There is no guarantee that any particular packet will get through undamaged (or at all), and no guarantee that the packets will get delivered in any particular order. There may even be duplicate packets delivered, depending on how the intermediary connections are configured. UDP transmissions are much faster than TCP, but applications must implement their own error checking and recovery procedures.
- Sockets are considered a low-level communications channel, and processes may often choose to use something at a higher level, such as those covered in the next two sections.
- Figure 3.19 and 3.20 illustrate a client-server system for determining the current date using sockets in Java.

```
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      // now listen for connections
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

        // write the Date to the socket
        pout.println(new java.util.Date().toString());

        // close the socket and resume
        // listening for connections
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Figure 3.19 Date server.

```
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args) {
    try {
      //make connection to server socket
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      // read the date from the socket
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      // close the socket connection
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Figure 3.20 Date client.

Figure 3.21 and Figure 3.22

3.6.2 Remote Procedure Calls, RPC

- The general concept of RPC is to make procedure calls similarly to calling on ordinary local procedures, except the procedure being called lies on a remote machine.
- Implementation involves ***stubs*** on either end of the connection.
  - The local process calls on the stub, much as it would call upon a local procedure.
  - The RPC system packages up (marshals) the parameters to the procedure call, and transmits them to the remote system.
  - On the remote side, the RPC daemon accepts the parameters and calls upon the appropriate remote procedure to perform the requested work.
  - Any results to be returned are then packaged up and sent back by the RPC system to the local system, which then unpackages them and returns the results to the local calling procedure.

- One potential difficulty is the formatting of data on local versus remote systems. (e.g. big-endian versus little-endian.) The resolution of this problem generally involves an agreed-upon intermediary format, such as XDR (external data representation.)
- Another issue is identifying which procedure on the remote system a particular RPC is destined for.
    - Remote procedures are identified by *ports*, though not the same ports as the socket ports described earlier.
    - One solution is for the calling procedure to know the port number they wish to communicate with on the remote system. This is problematic, as the port number would be compiled into the code, and it makes it break down if the remote system changes their port numbers.
    - More commonly a *matchmaker* process is employed, which acts like a telephone directory service. The local process must first contact the matchmaker on the remote system (at a well-known port number), which looks up the desired port number and returns it. The local process can then use that information to contact the desired remote procedure. This operation involves an extra step, but is much more flexible. An example of the matchmaker process is illustrated in Figure 3.21 below.
- One common example of a system based on RPC calls is a networked file system. Messages are passed to read, write, delete, rename, or check status, as might be made for ordinary local disk access requests.

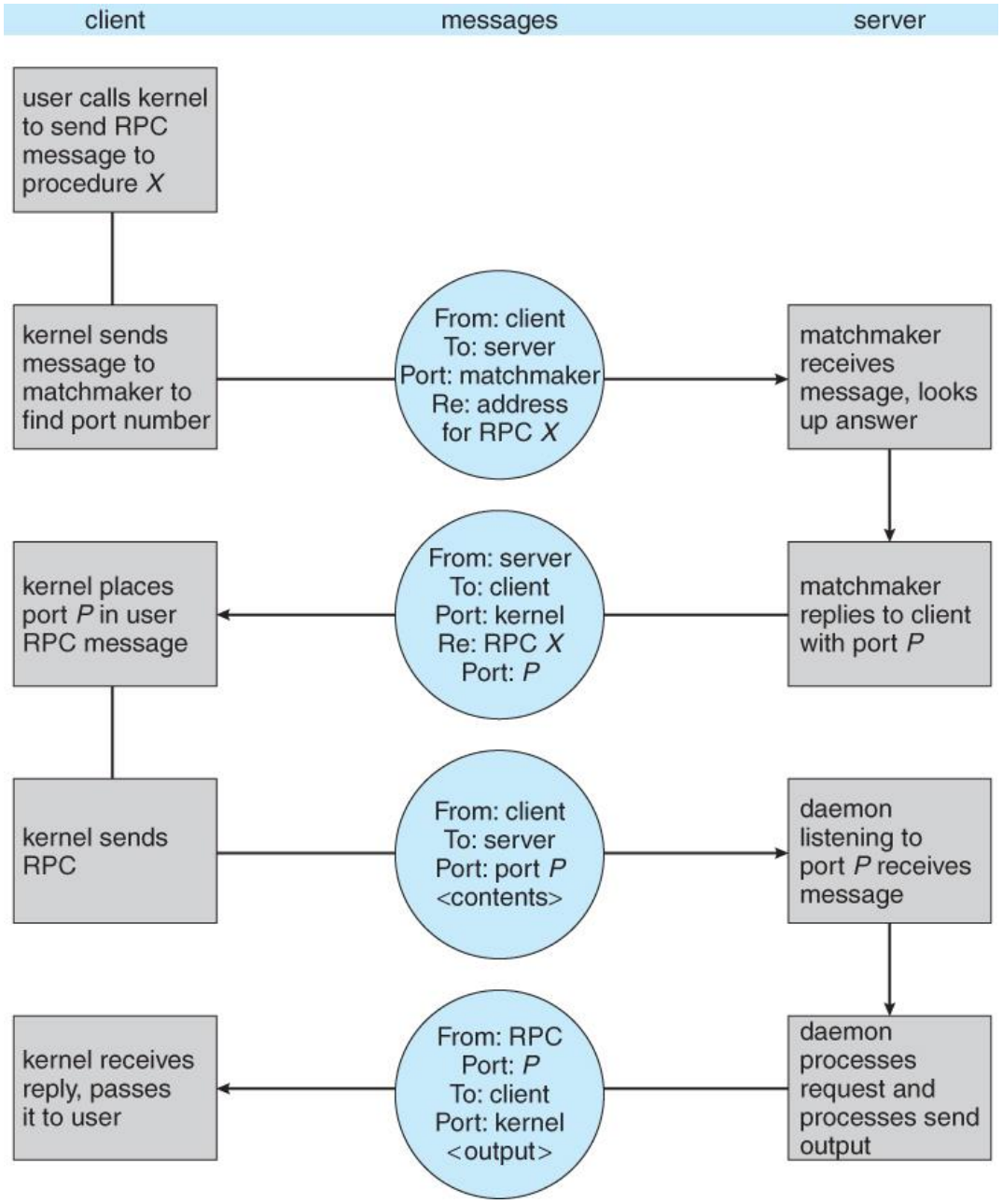


Figure 3.23 - Execution of a remote procedure call (RPC).

3.6.3 Pipes

- ***Pipes*** are one of the earliest and simplest channels of communications between (UNIX) processes.
- There are four key considerations in implementing pipes:
  1. Unidirectional or Bidirectional communication?
  2. Is bidirectional communication half-duplex or full-duplex?
  3. Must a relationship such as parent-child exist between the processes?
  4. Can pipes communicate over a network, or only on the same machine?
- The following sections examine these issues on UNIX and Windows

3.6.3.1 Ordinary Pipes

- Ordinary pipes are uni-directional, with a reading end and a writing end. (If bidirectional communications are needed, then a second pipe is required.)
- In UNIX ordinary pipes are created with the system call "`int pipe( int fd [ ] )`".

- The return value is 0 on success, -1 if an error occurs.
- The int array must be allocated before the call, and the values are filled in by the pipe system call:
  - fd[0] is filled in with a file descriptor for the reading end of the pipe
  - fd[1] is filled in with a file descriptor for the writing end of the pipe
- UNIX pipes are accessible as files, using standard read( ) and write( ) system calls.
- Ordinary pipes are only accessible within the process that created them.
  - Typically a parent creates the pipe before forking off a child.
  - When the child inherits open files from its parent, including the pipe file(s), a channel of communication is established.
  - Each process (parent and child) should first close the ends of the pipe that they are not using. For example, if the parent is writing to the pipe and the child is reading, then the parent should close the reading end of its pipe after the fork and the child should close the writing end.

- Figure 3.22 shows an ordinary pipe in UNIX, and Figure 3.23 shows code in which it is used.

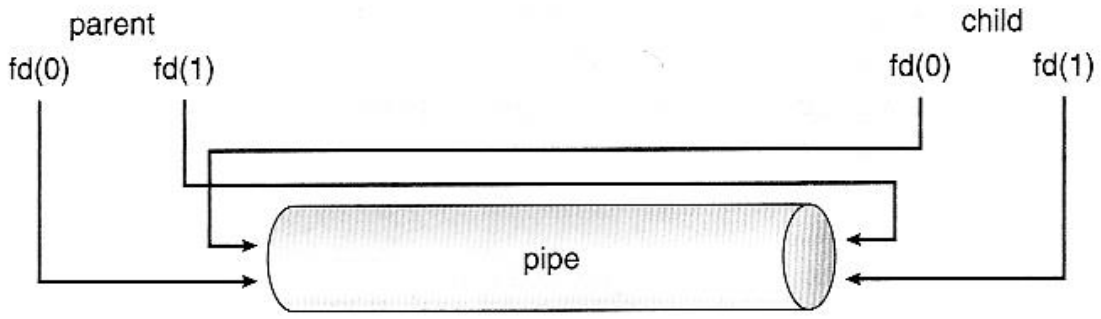


Figure 3.22 File descriptors for an ordinary pipe.

Figure 3.24

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

/* create the pipe */
if (pipe(fd) == -1) {
   fprintf(stderr,"Pipe failed");
   return 1;
}

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}

if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
   /* close the unused end of the pipe */
   close(fd[WRITE_END]);

   /* read from the pipe */
   read(fd[READ_END], read_msg, BUFFER_SIZE);
   printf("read %s",read_msg);

   /* close the write end of the pipe */
   close(fd[READ_END]);
}

return 0;
}
```

Figure 3.23 Ordinary pipes in UNIX.

Figure 3.24 Continuation of Figure 3.23 program.

Figure 3.25 and Figure 3.26

- Ordinary pipes in Windows are very similar
  - Windows terms them ***anonymous*** pipes
  - They are still limited to parent-child relationships.
  - They are read from and written to as files.
  - They are created with `CreatePipe( )` function, which takes additional arguments.
  - In Windows it is necessary to specify what resources a child inherits, such as pipes.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;

/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
  fprintf(stderr, "Create Pipe Failed");
  return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
 TRUE, /* inherit handles */
 0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
  fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Figure 3.25 Windows anonymous pipes – parent process.

Figure 3.26 Continuation of Figure 3.25 program.

Figure 3.27 and Figure 3.28

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

    /* get the read handle of the pipe */
    ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

    /* the child reads from the pipe */
    if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
      printf("child read %s",buffer);
    else
      fprintf(stderr, "Error reading from pipe");

    return 0;
}
```

Figure 3.27 Windows anonymous pipes — child process.

Figure 3.29

3.6.3.2 Named Pipes

- Named pipes support bidirectional communication, communication between non parent-child related processes, and persistence after the process which created them exits. Multiple processes can also share a named pipe, typically one reader and multiple writers.
- In UNIX, named pipes are termed fifos, and appear as ordinary files in the file system.
  - (Recognizable by a "p" as the first character of a long listing, e.g. /dev/initctl)
  - Created with `mkfifo( )` and manipulated with `read( ), write( ), open( ), close( )`, etc.
  - UNIX named pipes are bidirectional, but half-duplex, so two pipes are still typically used for bidirectional communications.
  - UNIX named pipes still require that all processes be running on the same machine. Otherwise sockets are used.
- Windows named pipes provide richer communications.
- Full-duplex is supported.
- Processes may reside on the same or different machines
- Created and manipulated using `CreateNamedPipe( ), ConnectNamedPipe( ), ReadFile( ),and WriteFile( ).`

Race Conditions (Not from the book)

Any time there are two or more processes or threads operating concurrently, there is potential for a particularly difficult class of problems known as ***race conditions.*** The identifying characteristic of race conditions is that the performance varies depending on which process or thread executes their instructions before the other one, and this becomes a problem when the program runs correctly in some instances and incorrectly in others. Race conditions are notoriously difficult to debug, because

they are unpredictable, unrepeatable, and may not exhibit themselves for years.
Here is an example involving a server and a client communicating via sockets:
1. First the server writes a greeting message to the client via the socket:

```
const int BUFFLENGTH = 100;
char buffer[ BUFFLENGTH ];
sprintf( buffer, "Hello Client %d!", i );
write( clientSockets[ i ], buffer, strlen( buffer ) + 1 );
```

2. The client then reads the greeting into its own buffer. The client does not know for sure how long the message is, so it allocates a buffer bigger than it needs to be. The following will read all available characters in the socket, up to a maximum of BUFFLENGTH characters:

```
const int BUFFLENGTH = 100;
char buffer[ BUFFLENGTH ];
read( mysocket, buffer, BUFFLENGTH );
cout << "Client received: " << buffer << "\n";
```

3. Now the server prepares a packet of work and writes that to the socket:

```
write( clientSockets[ i ], & wPacket, sizeof( wPacket ) );
```

4. And finally the client reads in the work packet and processes it:

```
read( mysocket, & wPacket, sizeof( wPacket ) );
```

The Problem: The problem arises if the server executes step 3 before the client has had a chance to execute step 2, which can easily happen depending on process scheduling. In this case, when the client finally gets around to executing step 2, it will read in not only the original greeting, but also the first part of the work packet. And just to make things harder to figure out, the cout << statement in step 2 will only print out the greeting message, since there is a null byte at the end of the greeting. This actually isn't even a problem at this point, but then later when the client executes step 4, it does not accurately read in the work packet because part of it has already been read into the buffer in step 2.
Solution I: The easiest solution is to have the server write the entire buffer in step 1, rather than just the part filled with the greeting, as:

```
write( clientSockets[ i ], buffer, BUFFLENGTH );
```

Unfortunately this solution has two problems: (1) It wastes bandwidth and time by writing more than is needed, and more importantly, (2) It leaves the code open to future problems if the BUFFLENGTH is not the same in the client and in the server.
Solution II: A better approach for handling variable-length strings is to first write the length of the string, followed by the string itself as a separate write. Under this solution the server code changes to:

```
sprintf( buffer, "Hello Client %d!", i );
int length = strlen( buffer ) + 1;
write( clientSockets[ i ], &length, sizeof( int ) );
write( clientSockets[ i ], buffer, length );
```

and the client code changes to:

```
int length;
if( read( mysocket, &length, sizeof( int ) ) != sizeof( int ) ) {
     perror( "client read error: " );
     exit( -1 );
}

if( length < 1 || length > BUFFLENGTH ) {
     cerr << "Client read invalid length = " << length << endl;
     exit( -1 );
}

if( read( mysocket, buffer, length ) != length ) {
perror( "client read error: " );
exit( -1 );
}
```

```
    cout << "Client received: " << buffer << "\n";
```
Note that the above solution also checks the return value from the read system call, to verify that the number of characters read is equal to the number expected. (Some of those checks were actually in the original code, but were omitted from the notes for clarity. The real code also uses select( ) before reading, to verify that there are characters present to read and to delay if not.)
Note also that this problem could not be (easily) solved using the synchronization tools covered in chapter 6, because the problem is not really one of two processes accessing the same data at the same time.

What are Threads?

Thread is an execution unit that consists of its own program counter, a stack, and a set of registers where the program counter mainly keeps track of which instruction to execute next, a set of registers mainly hold its current working variables, and a stack mainly contains the history of execution

Threads are also known as Lightweight processes. Threads are a popular way to improve the performance of an application through parallelism. Threads are mainly used to represent a software approach in order to improve the performance of an operating system just by reducing the overhead thread that is mainly equivalent to a classical process.

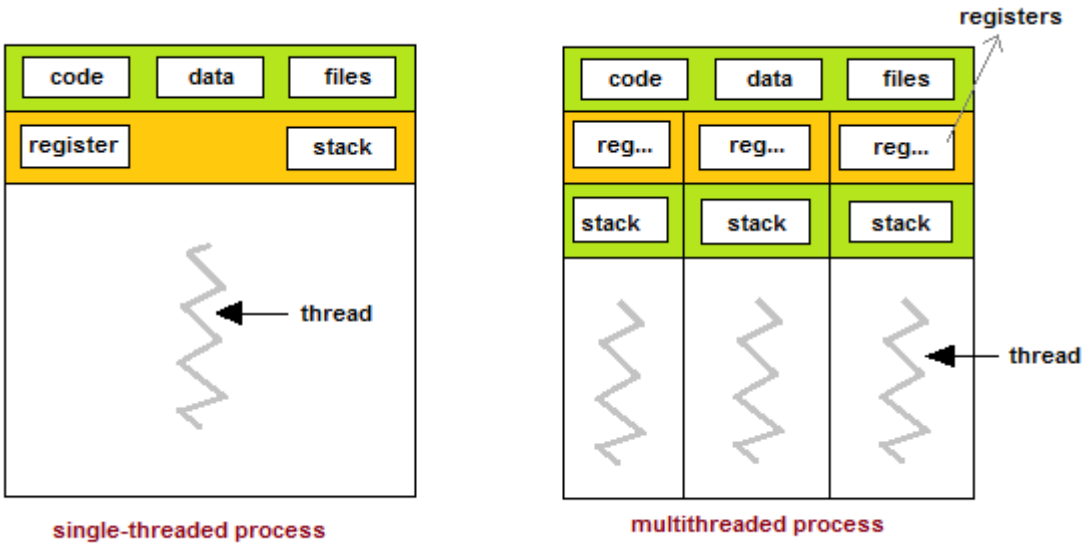
The CPU switches rapidly back and forth among the threads giving the illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution; thus Multiple processes can be executed parallelly by increasing the number of threads.

It is important to note here that each thread belongs to exactly one process and outside a process no threads exist. Each thread basically represents the flow of control separately. In the implementation of network servers and web servers threads have

been successfully used. Threads provide a suitable foundation for the parallel execution of applications on shared-memory multiprocessors.

The given below figure shows the working of a single-threaded and a multithreaded process:



Before moving on further let us first understand the difference between a process and a thread.

Process	Thread
A Process simply means any program in execution.	Thread simply means a segment of a process.
The process consumes more resources	Thread consumes fewer resources.

Process	Thread
The process requires more time for creation.	Thread requires comparatively less time for creation than process.
The process is a heavyweight process	Thread is known as a lightweight process
The process takes more time to terminate	The thread takes less time to terminate.
Processes have independent data and code segments	A thread mainly shares the data segment, code segment, files, etc. with its peer threads.
The process takes more time for context switching.	The thread takes less time for context switching.
Communication between processes needs more time as compared to thread.	Communication between threads needs less time as compared to processes.

Process	Thread
For some reason, if a process gets blocked then the remaining processes can continue their execution	In case if a user-level thread gets blocked, all of its peer threads also get blocked.

Advantages of Thread

Some advantages of thread are given below:

1. Responsiveness
2. Resource sharing, hence allowing better utilization of resources.
3. Economy. Creating and managing threads becomes easier.
4. Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.
5. Context Switching is smooth. Context switching refers to the procedure followed by the CPU to change from one task to another.
6. Enhanced Throughput of the system. Let us take an example for this: suppose a process is divided into multiple threads, and the function of each thread is considered as one job, then the number of jobs completed per unit of time increases which then leads to an increase in the throughput of the system.

Types of Thread

There are two types of threads:

1. User Threads
2. Kernel Threads

User threads are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel-level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

Let us now understand the basic difference between User level Threads and Kernel level threads:

User Level threads	**Kernel Level Threads**
These threads are implemented by users.	These threads are implemented by Operating systems
These threads are not recognized by operating systems,	These threads are recognized by operating systems,
In User Level threads, the Context switch requires no hardware support.	In Kernel Level threads, hardware support is needed.

User Level threads	Kernel Level Threads
These threads are mainly designed as dependent threads.	These threads are mainly designed as independent threads.
In User Level threads, if one user-level thread performs a blocking operation then the entire process will be blocked.	On the other hand, if one kernel thread performs a blocking operation then another thread can continue the execution.
Example of User Level threads: Java thread, POSIX threads.	Example of Kernel level threads: Window Solaris.
Implementation of User Level thread is done by a thread library and is easy.	While the Implementation of the kernel-level thread is done by the operating system and is complex.
This thread is generic in nature and can run on any operating system.	This is specific to the operating system.

What is Multicore Programming?

Multicore programming helps to create concurrent systems for deployment on multicore processor and multiprocessor systems. A multicore processor system is basically a single processor with multiple execution cores in one chip. It has multiple processors on the motherboard or chip. A Field-Programmable Gate Array (FPGA)

is might be included in a multiprocessor system. A FPGA is an integrated circuit containing an array of programmable logic blocks and a hierarchy of reconfigurable interconnects. Input data is processed by to produce outputs. It can be a processor in a multicore or multiprocessor system, or a FPGA.

The multicore programming approach has following advantages −

- Multicore and FPGA processing helps to increase the performance of an embedded system.
- Also helps to achieve scalability, so the system can take advantage of increasing numbers of cores and FPGA processing power over time.

Concurrent systems that we create using multicore programming have multiple tasks executing in parallel. This is known as concurrent execution. When multiple parallel tasks are executed by a processor, it is known as multitasking. A CPU scheduler, handles the tasks that execute in parallel. The CPU implements tasks using operating system threads. So that tasks can execute independently but have some data transfer between them, such as data transfer between a data acquisition module and controller for the system. Data transfer occurs when there is a data dependency.

Multiprocessor and Multicore System in Operating System

Multicore and multiprocessor systems both serve to accelerate the computing process. A multicore contains multiple cores or processing units in a single CPU. A multiprocessor is made up of several CPUs. A multicore processor does not need complex configurations like a multiprocessor. In contrast, A multiprocessor is much reliable and capable of running many programs. In this article, you will learn about the Multiprocessor and Multicore system in the operating system with their advantages and disadvantages.

What is a Multiprocessor System?

A multiprocessor has multiple CPUs or processors in the system. Multiple instructions are executed simultaneously by these systems. As a result, throughput is increased. If one CPU fails, the other processors will continue to work normally. So, multiprocessors are more reliable.

Shared memory or distributed memory can be used in multiprocessor systems. Each processor in a shared memory multiprocessor shares main memory and peripherals to execute instructions concurrently. In these systems, all CPUs access the main memory over the same bus. Most CPUs will be idle as the bus traffic increases. This type of

multiprocessor is also known as the symmetric multiprocessor. It provides a single memory space for all processors.

Each CPU in a distributed memory multiprocessor has its own private memory. Each processor can use local data to accomplish the computational tasks. The processor may use the bus to communicate with other processors or access the main memory if remote data is required.

Advantages and disadvantages of Multiprocessor System

There are various advantages and disadvantages of the multiprocessor system. Some advantages and disadvantages of the multiprocessor system are as follows:

Advantages

There are various advantages of the multiprocessor system. Some advantages of the multiprocessor system are as follows:

1. It is a very reliable system because multiple processors may share their work between the systems, and the work is completed with collaboration.
2. It requires complex configuration.
3. Parallel processing is achieved via multiprocessing.
4. If multiple processors work at the same time, the throughput may increase.
5. Multiple processors execute the multiple processes a few times.

Disadvantages

There are various disadvantages of the multiprocessor system. Some disadvantages of the multiprocessor system are as follows:

1. Multiprocessors work with different systems, so processors require memory space.
2. If one of the processors fails, the work is shared among the remaining processors.
3. These types of systems are very expensive.
4. If any processor is already utilizing an I/O device, additional processors may not utilize the same I/O device that creates deadlock.
5. The operating system implementation is complicated because multiple processors communicate with each other.

What is a Multicore System?

A single computing component with multiple cores (independent processing units) is known as a multicore processor. It denotes the presence of a single CPU with several cores in the system. Individually, these cores may read and run computer instructions. They work in such a way that the computer system appears to have several processors, although they are cores, not processors. These cores may execute normal processors instructions, including add, move data, and branch.

A single processor in a multicore system may run many instructions simultaneously, increasing the overall speed of the system's program execution. It decreases the amount of heat generated by the CPU while enhancing the speed with which instructions are executed. Multicore processors are used in various applications, including general-purpose, embedded, network, and graphics processing (GPU).

The software techniques used to implement the cores in a multicore system are responsible for the system's performance. The extra focus has been put on developing software that may execute in parallel because you want to achieve parallel execution with the help of many cores'

Advantages and disadvantages of Multicore System

There are various advantages and disadvantages of the multicore system. Some advantages and disadvantages of the multicore system are as follows:

Advantages

There are various advantages of the multicore system. Some advantages of the multicore system are as follows:

1. Multicore processors may execute more data than single-core processors.
2. When you are using multicore processors, the PCB requires less space.
3. It will have less traffic.
4. Multicores are often integrated into a single integrated circuit die or onto numerous dies but packaged as a single chip. As a result, Cache Coherency is increased.
5. These systems are energy efficient because they provide increased performance while using less energy.

Disadvantages

There are various disadvantages of the multicore system. Some disadvantages of the multicore system are as follows:

1. Some OSs are still using the single-core processor.
2. These are very difficult to manage than single-core processors.
3. These systems use huge electricity.
4. Multicore systems become hot while doing the work.
5. These are much expensive than single-core processors.
6. Operating systems designed for multicore processors will run slightly slower on single-core processors.

Main Differences between the Multiprocessor and Multicore System

Here, you will learn the main differences between the Multiprocessor and Multicore systems. Various differences between the Multiprocessor and Multicore system are as follows:

1. A multiprocessor system with multiple CPUs allows programs to be processed simultaneously. On the other hand, the multicore system is a single processor with multiple independent processing units called cores that may read and execute program instructions.
2. Multiprocessor systems outperform multicore systems in terms of reliability. A multiprocessor is a computer with many processors. If one of any processors fails in the system, the other processors will not be affected.
3. Multiprocessors run multiple programs faster than the multicore system. On the other hand, a multicore system quickly executes a single program.

4. Multicore systems have less traffic than multiprocessors system because the cores are integrated into a single chip.
5. Multiprocessors require complex configuration. On the other hand, a multicore system doesn't need to be configured.
6. Multiprocessors are expensive as compared to multicore systems. On the other hand, multicore systems are cheaper than multiprocessors systems.

Head-to-head Comparison between the Multiprocessors and Multicore Systems

Here, you will learn the head-to-head comparison between the Multiprocessors and Multicore systems. The main differences between the Multiprocessors and Multicore systems are as follows:

Features	Multiprocessors	Multicore
Definition	It is a system with multiple CPUs that allows processing programs simultaneously.	A multicore processor is a single processor that contains multiple independent processing units known as cores that may read and execute program instructions.
Execution	Multiprocessors run multiple programs faster than a multicore system.	The multicore executes a single program faster.
Reliability	It is more reliable than the multicore system. If one of any processors fails in the system, the other processors will not be affected.	It is not much reliable than the multiprocessors.
Traffic	It has high traffic than the multicore system.	It has less traffic than the multiprocessors.
Cost	It is more expensive as compared to a multicore system.	These are cheaper than the multiprocessors system.
Configuration	It requires complex configuration.	It doesn't need to be configured.

Conclusion

The terms multicore and multiprocessor differ in which multicore system refers to a single CPU with several execution units, while multiprocessor refers to a system with multiple CPUs. A multicore system would be more efficient if you only need to run one program. However, a multiprocessor machine would be faster if you have numerous apps running. Modern systems have multiple processors, and each of them has multiple cores.

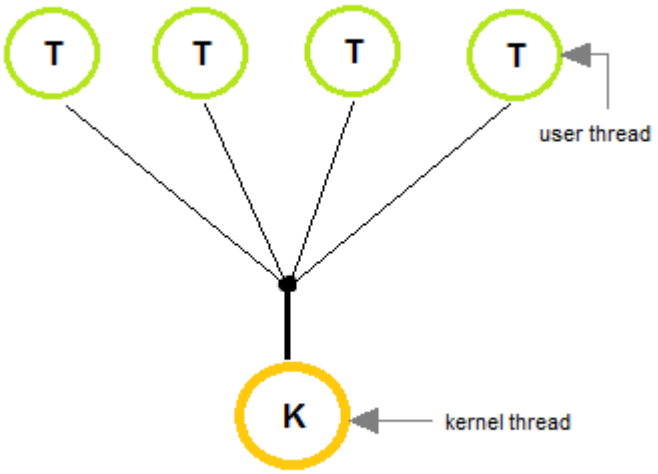
Multithreading Models

The user threads must be mapped to kernel threads, by one of the following strategies:

- Many to One Model
- One to One Model
- Many to Many Model

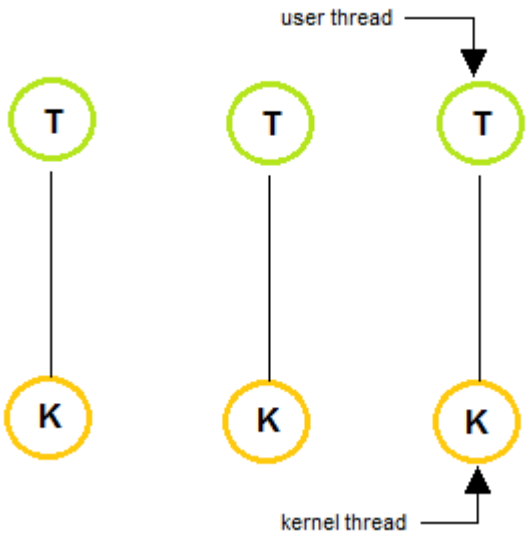
Many to One Model

- In the **many to one** model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is efficient in nature.
- In this case, if user-level thread libraries are implemented in the operating system in some way that the system does not support them, then the Kernel threads use this many-to-one relationship model.



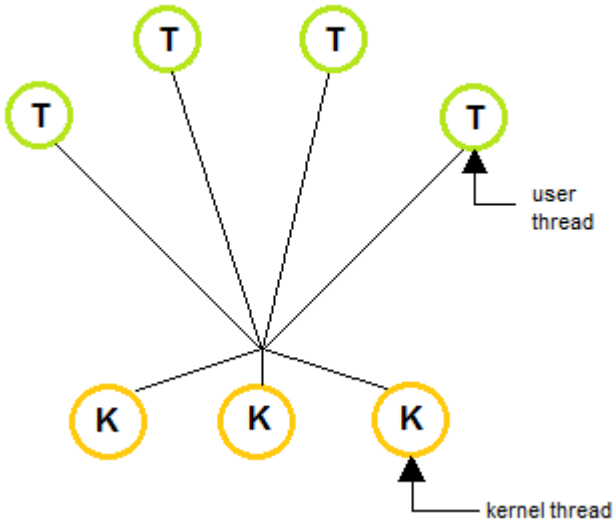
One to One Model

- The **one to one** model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.
- This model provides more concurrency than that of many to one Model.



Many to Many Model

- The **many to many** model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.



What are Thread Libraries?

Thread libraries provide programmers with API for the creation and management of threads.

Thread libraries may be implemented either in user space or in kernel space. The user space involves API functions implemented solely within the user space, with no kernel support. The kernel space involves system calls and requires a kernel with thread library support.

Three types of Thread

1. **POSIX Pitheads** may be provided as either a user or kernel library, as an extension to the POSIX standard.
2. **Win32 threads** are provided as a kernel-level library on Windows systems.
3. **Java threads**: Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system.

Multithreading Issues

Below we have mentioned a few issues related to multithreading. Well, it's an old saying, *All good things, come at a price*.

Thread Cancellation

Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be canceled.

Signal Handling

Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all or a single thread.

fork() System Call

fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in the Multithreaded process is, if one thread forks, will the entire process be copied or not?

Security Issues

Yes, there can be security issues because of the extensive sharing of resources between multiple threads.

There are many other issues that you might face in a multithreaded process, but there are appropriate solutions available for them. Pointing out some issues here was just to study both sides of the coin.

Process Synchronization

In this tutorial, we will be covering the concept of Process synchronization in an Operating System.

Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

- Independent Process
- Cooperative Process

Independent Processes

Two processes are said to be independent if the execution of one process does not affect the execution of another process.

Cooperative Processes

Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes

need to be synchronized so that the order of execution can be guaranteed.

Process Synchronization

It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

- It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
- In order to synchronize the processes, there are various synchronization mechanisms.
- Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time.

Race Condition

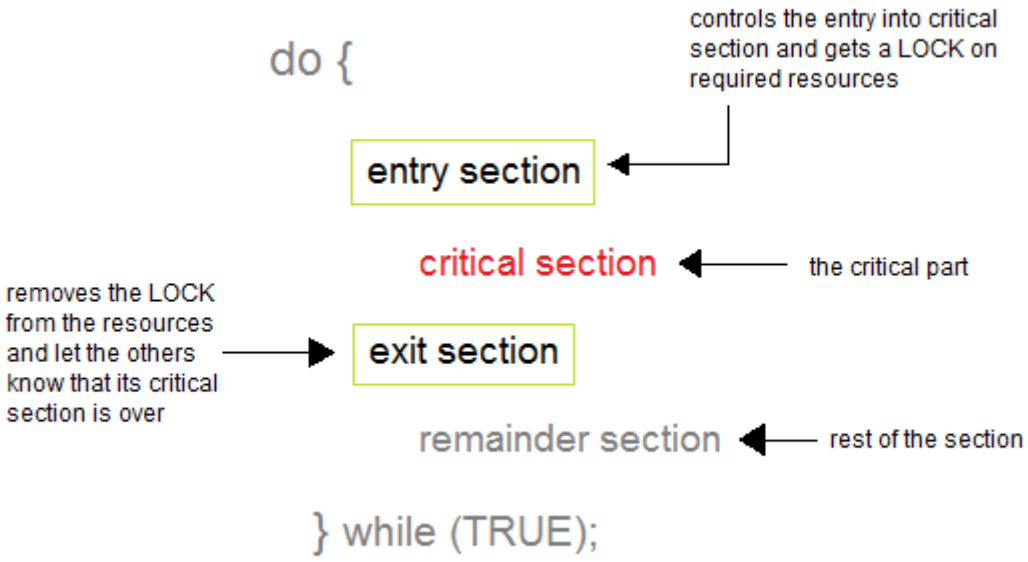
At the time when more than one process is either executing the same code or accessing the same memory or any shared variable; In that condition, there is a possibility that the output or the value of the shared variable is wrong so for that purpose all the processes are doing the race to say that my output is correct. This condition is commonly known as **a race condition.** As several processes access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the access of data takes place.

Mainly this condition is a situation that may occur inside the **critical section**. Race condition in the critical section happens when the result of multiple thread execution differs according to the order in which the threads execute. But this condition is critical sections can be avoided if the critical section is treated as an atomic instruction.

Proper thread synchronization using locks or atomic variables can also prevent race conditions.

Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes. The entry to the critical section is mainly handled by `wait()` function while the exit from the critical section is controlled by the `signal()` function.



Entry Section

In this section mainly the process requests for its entry in the critical section.

Exit Section

This section is followed by the critical section.

The solution to the Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

Solutions for the Critical Section

The critical section plays an important role in Process Synchronization so that the problem must be solved.

Some widely used method to solve the critical section problem are as follows:

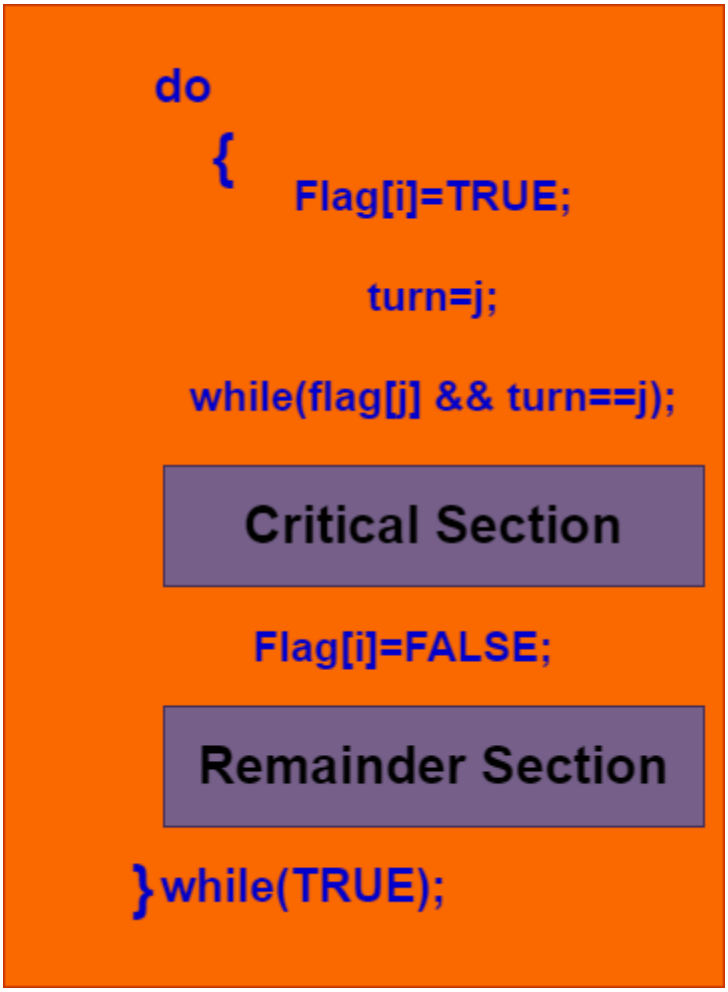
1.Peterson's Solution

This is widely used and software-based solution to critical section problems. Peterson's solution was developed by a computer scientist Peterson that's why it is named so.

With the help of this solution whenever a process is executing in any critical state, then the other process only executes the rest of the code, and vice-versa can happen. This method also helps to make sure of the thing that only a single process can run in the critical section at a specific time.

This solution preserves all three conditions:

- Mutual Exclusion is comforted as at any time only one process can access the critical section.
- Progress is also comforted, as a process that is outside the critical section is unable to block other processes from entering into the critical section.
- Bounded Waiting is assured as every process gets a fair chance to enter the Critical section.



The above shows the structure of process **Pi in Peterson's solution.**

- Suppose there are **N processes (P1, P2, ... PN)** and as at some point of time every process requires to enter in the **Critical Section**
- A **FLAG[]** array of size N is maintained here which is by default false. Whenever a process requires to enter in the critical section, it has to set its flag as true. Example: If Pi wants to enter it will set **FLAG[i]=TRUE.**
- Another variable is called **TURN** and is used to indicate the process number that is currently waiting to enter into the critical section.
- The process that enters into the critical section while exiting would change the **TURN** to another number from the list of processes that are ready.

- Example: If the turn is 3 then P3 enters the Critical section and while exiting turn=4 and therefore P4 breaks out of the wait loop.

Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time-consuming as the message is passed to all the processors.

This message transmission lag delays the entry of threads into the critical section, and the system efficiency decreases.

Mutex Locks

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside the critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

Classical Problems of Synchronization

In this tutorial we will discuss about various classical problem of synchronization.

Semaphore can be used in other synchronization problems besides Mutual Exclusion.

Below are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.

We will discuss the following three problems:

1. Bounded Buffer (Producer-Consumer) Problem
2. Dining Philosophers Problem
3. The Readers Writers Problem

Bounded Buffer Problem

Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- In this Producers mainly produces a product and consumers consume the product, but both can use of one of the containers each time.
- The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.

Dining Philosophers Problem

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

The Readers Writers Problem

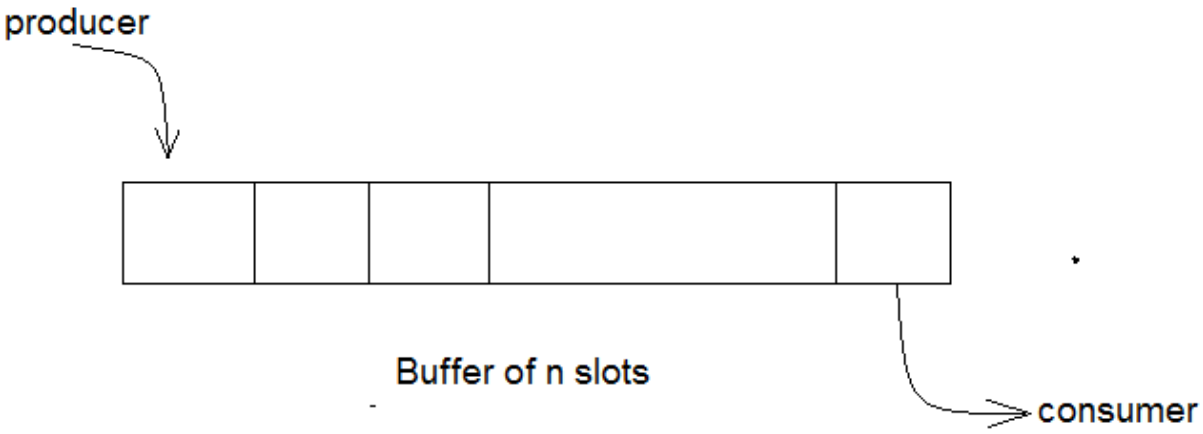
- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centred on relative priorities of readers and writers.
- The main complexity with this problem occurs from allowing more than one reader to access the data at the same time.

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- `m`, a **binary semaphore** which is used to acquire and release the lock.
- `empty`, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- `full`, a **counting semaphore** whose initial value is `0`.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do

{

    // wait until empty > 0 and then decrement
'empty'

    wait(empty);

    // acquire lock

    wait(mutex);



    /* perform the insert operation in a slot */
```

```
    // release lock
    signal(mutex);
    // increment 'full'
    signal(full);
}
while(TRUE)
```

Copy

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
```

```
{

    // wait until full > 0 and then decrement
'full'

    wait(full);

    // acquire the lock

    wait(mutex);



    /* perform the remove operation in a slot */



    // release the lock

    signal(mutex);

    // increment 'empty'

    signal(empty);

}

while(TRUE);
```

Copy

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.

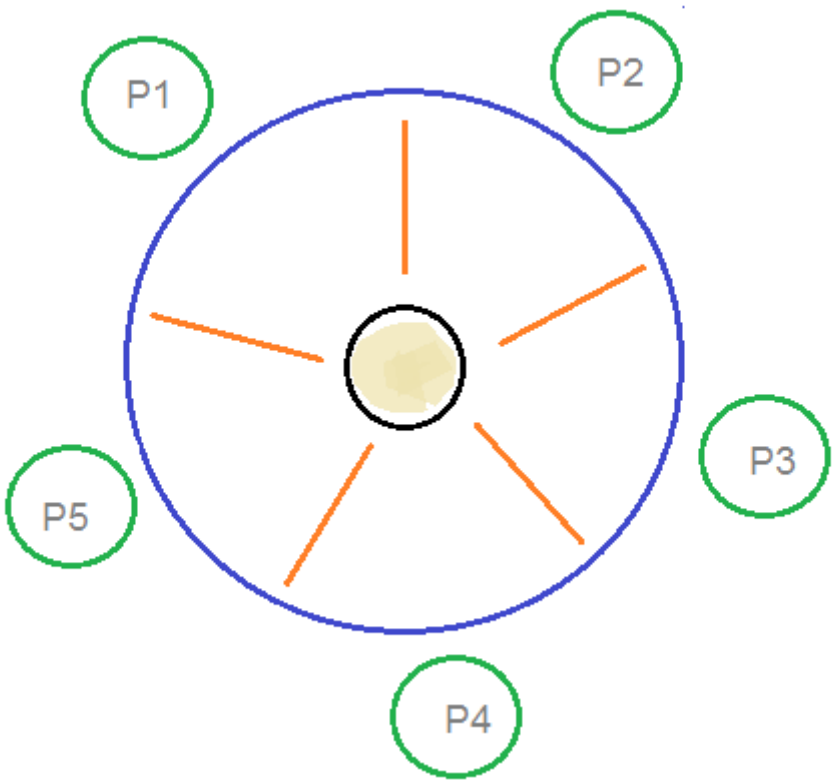
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Dining Philosophers Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



Dining Philosophers Problem

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, `stick[5]`, for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE)

{

    wait(stick[i]);

    /*

        mod is used because if i=5, next

        chopstick is 1 (dining table is circular)

    */

    wait(stick[(i+1) % 5]);



    /* eat */

    signal(stick[i]);



    signal(stick[(i+1) % 5]);

    /* think */

}
```

Copy

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

What is Readers Writer Problem?

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex** `m` and a **semaphore** `w`. An integer variable `read_count` is used to maintain the number of readers currently accessing the resource. The variable `read_count` is initialized to `0`. A value of `1` is given initially to `m` and `w`.

Instead of having the process to acquire lock on the shared resource, we use the mutex `m` to make the process to acquire and release lock whenever it is updating the `read_count` variable.

The code for the **writer** process looks like this:

```
while(TRUE)

{

     wait(w);



   /* perform the write operation */



   signal(w);

}
```

Copy

And, the code for the **reader** process looks like this:

```
while(TRUE)

{

    //acquire lock

    wait(m);

    read_count++;

    if(read_count == 1)

        wait(w);


    //release lock

    signal(m);


    /* perform the reading operation */


    // acquire lock

    wait(m);

    read_count--;

    if(read_count == 0)

        signal(w);
```

```
    // release lock

    signal(m);

}
```

Copy

Here is the Code uncoded(explained)

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are

zero readers now and a writer can have the chance to access the resource.

Introduction to Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called a **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by `P(S)` and `V(S)` respectively.

In very simple words, the **semaphore** is a variable that can hold only a non-negative Integer value, shared between all the threads, with operations **wait** and **signal**, which work as follow:

```
P(S): if S >= 1 then S := S - 1
      else <block and enqueue the process>;


V(S): if <some process is blocked on the queue>
        then <unblock a process>
      else S := S + 1;
```

Copy

The classical definitions of **wait** and **signal** are:

- **Wait**: This operation decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1). This Operation mainly helps you to control the entry of a task into the critical section. In the case of the negative or zero value, no operation is executed. wait() operation was originally termed as P; so it is also known as **P(S) operation**. The definition of wait operation is as follows:

```
wait(S)

{

   while (S<=0);//no operation

   S--;

}
```

Note:

When one process modifies the value of a semaphore then, no other process can simultaneously modify that same semaphore's value. In the above case the integer value of S(S<=0) as well as the possible modification that is S-- must be executed without any interruption.

- **Signal**: Increments the value of its argument S, as there is no more process blocked on the queue. This Operation is mainly used to control the exit of a task from the critical section.signal() operation was originally termed as V; so it is also known as **V(S) operation**. The definition of signal operation is as follows:

```
signal(S)

{
```

```
S++;

}
```

Also, note that all the modifications to the integer value of semaphore in the `wait()` and `signal()` operations must be executed indivisibly.

Properties of Semaphores

1. It's simple and always have a non-negative integer value.
2. Works with many processes.
3. Can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into the critical section at once, if desirable.

We will now cover the types of semaphores in the Operating system;

Types of Semaphores

Semaphores are mainly of two types in Operating system:

1. **Binary Semaphore:**

   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**. A binary semaphore is initialized to `1` and only takes the values `0` and `1` during the execution of a program. In Binary Semaphore, the wait operation works only if the value of semaphore = 1, and the signal operation succeeds when the semaphore= 0. Binary Semaphores are easier to implement than counting semaphores.

2. **Counting Semaphores:**

   These are used to implement **bounded concurrency**. The Counting semaphores can range over an **unrestricted domain**. These can be used to control access to a given resource that consists of a finite number of Instances. Here the semaphore count is used to indicate the number of available resources. If the resources are added then the semaphore count automatically gets incremented and if the resources are removed, the count is decremented. Counting Semaphore has no mutual exclusion.

Example of Use

Here is a simple step-wise implementation involving declaration and usage of semaphore.

```
Shared var mutex: semaphore = 1;

Process i

   begin

   .

   .

   P(mutex);

   execute CS;

   V(mutex);

   .

   .

   End;
```

Advantages of Semaphores

Benefits of using Semaphores are as given below:

- With the help of semaphores, there is a flexible management of resources.
- Semaphores are machine-independent and they should be run in the machine-independent code of the microkernel.
- Semaphores do not allow multiple processes to enter in the critical section.
- They allow more than one thread to access the critical section.
- As semaphores follow the mutual exclusion principle strictly and these are much more efficient than some other methods of synchronization.
- No wastage of resources in semaphores because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if any condition is fulfilled in order to allow a process to access the critical section.

Disadvantages of Semaphores

- One of the biggest limitations is that semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the critical section later.
- To avoid deadlocks in the semaphore, the Wait and Signal operations are required to be executed in the correct order.
- Using semaphores at a large scale is impractical; as their use leads to loss of modularity and this happens because the wait() and signal() operations prevent the creation of the structured layout for the system.
- Their use is not enforced but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called **Deadlock**. We will be studying deadlocks in detail in coming lessons.