**Program 1.1:** Write a C Program to simulate the FCFS (First Come First Server) CPU scheduling algorithm.

**Aim: To implement FCFS CPU scheduling algorithm in c**

**Description:**

First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue. In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed. We are given with the n number of processes i.e. P1, P2, P3,.......,Pn and their corresponding burst times. The task is to find the average waiting time and average turnaround time using FCFS CPU Scheduling algorithm. Here we are considering that arrival time for all processes is 0.

* Turnaround Time is the time interval between the submission of a process and its completion.

Turnaround Time = completion of a process – submission of a process

* Waiting Time is the difference between turnaround time and burst time

Waiting Time = turnaround time – burst time

**Algorithm:**

1. Start
2. Declare the following array to maximum size 15.
   1. WT[ MAX ] to store waiting time of each process.
   2. BT[ MAX ] to store Burst time of each process.
   3. TT [ MAX ] to store turnaround time of each process.
3. Read the number of processes.
4. Read the Burst time for all the processes.
5. Calculate the waiting time of each process
   1. WT [ i ] = WT [ i-1] + BT [i-1]
6. Calculate the Turnaround time of each process
   1. TT [ i ] = WT [ i ] + BT [ i ]
7. Calculate average waiting time and average turnaround time.
8. Display all the values
9. Stop.

**Program:**

#include<stdio.h>

int main()

{

int processes[] = { 1, 2, 3};

int n =3;

int bt[] = {10, 5, 8};

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

wt[0] = 0;

// calculating waiting time

for (int i = 1; i < n ; i++ )

wt[i] = bt[i-1] + wt[i-1] ;

// calculating turnaround time

for (int i = 0; i < n ; i++)

tat[i] = bt[i] + wt[i];

//Display processes along with all details

printf("Processes Burst time Waiting time Turn around time\n");

// Calculate total waiting time and total turn

// around time

for (int i=0; i<n; i++)

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

printf(" %d ",(i+1));

printf(" %d ", bt[i] );

printf(" %d",wt[i] );

printf(" %d\n",tat[i] );

}

int s=(float)total\_wt / (float)n;

int t=(float)total\_tat / (float)n;

printf("Average waiting time = %d",s);

printf("\n");

printf("Average turn around time = %d ",t);

return 0;

}

**OUTPUT**

**RUN 1:**

Processes    Burst time    Waiting time    Turn around time

   1         10         0        10

   2          5         10        15

   3         8         15        23

Average waiting time = 8

Average turn around time = 16

**VIVA Questions:**

1. **What are the features of FCFS scheduling algorithm?**
   1. Jobs are executed on first come, first serve basis.
   2. It is a non-preemptive scheduling algorithm.
   3. Easy to understand and implement.
   4. Its implementation is based on FIFO queue.
   5. Poor in performance as average wait time is high.
2. **What is CPU scheduling ()?**

Whenever the CPU becomes idle, it is the job of the CPU Scheduler to select another process from the ready queue to run next by CPU.

or

CPU scheduling is technique through which a process from the ready queue is selected for execution by CPU.

1. **Name some of the CPU Scheduling Algorithm.**
   1. FCFS (First-Come First-Serve Scheduling)
   2. SJF(Shortest-Job-First Scheduling)
   3. Priority scheduling.
   4. Round Robin scheduling.
   5. Multilevel Queue scheduling.
   6. Multilevel feedback queue scheduling.
2. **What is preemptive scheduling?**

In preemptive scheduling the resources are allocated to a process for a limited time, and process can be interrupted in between.

1. **What is non- preemptive scheduling?**

In non-preemptive scheduling once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state. Process cannot be interrupted till it terminates or switches to waiting state.

1. **What are CPU scheduling criteria ?**

There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:

* + 1. CPU utilization
    2. Throughput
    3. Turnaround time
    4. Waiting time
    5. Response time

**7. What is the Process in OS?**

Applications run on the computer. The running or execution instance of an application is called as the process. It is also considered as an execution unit.

**8. What is the Thread in OS?**

The process may contain multiple threads. Suppose you have notepad application installed on your system. You click on notepad icon to note down some points. The process starts running.  Then you open another instance of notepad to save other data. The two threads are created to run two instances of the notepad. You can work on any of the opened notepad windows independently.

## 9. Difference Between Process and Thread

* The thread is also called as a lightweight process. It requires fewer resources as compare to process to run.
* One single process can consist of multiple threads.
* Every process requires its own address space to run on a processor. Whereas threads within a single process share the single address space. So threads are easier to create than process.
* Threads read and write at the same place. It is good and easy for the communication between multiple threads in the single process.
* The data communications between multiple processes are difficult and it is carried out by IPC (inter-process communication).
* Context switching overhead is less in the thread as compare to process. So threads decrease overall execution time and the cost of the communication.
* Threads are useful to execute lightweight task whereas processes are responsible for running heavyweight tasks.

**Program 1.2 :** Write a C Program to simulate the SJF ( Shortest Job First) CPU scheduling algorithm.

**Aim: To implement SJF CPU scheduling algorithm in c**

**Description:** **Shortest Job First (SJF)** is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

**Algorithm:**

1. Start
2. Declare Process structure with following members.
   1. PID : to Store Process ID
   2. WT : to store waiting time of process.
   3. BT : to store Burst time of process.
   4. TT : to store turnaround time of process.
3. Read the number of processes.
4. Read the Burst time and Process Id for all the processes.
5. Sort the Array of process structure based on Burt time.
6. Calculate the waiting time of each process
   1. After sorting first process in the list will have waiting time 0
      1. Proc[0] .wt = 0
   2. Find Waiting time for all other process
      1. Proc[ i ] . wt = Proc[ i-1] .wt + Pro [ i-1] .bt
7. Calculate the Turnaround time of each process
   * 1. Proc[ i ] . tt = Proc[ i] .wt + Pro [ i] .bt
8. Calculate average waiting time
   1. Total\_waitingTime / Number of Process
9. Calculate average turnaround time.
   1. Total\_waitingTime / Number of Process
10. Display all the values
11. Stop.

**Program:**

#include <stdio.h>

#define MAX 15

struct process

{

int pid;

int bt;

int wt;

int tt;

};

int main()

{

struct process proc[MAX], temp;

int n , i , j;

int total\_wt=0 , total\_tt=0;

float avg\_wt, avg\_tt;

printf("enter number of process (max of 15) \n");

scanf("%d",&n);

printf(" Enter Process Id and Brust time of the process \n");

for( i=0; i< n; i++)

{

scanf("%d%d", &proc[i].pid , &proc[i].bt );

}

//Bubble sort, to sort the process

for (i = 0; i < n-1; i++)

for (j = 0; j < n-i-1; j++)

if (proc[j].bt > proc[j+1].bt)

{

temp=proc[j];

proc[j]= proc[j+1];

proc[j+1]=temp;

}

for (i = 0; i < n; i++)

{

if(i= =0)

{

proc[0].wt=0;

proc[0].tt = proc[0].bt;

}

else

{

proc[i].wt = proc[i-1].wt + proc[i-1].bt; // waiting time calculation

total\_wt = total\_wt+proc[i].wt; // total waiting time calculation

proc[i].tt = proc[i].wt+proc[i].bt ; // turn around time calculation

total\_tt = total\_tt+proc[i].tt; // total turn around time calculation

}

}

avg\_wt = (float) total\_wt/n;

avg\_tt = (float) total\_tt/n;

printf(" Process ID Burst time Waiting Time Turn Around Time \n");

for (i = 0; i < n; i++)

printf(" %d %d %d %d \n", proc[i].pid , proc[i].bt, proc[i].wt, proc[i].tt);

printf("Average waiting time = %f",avg\_wt);

printf("\n");

printf("Average turn around time = %f ",avg\_tt);

return 0;

}

OUTPUT

RUN1 :

enter number of process (max of 15)   4

Enter Process Id  and  Brust time of the process

2 45

3 56

1 78

4 7

Process ID    Burst time   Waiting Time   Turn Around Time

      4              7           0              7

      2              45           7              52

      3              56           52              108

      1              78           108              186

Viva Questions:

1. **What dose scheduling mean?**
   * + 1. **CPU scheduling** deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms
       2. Scheduling is done to balance the load on the system and ensure equal distribution of resources and give some prioritization according to set rules. This ensures that a computer system is able to serve all requests and achieve a certain quality of service.
       3. Scheduling is also known as process scheduling.
2. **What are the main concern of scheduler?**
   * + 1. Throughput, or how fast it can finish a certain number of tasks from beginning to end per unit of time
       2. Latency, which is the turnaround time or the time it takes to finish the task from the time of request or submission until finish, which includes the waiting time before it could be served
       3. Response time, which is the time it takes for the process or request to be served, in short the waiting time.
3. **What does *Shortest Job First (SJF)* mean?**
   * + 1. Shortest job first is a scheduling algorithm in which the process with the smallest execution time is selected for execution next. Shortest job first can be either preemptive or non-preemptive.
       2. It also reduces the average waiting time for other processes awaiting execution.
       3. Shortest job first is also known as shortest job next (SJN) and shortest process next (SPN).
4. **What are different CPU scheduling criteria?**
   * + 1. CPU utilization – keep the CPU as busy as possible
       2. Throughput – Number of processes that complete their execution per time unit
       3. Turnaround time – amount of time to execute a particular process
       4. Waiting time – amount of time a process has been waiting in the ready queue
       5. Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
5. **What are the disadvantages of SJF**
   * + 1. It can cause process starvation for longer jobs if there are a large number of shorter processes.
       2. The need to know the execution time for each process beforehand. Often, this is almost impossible in many environments.
6. **What are Scheduling Algorithm Optimization Criteria**
   * + 1. Max CPU utilization
       2. Max throughput
       3. Min turnaround time
       4. Min waiting time
       5. Min response time
7. **What is starvation ?**

**Starvation** or indefinite blocking is phenomenon associated with the Priority scheduling algorithms, in which a process ready to run for CPU can wait indefinitely because of low priority. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

**Program 1.3:** Write a C Program to simulate the RR ( Round Robin ) CPU scheduling algorithm.

**AIM: To implement RR CPU scheduling algorithm in c**

**Description:** The *round-robin* (*RR*) scheduling is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between process. A small unit of time, called a time quantum, or time slice, is defined. A time quantum is generally from 10 to 100ms. The ready queue is treated as a circular queue. RR goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. RR is preemptive. The performance of RR depends heavily on the size of the time quantum selected. At one extreme, if the quantum is very large (infinite), RR is the same as the FCFS policy. If the time quantum is very small, it would appear each process among n processes has its own processor running at 1/n the speed of the real processor.

**ALGORITHM:**

1. Start
2. Declare Process structure with following members.
   1. PID : to Store Process ID
   2. WT : to store waiting time of process.
   3. BT : to store Burst time of process.
   4. TT : to store turnaround time of process.
   5. rem\_bt : to store remaining burst time
3. Read the number of processes.
4. Read the Burst time and Process Id for all the processes.
5. Read the time quantum value.
6. The process are selected from the queue in first come first serve order and following steps are repeated until remaining burst time for all process is zero.
   1. If Burst time > time quantum
      1. The process is preempted after time quantum and added to the end of the queue. Remaing burst time is recorded. CPU execution time is also recorded
         1. t = t + quantum
         2. Proc[i] . rem\_bt = Proc[i].rem\_bt - quantum
   2. If Burst Time <= time quantum
      1. Process remaining burst time is set to zero and waiting time is calculated
         1. t = t + Proc[i] . rem\_bt
         2. Proc[i] . rem\_bt = 0
         3. Proc[ i ] .wt = t - Proc [ i ] .bt
7. Calculate the Turnaround time of each process
   * 1. Proc[ i ] . tt = Proc[ i] .wt + Pro [ i] .bt
8. Calculate average waiting time
   1. Total\_waitingTime / Number of Process
9. Calculate average turnaround time.
   1. Total\_waitingTime / Number of Process
10. Display all the values
11. Stop.

**Program:**

#include <stdio.h>

#define MAX 15

struct process

{

int pid;

int bt;

int wt;

int tt;

int rem\_bt;

};

int main()

{

struct process proc[MAX] ;

int n , i , j ,quantum , t;

int total\_wt=0 , total\_tt=0;

float avg\_wt, avg\_tt;

int done=0;

printf("enter number of process (max of 15) \n");

scanf("%d",&n);

printf(" Enter Process Id and Brust time of the process \n");

for( i=0; i< n; i++)

{

scanf("%d%d", &proc[i].pid , &proc[i].bt );

proc[i].rem\_bt = proc[i].bt;

}

printf("Enter time Quantum \n");

scanf("%d",&quantum);

t = 0;

// Keep traversing processes in round robin manner until all of them are not done.

printf("Order of Process Excecution is \n");

do

{

done = 1;

for (int i = 0 ; i < n; i++) // Traverse all processes one by one repeatedly

{

// If burst time of a process is greater than 0 then only need to process further

if (proc[i].rem\_bt > 0)

{

done = 0; // There is a pending process

printf(" %d : ",proc[i].pid);

if (proc[i].rem\_bt > quantum)

{

// Increase the value of t which shows how much time a process has been processed

t += quantum;

// Decrease the burst\_time of current process by quantum

proc[i].rem\_bt= proc[i].rem\_bt - quantum;

}

// If burst time is smaller than or equal to quantum. Last cycle for this process

else

{

// Increase the value of t, which shows how much time a process has been processed

t = t + proc[i].rem\_bt;

// Waiting time is current time minus time used by this process

proc[i].wt = t - proc[i].bt;

// As the process gets fully executed make its remaining burst time = 0

proc[i].rem\_bt = 0;

}

}

}

}while(done != 1);

printf("\n");

for (i = 0; i < n; i++)

{

total\_wt = total\_wt + proc[i].wt; // total waiting time calculation

proc[i].tt = proc[i].wt + proc[i].bt ; // turn around time calculation

total\_tt = total\_tt + proc[i].tt; // total turn around time calculation

}

avg\_wt = (float) total\_wt/n;

avg\_tt = (float) total\_tt/n;

printf(" | PID | BT | WT | TaT | \n");

for (i = 0; i < n; i++)

printf(" %d : %d : %d : %d \n", proc[i].pid , proc[i].bt , proc[i].wt , proc[i].tt );

printf("Average waiting time %f \n, ", avg\_wt);

printf("Average Turn around time %f \n " ,avg\_tt);

return 0;

}

enter number of process (max of 15)

4

Enter Process Id and Brust time of the process

1 4

2 3

3 5

4 6

Enter time Quantum

2

Order of Process Excecution is

1 : 2 : 3 : 4 : 1 : 2 : 3 : 4 : 3 : 4 :

Processes are executed in following order

PID | BT | WT | TaT

1 : 4 : 6 : 10

2 : 3 : 8 : 11

3 : 5 : 11 : 16

4 : 6 : 12 : 18

Average waiting time 9.250000

Average Turnaround time 13.750000

VIVA QUESTIONS

1. **What are the characteristic of RR?**
   1. Round Robin is a [CPU scheduling algorithm](http://quiz.geeksforgeeks.org/gate-notes-operating-system-process-scheduling/) where each process is assigned a fixed time slot in a cyclic way.
   2. It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
   3. One of the most commonly used technique in CPU scheduling as a core.
   4. It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
   5. The disadvantage of it is more overhead of context switching.
2. **What are the advantage of RR?**

A big advantage of round robin scheduling over non-preemptive schedulers is that it dramatically improves average response times. By limiting each task to a certain amount of time, the operating system can ensure that it can cycle through all ready tasks, giving each one a chance to run.

1. **What are disadvantages of RR ?**

Giving every process an equal share of the CPU is not always a good idea. For instance, highly interactive processes will get scheduled no more frequently than CPU-bound processes.

1. **What does *Round Robin Scheduling (RRS)* mean?**Round robin scheduling (RRS) is a job-scheduling algorithm that is considered to be very fair, as it uses time slices that are assigned to each process in the queue or line. Each process is then allowed to use the CPU for a given amount of time, and if it does not finish within the allotted time, it is preempted and then moved at the back of the line so that the next process in line is able to use the CPU for the same amount of time.

**Program 1.4 :** Write a C Program to simulate the Priority CPU scheduling algorithm.

**Aim: To implement Priority CPU scheduling algorithm in c**

**Description:** In *priority* scheduling, a priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.  SJF is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The large the CPU burst, the lower the priority. Priorities are generally some fixed range of numbers, such as 0 to 7. In this text, we assume that low numbers represent high priority.

**Algorithm:**

* 1. Start
  2. Declare Process structure with following members.
     1. PID : to Store Process ID
     2. Priority : to store priority of the process
     3. WT : to store waiting time of the process.
     4. BT : to store Burst time of the process.
     5. TT : to store turnaround time of the process.
  3. Read the number of processes.
  4. Read the Process Id, Burst time and Priority for all the processes.
  5. Sort the Array of process structure based on Priority.
  6. Calculate the waiting time of each process
  7. After sorting first process in the list will have waiting time 0
     1. Proc[0] .wt = 0
  8. Find Waiting time for all other process
     1. Proc[ i ] . wt = Proc[ i-1] .wt + Pro [ i-1] .bt

1. Calculate the Turnaround time of each process
   * 1. Proc[ i ] . tt = Proc[ i] .wt + Pro [ i] .bt
2. Calculate average waiting time
   1. Total\_waitingTime / Number of Process
3. Calculate average turnaround time.
   1. Total\_waitingTime / Number of Process
4. Display all the values
5. Stop.

#include <stdio.h>

#define MAX 15

struct process

{

int pid;

int priority;

int bt;

int wt;

int tt;

};

int main()

{

struct process proc[MAX], temp;

int n , i , j;

int total\_wt=0 , total\_tt=0;

float avg\_wt, avg\_tt;

printf("enter number of process (max of 15) \n");

scanf("%d",&n);

printf(" Enter Process Id and Brust time and priority of the process \n");

for( i=0; i< n; i++)

{

scanf("%d%d%d", &proc[i].pid , &proc[i].bt , &proc[i].priority);

}

//Bubble sort, to sort the process

for (i = 0; i < n-1; i++)

for (j = 0; j < n-i-1; j++)

if (proc[j].priority > proc[j+1].priority)

{

temp=proc[j];

proc[j]= proc[j+1];

proc[j+1]=temp;

}

for (i = 0; i < n; i++)

{

if(i==0)

{

proc[0].wt=0;

proc[0].tt = proc[0].bt;

}

else

{

proc[i].wt = proc[i-1].wt + proc[i-1].tt; // waiting time calculation

total\_wt = total\_wt + proc[i].wt; // total waiting time calculation

proc[i].tt = proc[i].wt + proc[i].bt ; // turn around time calculation

total\_tt = total\_tt + proc[i].tt; // total turn around time calculation

}

}

avg\_wt = (float) total\_wt/n;

avg\_tt = (float) total\_tt/n;

printf(" | Process ID | Priority | Burst time | Waiting Time | Turn Around Time | \n");

for (i = 0; i < n; i++)

printf(" | %d | %d | %d | %d | %d | \n", proc[i].pid , proc[i].priority, proc[i].bt, proc[i].wt, proc[i].tt);

return 0;

}

OUTPUT:

enter number of process (max of 15)

4

Enter Process Id and Brust time and priority of the process

3 55 6

4 5 9

3 10 2

7 20 1

| Process ID | Priority | Burst time | Waiting Time | Turn Around Time |

| 7 | 1 | 20 | 0 | 20 |

| 3 | 2 | 10 | 20 | 30 |

| 3 | 6 | 55 | 50 | 105 |

| 4 | 9 | 5 | 155 | 160 |

**VIVA Questions**

1. **What is Priority Scheduling?**

Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler chooses the tasks to work as per the priority. Priority scheduling involves priority assignment to every process, and processes with higher priorities are carried out first, whereas tasks with equal priorities are carried out on a first-come-first-served (FCFS).

1. **Is Priority Scheduling Preemptive or non-preemtive?**

Priority scheduling can be either of the following:

Preemptive: This type of scheduling may preempt the central processing unit (CPU) in the case the priority of the freshly arrived process being greater than those of the existing processes.

Non-preemptive: This type of scheduling algorithm simply places the new process at the top of the ready queue.

1. **Disadvantage of Priority Scheduling ?**

Indefinite blocking, otherwise called starvation, is one of the major issues concerning priority scheduling algorithms. It is a state where a process is ready to be executed, but faces a long wait in getting assigned to the CPU.It is often possible that a priority scheduling algorithm can make a low-priority process wait indefinitely. For example, in an intensely loaded system, if there are a number of higher priority processes, the low-priority processes may never get the CPU for execution.

1. **What is aging?**

A remedy to starvation is aging, which is a technique used to gradually increase the priority of those processes that wait for long periods in the system.

2. Programs to implement following system calls of UNIX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir ,closedir and lseek.

**Description:**

**System Calls:**

The way that programs talk to the operating system is via ``*system calls*.''

A system call looks like a procedure call , but it's different -- **it is a request to the operating system to perform some activity**.

* System calls are expensive. While a procedure call can usually be performed in a few machine instructions, a system call requires the computer to save its state, let the operating system take control of the CPU, have the operating system perform some function, have the operating system save its state, and then have the operating system give control of the CPU back to you.
* System calls are system dependent. Knowing this, it would not be a good idea to directly use system calls when portability cannot be neglected. System calls are also quite complex. Most often, it involves the duo of TRAP and RET (or some variations of those two). To implement system call, one would need specialized knowledge of I/O registers, the sequence of operations needed to use them; and most important of all, implement enough protection because I/O resources are generally shared among multiple users and/or processes.

## *System Calls for I/O*

There are 5 basic system calls that Unix provides for file I/O.

1. int open(char \*path, int flags [ , int mode ] );

2. int close(int fd);

3. int read(int fd, char \*buf, int size);

4. int write(int fd, char \*buf, int size);

5. off\_t lseek(int fd, off\_t offset, int whence);

**Program** 2.1 Write a C program to read from file and write to another file.

**AIM: To write a C program to copy content from one file to another file.**

**Program:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#define BUFF\_SIZE 1000

int main(void)

{

int n,fd1,fd2;

char buff[BUFF\_SIZE];

//open the file for reading

fd1 = open("testfile.txt",O\_RDWR,0644);

//read the data from file

n=read(fd1,buff,BUFF\_SIZE);

// creating a new file using open.

fd2=open("fileforcopy.txt", O\_CREAT | O\_RDWR, 0777);

//writting data to file (fd)

if( write(fd2, buff, n) == n)

printf("file copying is successful. and the data is:\n");

//Write to display (1 is standard fd for output device)

write(1, buff, n);

//closing the files

int close(int fd1);

int close(int fd2);

return 0;

}

**INPUT &OUTPUT:**

testfile.txt file copied to fileforcopy.txt

file copying is successful. and the data is: hello mgit

**Program 2.2.** Program using lseek() system call that reads 10 characters from file “seeking” and print on screen. Skip next 5 characters and again read 10 characters and write on screen.

**AIM: To write a C program to demonstrate the usage of lseek() system call.**

**Description:**

**lseek (C System Call)**: lseek is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.

off\_t lseek(int *fildes*, off\_t *offset*, int *whence*);

The *lseek*() function shall set the file offset for the open file description associated with the file descriptor *fildes,* as follows:

* If *whence* is SEEK\_SET, the file offset shall be set to *offset* bytes.
* If *whence* is SEEK\_CUR, the file offset shall be set to its current location plus *offset*.
* If *whence* is SEEK\_END, the file offset shall be set to the size of the file plus *offset*.

The symbolic constants SEEK\_SET, SEEK\_CUR, and SEEK\_END are defined in [*<unistd.h>*](https://pubs.opengroup.org/onlinepubs/009696799/basedefs/unistd.h.html).

Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned. Otherwise, (**off\_t**)-1 shall be returned, *errno* shall be set to indicate the error, and the file offset shall remain unchanged.

**Program:**

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h

#include<sys/stat.h>

#int main()

{

int n,f;

char buff[10];

f=open("seeking",O\_RDWR);

read(f,buff,10);

write(1,buff,10);

read(f,buff,10);

write(1,buff,10);

}

***Pre-requisite: Create a file “seeking” and write “1234567890abcdefghijxxxxxxxxxx” into it.***

*Output:*

The output will be the first 10 characters “1234567890” followed by “fghijxxxxx”. The inbetween 5 characters are skipped because we used lseek to reposition the pointer 5 characters ahead from the current (SEEK\_CUR) position.

**Program** 2.3: Write a C program to print file status information using stat function.

**AIM: To write a C program to demonstrate the usage of stat() system call.**

**Description:**

Stat system call is a system call in Linux to check the status of a file such as to check when the file was accessed. The stat() system call actually returns file attributes. The file attributes of an inode are basically returned by Stat() function. An inode contains the metadata of the file. An inode contains: the type of the file, the size of the file, when the file was accessed (modified, deleted) that is time stamps, and the path of the file, the user ID and the group ID, links of the file, and physical address of file content.

To use the stat system call in C programming language we should include the following header file:

#include <sys/stat.h>

int stat(const char \*path, struct stat \*buf)

* The return type of the function in int, if the function is executed successfully, 0 is returned if there are any errors, -1 will be returned.
* **const char \*path** specifies the name of the file.
* stat structure The data or information about the file is stored which uses a pointer named **buf,** which is passed in as a paramteter and filled in during the execution of the call and readable by the user after the call.

**Program :**

#include<stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

#include<unistd.h>

struct stat statbuf;

char dirpath[256];

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

{

// getcwd is to get the name of the current working directory if found

getcwd(dirpath,256);

DIR \*dir = opendir(dirpath);

struct dirent \*dp;

for (dp=readdir(dir); dp != NULL ; dp=readdir(dir))

{

stat(dp->d\_name, &statbuf);

printf("the file name is %s \n", dp->d\_name);

printf("dir = %d\n", S\_ISDIR(statbuf.st\_mode));

printf("file size is %ld in bytes \n", statbuf.st\_size);

printf("last modified time is %ld in seconds \n", statbuf.st\_mtime);

printf("last access time is %ld in seconds \n", statbuf.st\_atime);

printf("The device containing the file is %ld\n", statbuf.st\_dev);

printf("File serial number is %ld\n\n", statbuf.st\_ino);

}

}

**OUTPUT:**

the file name is fileone.txt

dir = 0

file size is 31 in bytes

last modified time is 1581656490 in seconds

last access time is 1581656969 in seconds

The device containing the file is 2053

File serial number is 2099241

the file name is filestatus.c

dir = 0

file size is 772 in bytes

last modified time is 1581146159 in seconds

last access time is 1581920990 in seconds

The device containing the file is 2053

File serial number is 2099181

**Program** 2.4 : C Program for opendir(), readdir() and closedir system calls

**AIM: To write a C program to demonstrate the usage of opendir(), readdir() and closedir() system call.**

**Description:**

The *opendir*() function shall open a directory stream corresponding to the directory named by the *dirname* argument. Directory entries represent files or subdirectory.

#include <[dirent.h](https://pubs.opengroup.org/onlinepubs/009604599/basedefs/dirent.h.html)>  
DIR \*opendir(const char \**dirname*);

* Upon successful completion, *opendir*() shall return a pointer to an object of type **DIR**. Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

The *readdir*() function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*, and position the directory stream at the next entry. It shall return a null pointer upon reaching the end of the directory stream.

\* readdir(DIR \**dirp*);

\* Upon successful completion, *readdir*() shall return a pointer to an object of type **struct dirent**. When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate the error.

The *closedir*() function shall close the directory stream referred to by the argument *dirp*. Upon successful completion, *closedir*() shall return 0; otherwise, -1 shall be returned and *errno* set to indicate the error.

**PROGRAM:**

#include<stdio.h>

#include<dirent.h>

struct dirent \*dptr;

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

{

char buff[100];

DIR \*dirp;

printf("\n\n ENTER DIRECTORY NAME");

scanf("%s", buff);

if( ( dirp=opendir( buff ) ) == NULL)

{

printf("The given directory does not exist");

exit(1);

}

while(dptr = readdir ( dirp ) )

{

printf("%s\n",dptr->d\_name);

} closedir(dirp); }

Program 2.5: C Program for fork() and getpid() system call

**AIM: To write a C program to demonstrate the usage of fork() and getpid() system call.**

**Description: fork**() creates a new process by duplicating the calling process.The new process is referred to as the *child* process. The calling process is referred to as the *parent* process. The child has its own unique process ID.The child's parent process ID is the same as the parent's process ID.

**#include <unistd.h>**

**pid\_t fork(void);**

* On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and [*errno*](https://man7.org/linux/man-pages/man3/errno.3.html) is set to indicate the error.

Getpid() is the function used to get the process ID of the process that calls that function.

**ALGORITHM:**

STEP 1: Start the program.

STEP 2: Declare the variables pid,pid1,pid2.

STEP 3: Call fork() system call to create process.

STEP 4: If pid==-1, exit.

STEP 5: Ifpid!=-1 , get the process id using getpid().

STEP 6: Print the process id.

STEP 7:Stop the program

**PROGRAM:**

#include<stdio.h>

#include<unistd.h>

main()

{

int pid,pid1,pid2;

pid=fork();

if(pid== -1)

{

printf("ERROR IN PROCESS CREATION \n");

exit(1);

}

if(pid!=0)

{

pid1=getpid();

printf("\n the parent process ID is %d\n", pid1);

}

else

{

pid2=getpid();

printf("\n the child process ID is %d\n", pid2);

} }

2.6 C program to execute another C program using exec system call.

**AIM: To write a C program to execute another C program using exec system call**

**Description:**

The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program. It comes under the header file **unistd.h.**There are many members in the exec family which are shown below with examples.

* **execvp** : Using this command, the created child process does not have to run the same program as the parent process does. The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script . **Syntax:**

int execvp (const char \*file, char \*const argv[]);

**file:** points to the file name associated with the file being executed.  
**argv:**  is a null terminated array of character pointers.

We will have two .C files , **EXEC.c** and **execDemo.c** and we will replace the execDemo.c with EXEC.c by calling execvp() function in execDemo.c .

| //EXEC.c    #include<stdio.h>  #include<unistd.h>    int main()  {      int i;        printf("I am EXEC.c called by execvp() ");      printf("\n");        return 0;  } |
| --- |

Now,create an executable file of EXEC.c using command

gcc EXEC.c -o EXEC

| //execDemo.c    #include<stdio.h>  #include<stdlib.h>  #include<unistd.h>  int main()  {          //A null terminated array of character          //pointers          char \*args[]={"./EXEC",NULL};          execvp(args[0],args);            /\*All statements are ignored after execvp() call as this whole          process(execDemo.c) is replaced by another process (EXEC.c)          \*/          printf("Ending-----");        return 0;  } |
| --- |

create an executable file of execDemo.c using command

gcc execDemo.c -o execDemo

After running the executable file of execDemo.cby using command ./excDemo, we get the following output:

I AM EXEC.c called by execvp()

When the file execDemo.c is compiled, as soon as the statement execvp(args[0],args) is executed, this very program is replaced by the program EXEC.c. “Ending—–” is not printed because because as soon as the execvp() function is called, this program is replaced by the program EXEC.c.

**Program** 3. Write a C Program to implement Banker’s Algorithm for Deadlock Avoidance and Detection

**AIM: To Simulate bankers algorithm for Dead Lock Avoidance (Banker‘s Algorithm)**

**Description:**

The **Banker algorithm**, sometimes referred to as the detection **algorithm**, is a resource allocation and deadlock avoidance **algorithm** developed by Edsger Dijkstra.

The algorithm tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue .

**Algorithm:**

The banker's algorithm relies on several key data structures:

( where n is the number of processes and m is the number of resource categories. )

1. Available[ m ] indicates how many resources are currently available of each type.
2. Max[ n ][ m ] indicates the maximum demand of each process of each resource.
3. Allocation[ n ][ m ] indicates the number of each resource category allocated to each process.
4. Need[ n ][ m ] indicates the remaining resources needed of each type for each process. ( Note that Need[ i ][ j ] = Max[ i ][ j ] - Allocation[ i ][ j ] for all i, j. )

In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.This algorithm determines if the current state of a system is safe, according to the following steps:

1. Let Work and Finish be vectors of length m and n respectively.
   * Work is a working copy of the available resources, which will be modified during the analysis.
   * Finish is a vector of booleans indicating whether a particular process can finish. ( or has finished so far in the analysis. )
   * Initialize Work to Available, and Finish to false for all elements.
2. Find an i such that both
   * Finish[ i ] == false, and
   * Need[ i ] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
3. Set Work = Work + Allocation[ i ], and
   * set Finish[ i ] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
4. If finish[ i ] == true for all i, then the state is a safe state, because a safe sequence has been found.

#### Resource-Request Algorithm ( The Bankers Algorithm )

Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself. This algorithm determines if a new request is safe, and grants it only if it is safe to do so.

* When a request is made ( that does not exceed currently available resources ), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
  1. Let Request[ n ][ m ] indicate the number of resources of each type currently requested by processes.
     + If Request[ i ] > Need[ i ] for any process i, raise an error condition.
  2. If Request[ i ] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
  3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely. The procedure for granting a request ( or pretending to for testing purposes ) is:
     + Available = Available – Request ,
     + Allocation = Allocation + Request
     + Need = Need - Request

**Program:**

#include <stdio.h>

int main()

{

// P0, P1, P2, P3, P4 are the Process names here

int n, m, i, j, k;

n = 5; // Number of processes

m = 3; // Number of resources

int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

{ 2, 0, 0 }, // P1

{ 3, 0, 2 }, // P2

{ 2, 1, 1 }, // P3

{ 0, 0, 2 } }; // P4

int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

{ 3, 2, 2 }, // P1

{ 9, 0, 2 }, // P2

{ 2, 2, 2 }, // P3

{ 4, 3, 3 } }; // P4

int avail[3] = { 3, 3, 2 }; // Available Resources

int newRequest[5][3]={0} , ProNo;

//banker algorithm : check for new request

/\*1. What will happen if process P1 requests one additional

instance of resource type A and two instances of resource type C?

P1 ( 1 0 2)

\*/

/\*2.Can request for (3,3,0) by P4 be granted?

3. Can request for (0,2,0) by P0 be granted?

\*/

printf("Banker's Algorithm \n");

printf("For the new request: enter process no \n");

scanf("%d",&ProNo);

printf("Enter the new request of the process%d in the order resources[A, B,C]\n", ProNo);

scanf("%d",&newRequest[ProNo][0]);

scanf("%d",&newRequest[ProNo][1]);

scanf("%d",&newRequest[ProNo][2]);

//pretend it has been granted thus :Available = Available - NewRequest, Allocation = Allocation + Request

avail[0]= avail[0]-newRequest[ProNo][0];

avail[1]= avail[1]-newRequest[ProNo][1];

avail[2]= avail[2]-newRequest[ProNo][2];

alloc[ProNo][0]= alloc[ProNo][0] + newRequest[ProNo][0] ;

alloc[ProNo][1]= alloc[ProNo][1] + newRequest[ProNo][1] ;

alloc[ProNo][2]= alloc[ProNo][2] + newRequest[ProNo][2] ;

int f[n], ans[n], ind = 0;

for (k = 0; k < n; k++)

{

f[k] = 0;

}

int need[n][m];

for (i = 0; i < n; i++) //Process

{

for (j = 0; j < m; j++) //resource

need[i][j] = max[i][j] - alloc[i][j];

}

int flag = 0;

int y = 0;

for (k = 0; k < 5; k++)//numer of cycles to complete exceution

{

for (i = 0; i < n; i++)

{

if (f[i] == 0)

{

flag = 0;

for (j = 0; j < m; j++)

{

if (need[i][j] > avail[j])

{

flag = 1;

break;

}

}

if (flag == 0)

{

ans[ind++] = i; //creating the Safe State

for (y = 0; y < m; y++)

avail[y] += alloc[i][y]; //update the Av

f[i] = 1;

}

}

}

}

if(flag == 1)

{

printf("Request cannot be granted now , as it may lead to Deadlock\n as their is no safe sequence\n");

printf("your request is not granted to Avoid DeadLock");

printf(" Process has to wait\n");

return 0;

}

printf("Request can be granted, as a safe sequence present\n");

printf("Their will be no deadlock\n");

printf("Following is the SAFE Sequence\n");

for (i = 0; i < n - 1; i++)

printf(" P%d ->", ans[i]);

printf(" P%d", ans[n - 1]);

return (0);

}OUTPUT:

**RUN 1:**

Banker's Algorithm

For the new request:  enter process no

1

Enter the new  request of the process1 in the order resources[A, B,C]                                                                  1   0 2

Request can be granted, as a  safe sequence present

Their will be no deadlock

Following is the SAFE Sequence

 P1 -> P3 -> P4 -> P0 -> P2

**RUN 2:**

Banker's Algorithm

For the new request:  enter process no

0

Enter the new  request of the process0 in the order resources[A, B,C]

0 2 0

Request can be granted, as a  safe sequence present

Their will be no deadlock

Following is the SAFE Sequence

 P3 -> P1 -> P2 -> P0 -> P4

**RUN 3:**

Banker's Algorithm

For the new request:  enter process no

4

Enter the new  request of the process4 in the order resources[A, B,C]

 3 3 0

Request cannot be granted now , as it may lead to Deadlock  as their is no safe sequence

 Process has to wait

**Viva Questions**

## Who is the banker in banker's algorithm?

Operating System's resource allocator acts as the **banker**. **Bankers algorithm** or detection **algorithm** is based on the concept that banks lend to the customer only if it's resources permits.

## How can deadlock be prevented?

**Deadlocks can be prevented** by avoiding at least one of the four conditions, because all this four conditions are required simultaneously to cause **deadlock**. Resources shared such as read-only files do not lead to **deadlocks** but resources, such as printers and tape drives, requires exclusive access by a single process.

### What are the different types of Deadlock Types and Handling Strategies?

There are multiple deadlocks handling strategies to avoid deadlock.

We can prevent the deadlock not to occur and it is the optimal solution. But preventing the system not to occur deadlock is not easy as we require prior information about processes as well as resources.

On other hands, if it occurs we have a recovery mechanism.

There are four strategies to handle deadlock.

* Prevention of Deadlock
* Avoidance of Deadlock
* Deadlock Detection and Recover
* Deadlock Ignorance

#### What is Deadlock in the Operating System?

Deadlock is the condition that occurs when two processes wait for each other to complete and halt without proceeding further. Here two processes wait for the resources acquired by each other.

**What is resource allocation graph?**

This is the graphical description of deadlocks. This graph consists of a set of edges E and a set of vertices V. The set of vertices V is partitioned into two different types of nodes P={p1,p2,…,pn},the set consisting of all the resources in the system,R={r1,r2,…rn}.A directed edge Pi?Rj is called a request edge; a directed edge Rj? Pi is called an assignment edge. Pictorially we represent a process Pi as a circle, and each resource type Rj as square.

**What are deadlock prevention techniques?**

1. Mutual exclusion : Some resources such as read only files shouldn’t be mutually exclusive. They should be sharable. But some resources such as printers must be mutually exclusive.

2. Hold and wait : To avoid this condition we have to ensure that if a process is requesting for a resource it should not hold any resources.

3. No preemption : If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is the process must wait),then all the resources currently being held are

preempted(released autonomously).

4. Circular wait : the way to ensure that this condition never holds is to impose a total ordering of all the resource types, and to require that each process requests resources in an increasing order of enumeration.

Program 4: Producer Consumer Problem using Semaphore – using PThread

**AIM: To Write a C program to simulate producer - consumer problem using semaphores.**

**Description:**

**About Producer-Consumer problem**

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes.

In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.

The following are the problems that might occur in the Producer-Consumer:

* The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
* The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
* The producer and consumer should not access the buffer at the same time.

**The above three problems can be solved with the help of semaphores. In the producer-consumer problem, we use three semaphore variables:**

1. Semaphore S: This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.
2. Semaphore E: This semaphore variable is used to define the empty space in the buffer. Initially, it is set to the whole space of the buffer i.e. "n" because the buffer is initially empty.
3. Semaphore F: This semaphore variable is used to define the space that is filled by the producer. Initially, it is set to "0" because there is no space filled by the producer initially.

By using the above three semaphore variables and by using the *wait()* and *signal()* function, we can solve our problem(the *wait()* function decreases the semaphore variable by 1 and the *signal()* function increases the semaphore variable by 1).

**Semaphore related System Calls used in the program:**

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to :

1. Include semaphore.h
2. Compile the code by linking with -lpthread –lrt

To lock a semaphore or wait we can use the sem\_wait function:

int sem\_wait(sem\_t \*sem);

To release or signal a semaphore, we use the sem\_post function:

int sem\_post(sem\_t \*sem);

A semaphore is initialised by using sem\_init(for processes or threads) or sem\_open (for IPC).

sem\_init(sem\_t \*sem, int pshared, unsigned int value);

Where,

* + sem : Specifies the semaphore to be initialized.
  + pshared : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
  + value : Specifies the value to assign to the newly initialized semaphore.

**Thread System Calls used in the program:**

*1. pthread\_create:* used to create a new thread

Syntax:

int pthread\_create(pthread\_t \* thread,

const pthread\_attr\_t \* attr,

void \* (\*start\_routine)(void \*),

void \*arg);

Parameters:

* thread: pointer to an unsigned integer value that returns the thread id of the thread created.
* attr: pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
* start\_routine: pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void \*. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
* arg: pointer to void that contains the arguments to the function defined in the earlier argument

*2. pthread\_join:* used to wait for the termination of a thread.

int pthread\_join(pthread\_t th, void \*\*thread\_return);

Parameter: This method accepts following parameters:

* th: thread id of the thread for which the current thread waits.
* thread\_return: pointer to the location where the exit status of the thread mentioned in th is stored.

**Algorithm:**

Algorithm The following is the pseudo-code for the producer:

void producer()

{

for(n<=10)

{

wait(Empty)

wait(Mutex)

produce()

*append()*

signal(Mutex)

signal(Full)

}

}

void consumer()

{

for(n<=10)

{

wait(Full)

wait(Mutex)

take()

signal(Mutex)

signal(Empty)

use()

}

}

**Program :**

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>

#include<semaphore.h>

int buf[5], f, r; //Circular queue

sem\_t mutex, full, empty;

void \*produce(void \*arg)

{

int i;

for(i=0;i<10;i++)

{

sem\_wait(&empty);//Wait of empty slot and decrement the empty

sem\_wait(&mutex); //Wait if consumer is comsuming a item

printf("produced item is %d\n",i);

buf[(++r)%5]=i;

sleep(1);

sem\_post(&mutex); // signal the consumer to consume the item mutex=0

sem\_post(&full); // full will be increamented

}

}

void \*consume(void \*arg)

{

int item,i;

for(i=0;i<10;i++)

{

sem\_wait(&full); //Wait for item and decrement the full

sem\_wait(&mutex); // Wait if producer s producing a item

item=buf[(++f)%5];

printf("consumed item is %d\n",item);

sleep(1);

sem\_post(&mutex); // signal the producer to produce the item mutex=0

sem\_post(&empty); // incrementing the empty variable

}

}

int main()

{

pthread\_t tid1,tid2;

sem\_init(&mutex, 0, 1);

sem\_init(&full, 0, 0);

sem\_init(&empty, 0, 5);

pthread\_create(&tid1, NULL, produce, NULL);

pthread\_create(&tid2,NULL, consume, NULL);

pthread\_join(tid1,NULL);

pthread\_join(tid2,NULL);

return 0;

}

VIVA Questions

**What is a Thread?**  
A thread is a path of execution within a process. A process can contain multiple threads.

**Why Multithreading?**  
A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

**Process vs Thread?**  
The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.  
Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

**What is a Monitor?**

A Monitor is characterized by a set of programmer-defined operators. The representation of a Monitor

type consists of declaration of variables whose value define the state of an instance of the type, as well as

the bodies of procedures or functions that implement operations on the type.

**What is meant by Counting Semaphore?**

A Counting Semaphore is a semaphore whose integer value that can range between 0 & 1.

5.1 C program to implement interprocess communication using Pipes

**Description:**

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).

1. **int pipe(int pipedes[2]);**

This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor pipedes[0] is for reading and pipedes[1] is for writing. Whatever is written into pipedes[1] can be read from pipedes[0].

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

1. int open(const char \*pathname, int flags);

int open(const char \*pathname, int flags, mode\_t mode);

The arguments passed to open system call are pathname (relative or absolute path), flags mentioning the purpose of opening file (say, opening for read, O\_RDONLY, to write, O\_WRONLY, to read and write, O\_RDWR, to append to the existing file O\_APPEND, to create file, if not exists with O\_CREAT and so on) and the required mode providing permissions of read/write/execute for user or owner/group/others. Mode can be mentioned with symbols.

Read – 4, Write – 2 and Execute – 1.

For example: Octal value (starts with 0), 0764 implies owner has read, write and execute permissions, group has read and write permissions, other has read permissions. This can also be represented as S\_IRWXU | S\_IRGRP | S\_IWGRP | S\_IROTH, which implies or operation of 0700|0040|0020|0004 → 0764.

1. ssize\_t read(int fd, void \*buf, size\_t count)

* The above system call is to read from the specified file with arguments of file descriptor fd, proper buffer with allocated memory (either static or dynamic) and the size of buffer.
* The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call. The file needs to be opened before reading from the file. It automatically opens in case of calling pipe() system call.
* This call would return the number of bytes read (or zero in case of encountering the end of the file) on success and -1 in case of failure. The return bytes can be smaller than the number of bytes requested, just in case no data is available or file is closed.

1. ssize\_t write(int fd, void \*buf, size\_t count)

* The above system call is to write to the specified file with arguments of the file descriptor fd, a proper buffer with allocated memory (either static or dynamic) and the size of buffer.
* The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call.
* The file needs to be opened before writing to the file. It automatically opens in case of calling pipe() system call.
* Program to write and read two messages through the pipe using the parent and the child processes.

**Algorithm**

**Step 1** − Create a pipe.

**Step 2** − Create a child process.

**Step 3** − Parent process writes to the pipe.

**Step 4** − Child process retrieves the message from the pipe and writes it to the standard output.

**Step 5** − Repeat step 3 and step 4 once again.

**Source Code: pipewithprocesses.c**

#include<stdio.h>

#include<unistd.h>

int main()

{

int pipefds[2];

int returnstatus;

int pid;

char writemessages[2][20]={"Hi", "Hello"};

char readmessage[20];

returnstatus = pipe(pipefds);

if (returnstatus == -1)

{

printf("Unable to create pipe\n");

return 1;

}

pid = fork();

// Child process

if (pid == 0)

{

read(pipefds[0], readmessage, sizeof(readmessage));

printf("Child Process - Reading from pipe – Message 1 is %s\n", readmessage);

read(pipefds[0], readmessage, sizeof(readmessage));

printf("Child Process - Reading from pipe – Message 2 is %s\n", readmessage);

} else

{ //Parent process

printf("Parent Process - Writing to pipe - Message 1 is %s\n", writemessages[0]);

write(pipefds[1], writemessages[0], sizeof(writemessages[0]));

printf("Parent Process - Writing to pipe - Message 2 is %s\n", writemessages[1]);

write(pipefds[1], writemessages[1], sizeof(writemessages[1]));

}

return 0;

}

## Two-way Communication Using Pipes

Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.

Following are the steps to achieve two-way communication −

**Step 1** − Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

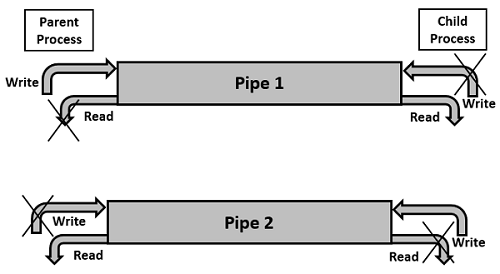
**Step 2** − Create a child process.

**Step 3** − Close unwanted ends as only one end is needed for each communication.

**Step 4** − Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

**Step 5** − Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

**Step 6** − Perform the communication as required.



**Sample program 1** − Achieving two-way communication using pipes.

### Algorithm

**Step 1** − Create pipe1 for the parent process to write and the child process to read.

**Step 2** − Create pipe2 for the child process to write and the parent process to read.

**Step 3** − Close the unwanted ends of the pipe from the parent and child side.

**Step 4** − Parent process to write a message and child process to read and display on the screen.

**Step 5** − Child process to write a message and parent process to read and display on the screen.

**Source Code: twowayspipe.c**

#include<stdio.h>

#include<unistd.h>

int main()

{

int pipefds1[2], pipefds2[2];

int returnstatus1, returnstatus2;

int pid;

char pipe1writemessage[20] = "Hi";

char pipe2writemessage[20] = "Hello";

char readmessage[20];

returnstatus1 = pipe(pipefds1);

if (returnstatus1 == -1)

{

printf("Unable to create pipe 1 \n");

return 1;

}

returnstatus2 = pipe(pipefds2);

if (returnstatus2 == -1)

{

printf("Unable to create pipe 2 \n");

return 1;

}

pid = fork();

if (pid != 0) // Parent process

{

close(pipefds1[0]); // Close the unwanted pipe1 read side

close(pipefds2[1]); // Close the unwanted pipe2 write side

printf("In Parent: Writing to pipe 1 – Message is %s\n", pipe1writemessage);

write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));

read(pipefds2[0], readmessage, sizeof(readmessage));

printf("In Parent: Reading from pipe 2 – Message is %s\n", readmessage);

}

else

{ //child process

close(pipefds1[1]); // Close the unwanted pipe1 write side

close(pipefds2[0]); // Close the unwanted pipe2 read side

read(pipefds1[0], readmessage, sizeof(readmessage));

printf("In Child: Reading from pipe 1 – Message is %s\n", readmessage);

printf("In Child: Writing to pipe 2 – Message is %s\n", pipe2writemessage);

write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));

}

return 0;

}

**1. What Is Ipc? What Are The Various Schemes Available?**

Inter Process Communication. IPC is used to pass information between two or more processes.Schemes are pipes, shared memory & semaphore.**Below are the different IPC methods:**

1. Semaphores
2. FIFO's (Also called Named Pipes)
3. Message Queues
4. Shared Memory

**2. What are the models of IPC?**  
Two basic models are used in IPC:  
**Shared memory:** “shared data” are directly available to each process in their address space.  
**Message passing:** “shared data” are explicitly exchanged.

**3. Differentiate between synchronous and asynchronous communication?:**The IPC operations may provide the synchronization necessary using a blocking operation issued by a process will block further processing of the process until the operation is fulfilled.

Alternatively, IPC operations may be asynchronous or nonblocking.An asynchronous operation issued by a process will not block further processing of the process.  Instead, the process is free to proceed with its processing, and may optionally be notified by the system when the operation is fulfilled.

Program 5.2 : C program to implement interprocess communication using FIFO(Named Pipes)

FIFOs are pipes with a name and are also commonly referred to as named pipes. A FIFO is different because it has a name like a file.

We can create a FIFO from the shell using the mknod command. We can remove a FIFO just like a file using the rm command.

### mkfifo system call

We can create a FIFO from a program using the mkfifo system call.

#include <sys/stat.h>

int mkfifo (const char \*path, mode\_t perms);// Returns -1 on error.

1. A FIFO can be opened using the open system call. After open, we can use the read and write system calls for reading from and writing to the FIFO, using the file descriptor returned by open.

**Step 1** − Create two processes, one is fifoserver and another one is fifoclient.

**Step 2** − Server process performs the following −

* Creates a named pipe (using system call mknod()) with name “MYFIFO”, if not created.
* Opens the named pipe for read only purposes.
* Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.
* Waits infinitely for message from the Client.
* If the message received from the client is not “end”, prints the message. If the message is “end”, closes the fifo and ends the process.

**Step 3** − Client process performs the following −

* Opens the named pipe for write only purposes.
* Accepts the string from the user.
* Checks, if the user enters “end” or other than “end”. Either way, it sends a message to the server. However, if the string is “end”, this closes the FIFO and also ends the process.
* Repeats infinitely until the user enters string “end”.

**Process 1: (Writes First):**

// C program to implement one side of FIFO

// This side writes first, then reads

#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

    int fd;

    // FIFO file path

    char \* myfifo = "/tmp/myfifo";

    // Creating the named file(FIFO)

    // mkfifo(<pathname>, <permission>)

    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];

    while (1)

    {

        // Open FIFO for write only

        fd = open(myfifo, O\_WRONLY);

        // Take an input arr2ing from user.

        // 80 is maximum length

        fgets(arr2, 80, stdin);

        // Write the input arr2ing on FIFO

        // and close it

        write(fd, arr2, strlen(arr2)+1);

        close(fd);

        // Open FIFO for Read only

        fd = open(myfifo, O\_RDONLY);

        // Read from FIFO

        read(fd, arr1, sizeof(arr1));

        // Print the read message

        printf("User2: %s\n", arr1);

        close(fd);

    }

    return 0;

}

**Process 2: (Reads First):**

#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

    int fd;

    // FIFO file path

    char \* myfifo = "/tmp/myfifo";

    // Creating the named file(FIFO)

    // mkfifo(<pathname>, <permission>)

    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];

    while (1)

    {

        // Open FIFO for write only

        fd = open(myfifo, O\_WRONLY);

        // Take an input arr2ing from user.

        // 80 is maximum length

        fgets(arr2, 80, stdin);

        // Write the input arr2ing on FIFO

        // and close it

        write(fd, arr2, strlen(arr2)+1);

        close(fd);

        // Open FIFO for Read only

        fd = open(myfifo, O\_RDONLY);

        // Read from FIFO

        read(fd, arr1, sizeof(arr1));

        // Print the read message

        printf("User2: %s\n", arr1);

        close(fd);

    }

    return 0;

}

OUTPUT

Hello

#### VIVA Question:

#### What is Safe State?

* A state is ***safe*** if the system can allocate all resources requested by all processes ( up to their stated maximums ) without entering a deadlock state.
* More formally, a state is safe if there exists a ***safe sequence*** of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. ( I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up. )
* If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock.

# What is Deadlock avoidance?

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states. In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource

**What is a device queue?**

A list of processes waiting for a particular I/O device is called device queue.

**What is a long term scheduler?**

Long term schedulers are the job schedulers that select processes from the job queue and load them into memory for execution.

**What are short term scheduler?**

The short term schedulers are the CPU schedulers that select a process form the ready queue and allocate the CPU to one of them.

**What is context switching?**

Transferring the control from one process to other process requires saving the state of the old process and loading the saved state for new process.

Program 5.3: To implement IPC using Shared Memory

**AIM : To write a C Program to implement IPC using shared memory technique.**

**Description:**

SYSTEM CALLS USED ARE:

**ftok()**: is use to generate a unique key.

**shmget()**: int shmget(key\_t,size\_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.

**shmat()**: Before you can use a shared memory segment, you have to attach process to it using shmat().

void \*shmat(int shmid ,void \*shmaddr ,int shmflg);  
shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.

**shmdt()**: When you’re done with the shared memory segment, your program should detach itself from it using shmdt(). int shmdt(void \*shmaddr);

**shmctl()**: when you detach from shared memory,it is not destroyed. So, to destroy shmctl() is used. shmctl(int shmid,IPC\_RMID,NULL);

Writer Processes Program

| #include <iostream>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <stdio.h>  using namespace std;    int main()  {      // ftok to generate unique key      key\_t key = ftok("shmfile",65);        // shmget returns an identifier in shmid      int shmid = **shmget**(key, 1024, 0666|IPC\_CREAT);        // shmat to attach to shared memory      char \*str = (char\*) **shmat**(shmid,(void\*)0,0);        cout<<"Write Data : ";      gets(str);        printf("Data written in memory: %s\n",str);        //detach from shared memory  **shmdt**(str);        return 0;  } |
| --- |

Reader Process Program

| #include <iostream>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <stdio.h>  using namespace std;    int main()  {      // ftok to generate unique key      key\_t key = ftok("shmfile",65);        // shmget returns an identifier in shmid      int shmid = shmget(key,1024,0666|IPC\_CREAT);        // shmat to attach to shared memory      char \*str = (char\*) shmat(shmid,(void\*)0,0);        printf("Data read from memory: %s\n",str);        //detach from shared memory      shmdt(str);        // destroy the shared memory      shmctl(shmid,IPC\_RMID,NULL);        return 0;  } |
| --- |

**Output:**

Run the two programs simultaneously on two terminals

Writer Terminal 1: Hello

Reader Terminal 2: Hello

VIVA Question:

**1. What is shared memory?**

Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.

**2. What are two types of process?**

A process can be of two types: Independent process and Co-operating process.

**3. What is Independent process?**

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes

Program 5.4: To implement IPC Using Message Queue

**AIM: To write a C Program to implement IPC using message queue technique.**

**Description:**

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**. New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don’t have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

**System Calls used**

* A new queue is created or an existing queue opened by **msgget()**.
* New messages are added to the end of a queue by **msgsnd()**.
* Messages are fetched from a queue by **msgrcv()**.
* **ftok()**: is use to generate a unique key.
* **msgctl()**:it is use to destroy message queue.

| // C Program for Message Queue (Writer Process)  #include <stdio.h>  #include <sys/ipc.h>  #include <sys/msg.h>    // structure for message queue  struct mesg\_buffer  {      long mesg\_type;      char mesg\_text[100];  } message;    int main()  {      key\_t key;      int msgid;        // ftok to generate unique key  **key = ftok("progfile", 65);**        // msgget creates a message queue      // and returns identifier  **msgid** = **msgget**(key, 0666 | IPC\_CREAT);      message.**mesg\_type** = **1**;        printf("Write Data : ");      gets(message.**mesg\_text**);        // msgsnd to send message  **msgsnd**(msgid, &message, sizeof(message), 0);        // display the message      //printf("Data send is : %s \n", message.mesg\_text);        return 0;  } |
| --- |

MESSAGE QUEUE FOR READER PROCESS

| // C Program for Message Queue (Reader Process)  #include <stdio.h>  #include <sys/ipc.h>  #include <sys/msg.h>    // structure for message queue  struct mesg\_buffer  {      long mesg\_type;      char mesg\_text[100];  } message;    int main()  {      key\_t key;      int msgid;        // ftok to generate unique key      key = ftok("progfile", 65);        // msgget creates a message queue      // and returns identifier      msgid = **msgget**(key, 0666 | IPC\_CREAT);        // msgrcv to receive message  **msgrcv**(msgid, &message, sizeof(message), **1**, 0);        // display the message      printf("Data Received is : %s \n",                      message.mesg\_text);        // to destroy the message queue      msgctl(msgid, IPC\_RMID, NULL);        return 0;  } |
| --- |

**OUTPUT:** Run the two programs simultaneously on two terminals.

**Terminal 1: Terminal 2:**

$cc writer.c -o writer

$./writer

Hello

$cc reader.c -o reader

$./reader

Received message is : Hello

Program 6.1: To simulate paging technique of memory management.

**AIM: To write a c program to simulate paging technique of memory management.**

**Description:**

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

* The main idea behind the paging is to divide each process in the form of pages. The main memory will also be divided in the form of frames.
* One page of the process is to be stored in one of the frames of the memory. The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes.
* Pages of the process are brought into the main memory only when they are required otherwise they reside in the secondary storage.
* Different operating system defines different frame sizes. The sizes of each frame must be equal. Considering the fact that the pages are mapped to the frames in Paging, page size needs to be as same as frame size

**Algorithm:**

Step 1: Read Memory size and Frame size

Step 2: Find no of frames

Step 3: Read new process page requriment

Step 4: Read Page Table

Step 5: Read Logical address( Page no and offset value)

Step 6: With page no as index into page table find frame no

Step 6: Calculate and print physical address( Physical address = Frame no\* Frame size + Offset value )

**Program:**

#include<stdio.h>

int main()

{

int ms, Fsize, NoFrames, NoProcess, RemFrames, i, j, x, y, pa, offset;

int NoPages, PageTable[10];

printf("\nEnter the memory size -- ");

scanf("%d", &ms);

printf("\nEnter the Frame size -- ");

scanf("%d", &Fsize);

NoFrames = ms/Fsize;

printf("\n The no. of Frames available in memory are -- %d ", NoFrames);

RemFrames = NoFrames;

printf("\n Enter no. of pages required : ");

scanf("%d", &NoPages);

if(NoPages >RemFrames)

{

printf("\n Memory is Full");

return (0);

}

RemFrames = RemFrames - NoPages;

printf("\n ---Enter page table --- ");

for(j=0;j<NoPages;j++)

scanf("%d", &PageTable[j]);

printf("\n ---page table --- ");

printf(" \n| PNo || FNo |");

for(j=0;j<NoPages;j++)

printf(" \n| %d || %d |",j,PageTable[j]);

int yes=1;

do

{

printf("\nEnter Logical Address to find Physical Address ");

printf("\nEnter page number and offset -- ");

scanf(" %d %d",&y, &offset);

if( y>=NoPages || offset>=Fsize)

{

printf("\n trap: Page Number or offset illegal");

return(0);

}

else

{

pa = (PageTable[y]\*Fsize) + offset;

printf("Fsize=%d,offset=%d\n,frame no=%d",Fsize,offset,PageTable[y]);

printf("\n The Physical Address is -- %d", pa);

}

printf("\nContinue : yes=1,no=0\n");

scanf("%d",&yes);

}while(yes==1);

return 0;

}

Enter the memory size -- 32

Enter the Frame size -- 4

The no. of Frames available in memory are -- 8

Enter no. of pages required : 4

---Enter page table ---

5

6

1

2

---page table ---

| PNo || FNo |

| 0 || 5 |

| 1 || 6 |

| 2 || 1 |

| 3 || 2 |

Enter Logical Address to find Physical Address

Enter page number and offset -- 0 0

Fsize=4,offset=0

,frame no=5

The Physical Address is -- 20

Continue : yes=1,no=0

1

Enter Logical Address to find Physical Address

Enter page number and offset -- 1 3

Fsize=4,offset=3

,frame no=6

The Physical Address is -- 27

Continue : yes=1,no=0

VIVA Questions

1. What is paging?

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

2. What is Page Table?

Page table is a data structure. It maps the page number referenced by the CPU to the frame number where that page is stored.

* Page table is stored in the main memory.
* Number of entries in a page table = Number of pages in which the process is divided.
* Page Table Base Register (PTBR) contains the base address of page table.
* Each process has its own independent page table.

3. How to implement Page Table?

Page Table can be implemented as following way

1. Using fast dedicated registers
2. Keep the page table in main memory
3. Use content-addressable associative registers or TLB

4. What is TLB(**Translation Look-aside Buffers**)

* A Translation look aside buffer can be defined as a memory cache which can be used to reduce the time taken to access the page table again and again.
* It is a memory cache which is closer to the CPU and the time taken by CPU to access TLB is lesser then that taken to access main memory.
* TLB is faster and smaller than the main memory but cheaper and bigger than the register.
* TLB follows the concept of locality of reference which means that it contains only the entries of those many pages that are frequently accessed by the CPU.

**5. How memory Protection is provided for Page table?**

* Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed. Can also add more bits to indicate page execute-only, and so on
* **Valid-invalid** bit attached to each entry in the page table:“valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.“invalid” indicates that the page is not in the process’ logical address space, Or use **page-table length register** (**PTLR**).

**6. What is PTLR?**

Page-table length register (PTLR) indicates size of the page table. These registers are reloaded on context switch by dispatcher.

**7. What is Page-table base register (PTBR)**

The starting location of the page table is stored in a register called page table base register (PTBR)

**8. What is the purpose of paging the page tables?**

**Answer:** In certain situations the page tables could become large enough that by paging the page tables, one could simplify the memory allocation problem (by ensuring that everything is allocated as fixed-size pages as opposed to variable-sized chunks) and also enable the swapping of portions of page table that are not currently used.

**9. What is Protection ?**

If paging or segmentation is provided, different sections of a user program can be declared execute-only, read-only, or read -write. This restriction is necessary with shared code or data and is generally useful in any case to provide simple run-time checks for common programming errors.

10. What is relocation ?

One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory in such a way that the program does not notice the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.

* **compaction**.

11. Difference between Symbolic address, Relative address and Physical address?

**Symbolic addresses:** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.

**Relative addresses:** At the time of compilation, a compiler converts symbolic addresses into relative addresses

**Physical addresses:** The loader generates these addresses at the time when a program is loaded into main memory.

Program 6.2: To simulate Segmentation technique of memory management.

**AIM: To Write** [**C Program**](http://sourcecodesonline.blogspot.com/search/label/C%20Programs) **for the Implementation of Segmentation technique of Memory management.**

**Description: Segmentation** is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name

and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset.

Segment table contains mainly two information about segment:

1. Base: It is the base address of the segment
2. Limit: It is the length of the segment.

The Segment number is mapped to the segment table. The limit of the respective segment is compared with the offset. If the offset is less than the limit then the address is valid otherwise it throws an error as the address is invalid. In the case of valid address, the base address of the segment is added to the offset to get the physical address of actual word in the main memory.

**Algorithm:**

Step 1: Read number of segments

Step 2: Read Segmentation Table data( Base value and Limit)

Step 3: Read Logical address( Segment no and offset value)

Step 4: With segment no as index into Segmentation Table find Limit value and base value

Step 5: Compare Limit value with offset value

1. If offset value is valid find physical address

Physical address= Base address + offset

2. Else print error message

**Program:**

#include<stdio.h>

int main()

{

int i, y, PhyAddre, offset;

int NoSeg, SegmentTable[10][10];

printf("\nEnter number segments -- ");

scanf("%d", &NoSeg);

printf("\nEnter the Segmentation Table data: Base value & Limit Value\n -- ");

for(i=0;i<NoSeg;i++)

scanf("%d%d", &SegmentTable[i][0],&SegmentTable[i][1]);

printf("\n----Enter the Segmentation Table data---\n");

printf("Segment NO || Base || Limit || \n");

for(i=0;i<NoSeg;i++)

printf("|| %d || %d || %d || \n",i,SegmentTable[i][0],SegmentTable[i][1]);

int yes=1;

do

{

printf("\nEnter Logical Address to find Physical Address ");

printf("\nEnter segment number and offset\n ");

scanf(" %d %d",&y, &offset);

if(offset>SegmentTable[y][1])

{

printf("Trap : Addressing Error\n");

}

else

{

PhyAddre=SegmentTable[y][0]+offset;

printf("\n The Physical Address is -- %d", PhyAddre);

printf("\nContinue : yes=1,no=0\n");

scanf("%d",&yes);

}

}while(yes==1);

return 0;

}

**OUTPUT:**  
  
Enter number of segments -- 5

Enter the Segmentation Table data: Base value & Limit Value

--

219 600

2300 14

90 100

1327 580

1952 96

----Enter the Segmentation Table data---

Segment NO || Base || Limit ||

|| 0 || 219 || 600 ||

|| 1 || 2300 || 14 ||

|| 2 || 90 || 100 ||

|| 3 || 1327 || 580 ||

|| 4 || 1952 || 96 ||

Enter Logical Address to find Physical Address

Enter segment number and offset

0 430

The Physical Address is -- 649

Continue : yes=1,no=0

1

Enter Logical Address to find Physical Address

Enter segment number and offset

1 10

The Physical Address is -- 2310

Continue : yes=1,no=0

1

Enter Logical Address to find Physical Address

Enter segment number and offset

2 500

Trap : Addressing Error

Continue : yes=1,no=0  
Viva Questions  
1. What is the functionality of Memory Management Component?

1. Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.
2. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free.
3. It checks how much memory is to be allocated to processes.
4. It decides which process will get memory at what time.
5. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

2. Difference between Physical and logical address?

* Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.
* The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**

3. What is MMU?

* The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

4. What is swapping?

* Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes.
* Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory**

Program to implement and simulate the MFT (Memory Management with Fixed Partition)algorithm and MVT(Memory Management with Variable Partition).

**DESCRIPTION**

**MFT (Multiprogramming with a Fixed number of Tasks**) is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change.

**MVT (Multiprogramming with a Variable number of Tasks)** is the memory management technique in which each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system.

MVT is a more ``efficient'' user of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

**MFT ALGORITHM:**

Step1: start

Step2: Declare variables.

Step3: Enter total memory size.

Step4: Read the no of partitions to be divided.

Step5: Allocate memory for os.

Step6: calculate available memory by subtracting the memory of os from total memory

Step7: calculate the size of each partition by dividing available memory with no of partitions.

Step8: Read the number of processes and the size of each process.

Step9: If size of process<= size of partition then allocate memory to that process.

Step10: Display the wastage of memory.// Internal Fragmentation

Step11: Stop

**PROGRAM:**

#include<stdio.h>

void main()

{

int ms,i,ps[20],n,size,p[20],s,intr=0;

printf("Enter size of memory:");

scanf("%d",&ms);

printf("Enter memory for OS:");

scanf("%d",&s);

ms-=s;

printf("Enter no.of partitions to be divided:");

scanf("%d",&n);

size=ms/n;

printf("Size of each partition is %d \n",size);

for(i=0;i<n;i++)

{

printf("Enter process and process size");

scanf("%d %d",&p[i],&ps[i]);

if(ps[i]<=size)

{

intr=intr+size-ps[i];

printf("process[%d] is allocated\n",p[i]);

}

else

printf("process[%d] size is greater that size of the partition : blocked",p[i]);

}

printf("total Internal fragmentation is %d",intr);

}

**OUTPUT:**

Enter size of memory:50

Enter memory for OS:10

Enter no.of partitions to be divided:4

Enter process and process size1 10

process[1] is allocated

Enter process and process size2 9

process[2] is allocated

Enter process and process size3 9

process[3] is allocated

Enter process and process size4 8

process[4] is allocated

total Internal fragmentation is 4

**MVT ALGORITHM:**

Step1: start

Step2: Declare variables.

Step3: Enter total memory size.

Step4: Read the no of processes

Step5: Allocate memory for os.

Step6: read the size of each process

Step7:calculate available memory by subtracting the memory of os from total memory

Step8: If available memory >= size of process then allocate memory to that process.

Step9: Display the wastage of memory.

Step10: Stop .

**PROGRAM:**

#include<stdio.h>

void main()

{

int i,m,n,tot,s[20];

printf("Enter total memory size:");

scanf("%d",&tot);

printf("Enter no. of processes:");

scanf("%d",&n);

printf("Enter memory for OS:");

scanf("%d",&m);

for(i=0;i<n;i++)

{

printf("Enter size of process %d:",i+1);

scanf("%d",&s[i]);

}

tot=tot-m;

for(i=0;i<n;i++)

{

if(tot>=s[i])

{

printf("Allocate memory to process %d\n",i+1);

tot=tot-s[i];

}

else

printf("process p%d is blocked\n",i+1);

}

printf("External Fragmentation is=%d",tot);

}

**OUTPUT:**

Enter total memory size : 50

Enter no.of pages : 4

Enter memory for OS :10

Enter size of page : 10

Enter size of page : 9

Enter size of page : 9

Enter size of page : 10

External Fragmentation is = 2

**VIVA QUESTIONS:**

1. Differentiate MFT and MVT?

1. MFT or fixed partitioning Scheme:

* In fixed scheme, the OS will be divided into fixed sized blocks. It takes place at the time of installation.
* At compile time, we can bind only addresses
* Degree of multiprogramming is not flexible. This is because the number of blocks is fixed resulting in memory wastage due to fragmentation.

2. MVT OR variable partitioning Scheme**:**

* In variable partitioning scheme there are no partitions at the beginning. There is only the OS area and the rest of the available RAM.
* The memory is allocated to the processes as they enter. This method is more flexible as there is no internal fragmentation and there is no size limitation.

2. What is fragmentation?

As processes are loaded and then removed from the memory due to which the free space in the memory gets broken down into smaller pieces. This mainly creates small non utilized fragmented memory spaces that are so small due to which normal processes can not fit into them. The problem due to which memory space is not getting utilized at all is commonly known as **Fragmentation** in the operating system

3. What is internal fragmentation?

Internal Fragmentation is a problem that occurs when the process is allocated to a memory block whose size is more than the size of that process and due to which some part of the memory is left unused. Thus the space wasted inside the allocated memory block is due to the restriction on the allowed sizes of allocated blocks.

4. What is external fragmentation?

When the memory space in the system can easily satisfy the requirement of the processes, but this available memory space is **non-contiguous**, So it can’t be utilized further. Then this problem is referred to as **External Fragmentation.**

5. What is Compaction?

Compaction is a solution for the problem of External Fragmentation .The main goal is to shuffle the memory contents so as to place all the free memory together in one large block

A C program to implement FIFO page replacement algorithm

**AIM: To write c program to implement FIFO page replacement algorithm.**

**Description:**

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

**Page Fault –** A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

**First In First Out (FIFO) –**   
This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

**Program:**

#include<stdio.h>

int main()

{

int i,j,n,a[50],frame[10],no,k,avail,count=0;

printf("\n ENTER THE NUMBER OF PAGES:\n");

scanf("%d",&n);

printf("\n ENTER THE PAGE NUMBER :\n");

for(i=1;i<=n;i++)

scanf("%d",&a[i]);

printf("\n ENTER THE NUMBER OF FRAMES :");

scanf("%d",&no);

for(i=0;i<no;i++)

frame[i]= -1;

j=0;

printf("\tref string\t page frames\n");

for(i=1;i<=n;i++)

{

printf("%d\t\t",a[i]);

avail=0;

for(k=0;k<no;k++)

if(frame[k]==a[i])

avail=1;

if (avail==0)

{

frame[j]=a[i];

j=(j+1)%no;

count++;

for(k=0;k<no;k++)

printf("%d\t",frame[k]);

}

printf("\n");

}

printf("Page Fault Is %d",count);

return 0;

}

OUTPUT

ENTER THE NUMBER OF PAGES: 20

ENTER THE PAGE NUMBER : 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

ENTER THE NUMBER OF FRAMES :3

ref string page frames

7 7 -1 -1

0 7 0 -1

1 7 0 1

2 2 0 1

0

3 2 3 1

0 2 3 0

4 4 3 0

2 4 2 0

3 4 2 3

0 0 2 3

3

2

1 0 1 3

2 0 1 2

0

1

7 7 1 2

0 7 0 2

1 7 0 1

Page Fault Is 15

**Viva questions**

**What is Belay’s anomaly?**

**Bélády's anomaly** is the phenomenon in which increasing the number of page frames results in an increase in the number of [page faults](https://en.wikipedia.org/wiki/Page_fault) for certain memory access patterns. This phenomenon is commonly experienced when using the first-in first-out ([FIFO](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))) [page replacement algorithm](https://en.wikipedia.org/wiki/Page_replacement_algorithm).

**What is page replacement?**

Page replacement is a process of swapping out an existing page from the frame of a main memory and replacing it with the required page.

**What are different types of page replacement algorithm?**

FIFO Page Replacement Algorithm

LRU Page Replacement Algorithm

Optimal Page Replacement Algorithm

Preemptive Shortest Job First Scheduling in C Programming

**AIM: To write a C program to implement priority scheduling algorithm.**

**Program:**

#include <stdio.h>

int main()

{

      int arrival\_time[10], burst\_time[10], temp[10];

      int i, smallest, count = 0, time, limit;

      double wait\_time = 0, turnaround\_time = 0, end;

      float average\_waiting\_time, average\_turnaround\_time;

      printf("nEnter the Total Number of Processes:t");

      scanf("%d", &limit);

      printf("nEnter Details of %d Processesn", limit);

      for(i = 0; i < limit; i++)

      {

            printf("nEnter Arrival Time:t");

            scanf("%d", &arrival\_time[i]);

            printf("Enter Burst Time:t");

            scanf("%d", &burst\_time[i]);

            temp[i] = burst\_time[i];

      }

      burst\_time[9] = 9999;

      for(time = 0; count != limit; time++)

      {

            smallest = 9;

            for(i = 0; i < limit; i++)

            {

                  if(arrival\_time[i] <= time && burst\_time[i] < burst\_time[smallest] && burst\_time[i] > 0)

                  {

                        smallest = i;

                  }

            }

            burst\_time[smallest]--;

            if(burst\_time[smallest] == 0)

            {

                  count++;

                  end = time + 1;

                  wait\_time = wait\_time + end - arrival\_time[smallest] – temp[smallest];

                  turnaround\_time = turnaround\_time + end - arrival\_time[smallest];

            }

      }

OUTPUT:

Enter the number of the process

3

Enter the arrival time and burst time of the process

AT WT

0 7

2 3

7 2

Id WT TAT

1 5 12

2 0 3

3 0 2

Avg waiting time of the process is 1.666667

Avg turn around time of the process 5.666667

**VIVA Qeuestions:**

**What is a process?**

A program in execution is called a process.

**What are the states of a process?**

1. New

2. Running

3. Waiting

4. Ready

5. Terminated

**What is known as Resource Reservation in Real time Scheduling?**

The Scheduler either admits the process, guaranteeing that the process will complete on time or rejects

the request as impossible. This is known as Resource Reservation.

**What is known as Priority inversion?**

The high priority process would be waiting for a lower -priority one to finish. This situation is known

as Priority Inversion.

**What is meant by Dispatch latency?**

The time taken by the dispatcher to stop one process and start another running is known as Dispatch

Latency.