Dr. D. Y. Patil Pratishthan's

DR. D. Y. PATIL INSTITUTE OF ENGINEERING, MANAGEMENT & RESEARCH

Approved by A.I.C.T.E, New Delhi , Maharashtra State Government, Affiliated to Savitribai Phule Pune University Sector No. 29, PCNTDA , Nigidi Pradhikaran, Akurdi, Pune 411044. Phone: 020–27654470, Fax: 020-27656566 Website :www.dypiemr.ac.in Email : principal.dypiemr@gmail.com

Department of Artificial Intelligence and Data Science

LAB MANUAL

OPERATING SYSTEM
Second Year Engineering (2020 Course)
Semester-I

Prepared By: Mrs. Mital P. Kadu
Mrs. Ashwini Dhumal

217524: Operating Systems Laboratory

Teaching Scheme Practical: 02 Hours/Week	Credit Scheme 01	Examination Scheme and Marks Term Work: 25 Marks

Companion Course: 217525: Operating Systems

Course Objectives:

- To learn and understand process, resource and memory management
- To understand shell scripting and shell programming

Course Outcomes:

On completion of the course, learner will be able to–

- CO1: Choose the best CPU scheduling algorithm for a given problem instance
- CO2: Demonstrate interprocess communication
- CO3: Apply deadlock avoidance algorithm
- CO4: Compare performance of page replacement algorithms CO5: Demonstrate the fundamental UNIX commands & system calls

Table of Contents

EXPERIMENT NO. 1 (Group A)

- **Aim:** Given the list of processes, their CPU burst times. Display/print the Gantt chart for FCFS, SJF, Priority and Round Robin scheduling algorithm. Compute and print the average waiting time and average turnaround time

- **Outcome: At end of this experiment, student will be able understand the scheduler, and how its behaviour influences the performance of the system**

- **Hardware Requirement:**

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

- **Software Requirement:**

Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

- **Theory:**

First come first serve (FCFS)

(FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

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.

Implementation:

1- Input the processes along with their burst time (bt).
2- Find waiting time (wt) for all processes.
3- As first process that comes need not to wait so Waiting time for process 1 will be 0 i.e. wt[0] = 0.
4- Find **waiting time** for all other processes i.e. for Process i -> wt[i] = bt[i-1] + wt[i-1] .
5- Find **turnaround time** = waiting_time + burst_time for all processes.
6- Find **average waiting time** = total_waiting_time / no_of_processes.
7- Similarly, find **average turnaround time** = total_turn_around_time / no_of_processes.

Shortest Job First (SJF) Scheduling

SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready

queue, is going to be scheduled next. However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.

Implementation :

1. Sort all the process according to the arrival time.
2. Then select that process which has minimum arrival time and minimum Burst time.
3. After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

Priority CPU Scheduling

Priority scheduling is one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with the highest priority is to be executed first and so on. Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Implementation:

1- First input the processes with their burst time and priority.
2- Sort the processes, burst time and priority according to the priority.
3- Now simply apply FCFS algorithm.

Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

How to compute below times in Round Robin using a program?

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time.
   Waiting Time = Turn Around Time – Burst Time

// C++ program for implementation of FCFS

```
#include<iostream>
using namespace std;
// Function to find the waiting time for all
// processes

void findWaitingTime(int processes[], int n,int bt[], int wt[])

{

        // waiting time for first process is 0
        wt[0] = 0;
        // calculating waiting time
        for (int i = 1; i < n ; i++ )
                wt[i] = bt[i-1] + wt[i-1] ;

}



// Function to calculate turn around time

void findTurnAroundTime( int processes[], int n,

                                int bt[], int wt[], int tat[])

{

        // calculating turnaround time by adding
        // bt[i] + wt[i]
        for (int i = 0; i < n ; i++)
                tat[i] = bt[i] + wt[i];

}



//Function to calculate average time

void findavgTime( int processes[], int n, int bt[])

{

        int wt[n], tat[n], total_wt = 0, total_tat = 0;
```

```
        //Function to find waiting time of all processes

        findWaitingTime(processes, n, bt, wt);



        //Function to find turn around time for all processes

        findTurnAroundTime(processes, n, bt, wt, tat);

        //Display processes along with all details

        cout << "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];
                cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
                        << wt[i] <<"\t\t " << tat[i] <<endl;

        }



        cout << "Average waiting time = "

                << (float)total_wt / (float)n;

        cout << "\nAverage turn around time = "

                << (float)total_tat / (float)n;

}
```

```
// Driver code

int main()

{

        //process id's
        int processes[] = { 1, 2, 3};
        int n = sizeof processes / sizeof processes[0];
        //Burst time of all processes
        int burst_time[] = {10, 5, 8};
        findavgTime(processes, n, burst_time);
        return 0;

}
```

----------Output: -

Processes Burst time Waiting time Turn around time

1	10	0	10
2	5	10	15
3	8	15	23

Average waiting time = 8.33333

Average turn around time = 16

// C++ program to implement Shortest Job first with Arrival time

```
#include <iostream>
using namespace std;
int mat[10][6];

void swap(int* a, int* b)

{
        int temp = *a;
        *a = *b;
        *b = temp;

}



void arrangeArrival(int num, int mat[][6])

{

        for (int i = 0; i < num; i++) {

                for (int j = 0; j < num - i - 1; j++) {

                        if (mat[j][1] > mat[j + 1][1]) {

                                for (int k = 0; k < 5; k++) {

                                        swap(mat[j][k], mat[j + 1][k]);

                                }

                        }

                }

        }

}

void completionTime(int num, int mat[][6])

{

        int temp, val;

        mat[0][3] = mat[0][1] + mat[0][2];
```

```
        mat[0][5] = mat[0][3] - mat[0][1];

        mat[0][4] = mat[0][5] - mat[0][2];


        for (int i = 1; i < num; i++) {

                temp = mat[i - 1][3];

                int low = mat[i][2];

                for (int j = i; j < num; j++) {

                        if (temp >= mat[j][1] && low >= mat[j][2]) {

                                low = mat[j][2];

                                val = j;

                        }

                }

                mat[val][3] = temp + mat[val][2];

                mat[val][5] = mat[val][3] - mat[val][1];

                mat[val][4] = mat[val][5] - mat[val][2];

                for (int k = 0; k < 6; k++) {

                        swap(mat[val][k], mat[i][k]);

                }

        }

}

int main()

{

        int num, temp;
        cout << "Enter number of Process: ";
        cin >> num;
```

```
        cout << "...Enter the process ID...\n";
        for (int i = 0; i < num; i++) {
                cout << "...Process " << i + 1 << "...\n";
                cout << "Enter Process Id: ";
                cin >> mat[i][0];
                cout << "Enter Arrival Time: ";
                cin >> mat[i][1];
                cout << "Enter Burst Time: ";
                cin >> mat[i][2];

        }



        cout << "Before Arrange...\n";
        cout << "Process ID\tArrival Time\tBurst Time\n";
        for (int i = 0; i < num; i++)
        {
                cout << mat[i][0] << "\t\t" << mat[i][1] << "\t\t"
                        << mat[i][2] << "\n";

        }



        arrangeArrival(num, mat);

        completionTime(num, mat);

        cout << "Final Result...\n";

        cout << "Process ID\tArrival Time\tBurst Time\tWaiting "

                        "Time\tTurnaround Time\n";

        for (int i = 0; i < num; i++) {

                cout << mat[i][0] << "\t\t" << mat[i][1] << "\t\t"

                        << mat[i][2] << "\t\t" << mat[i][4] << "\t\t"

                        << mat[i][5] << "\n";

        }

    }
```

----------Output----------

Enter number of Process: ...Enter the process ID...

Before Arrange...

Process ID Arrival Time Burst Time

Final Result...

Process ID Arrival Time Burst Time Waiting Time Turnaround Time

Output:

Process ID	Arrival Time	Burst Time
1	2	3
2	0	4
3	4	2
4	5	4

Final Result...

Process ID	Arrival Time	Burst Time	Waiting Time	Turnaround Time
2	0	4	0	4
3	4	2	0	2
1	2	3	4	7
4	5	4	4	8

// C++ program for implementation of RR scheduling

```
#include<iostream>
using namespace std;



// Function to find the waiting time for all

// processes

void findWaitingTime(int processes[], int n,

                        int bt[], int wt[], int quantum)

{

        // Make a copy of burst times bt[] to store remaining

        // burst times.

        int rem_bt[n];

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

                rem_bt[i] = bt[i];

        int t = 0; // Current time

        // Keep traversing processes in round robin manner

        // until all of them are not done.

        while (1)

        {

                bool done = true;

                // Traverse all processes one by one repeatedly

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

                {

                        // If burst time of a process is greater than 0

                        // then only need to process further
```

```
            if (rem_bt[i] > 0)
            {
                done = false; // There is a pending process
                if (rem_bt[i] > quantum)
                {
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t += quantum;
                    // Decrease the burst_time of current process
                    // by quantum
                    rem_bt[i] -= quantum;
                }

                // If burst time is smaller than or equal to
                // quantum. Last cycle for this process
                else
                {
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t = t + rem_bt[i];
                    // Waiting time is current time minus time
                    // used by this process
                    wt[i] = t - bt[i];
```

```
                                        // As the process gets fully executed
                                        // make its remaining burst time = 0
                                        rem_bt[i] = 0;
                                    }
                                }
                            }
                            // If all processes are done
                            if (done == true)
                            break;
                    }
            }

            // Function to calculate turn around time
            void findTurnAroundTime(int processes[], int n,     int bt[], int wt[], int tat[])
            {
                    // calculating turnaround time by adding
                    // bt[i] + wt[i]
                    for (int i = 0; i < n ; i++)
                            tat[i] = bt[i] + wt[i];
            }

            // Function to calculate average time
            void findavgTime(int processes[], int n, int bt[],int quantum)
            {
```

```
        int wt[n], tat[n], total_wt = 0, total_tat = 0;



        // Function to find waiting time of all processes

        findWaitingTime(processes, n, bt, wt, quantum);

        // Function to find turn around time for all processes

        findTurnAroundTime(processes, n, bt, wt, tat);

        // Display processes along with all details

        cout << "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];
                cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
                        << wt[i] <<"\t\t " << tat[i] <<endl;

        }



        cout << "Average waiting time = "

                << (float)total_wt / (float)n;

        cout << "\nAverage turn around time = "

                << (float)total_tat / (float)n;

}

// Driver code

int main()
```

```
{

        // process id's
        int processes[] = { 1, 2, 3};
        int n = sizeof processes / sizeof processes[0];

        // Burst time of all processes
        int burst_time[] = {10, 5, 8};

        // Time quantum
        int quantum = 2;
        findavgTime(processes, n, burst_time, quantum);
        return 0;

}
```

Output

Processes Burst time Waiting time Turn around time

1 10 13 23

2 5 10 15

3 8 13 21

Average waiting time = 12

Average turn around time = 19.6667

EXPERIMENT NO. 2 (Group A)

- **Aim**: Implement producer-consumer problem with counting semaphores and mutex.

- **Outcome: At end of this experiment, student will be able apply the semaphores and mutex.**

- **Hardware Requirement:**

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

- **Software Requirement:**
  Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

- **Theory:**

  Producer-Consumer Problem is also known as **bounded buffer problem**. The Producer-Consumer **Problem** is one of the classic problems of synchronization.

  There are certain restrictions/conditions for both the **producer** and **consumer** process, so that data synchronization can be done without interruption. These are as follows:

- The **producer** tries to insert data into an empty slot of the buffer.
- The **consumer** tries to remove data from a filled slot in the buffer.
- The **producer** must not insert data when the buffer is full.
- The **consumer** must not remove data when the buffer is empty.
- The **producer** and **consumer** should not insert and remove data simultaneously.

For solving the **producer-consumer** problem, **three semaphores** are used:

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

Implementation of producer code

```
void Producer()
{
```

```
    do
{
        //wait until empty > 0
        wait(Empty);
        wait(mutex);
        add()
        signal(mutex);
        signal(Full);
    } while(TRUE);
}
```

In the above code:

- **wait(empty):** If the producer has to produce/insert something into the buffer, it needs to first check whether there are empty slots in the buffer. If true, the producer inserts data into the buffer and then decrements one empty slot.
- **wait(mutex):** It is a binary semaphore, hence acquires the lock. This is shared among the **producer** and **consumer**. Hence, if the **producer** is acquiring the lock, the **consumer** cannot make any change in the buffer, until the lock is released.
- **add():** It adds the data to the buffer.
- **signal(mutex):** It simply releases the lock acquired by the **producer** since the addition of data has been done in the buffer.
- **signal(full):** This increments the **full semaphore** since one of the empty slots has now been filled.

Implementation of consumer code

```
void Producer(){

    do{
        //wait until empty > 0
        wait(full);
        wait(mutex);
        consume()
        signal(mutex);
        signal(empty);
    }while(TRUE);
}
```

In the above code:

- **wait(full):** If the consumer has to remove data from the buffer, it needs to first check whether the buffer contains some item or not. If true, the consumer removes the data from the buffer and then decrements one full slot.
- **wait(mutex):** It is a binary semaphore, hence acquires the lock. This is shared among the **producer** and **consumer**. Hence, if the **consumer** is acquiring the lock, the **producer** cannot make any change in the buffer, until the lock is released.
- **consumer():** It removes the data from the buffer.
- **signal(mutex):** It simply releases the lock acquired by the **producer** since the addition of data has been done in the buffer.
- **signal(empty):** This increments the **empty semaphore** since one of the empty slots have now been emptied.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>

/*
This program provides a possible solution for producer-consumer problem using mutex and semaphore.
I have used 5 producers and 5 consumers to demonstrate the solution. You can always play with these values.
*/

#define MaxItems 5 // Maximum items a producer can produce or a consumer can consume
#define BufferSize 5 // Size of the buffer

sem_t empty;
sem_t full;
int in = 0;
int out = 0;
int buffer[BufferSize];
pthread_mutex_t mutex;

void *producer(void *pno)
{
   int item;
   for(int i = 0; i < MaxItems; i++) {
      item = rand(); // Produce an random item
      sem_wait(&empty);
      pthread_mutex_lock(&mutex);
      buffer[in] = item;
      printf("Producer %d: Insert Item %d at %d\n", *((int *)pno),buffer[in],in);
      in = (in+1)%BufferSize;
      pthread_mutex_unlock(&mutex);
      sem_post(&full);
   }
}
```

```
void *consumer(void *cno)
{
   for(int i = 0; i < MaxItems; i++) {
      sem_wait(&full);
      pthread_mutex_lock(&mutex);
      int item = buffer[out];
      printf("Consumer %d: Remove Item %d from %d\n",*((int *)cno),item, out);
      out = (out+1)%BufferSize;
      pthread_mutex_unlock(&mutex);
      sem_post(&empty);
   }
}

int main()
{

   pthread_t pro[5],con[5];
   pthread_mutex_init(&mutex, NULL);
   sem_init(&empty,0,BufferSize);
   sem_init(&full,0,0);

   int a[5] = {1,2,3,4,5}; //Just used for numbering the producer and consumer

   for(int i = 0; i < 5; i++) {
      pthread_create(&pro[i], NULL, (void *)producer, (void *)&a[i]);
   }
   for(int i = 0; i < 5; i++) {
      pthread_create(&con[i], NULL, (void *)consumer, (void *)&a[i]);
   }

   for(int i = 0; i < 5; i++) {
      pthread_join(pro[i], NULL);
   }
   for(int i = 0; i < 5; i++) {
      pthread_join(con[i], NULL);
   }

   pthread_mutex_destroy(&mutex);
   sem_destroy(&empty);
   sem_destroy(&full);

   return 0;

}
```

Output

```
Producer 4: Insert Item 1804289383 at 0
Producer 4: Insert Item 846930886 at 1
Producer 4: Insert Item 1681692777 at 2
Producer 4: Insert Item 1714636915 at 3
Producer 4: Insert Item 1957747793 at 4
Consumer 3: Remove Item 1804289383 from 0
Consumer 3: Remove Item 846930886 from 1
Consumer 3: Remove Item 1681692777 from 2
Consumer 3: Remove Item 1714636915 from 3
Consumer 3: Remove Item 1957747793 from 4
Producer 1: Insert Item 424238335 at 0
Producer 1: Insert Item 719885386 at 1
Producer 1: Insert Item 1649760492 at 2
Producer 1: Insert Item 596516649 at 3
Producer 1: Insert Item 1189641421 at 4
Consumer 1: Remove Item 424238335 from 0
Consumer 1: Remove Item 719885386 from 1
Consumer 1: Remove Item 1649760492 from 2
Consumer 1: Remove Item 596516649 from 3
Consumer 1: Remove Item 1189641421 from 4
Producer 2: Insert Item 1025202362 at 0
Producer 2: Insert Item 1350490027 at 1
Producer 2: Insert Item 783368690 at 2
Producer 2: Insert Item 1102520059 at 3
Producer 2: Insert Item 2044897763 at 4
Consumer 2: Remove Item 1025202362 from 0
Consumer 2: Remove Item 1350490027 from 1
Consumer 2: Remove Item 783368690 from 2
Consumer 2: Remove Item 1102520059 from 3
Consumer 2: Remove Item 2044897763 from 4
Producer 5: Insert Item 1967513926 at 0
Producer 5: Insert Item 1540383426 at 1
Producer 5: Insert Item 304089172 at 2
Producer 5: Insert Item 1303455736 at 3
Producer 5: Insert Item 35005211 at 4
Consumer 5: Remove Item 1967513926 from 0
Consumer 5: Remove Item 1540383426 from 1
Consumer 5: Remove Item 304089172 from 2
Consumer 5: Remove Item 1303455736 from 3
Consumer 5: Remove Item 35005211 from 4
Producer 3: Insert Item 1365180540 at 0
Producer 3: Insert Item 521595368 at 1
Producer 3: Insert Item 294702567 at 2
Producer 3: Insert Item 1726956429 at 3
Producer 3: Insert Item 336465782 at 4
Consumer 4: Remove Item 1365180540 from 0
Consumer 4: Remove Item 521595368 from 1
Consumer 4: Remove Item 294702567 from 2
Consumer 4: Remove Item 1726956429 from 3
Consumer 4: Remove Item 336465782 from 4
```

EXPERIMENT NO. 3 (Group A)

- **Aim**: Demonstrate Reader-Writer problem with reader priority or writer

- **Outcome: At end of this experiment, student will be able apply the semaphores and mutex.**

- **Hardware Requirement:**

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

- **Software Requirement:**
  Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

- **Theory:**

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

This can be implemented using semaphores. The codes for the reader and writer process in the reader-writer problem are given as follows −

Reader Process

The code that defines the reader process is given below −

```
wait (mutex);
rc ++;
if (rc == 1)
wait (wrt);
signal(mutex);
.
. READ THE OBJECT
```

```
.
wait(mutex);
rc --;
if (rc == 0)
signal (wrt);
signal(mutex);
```

In the above code, mutex and wrt are semaphores that are initialized to 1. Also, rc is a variable that is initialized to 0. The mutex semaphore ensures mutual exclusion and wrt handles the writing mechanism and is common to the reader and writer process code.

The variable rc denotes the number of readers accessing the object. As soon as rc becomes 1, wait operation is used on wrt. This means that a writer cannot access the object anymore. After the read operation is done, rc is decremented. When re becomes 0, signal operation is used on wrt. So a writer can access the object now.

Writer Process

The code that defines the writer process is given below:

```
wait(wrt);
.
. WRITE INTO THE OBJECT
.
signal(wrt);
```

If a writer wants to access the object, wait operation is performed on wrt. After that no other writer can access the object. When a writer is done writing into the object, signal operation is performed on wrt.

```
#include<iostream>
#include<mutex>
using namespace std;
struct semaphore
{
int mutex;
int rcount;
int rwait;
bool wrt;
};

void addReader(struct semaphore *s)
```

```
{
if(s->mutex == 0 && s->rcount == 0)
{
cout<<"Sorry, File isopen in Write mode.\nNew Reader added to queue."<<endl;
s->rwait++;
}
else
{
cout<<"Reader Process added."<<endl;
s->rcount++;
s->mutex--;
}
return ;
}


void addWriter(struct semaphore *s)
{
if(s->mutex==1)
{
s->mutex--;
s->wrt=1;
cout<<"\nWriter Process added."<<endl;
}
else if(s->wrt)
cout<<"Sorry, Writer already operational."<<endl;
else
cout<<"Sorry, File open in Read mode."<<endl;
return ;
}

void removeReader(struct semaphore *s)
{
if(s->rcount == 0) cout<<"No readers to remove."<<endl;
else
{
cout<<"Reader Removed."<<endl;
s->rcount--;
s->mutex++;
}
return ;
}

void removeWriter(struct semaphore *s)
{
if(s->wrt==0) cout<<"No Writer to Remove"<<endl;
else
 {
cout<<"Writer Removed"<<endl;
s->mutex++;
s->wrt=0;
if(s->rwait!=0)
{
```

```
s->mutex-=s->rwait;
s->rcount=s->rwait;
s->rwait=0;
cout<<"waiting Readers Added:"<<s->rcount<<endl;
}
}
}



int main()
{
struct semaphore S1={1,0,0};
while(1)
{
cout<<"Options"<<endl<<"1. Add Reader."<<endl<<"2. Add Writer."<<endl<<"3. Remove Reader."<<endl<<"4.
Remove Writer."<<endl<<"5. Exit.<<Choice : "<<endl;
int choice;
cin>>choice;
switch(choice)
{
case 1: addReader(&S1); break;
case 2: addWriter(&S1); break;
case 3: removeReader(&S1); break;
case 4: removeWriter(&S1); break;
case 5: cout<<"\n\tGoodBye!";break;
default: cout<<"\nInvalid Entry!";
}
}
return 0;
}
```

-output

```
Options
1. Add Reader.
2. Add Writer.
3. Remove Reader.
4. Remove Writer.
5. Exit.<<Choice :
1
Reader Process added.
Options
1. Add Reader.
2. Add Writer.
3. Remove Reader.
4. Remove Writer.
5. Exit.<<Choice :
2
```

Sorry, File open in Read mode.
Options
1. Add Reader.
2. Add Writer.
3. Remove Reader.
4. Remove Writer.
5. Exit.<<Choice :
3
Reader Removed.
Options
1. Add Reader.
2. Add Writer.
3. Remove Reader.
4. Remove Writer.
5. Exit.<<Choice :
2

Writer Process added.
Options
1. Add Reader.
2. Add Writer.
3. Remove Reader.
4. Remove Writer.
5. Exit.<<Choice :
1
Sorry, File isopen in Write mode.
New Reader added to queue.
Options
1. Add Reader.
2. Add Writer.
3. Remove Reader.
4. Remove Writer.
5. Exit.<<Choice :

EXPERIMENT NO. 4 (Group A)

- **Aim**: Write a program to implement the Bankers Algorithm

- **Outcome: At end of this experiment, student will be able resource allocation and deadlock avoidance.**

- **Hardware Requirement:**

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

- **Software Requirement:**

Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

- **Theory:**

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that 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.

Why Banker's algorithm is named so?

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it. In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needsof all its customers. The bank would try to be in safe state always.

Following **Data structures** are used to implement the Banker's Algorithm:
Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.
Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R}_j$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P}_i$ may request at most **'k'** instances of resource type $\mathbf{R}_j$.
-

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P}_i$ is currently allocated **'k'** instances of resource type $\mathbf{R}_j$

Need:

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P}_i$ currently need **'k'** instances of resource type $\mathbf{R}_j$
- Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation specifies the resources currently allocated to process $\mathbf{P}_i$ and Need specifies the additional resources that process $\mathbf{P}_i$ may still request to complete its task. Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both

a) Finish[i] = false

b) $Need_i <= Work$

if no such i exists goto step (4)

3) Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

4) if Finish [i] = true for all i

then the system is in a safe state

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i[j] = k$ means process P_i wants k instances

of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i$ <= $Need_i$

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$ <= Available

Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifyin g the state as follows:

Available = Available – Requesti

$Allocation_i$ = $Allocation_i$ + $Request_i$

$Need_i$ = $Need_i$– $Request_i$

```
// Banker's Algorithm
#include <iostream>
using namespace std;

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 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++) {
for (j = 0; j < m; j++)
need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
for (i = 0; i < n; i++) {
if (f[i] == 0) {

        int flag = 0;
        for (j = 0; j < m; j++) {
        if (need[i][j] > avail[j]){
                flag = 1;
                break;
        }
        }

        if (flag == 0) {
        ans[ind++] = i;
        for (y = 0; y < m; y++)
                avail[y] += alloc[i][y];
        f[i] = 1;
        }
}
}
}

cout << "Following is the SAFE Sequence" << endl;
for (i = 0; i < n - 1; i++)
cout << " P" << ans[i] << " ->";
cout << " P" << ans[n - 1] <<endl;

return (0);
}
```

----------output----------

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

EXPERIMENT NO. 5 (Group A)

- **Aim**: Write a program to implement page Replacement strategies (FIFO, LRU,Optimal)

- **Outcome: At end of this experiment, student will be find the page Fault & able to know which algorithm is best for page Replacement strategies**

- **Hardware Requirement:**

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

- **Software Requirement:**

Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

- **Theory:**

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.

Since actual physical memory is much smaller than virtual memory, page faults happen. **In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page.** Different page replacement algorithms suggest different ways to decide which page to replace. **The target for all algorithms is to reduce the number of page faults.**

Page Replacement Algorithms:

1. First In First Out (FIFO) –

This is the simplest page replacement algorithm.

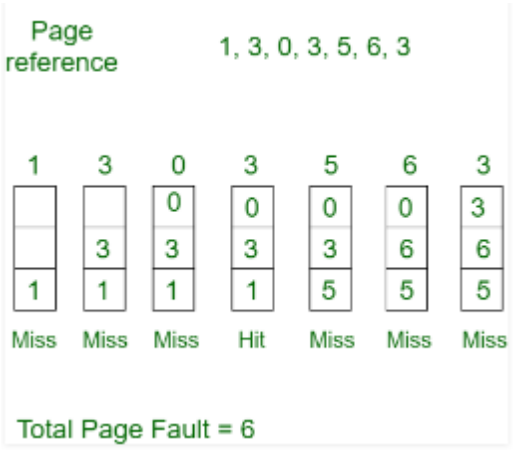
In this algorithm, the operating system keeps track of all pages in the memory in a queue; the oldest page is in the front of the queue.

When a page needs to be replaced page in the front of the queue is selected for removal.

Example-1

Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames.

Find number of page faults.



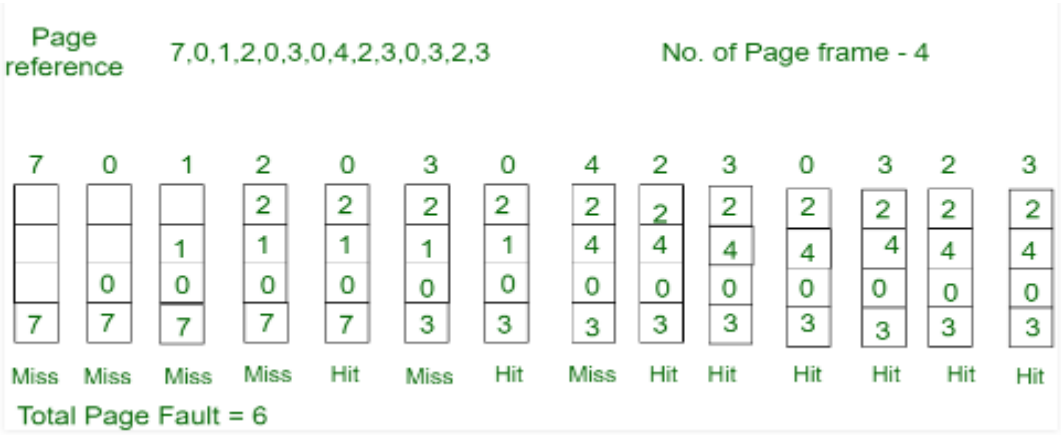
Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.**
when 3 comes, it is already in memory so —> **0 Page Faults.**
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>**1 Page Fault**.
6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>**1 Page Fault**.
Finally when 3 come it is not available so it replaces 0 —>1 **page fault**

2. Optimal Page replacement –

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example-2:

Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.



Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**

0 is already there so —> **0 Page fault.**

when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.**

0 is already there so —> **0 Page fault.**.

4 will takes place of 1 —> **1 Page Fault.**

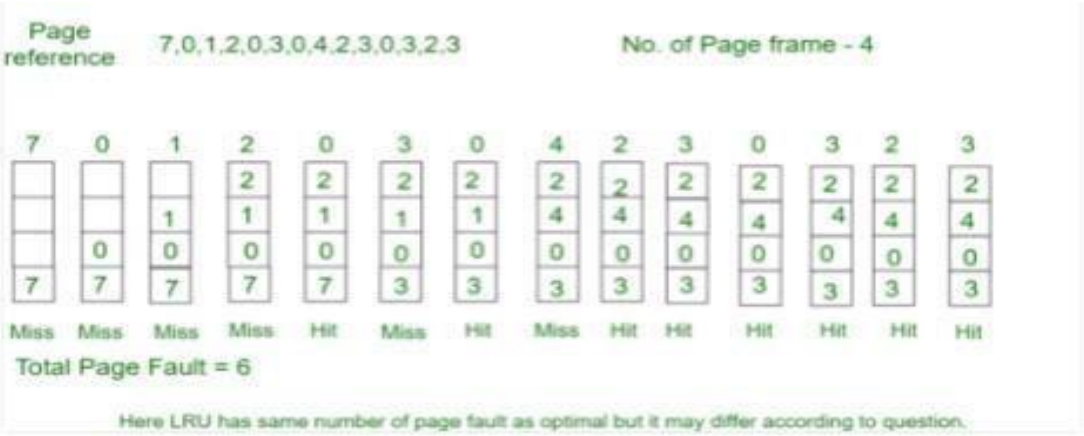
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

3. Least Recently Used –

In this algorithm page will be replaced which is least recently used. **Example-3**

Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames. Find number of page faults.



Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**
0 is already in memory so —> **0 Page fault**.
4 will takes place of 1 —> **1 Page Fault**
Now for the further page reference string —> **0 Page fault**
because they are already available in the memory.

```
// C  implementation of FIFO page replacement
// in Operating Systems.
#include <stdio.h>
int main()
{
    int referenceString[10], pageFaults = 0, m, n, s, pages, frames;
    printf("\nEnter the number of Pages:\t");
    scanf("%d", &pages);
    printf("\nEnter reference string values:\n");
    int(m = 0; m < pages; m++)
    {
        printf("Value No. [%d]:\t", m + 1);
        scanf("%d", &referenceString[m]);
    }
    printf("\n What are the total number of frames:\t");
    {
        scanf("%d", &frames);
    }
    inttemp[frames];
    for(m = 0; m < frames; m++)
    {
        temp[m] = -1;
    }
    for(m = 0; m < pages; m++)
    {
        s = 0;
        for(n = 0; n < frames; n++)
        {
            if(referenceString[m] == temp[n])
            {
                s++;
                pageFaults--;
            }
        }
        pageFaults++;
        if((pageFaults <= frames) && (s == 0))
        {
            temp[m] = referenceString[m];
        }
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = referenceString[m];
        }
        printf("\n");
        for(n = 0; n < frames; n++)
        {
            printf("%d\t", temp[n]);
        }
    }
    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

----------output----------

Enter the number of Pages: 5
Enter reference string values:
Value No. [1]: 4
Value No. [2]: 1
Value No. [3]: 2
Value No. [4]: 4
Value No. [5]: 5
What are the total number of frames: 3

4	**-1**	**-1**
4	**1**	**-1**
4	**1**	**2**
4	**1**	**2**
5	**1**	**2**

Total number of page faults: 4

```
#include<stdio.h>

int findLRU(int time[], int n){
int i, minimum = time[0], pos = 0;

for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum = time[i];
pos = i;
}
}
return pos;
}

int main()
{
   int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j, pos, faults
= 0;
printf("Enter number of frames: ");
scanf("%d", &no_of_frames);
printf("Enter number of pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string: ");
   for(i = 0; i < no_of_pages; ++i){
    scanf("%d", &pages[i]);
   }

for(i = 0; i < no_of_frames; ++i){
    frames[i] = -1;
   }

   for(i = 0; i < no_of_pages; ++i){
    flag1 = flag2 = 0;

    for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == pages[i]){
    counter++;
    time[j] = counter;
  flag1 = flag2 = 1;
  break;
  }
   }

    if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == -1){
    counter++;
    faults++;
```

```
        frames[j] = pages[i];
        time[j] = counter;
        flag2 = 1;
        break;
        }
        }
        }

        if(flag2 == 0){
        pos = findLRU(time, no_of_frames);
        counter++;
        faults++;
        frames[pos] = pages[i];
        time[pos] = counter;
        }

        printf("\n");

        for(j = 0; j < no_of_frames; ++j){
        printf("%d\t", frames[j]);
        }
    }
    printf("\n\nTotal Page Faults = %d", faults);

       return 0;
    }
```

----------output----------

Enter number of frames: 3
Enter number of pages: 6
Enter reference string: 5 7 5 6 7 3

5 -1 -1
5 7 -1
5 7 -1
5 7 6
5 7 6
3 7 6

Total Page Faults = 4

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

```
int  main()
{
//variable declaration and initialization
int frames_number, pages_number, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, miss = 0;
//code to input the frame number
printf("Enter number of frames: ");
scanf("%d", & frames_number);
//code to input number of pages
printf("Enter number of pages: ");
scanf("%d", &pages_number);
//code to define reference string, page numbers, and frame numbers
printf("Enter page reference string: ");
for(i = 0; i < pages_number; ++i){
scanf("%d", &pages[i]);
}
for(i = 0; i < frames_number; ++i){
frames[i] = -1;
}
for(i = 0; i < pages_number; ++i){
flag1 = flag2 = 0;
for(j = 0; j < frames_number; ++j){
if(frames[j] == pages[i]){
flag1 = flag2 = 1;
break;
}
}
//definition of the flag at the starting of the string
if(flag1 == 0){
for(j = 0; j < frames_number; ++j){
if(frames[j] == -1){
faults++;
frames[j] = pages[i];
flag2 = 1;
break;
}
}
}
// definition of the flag at the mid position
if(flag2 == 0){
flag3 =0;
for(j = 0; j < frames_number; ++j){
temp[j] = -1;
for(k = i + 1; k < pages_number; ++k){
if(frames[j] == pages[k]){
temp[j] = k;
break;
}
}
}
for(j = 0; j < frames_number; ++j){
if(temp[j] == -1){
pos = j;
```

```
flag3 = 1;
break;
}
}
//definition of flag at the rear position
if(flag3 ==0){
max = temp[0];
pos = 0;
for(j = 1; j < frames_number; ++j){ if(temp[j] > max){
max = temp[j];
pos = j;
}
}
}
frames[pos] = pages[i];
miss++;
}
printf("\n");
for(j = 0; j < frames_number; ++j){
printf("%d\t", frames[j]);
}
}
printf("\n\nTotal Page miss = %d", miss);
return 0;
}
```

----------output----------

Enter number of frames: 3
Enter number of pages: 6
Enter page reference string: 1
0
2
0
3
4

1	**-1**	**-1**
1	**0**	**-1**
1	**0**	**2**
1	**0**	**2**
3	**0**	**2**
4	**0**	**2**

Total Page miss = 2

EXPERIMENT NO. 6 (Group A)

- **Aim**: Write a Program to implement paging simulation using Least Recently Used (LRU) and Optimal algorithm

- **Outcome: At end of this experiment, student will be able**

- **Hardware Requirement:**

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

- **Software Requirement:**

Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

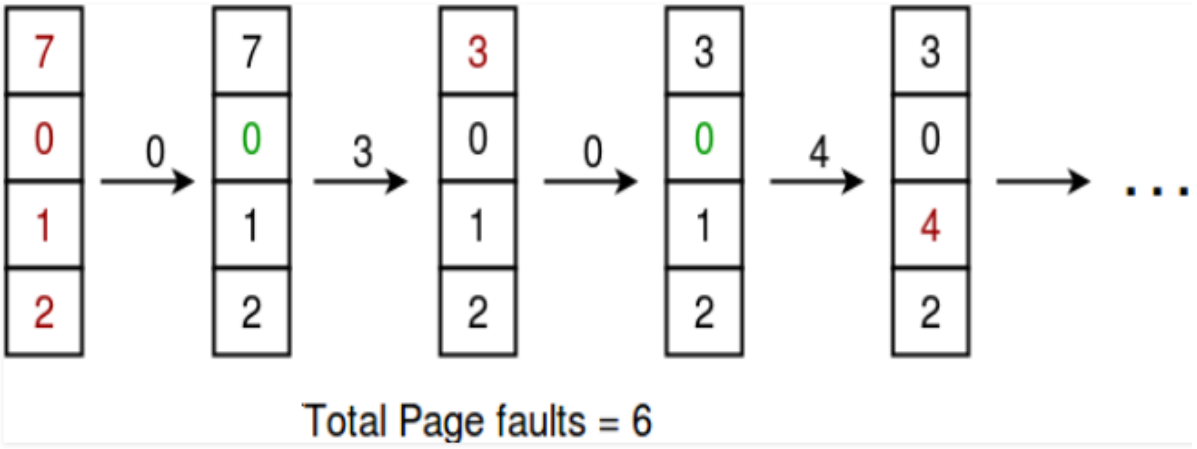
- **Theory:**

Least Recently Used (LRU)

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

In Least Recently Used (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely

Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.
Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults
0 is already their so —> 0 Page fault.
when 3 came it will take the place of 7 because it is least recently used —>1 Page fault
0 is already in memory so —> 0 Page fault.
4 will takes place of 1 —> 1 Page Fault
Now for the further page reference string —> 0 Page fault because they are already available in the memory.



Optimal Page Replacement Algorithm

In operating systems, whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of pagefaults.

In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

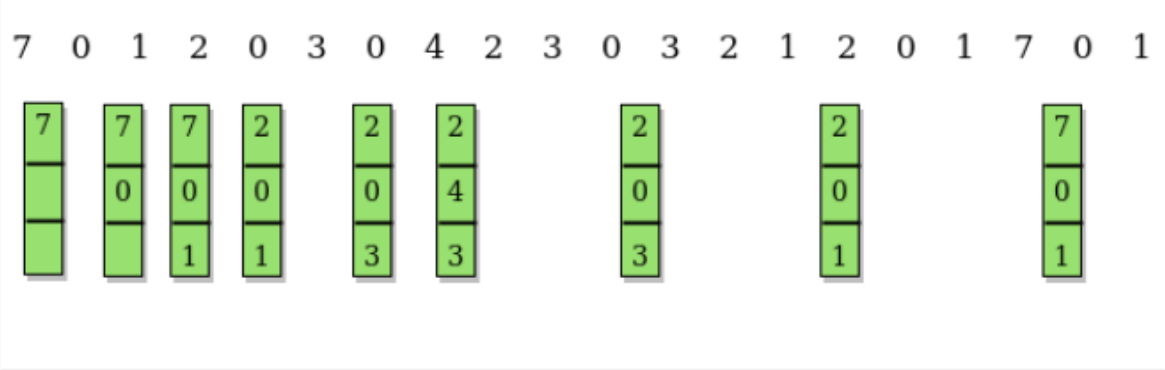
Examples:

```
Input : Number of frames, fn = 3
        Reference String, pg[] = {7, 0, 1, 2,
               0, 3, 0, 4, 2, 3, 0, 3, 2, 1,
               2, 0, 1, 7, 0, 1};
Output : No. of hits = 11
         No. of misses = 9

Input : Number of frames, fn = 4
        Reference String, pg[] = {7, 0, 1, 2,
                  0, 3, 0, 4, 2, 3, 0, 3, 2};
Output : No. of hits = 7
         No. of misses = 6
```

The idea is simple, for every reference we do following :

1. If referred page is already present, increment hit count.
2. If not present, find if a page that is never referenced in future. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced farthest in future. Replace this page with new page.




```
//C++ implementation of above algorithm  LRU
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using indexes
int pageFaults(int pages[], int n, int capacity)
{
    // To represent set of current pages. We use
    // an unordered_set so that we quickly check
    // if a page is present in set or not
    unordered_set<int> s;

    // To store least recently used indexes
    // of pages.
    unordered_map<int, int> indexes;

    // Start from initial page
    int page_faults = 0;
    for (int i=0; i<n; i++)
    {
            // Check if the set can hold more pages
            if (s.size() < capacity)
            {
                    // Insert it into set if not present
                    // already which represents page fault
                    if (s.find(pages[i])==s.end())
                    {
                            s.insert(pages[i]);
```

```
                        // increment page fault
                        page_faults++;
                }

                // Store the recently used index of
                // each page
                indexes[pages[i]] = i;
        }

        // If the set is full then need to perform lru
        // i.e. remove the least recently used page
        // and insert the current page
        else
        {
                // Check if current page is not already
                // present in the set
                if (s.find(pages[i]) == s.end())
                {
                        // Find the least recently used pages
                        // that is present in the set
                        int lru = INT_MAX, val;
                        for (auto it=s.begin(); it!=s.end(); it++)
                        {
                                if (indexes[*it] < lru)
                                {
                                        lru = indexes[*it];
                                        val = *it;
                                }
                        }

                        // Remove the indexes page
                        s.erase(val);

                        // insert the current page
                        s.insert(pages[i]);

                        // Increment page faults
                        page_faults++;
                }

                // Update the current page index
                indexes[pages[i]] = i;
        }
    }

    return page_faults;
}

// Driver code
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
```

```
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

6

```
// CPP program to demonstrate optimal page
// replacement algorithm.
#include <bits/stdc++.h>
using namespace std;

// Function to check whether a page exists
// in a frame or not
bool search(int key, vector<int>& fr)
{
    for (int i = 0; i < fr.size(); i++)
        if (fr[i] == key)
            return true;
    return false;
}

// Function to find the frame that will not be used
// recently in future after given index in pg[0..pn-1]
int predict(int pg[], vector<int>& fr, int pn, int index)
{
    // Store the index of pages which are going
    // to be used recently in future
    int res = -1, farthest = index;
    for (int i = 0; i < fr.size(); i++) {
        int j;
        for (j = index; j < pn; j++) {
            if (fr[i] == pg[j]) {
                if (j > farthest) {
                    farthest = j;
                    res = i;
                }
                break;
            }
        }

        // If a page is never referenced in future,
        // return it.
        if (j == pn)
            return i;
    }

    // If all of the frames were not in future,
    // return any of them, we return 0. Otherwise
    // we return res.
    return (res == -1) ? 0 : res;
```

```
}

void optimalPage(int pg[], int pn, int fn)
{
    // Create an array for given number of
    // frames and initialize it as empty.
    vector<int> fr;

    // Traverse through page reference array
    // and check for miss and hit.
    int hit = 0;
    for (int i = 0; i < pn; i++) {

            // Page found in a frame : HIT
            if (search(pg[i], fr)) {
                    hit++;
                    continue;
            }

            // Page not found in a frame : MISS

            // If there is space available in frames.
            if (fr.size() < fn)
                    fr.push_back(pg[i]);

            // Find the page to be replaced.
            else {
                    int j = predict(pg, fr, pn, i + 1);
                    fr[j] = pg[i];
            }
    }
    cout << "No. of hits = " << hit << endl;
    cout << "No. of misses = " << pn - hit << endl;
}

// Driver Function
int main()
{
    int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };
    int pn = sizeof(pg) / sizeof(pg[0]);
    int fn = 4;
    optimalPage(pg, pn, fn);
    return 0;
}
```

-Output-

No. of hits = 7

No. of misses = 6

EXPERIMENT NO. 7 (Group B)

- **Aim**: Implement UNIX system calls like ps, fork, join, exec family, and wait for process management (use shell script/ Java/ C programming)Shell programming

- **Outcome:** Understand the implementation of the UNIX system calls **like ps, fork, join, exec family,** and **wait** for process management**.**

Hardware Requirement:

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

Software Requirement:

Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

Theory:

exec family of functions in C

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.

Let us see a small example to show how to use execvp() function in C. 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;
}
```

Now, 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:

-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. is not printed because because as soon as the execvp()

function is called, this program is replaced by the program EXEC.c.

execv : This is very similar to execvp() function in terms of syntax as well. The syntax of execv() is as

shown below:Syntax:

int execv(const char *path, char *const argv[]);

path: should point to the path of the file being executed.

argv[]: is a null terminated array of character pointers.

Conclusion:

Hence we have implemented the UNIX system calls like ps, fork, join, exec family, and wait for process management

EXPERIMENT NO. 8 (Group B)

- **Aim**: Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit

- **Outcome:** Student is able to Understand the implementation of the shell script & able to run the shell program.

Hardware Requirement:

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

Software Requirement:

Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

Theory:

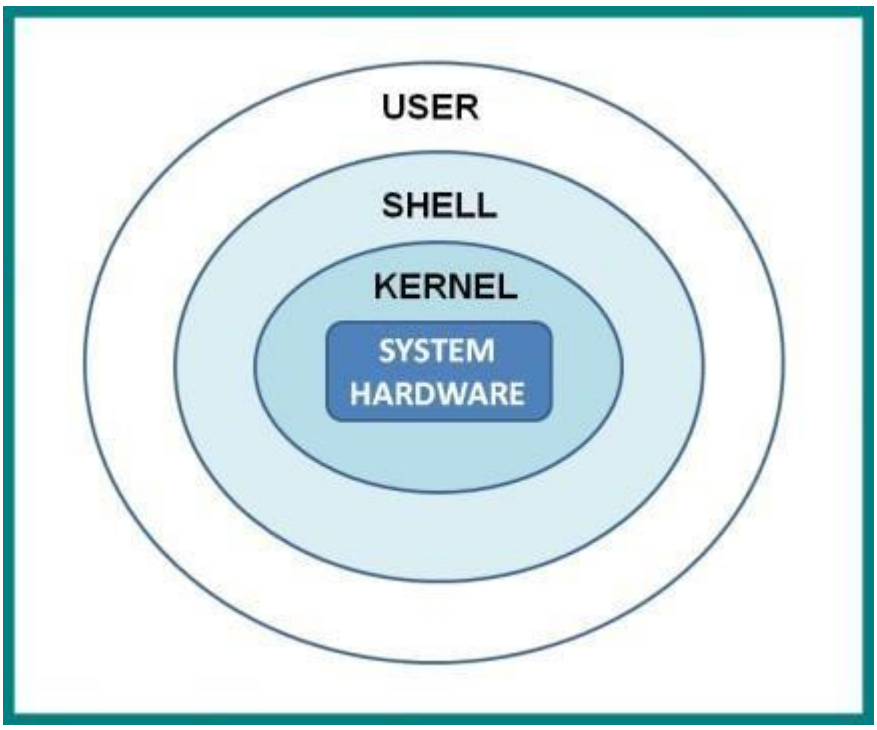
Shell is a UNIX term for an interface between a user and an operating system service.
Shell provides users with an interface and accepts human-readable commands into the system and executes those commands which can run automatically and give the program's output in a shell script.
In Linux, shells like bash and korn support programming construct which are saved as scripts. These scripts become shell commands and hence many Linux commands are script.
Shell Scripting is a program to write a series of commands for the shell to execute
Shell is the native command interpreter.

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

Shell is basically Bridge between **kernel** and the user, a **Command Interpreter** where user can type command and the command is conveyed to the kernel and it will be executed.



Creating and running a basic shell script

A shell script can be created using vi, cat command, or the normal text editor in GUI.

1. Let's create a basic shell script using vi

   $ vi basic_script.sh

2. This will take you to the vi editor. Add the following lines:

   #!/bin/bash
   Who am i
   date

This simple script should display the current user followed by the date.

3. To save and exit the vi editor:

- Press ESC
- Type **:**
- Type in ‘wq’
- Hit Enter

4. $ chmod +x basic_script.sh

This will give you (current user) the permission to execute the file

5. To run the script :

sh basic_script.sh

Another way of running the script is :

$ ./basic_script.sh

```
#! /bin/bash
it=0
a=1
while [ [ $op -lt 7 ] ]
 do
       echo enter the option
       echo "1 for create"
       echo "2 for add"
       echo "3 for display"
       echo "4 for search"
       echo "5 for delete"
       echo "6 for modify"
       echo "7 for exit"
       echo "enter u r choice"
       read op
 $word="$op"

case "$FRUIT" in
"1")
       if [ "$op" == "1" ]
```

```
        then
        echo "Enter the name for the database"
        read db
        touch "$db"
    fi
;;
"2")

    if [ "$op" == "2" ]
        then
        echo "in which database u want to add records"
        read db
        echo "enter the no. of records"
        read n
    while [ $it -lt $n ]
    do
        echo "enter id:"
        read id1
        echo "enter name:"
        read nm
        pa1="^[A-Za-z]"
    while [[ ! $add =~ $pa ]]
    do
        echo "enter valid address:"
        read add
    done

    echo "enter address:"
    read add
    pa="^[A-Za-z0-9]"
    while [[ ! $add =~ $pa ]]
    do
        echo "enter valid address:"
        read add
    done

    #echo $add

    echo "enter phone no.:"
    read ph
    pat="^[0-9]{10}$"
    while [[ ! $ph =~ $pat ]]
    do
        echo "please enter phone number as XXXXXXXXXX:"
        read ph
    done

    #echo $ph
```

```
echo "eter email:"
read em
patem="^[a-z0-9._%-+]+@[a-z]+\.[a-z]{2,4}$"
```

```
        while [[ ! $em =~ $patem ]]
        do
                echo "please enter valid email address"
                read em
        done

        #echo $em

        echo "$id1,$nm,$add,$ph,$em" >> "$db"

        it=`expr $it + 1`
        echo "$it record entered"
        done
        fi
;;
"3")
        if [ "$op" == "3" ]
                then
                echo "enter name of database from where data to be display:"
                read db
                cat $db
        fi
;;
"4")
        if [ "$op" == "4" ]
                then
                echo "enter name of database from where to search:"
                read db
                echo "enter email to be search:"
                read em1
                grep $em1  $db
                echo "record found"
                else
                echo "not found"
        fi
;;

"5")
        if [ "$op" == "5" ]
                then
                echo "enter name of database:"
                read db
                echo "enter id:"
                read id1
                echo "enter line no. u want to delete:"
                read linenumber

                for line in `grep -n "$id1" $db`
                do
```

```
        number=`echo "$line" | cut -c1`
        #echo $number
        if [ $number == $linenumber ]
        then
        lineRemove="${linenumber}d"
        sed -i -e "$lineRemove" $db
        echo "record removed"
        fi
        #echo
        cat $db
        done
    fi
;;
"6")

    if [ "$op" == "6" ]
        then
        echo "enter name of database:"
        read db
        echo "enter id:"
        read id1
        echo "enter line u want to modify:"
        read linenumber

        for line in `grep -n "$id1" "$db"`
        do
        number=`echo "$line" | cut -c1`

        if [ "$number" == "$linenumber" ]
        then
        echo "what would u like to change"
        echo "\"id,name,address,mobile,email\""
        read edit
        linechange="${linenumber}s"
        sed -i -e "$linechange/.*/$edit/" $db
        echo record edited
        fi
        done
    fi
;;
"7")

    echo "bye"
;;
*) echo invalid input
esac
done
```

Conclusion:

Hence we have implemented the address book by using shell script program

EXPERIMENT NO. 9 (Group B)

- **Aim**: Create a shell program to do mathematical operations.

- **Outcome:** Student able to implement mathematical operations in shell script

Hardware Requirement:

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

Software Requirement:

Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

Theory:

A variable in a shell script is a means of **referencing** a **numeric** or **character value**. And unlike formal programming languages, a shell script doesn't require you to **declare a type** for your variables

User Defined Variables

These variables are defined by **users**. A shell script allows us to set and use our **own variables** within the script. Setting variables allows you to **temporarily store data** and use it throughout the script, making the shell script more like a real computer program.

User variables can be any text string of up to **20 letters**, **digits**, or **an underscore character**. User variables are case sensitive, so the variable Var1 is different from the variable var1. This little rule often gets novice script programmers in trouble.

Values are assigned to user variables using an **equal sign.** No spaces can appear between the variable, the equal sign, and the value (another trouble spot for novices). Here are a few examples of assigning values to user variables:

var1=10
var2=-57
var3=testing
var4="still more testing"

The shell script **automatically determines the data type** used for the variable value. Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes.

Shell Arithmetic

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then	[$a == $b] would return

	returns true.	false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

Case Statement

case Statement Syntax

```
case EXPRESSION in

  PATTERN_1)
    STATEMENTS
    ;;

  PATTERN_2)
    STATEMENTS
    ;;

  PATTERN_N)
    STATEMENTS
    ;;

  *)
    STATEMENTS
    ;;
esac
```

- Each **Case** statement starts with the **Case** keyword, followed by the case
- expression and the **in** keyword. The statement ends with the **esac** keyword.
- You can use multiple patterns separated by the **| operator. The)** operator terminates a pattern list.
- A pattern can have **special characters**
- A pattern and its associated commands are known as a clause.
- Each clause must be terminated with ;;
- The commands corresponding to the first pattern that matches the expression are executed.
- It is a common practice to use the wildcard asterisk symbol (*) as a final pattern to define the default case. This pattern will always match.
- If no pattern is matched, the return status is zero. Otherwise, the return status is the **exit status** of the executed commands.

ALGORITHM:

Step 1: Start

Step 2: Read the two Numbers.

Step 3: Get the operation choice from the User

Step 4: Give the expressions for each case and solve them.

Step 5: Print the Result

```
echo "Enter Two Numbers"
read a b
echo "What do you want to do? (1 to 5)"
echo "1) Sum"
echo "2) Difference"
echo "3) Product"
echo "4) Quotient"
echo "5) Remainder"
echo "Enter your Choice"
read n
case "$n" in
1) echo "The Sum of $a and $b is `expr $a + $b`";;
```

```
2) echo "The Difference between $a and $b is `expr $a - $b`";;
3) echo "The Product of the $a and $b is `expr $a \* $b`";;
4) echo "The Quotient of $a by $b is `expr $a / $b`";;
5) echo "The Remainder of $a by $b is `expr $a % $b`";;
esac
```

SAMPLE OUTPUT:

```
[snjb@snjb-desktop]# sh arith_switch.sh
Enter Two Numbers
12 10
What do you want to do? (1 to 5)
1) Sum
2) Difference
3) Product
4) Quotient
5) Remainder
Enter your Choice
4
The Quotient of 12 by 10 is 1
```

Conclusion:

Hence we have implemented the mathematical operations using shell programming

EXPERIMENT NO. 10 (Group C)

- **Aim**: Inter process communication in Linux using Pipes

- **Outcome:** Student able to implement & understand the IPC using pipe

Hardware Requirement:

- 6 GB free disk space.
- 2 GB RAM.
- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

Software Requirement:

Red Hat Enterprise Linux 7/5 /Centos/Ubuntu

- **Theory:**

IPC (InterProcess Communication)

Pipes –

Communication between two related processes.

1. The mechanism is **half duplex** meaning the first process communicates with the second process.
2. To achieve a **full duplex** i.e., for the second process to communicate with the first process another pipe is required.

Pipe

A **Pipe** is a technique used for inter process communication.
A pipe is a mechanism by which the output of one process is directed into the input of another process. Thus it provides one way flow of data between two related processes. It is a mechanism whereby two or more processes communicate with each other to perform tasks
Although pipe can be accessed like an **ordinary file**, the system actually manages it

as **FIFO queue**.
A pipe file is created using the **pipe system call.**

A pipe has an input end and an output end.
One can write into a pipe from input end and read from the output end.
A **pipe descriptor** has an array that stores two pointers, one pointer is for its input end and the other pointer is for its output end.

Suppose **two processes**, **Process A and Process B**, need to communicate. In such a case, it is important that the process which writes, closes its read end of the pipe and the process which reads, closes its write end of a pipe. Essentially, for a communication from Process A to Process B the following should happen.

- **Process A** should keep its write end open and close the read end of the pipe.
- **Process B** should keep its read end open and close its write end. When a pipe is created, it is given a fixed size in bytes.

When a process attempts to write into the pipe, the write request is immediately executed if the pipe is not full. However, if pipe is full the process is blocked until the state of pipe changes. Similarly, a reading process is blocked, if it attempts to read more bytes that are currently in pipe, otherwise the reading process is executed. Only one process can access a pipe at a time.

Pipe Limitations

Pipes are the oldest form of IPC on UNIX
Pipes have been half duplex
Data flows in only one direction

1. Historically, they have been half duplex (i.e .data flows in only one direction)
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child

Pipe Creation

A pipe is created by calling the **pipe function**

#include<unistd.h>

int pipe(int fd[2]);

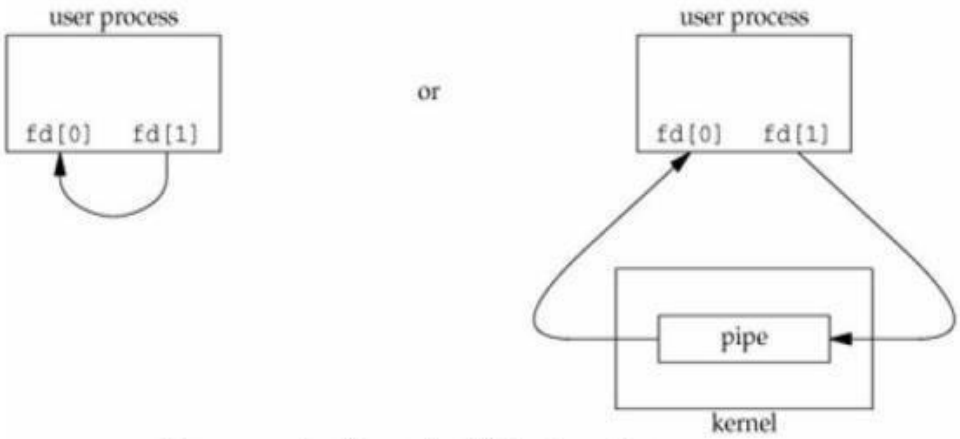
Returns: 0 if OK, –1 on error

PIPES

Two file descriptors are returned through the fd argument:
fd[0] is open for reading.
and
fd[1] is open for writing.
The output of **fd[1]** is the input for **fd[0].**

PIPES

Two ways to picture a half-duplex pipe are shown in the given diagrams below:

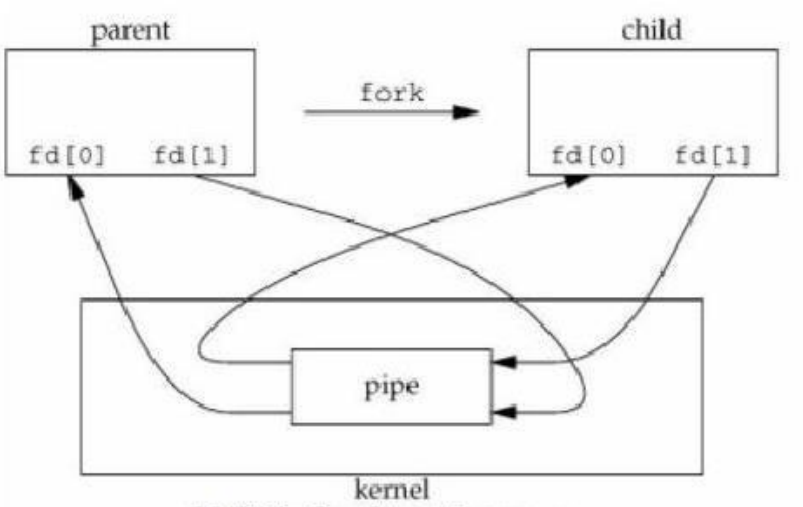


Two ways to view a half-duplex pipe

PIPES

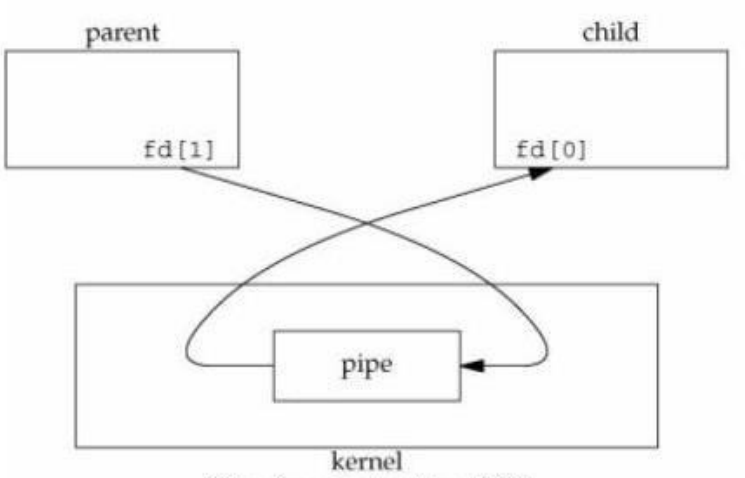
1. **The left half of the diagram shows the two ends of the pipe connected in a single process.**
2. **The right half of the diagram emphasizes that the data in the pipe flows through the kernel.**
3. **A pipe in a single process is next to useless.**
4. **Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa**

PIPES



Half-duplex pipe after a `fork`

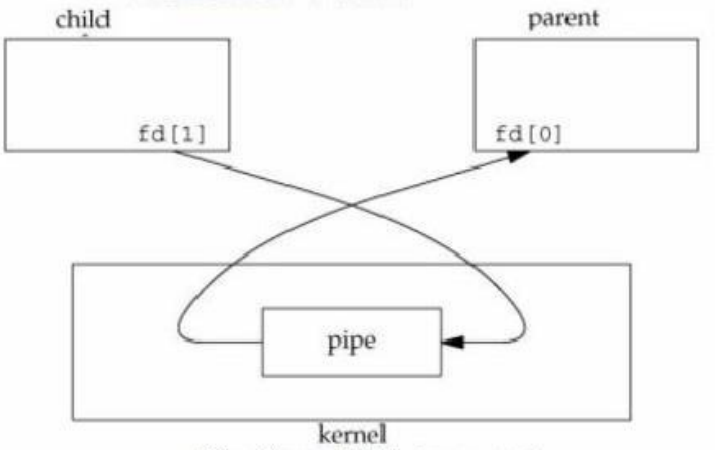
PIPES



Pipe from parent to child

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

PIPES



Pipe from child to parent

For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

PIPES

When one end of a pipe is closed, the following two rules apply.

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

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

Creating a Pipe

```
int fd[2];
pid_t   pid;

if (pipe (fd) < 0)                          1
      Error
if ((pid = fork()) < 0) {                   2
      Error
} else if (pid > 0) {
      close (fd[0]);
      write (fd[1], "Guten Tag\n", 10);     3
} else {
      close (fd[1]);
      n = read (fd[0], line, MAXLINE);      3
}
```

Example Programs

Following are some example programs

Example program 1 − Program to write and read two messages using pipe.

Algorithm

Step 1 − Create a pipe.

Step 2 − Send a message to the pipe.

Step 3 − Retrieve the message from the pipe and write it to the standard output.

Step 4 − Send another message to the pipe.

Step 5 − Retrieve the message from the pipe and write it to the standard output.

Note − Retrieving messages can also be done after sending all messages.

simplepipe.c

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

int main() {
   int pipefds[2];
   int returnstatus;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);

   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }

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

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

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

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

   printf("Writing to pipe - Message 2 is %s\n", writemessages[0]);

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

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

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

   return 0;
```

```
}
```

Execution Steps

gcc -o simplepipe simplepipe.c

Execution/Output

```
Writing to pipe - Message 1 is Hi
Reading from pipe – Message 1 is Hi
Writing to pipe - Message 2 is Hi
Reading from pipe – Message 2 is Hell
```

Example program 2 − 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;
}
```

Execution Steps

Compilation

gcc pipewithprocesses.c –o pipewithprocesses

Execution

Parent Process - Writing to pipe - Message 1 is Hi
Parent Process - Writing to pipe - Message 2 is Hello
Child Process - Reading from pipe – Message 1 is Hi
Child Process - Reading from pipe – Message 2 is Hello

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.

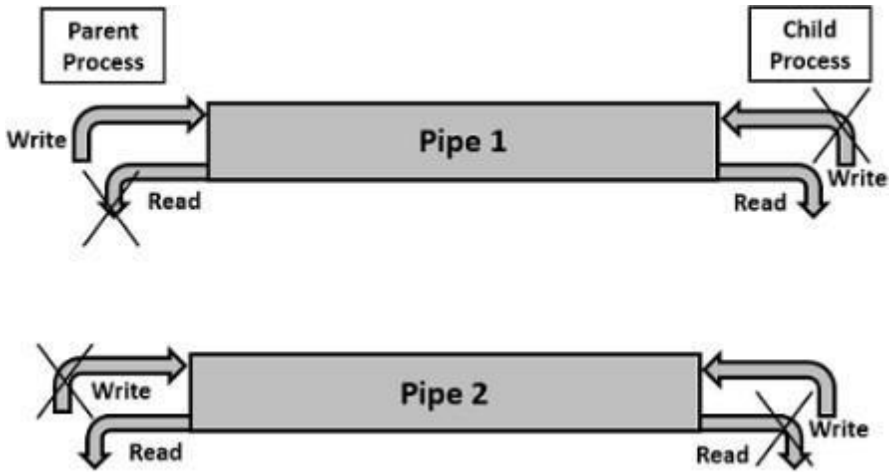
Step 2 − Create a child process.

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 Programs

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;
}
```

Output

```
In Parent: Writing to pipe 1 – Message is Hi
In Child: Reading from pipe 1 – Message is Hi
In Child: Writing to pipe 2 – Message is Hello
In Parent: Reading from pipe 2 – Message is Hello
```

Conclusion:

we have implemented IPC using the pipe, process able to communicate with pipe