**EID224 : OPERATING SYSTEMS LABORATORY LTPC0032**

1.Scheduling algorithms, simulation of first cum first serve cpuscheduling algorithm.

2.Simulation of shortest job first cpu scheduling algorithm.

3.Simulation of shortest job first preemptive cpu scheduling algorithm.

4.Bankers algorithm for dead lock avoidance.

5.Page replacement algorithms, implement first in first out pagereplacement algorithm.

6.Page replacement algorithms, implement least recently used pagereplacement algorithm.

7.Page replacement algorithms, implement optimal page replacementalgorithm.

8.Write a program to implement concurrent programming constructsthrough semaphores -dining philosophers' problem, consumer-producer, readers-writers etc.

9.Write a C program to implement deadlock avoidance algorithms.

10.Write a program to page replacement algorithms.

11.Write a program to implement virtual memory

**Week-1**

**Que**: Scheduling algorithms, simulation of First Come First Serve CPU Scheduling algorithm.

**FCFS CPU SCHEDULING ALGORITHM**

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm. 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. Here we are considering that arrival time for all processes is 0.

How to compute below times ?

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

**PROGRAM**

#include<stdio.h>

void main()

{

int bt[20], wt[20], tat[20], i, n;

float wtavg, tatavg;

printf("\nEnter the number of processes -- ");

scanf("%d", &n);

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

{

printf("\nEnter Burst Time for Process %d -- ", i);

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

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

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

{

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

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

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

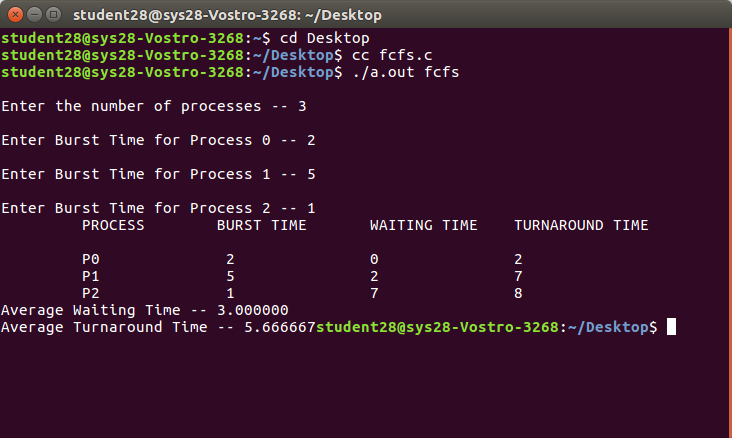
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

}

**OUTPUT:**



**Week-2**

**Que:** Simulation of Shortest job first CPU Scheduling algorithm.

**SJF CPU SCHEDULING ALGORITHM**

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.

* Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
* It is a Greedy Algorithm.
* It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging.
* It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

Algorithm:

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.

How to compute below times in SJF 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

**PROGRAM**

#include<stdio.h>

void main()

{

int p[20], bt[20], wt[20], tat[20], i, k, n, temp;

float wtavg, tatavg;

printf("\nEnter the number of processes -- ");

scanf("%d", &n);

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

{

p[i]=i;

printf("Enter Burst Time for Process %d -- ", i);

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

}

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

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

if(bt[i]>bt[k])

{

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=p[i];

p[i]=p[k];

p[k]=temp;

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

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

{

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

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

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

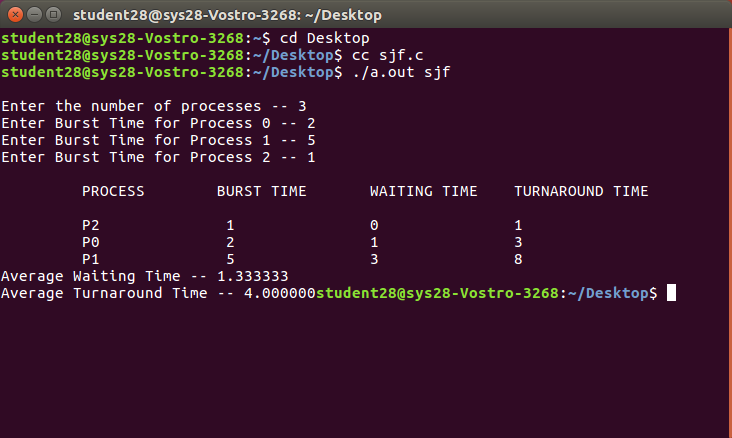
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

}

**OUTPUT:**

****

**Week-3**

**Que:** Simulation of Shortest Job First Preemptive CPU scheduling algorithm.

**SJF Preemptive CPU Scheduling Algorithm**

The preemptive version of SJF known as Shortest Remaining Time First (SRTF).

**Shortest Remaining Time First (SRTF) scheduling**

In the Shortest Remaining Time First (SRTF) scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

**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 Processes\n", 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];

}

}

average\_waiting\_time = wait\_time / limit;

average\_turnaround\_time = turnaround\_time / limit;

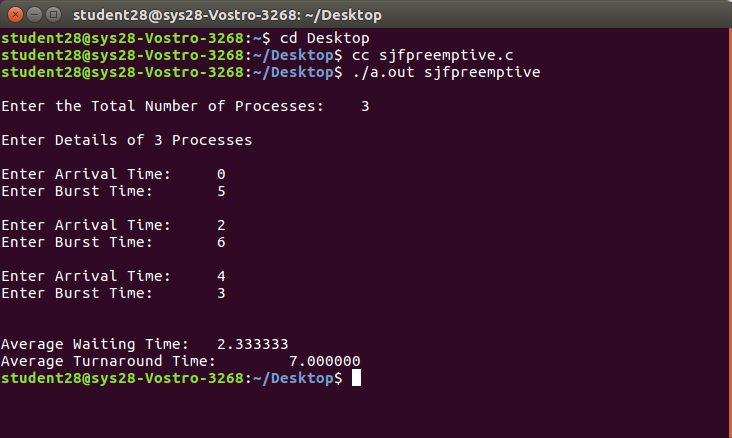
printf("\n\nAverage Waiting Time:\t%lf\n", average\_waiting\_time);

printf("Average Turnaround Time:\t%lf\n", average\_turnaround\_time);

return 0;

}

**OUTPUT:**



**Week-4**

**Que:** Bankers algorithm for dead lock avoidance.

**Bankers algorithm:**

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.

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 Rj

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 Pi may request at most ‘k’ instances of resource type Rj.

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 Pi is currently allocated ‘k’ instances of resource type Rj

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 Pi currently need ‘k’ instances of resource type Rj

for its execution.

* Need [ i, j ] = Max [ i, j ] – Allocation [ i, j ]
* Allocationi specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.

**PROGRAM:**

#include<stdio.h>

struct file

{

int all[10];

int max[10];

int need[10];

int flag;

};

void main()

{

struct file f[10];

int fl;

int i, j, k, p, b, n, r, g, cnt=0, id, newr;

int avail[10],seq[10];

printf("Enter number of processes -- ");

scanf("%d",&n);

printf("Enter number of resources -- ");

scanf("%d",&r);

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

{

printf("Enter details for P%d",i);

printf("\nEnter allocation\t -- \t");

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

scanf("%d",&f[i].all[j]);

printf("Enter Max\t\t -- \t");

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

scanf("%d",&f[i].max[j]);

f[i].flag=0;

}

printf("\nEnter Available Resources\t -- \t");

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

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

printf("\nEnter New Request Details -- ");

printf("\nEnter pid \t -- \t");

scanf("%d",&id);

printf("Enter Request for Resources \t -- \t");

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

{

scanf("%d",&newr);

f[id].all[i] += newr;

avail[i]=avail[i] - newr;

}

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

{

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

{

f[i].need[j]=f[i].max[j]-f[i].all[j];

if(f[i].need[j]<0)

f[i].need[j]=0;

}

}

cnt=0;

fl=0;

while(cnt!=n)

{

g=0;

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

{

if(f[j].flag==0)

{

b=0;

for(p=0;p<r;p++)

{

if(avail[p]>=f[j].need[p])

b=b+1;

else

b=b-1;

}

if(b==r)

{

printf("\nP%d is visited",j);

seq[fl++]=j;

f[j].flag=1;

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

avail[k]=avail[k]+f[j].all[k];

cnt=cnt+1;

printf("(");

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

printf("%3d",avail[k]);

printf(")");

g=1;

}

}

}

if(g==0)

{

printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");

printf("\n SYSTEM IS IN UNSAFE STATE");

goto y;

}

}

printf("\nSYSTEM IS IN SAFE STATE");

printf("\nThe Safe Sequence is -- (");

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

printf("P%d ",seq[i]);

printf(")");

y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");

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

{

printf("P%d\t",i);

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

printf("%6d",f[i].all[j]);

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

printf("%6d",f[i].max[j]);

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

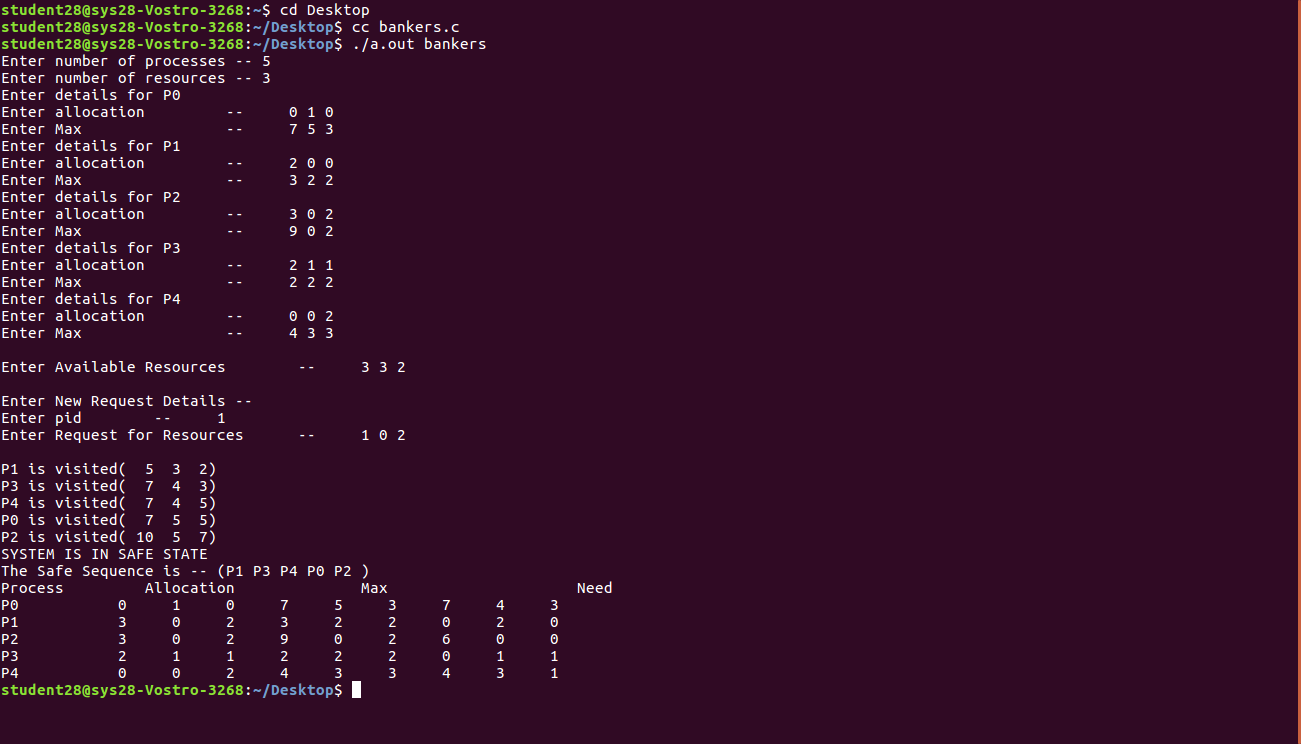
printf("%6d",f[i].need[j]);

printf("\n");

}

}

**OUTPUT:**



**Week- 5**

**Que:** Page replacement algorithms, implement first in first out page replacement algorithm.

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.

Belady’s anomaly:

Belady’s anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm.

**PROGRAM:**

void main()

{

int i, j, k, f, pf=0, count=0, rs[25], m[10], n;

printf("\n Enter the length of reference string -- ");

scanf("%d",&n);

printf("\n Enter the reference string -- ");

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

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

printf("\n Enter no. of frames -- ");

scanf("%d",&f);

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

m[i]=-1;

printf("\n The Page Replacement Process is -- \n");

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

{

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

{

if(m[k]==rs[i])

break;

}

if(k==f)

{

m[count++]=rs[i];

pf++;

}

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

printf("\t%d",m[j]);

if(k==f)

printf("\tPF No. %d",pf);

printf("\n");

if(count==f)

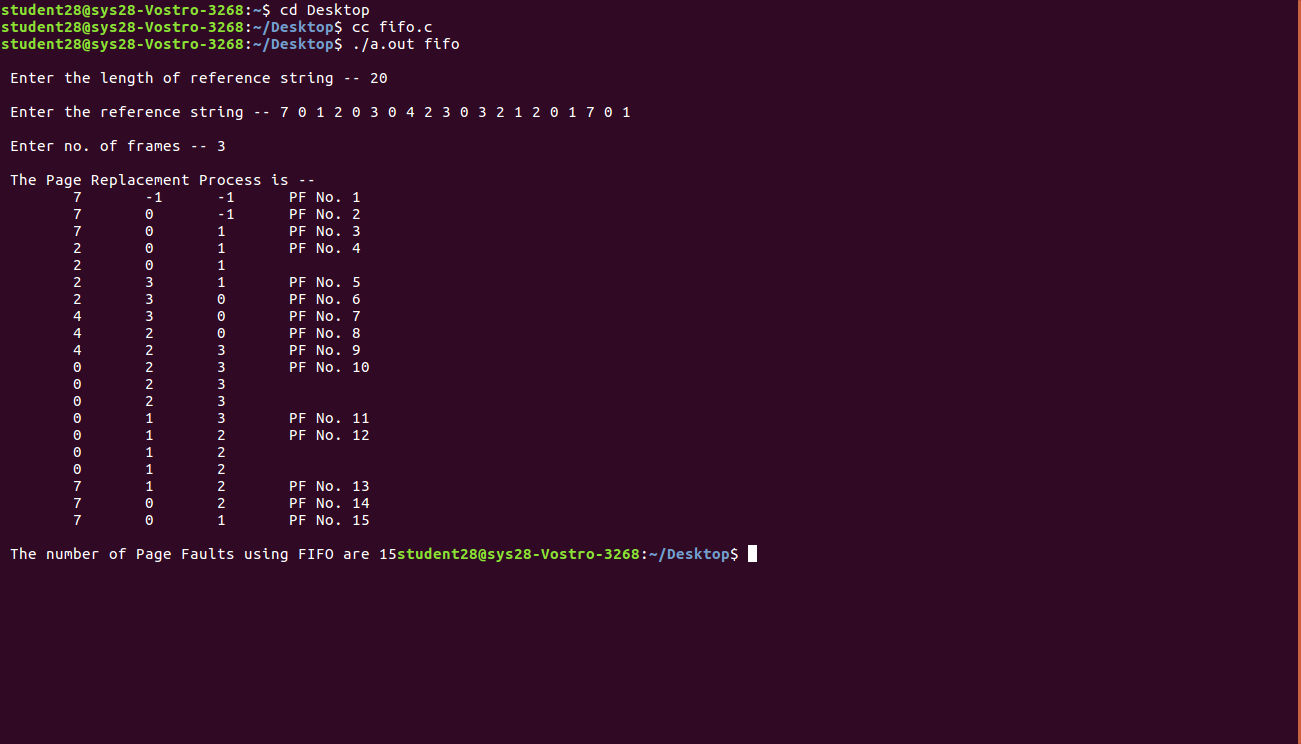
count=0;

}

printf("\n The number of Page Faults using FIFO are %d",pf);

}

**OUTPUT:**



**Week-6**

**Que:** Page replacement algorithms, implement least recently used page replacement algorithm.

**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.

**PROGRAM:**

#include<stdio.h>

void main()

{

int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;

printf("Enter the length of reference string -- ");

scanf("%d",&n);

printf("Enter the reference string -- ");

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

{

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

flag[i]=0;

}

printf("Enter the number of frames -- ");

scanf("%d",&f);

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

{

count[i]=0;

m[i]=-1;

}

printf("\nThe Page Replacement process is -- \n");

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

{

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

{

if(m[j]==rs[i])

{

flag[i]=1;

count[j]=next;

next++;

}

}

if(flag[i]==0)

{

if(i<f)

{

m[i]=rs[i];

count[i]=next;

next++;

}

else

{

min=0;

for(j=1;j<f;j++)

if(count[min] > count[j])

min=j;

m[min]=rs[i];

count[min]=next;

next++;

}

pf++;

}

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

printf("%d\t", m[j]);

if(flag[i]==0)

printf("PF No. -- %d" , pf);

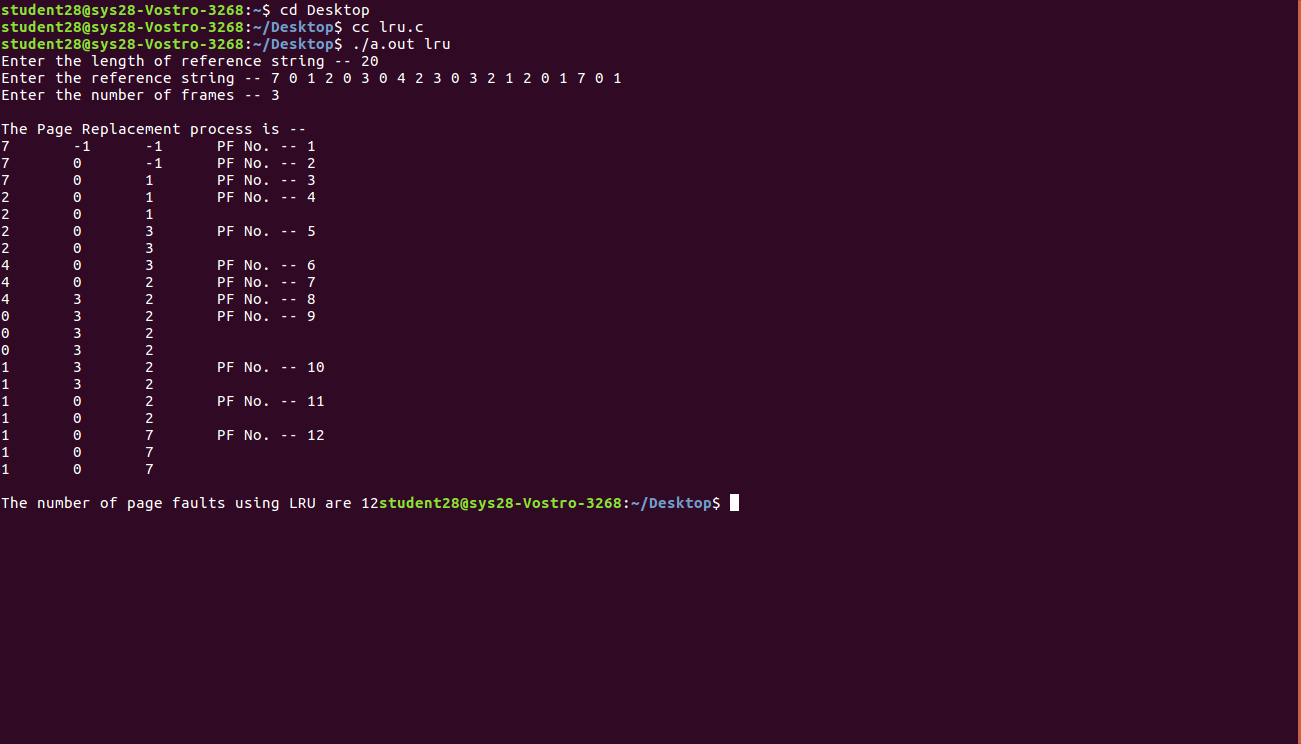
printf("\n");

}

printf("\nThe number of page faults using LRU are %d",pf);

}

**OUTPUT:**



**Week-7**

**Que:** Page replacement algorithms, implement optimal page replacement algorithm.

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

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.

**PROGRAM:**

#include<stdio.h>

int n;

void main()

{

int findmax(int a[]);

int seq[30],fr[5],pos[5],find,flag,max,i,j,m,k,t,s;

int count=1,pf=0,p=0;

float pfr;

printf("Enter maximum limit of the sequence: ");

scanf("%d",&max);

printf("\nEnter the sequence: ");

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

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

printf("\nEnter no. of frames: ");

scanf("%d",&n);

fr[0]=seq[0];

pf++;

printf("%d\t",fr[0]);

i=1;

while(count<n)

{

flag=1;

p++;

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

{

if(seq[i]==seq[j])

flag=0;

}

if(flag!=0)

{

fr[count]=seq[i];

printf("%d\t",fr[count]);

count++;

pf++;

}

i++;

}

printf("\n");

for(i=p;i<max;i++)

{

flag=1;

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

{

if(seq[i]==fr[j])

flag=0;

}

if(flag!=0)

{

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

{

m=fr[j];

for(k=i;k<max;k++)

{

if(seq[k]==m)

{

pos[j]=k;

break;

}

else

pos[j]=1;

}

}

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

{

if(pos[k]==1)

flag=0;

}

if(flag!=0)

s=findmax(pos);

if(flag==0)

{

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

{

if(pos[k]==1)

{

s=k;

break;

}

}

}

fr[s]=seq[i];

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

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

pf++;

printf("\n");

}

}

pfr=(float)pf/(float)max;

printf("\nThe no. of page faults are %d",pf);

printf("\nPage fault rate %f",pfr);

}

int findmax(int a[])

{

int max,i,k=0;

max=a[0];

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

{

if(max<a[i])

{

max=a[i];

k=i;

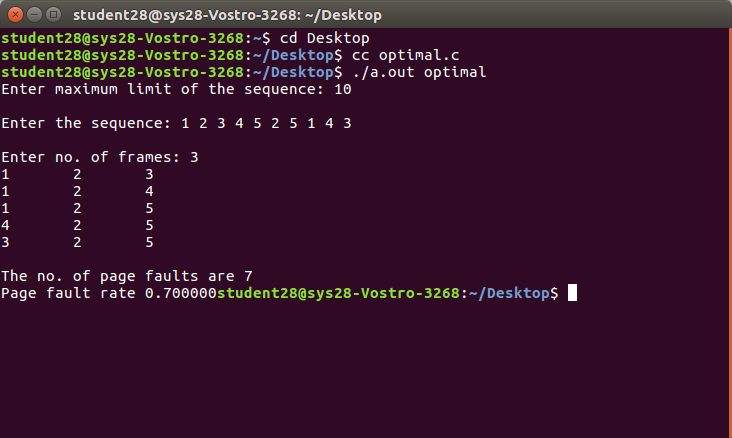
}

}

return k;

}

**OUTPUT:**



**Week-8**

**Que:** Write a program to implement concurrent programming constructs through semaphores -dining philosophers' problem, consumer-producer, readers-writers etc.

**1. Dining philosophers' problem**

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

There are three states of philosopher : **THINKING, HUNGRY and EATING**. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

int tph, philname[20], status[20], howhung, hu[20], cho;

void main()

{

void one();

void two();

int i;

printf("\n\nDINING PHILOSOPHER PROBLEM");

printf("\nEnter the total no. of philosophers: ");

scanf("%d",&tph);

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

{

philname[i] = (i+1);

status[i]=1;

}

printf("How many are hungry : ");

scanf("%d", &howhung);

if(howhung==tph)

{

printf("\nAll are hungry..\nDead lock stage will occur");

printf("\nExiting..");

}

else

{

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

{

printf("Enter philosopher %d position: ",(i+1));

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

status[hu[i]]=2;

}

do

{

printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your choice:");

scanf("%d", &cho);

switch(cho)

{

case 1: one(); consumer-producer

break;

case 2: two();

break;

case 3: exit(0); break;

default: printf("\nInvalid option..");

}

}while(1);

}

}

void one()

{

int pos=0, x, i;

printf("\nAllow one philosopher to eat at any time\n");

for(i=0;i<howhung; i++, pos++)

{

printf("\nP %d is granted to eat", philname[hu[pos]]);

for(x=pos;x<howhung;x++)

printf("\nP %d is waiting", philname[hu[x]]);

}

}

void two()

{

int i, j, s=0, t, r, x;

printf("\n Allow two philosophers to eat at same time\n");

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

{

for(j=i+1;j<howhung;j++)

{

if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)

{

printf("\n\ncombination %d \n", (s+1));

t=hu[i];

r=hu[j];

s++;

printf("\nP %d and P %d are granted to eat", philname[hu[i]],

philname[hu[j]]);

for(x=0;x<howhung;x++)

{

if((hu[x]!=t)&&(hu[x]!=r))

printf("\nP %d is waiting", philname[hu[x]]);

}

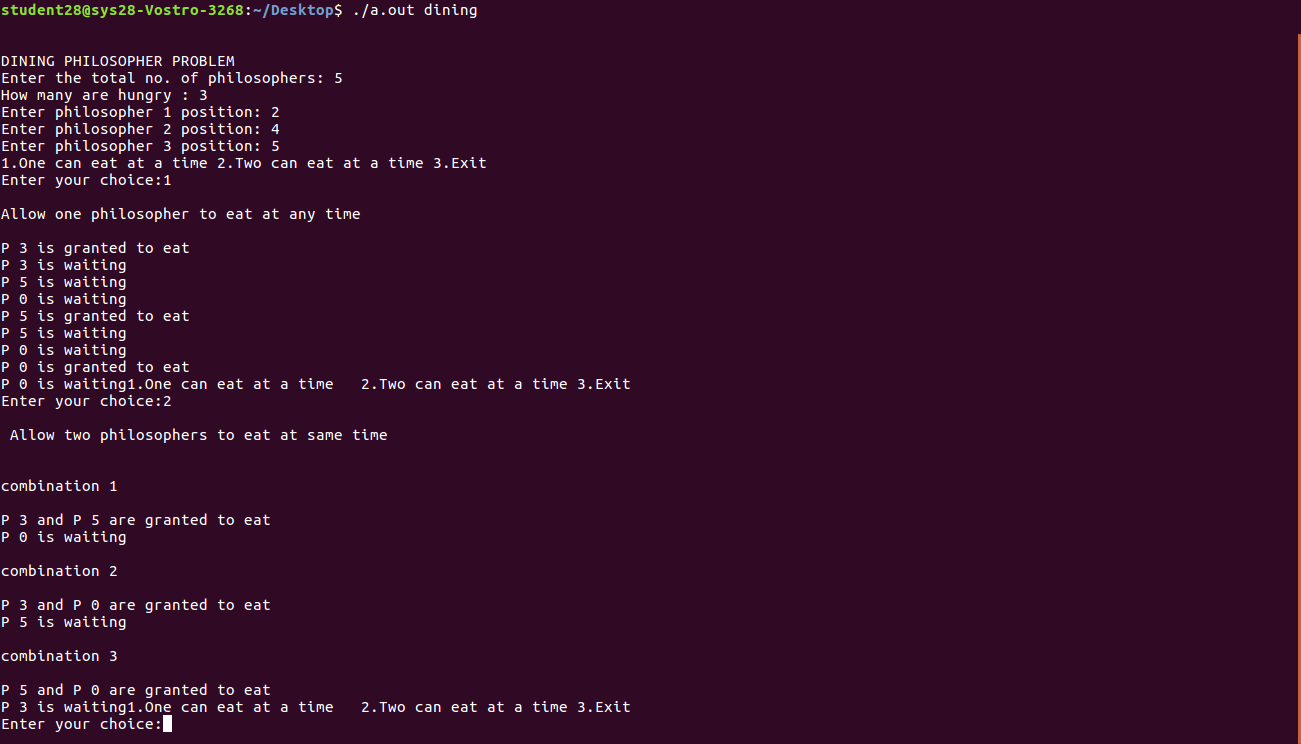
}

}

}

}

**OUTPUT:**



2. **Consumer-producer:**

Producer consumer problem is a classical synchronization problem. We can solve this problem by using semaphores. A Semaphore S is an integer variable that can be accessed only through two standard operations : wait() and signal(). The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

Semaphores are of two types:

1. Binary Semaphore – This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

2. Counting Semaphore – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Producer-consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

void main()

{

int buffer[10], bufsize, in, out, produce, consume, choice=0;

in = 0;

out = 0;

bufsize = 10;

while(choice!=3)

{

printf("\n 1. Produce \t 2. Consume \t 3. Exit");

printf("\nEnter your choice: ");

scanf("%d",&choice);

switch(choice) {

case 1: if((in+1)%bufsize==out)

printf("\nBuffer is Full");

else

{

printf("\nEnter the value:");

scanf("%d", &produce);

buffer[in] = produce;

in = (in+1)%bufsize;

}

break;

case 2: if(in==out)

printf("\nBuffer is Empty");

else

{

consume = buffer[out];

printf("\nThe consumed value is %d", consume);

out = (out+1)%bufsize;

}

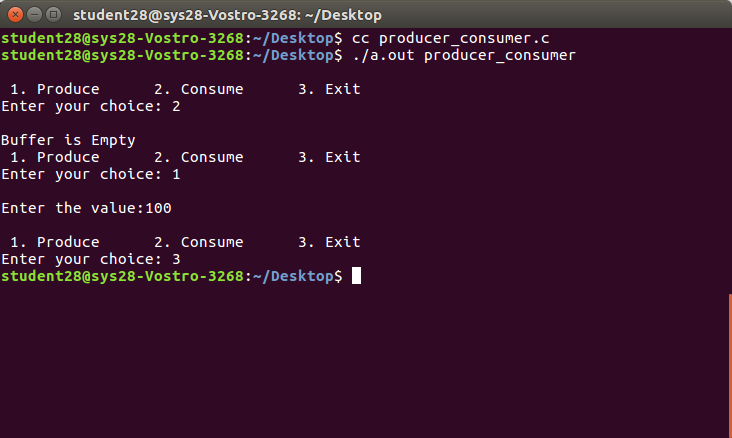
break;

}

}

}

**OUTPUT:**



**3. Readers-writers problem:**

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

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

Precisely in OS we call this situation as the readers-writers problem

Problem parameters:

One set of data is shared among a number of processes

* Once a writer is ready, it performs its write. Only one writer may write at a time
* If a process is writing, no other process can read it
* If at least one reader is reading, no other process can write
* Readers may not write and only read

Solution when Reader has the Priority over Writer

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: mutex, wrt, readcnt to implement solution

1. semaphore mutex, wrt; // semaphore mutex is used to ensure mutual exclusion when readcnt is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers
2. int readcnt; //readcnt tells the number of processes performing read in the critical section, initially 0

Functions for sempahore :

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

PROGRAM:

**Week-9**

**Que:** Write a C program to implement deadlock avoidance algorithms

Write a C program to simulate disk scheduling algorithms

a) FCFS b) SCAN c) C-SCAN

DESCRIPTION

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

**FCFS DISK SCHEDULING ALGORITHM**

#include<stdio.h>

#include<stdlib.h>

void main()

{

int t[20], n, i, j, tohm[20], tot=0;

float avhm;

printf("enter the no.of tracks");

scanf("%d",&n);

printf("enter the tracks to be traversed");

for(i=2;i<n+2;i++)

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

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

{

tohm[i]=t[i+1]-t[i];

if(tohm[i]<0)

tohm[i]=tohm[i]\*(-1);

}

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

tot+=tohm[i];

avhm=(float)tot/n;

printf("Tracks traversed\tDifference between tracks\n");

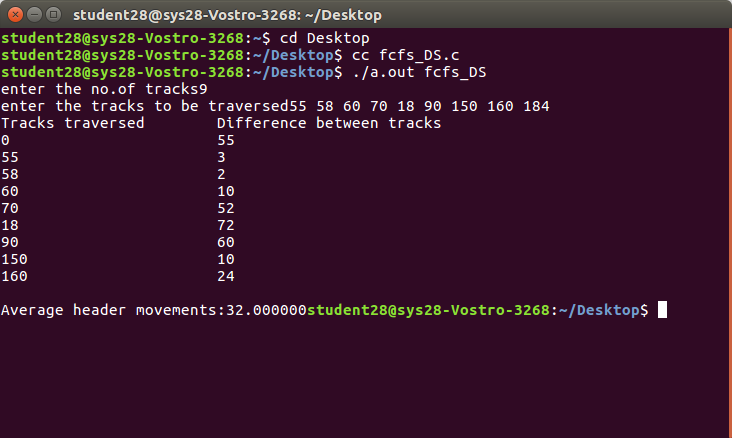
for(i=1;i<n+1;i++)

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

printf("\nAverage header movements:%f",avhm);

}

**OUTPUT:**



**SCAN DISK SCHEDULING ALGORITHM**

#include<stdio.h>

main()

{

int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;

clrscr();

printf("enter the no of tracks to be traveresed");

scanf("%d'",&n);

printf("enter the position of head");

scanf("%d",&h);

t[0]=0;t[1]=h;

printf("enter the tracks");

for(i=2;i<n+2;i++)

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

for(i=0;i<n+2;i++)

{

for(j=0;j<(n+2)-i-1;j++)

{

if(t[j]>t[j+1])

{

temp=t[j];

t[j]=t[j+1];

t[j+1]=temp;

} } }

for(i=0;i<n+2;i++)

if(t[i]==h)

j=i;k=i;

p=0;

while(t[j]!=0)

{

atr[p]=t[j];

j--;

p++;

}

atr[p]=t[j];

for(p=k+1;p<n+2;p++,k++)

atr[p]=t[k+1];

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

{

if(atr[j]>atr[j+1])

d[j]=atr[j]-atr[j+1];

else

d[j]=atr[j+1]-atr[j];

sum+=d[j];

}

printf("\nAverage header movements:%f",(float)sum/n);

getch();

}