**Experiment 2: Process Creation**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

int main(){

int pid1,pid2,pid3;

pid1=fork();

if(pid1==-1)

{

printf("Error");

exit(1);

}

if(pid1!=0)

{

pid2=getpid();

printf("Parent process id: %d",pid2);

}

else

{

pid3=getpid();

printf("Child process id: %d",pid3);

}

}

**Experiment 3: System calls**

#include <stdio.h>

#include<stdlib.h>

#include<dirent.h>

#include <unistd.h>

struct dirent \*dptr;

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

{

char buff[100];

DIR \*dirp;

printf("\n\nENTER THE DIRECTORY NAME:\n");

scanf("%s",buff);

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

{

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

exit(1);

}

while(dptr=readdir(dirp))

{

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

}

closedir(dirp);

}

**Experiment 4: thread Concept**

#include<pthread.h>

#include<stdio.h>

#include<stdlib.h>

void\* func(void\* arg)

{

pthread\_detach(pthread\_self());

printf("Inside the thread\n");

pthread\_exit(NULL);

}

void fun()

{

pthread\_t ptid;

pthread\_create(&ptid,NULL,&func,NULL);

printf("This line may be printed before thread terminates\n");

//pthread\_cancel(ptid);

if(pthread\_equal(ptid,pthread\_self()))

printf("Threads are equal");

else

printf("Threads are not equal");

pthread\_join(ptid,NULL);

printf("This line will be printed after thread ends");

pthread\_exit(NULL);

}

int main(){

fun();

return 0;

}

**Experiment 5: FCFS**

// C program for implementation of FCFS

// scheduling

#include<stdio.h>

// 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

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

// Calculate total waiting time and total turn

// around time

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

{

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

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

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

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

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

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

}

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

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

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

printf("\n");

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

}

// 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;

}

**Experiment 6: SJF**

#include<stdio.h>

void main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;

float avg\_wt,avg\_tat;

printf("Enter number of process:");

scanf("%d",&n);

printf("\nEnter Burst Time:\n");

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

{

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

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

p[i]=i+1; //contains process number

}

//sorting burst time in ascending order using selection sort

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

{

pos=i;

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

{

if(bt[j]<bt[pos])

pos=j;

}

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0; //waiting time for first process will be zero

//calculate waiting time

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

{

wt[i]=0;

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

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=(float)total/n; //average waiting time

total=0;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

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

{

tat[i]=bt[i]+wt[i]; //calculate turnaround time

total+=tat[i];

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

}

avg\_tat=(float)total/n; //average turnaround time

printf("\n\nAverage Waiting Time=%f",avg\_wt);

printf("\nAverage Turnaround Time=%f\n",avg\_tat);

}

**Experiment 7: RR**

#include<stdio.h>

int main()

{

int count,j,n,time,remain,flag=0,time\_quantum;

int wait\_time=0,turnaround\_time=0,at[10],bt[10],rt[10];

printf("Enter Total Process:\t ");

scanf("%d",&n);

remain=n;

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

{

printf("Enter Arrival Time and Burst Time for Process Process Number %d :",count+1);

scanf("%d",&at[count]);

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

rt[count]=bt[count];

}

printf("Enter Time Quantum:\t");

scanf("%d",&time\_quantum);

printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");

for(time=0,count=0;remain!=0;)

{

if(rt[count]<=time\_quantum && rt[count]>0)

{

time+=rt[count];

rt[count]=0;

flag=1;

}

else if(rt[count]>0)

{

rt[count]-=time\_quantum;

time+=time\_quantum;

}

if(rt[count]==0 && flag==1)

{

remain--;

printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);

wait\_time+=time-at[count]-bt[count];

turnaround\_time+=time-at[count];

flag=0;

}

if(count==n-1)

count=0;

else if(at[count+1]<=time)

count++;

else

count=0;

}

printf("\nAverage Waiting Time= %f\n",wait\_time\*1.0/n);

printf("Avg Turnaround Time = %f",turnaround\_time\*1.0/n);

return 0;

}

**Experiment 8: Producer Consumer**

// C program for the above approach

#include <stdio.h>

#include <stdlib.h>

// Initialize a mutex to 1

int mutex = 1;

// Number of full slots as 0

int full = 0;

// Number of empty slots as size

// of buffer

int empty = 10, x = 0;

// Function to produce an item and

// add it to the buffer

void producer()

{

// Decrease mutex value by 1

--mutex;

// Increase the number of full

// slots by 1

++full;

// Decrease the number of empty

// slots by 1

--empty;

// Item produced

x++;

printf("\nProducer produces"

"item %d",

x);

// Increase mutex value by 1

++mutex;

}

// Function to consume an item and

// remove it from buffer

void consumer()

{

// Decrease mutex value by 1

--mutex;

// Decrease the number of full

// slots by 1

--full;

// Increase the number of empty

// slots by 1

++empty;

printf("\nConsumer consumes "

"item %d",

x);

x--;

// Increase mutex value by 1

++mutex;

}

// Driver Code

int main()

{

int n, i;

printf("\n1. Press 1 for Producer"

"\n2. Press 2 for Consumer"

"\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'

// can give wrong value due to

// synchronization issues.

// 'critical' specifies that code is

// executed by only one thread at a

// time i.e., only one thread enters

// the critical section at a given time

#pragma omp critical

for (i = 1; i > 0; i++) {

printf("\nEnter your choice:");

scanf("%d", &n);

// Switch Cases

switch (n) {

case 1:

// If mutex is 1 and empty

// is non-zero, then it is

// possible to produce

if ((mutex == 1)

&& (empty != 0)) {

producer();

}

// Otherwise, print buffer

// is full

else {

printf("Buffer is full!");

}

break;

case 2:

// If mutex is 1 and full

// is non-zero, then it is

// possible to consume

if ((mutex == 1)

&& (full != 0)) {

consumer();

}

// Otherwise, print Buffer

// is empty

else {

printf("Buffer is empty!");

}

break;

// Exit Condition

case 3:

exit(0);

break;

}

}

}

**Experiment 9: Deadlock Detection**

#include<stdio.h>

static int mark[20];

int i,j,np,nr;

int main()

{

int alloc[10][10],request[10][10],avail[10],r[10],w[10];

printf("\nEnter the no of process: ");

scanf("%d",&np);

printf("\nEnter the no of resources: ");

scanf("%d",&nr);

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

{

printf("\nTotal Amount of the Resource R%d: ",i+1);

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

}

printf("\nEnter the request matrix:");

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

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

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

printf("\nEnter the allocation matrix:");

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

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

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

/\*Available Resource calculation\*/

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

{

avail[j]=r[j];

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

{

avail[j]-=alloc[i][j];

}

}

//marking processes with zero allocation

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

{

int count=0;

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

{

if(alloc[i][j]==0)

count++;

else

break;

}

if(count==nr)

mark[i]=1;

}

// initialize W with avail

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

w[j]=avail[j];

//mark processes with request less than or equal to W

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

{

int canbeprocessed=0;

if(mark[i]!=1)

{

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

{

if(request[i][j]<=w[j])

canbeprocessed=1;

else

{

canbeprocessed=0;

break;

}

}

if(canbeprocessed)

{

mark[i]=1;

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

w[j]+=alloc[i][j];

}

}

}

//checking for unmarked processes

int deadlock=0;

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

if(mark[i]!=1)

deadlock=1;

if(deadlock)

printf("\n Deadlock detected");

else

printf("\n No Deadlock possible");

}

**Experiment 10: FIFO page replacement**

// C program for FIFO page replacement algorithm

#include <stdio.h>

int main()

{

int incomingStream[] = {4, 1, 2, 4, 5};

int pageFaults = 0;

int frames = 3;

int m, n, s, pages;

pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");

int temp[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(incomingStream[m] == temp[n])

{

s++;

pageFaults--;

}

}

pageFaults++;

if((pageFaults <= frames) && (s == 0))

{

temp[m] = incomingStream[m];

}

else if(s == 0)

{

temp[(pageFaults - 1) % frames] = incomingStream[m];

}

printf("\n");

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

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

{

if(temp[n] != -1)

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

else

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

}

}

printf("\nTotal Page Faults:\t%d\n", pageFaults);

return 0;

}

**Experiment 11: LRU page replacement**

#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;

}

**Experiment 12: Optimal Page Replacement**

#include<stdio.h>

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter page 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]){

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == -1){

faults++;

frames[j] = pages[i];

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

flag3 =0;

for(j = 0; j < no\_of\_frames; ++j){

temp[j] = -1;

for(k = i + 1; k < no\_of\_pages; ++k){

if(frames[j] == pages[k]){

temp[j] = k;

break;

}

}

}

for(j = 0; j < no\_of\_frames; ++j){

if(temp[j] == -1){

pos = j;

flag3 = 1;

break;

}

}

if(flag3 ==0){

max = temp[0];

pos = 0;

for(j = 1; j < no\_of\_frames; ++j){

if(temp[j] > max){

max = temp[j];

pos = j;

}

}

}

frames[pos] = pages[i];

faults++;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

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

}

}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}