**DR B R AMBEDKAR NATIONAL INSTITUE OF TECHNOLOGY JALANDHAR**

****

LAB FILE

OF

OPERATING SYSTEMS

CSX-325

SESSION: JULY-DEC 2019

**SUBMITTED TO: SUBMITTED BY:** Dr. D.K. Gupta ABHINENDRA GAURAV

CSE Dept. 17103002

GROUP: G1

CSE 3rd year

**PRACTICAL-1**

**AIM:** Using Linux Commands

* **ls** - The **ls** command provides a listing of a directory's contents. Its function is similar to the **dir** command in DOS.

The **-l** option generates a listing that shows permissions, owner and group membership, size, modification date, and the name for everything in the directory.

* **pwd** - The **pwd** command shows the current directory (**pwd** stands for “print working directory”). When you first log in to your account, the working directory is **/home/**username. This directory is your home directory, and is also represented by the tilde sign **~**.
* **cd** - To change to another directory, use the **cd** command.
* **find** - The **find** command looks for a file or set of files in a directory and all subdirectories beneath it.
* **mkdir** - To create a directory, use the **mkdir** command.
* **cp** - To copy a file, use the **cp** command. When you copy a file, you specify the source and destination locations.
* **mv** - To move a file, use the **mv** command. As with the **cp** command, you specify the source and destination locations.
* **rm** - To delete a file, use the **rm** command.
* **cat** - This will print out the entire contents of a document file to the terminal.
* **man** - It shows manual page of a command.
* **help** - It shows all commands with their basic info.
* **echo -** The echo command helps us move some data, usually text into a file.
* **sudo** - A widely used command in the Linux command line, sudo stands for "SuperUser Do". So, if you want any command to be done with administrative or root privileges, you can use the sudo command.
* **chmod -** Use chmod to make a file executable and to change the permission granted in Linux.

**PRACTICAL-2**

**Aim:** Introduction to shell scripting.

**Theory:**

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with .sh file extension eg. myscript.sh

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

* Shell Keywords – if, else, break etc.
* Shell commands – cd, ls, echo, pwd, touch etc.
* Functions
* Control flow – if..then..else, case and shell loops etc.

**PRACTICAL-3**

**Aim:** Write a program for first come first serve CPU scheduling.

**Theory:**

Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non-preemptive scheduling algorithm.

**Algorithm:**

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] .

1. Find turnaround time = waiting\_time + burst\_time for all processes.
2. Find average waiting time =
   * 1. total\_waiting\_time / no\_of\_processes.
3. Similarly, find average turnaround time =
   * 1. total\_turn\_around\_time / no\_of\_processes.

**Program:**

#include<iostream>

using namespace std;

void avgTime(int process[],int burstTime[],int n)

{

int waitTime[n];

int turnAroundTime[n];

double avg=0;

waitTime[0]=0;

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

{

waitTime[i]=burstTime[i-1]+waitTime[i-1];

avg+=waitTime[i];

}

cout<<"Average wait time is "<<avg/n<<endl;

avg=0;

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

{

turnAroundTime[i]=burstTime[i]+waitTime[i];

avg+=turnAroundTime[i];

}

cout<<"Average turn around time is "<<avg/n;

}

int main()

{

int n;

cout<<"Enter no of processes ";

cin>>n;

int process[n];

int burstTime[n];

cout<<"Enter processes ";

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

{

cin>>process[i];

}

cout<<"Enter burst Time ";

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

{

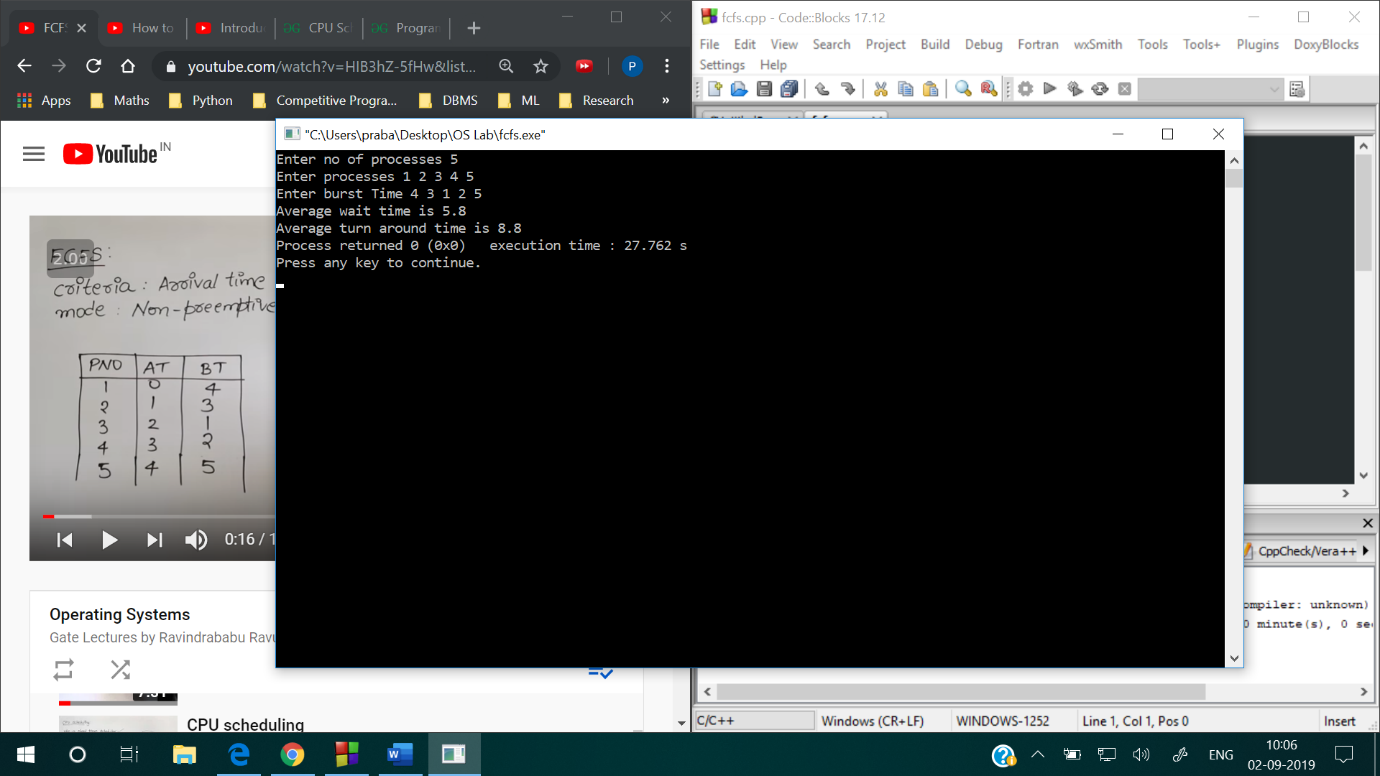
cin>>burstTime[i];

}

avgTime(process,burstTime,n);

return 0;

}

**Output:**

**PRACTICAL-4**

**Aim:** Write a program for shortest job first CPU scheduling.

**Theory:**

Process which have the shortest burst time are scheduled first.If two processes have the same bust time then FCFS is used to break the tie. It is a non-preemptive scheduling algorithm.

**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.

**Program:**

#include<iostream>

#include<queue>

using namespace std;

int main()

{

int n;

cout<<"Enter no of processes ";

cin>>n;

bool completed[n];

int arrivalTime[n];

int burstTime[n];

int completeTime[n]={0};

int turnAroundTime[n]={0};

int waitTime[n]={0};

cout<<"Enter arrival time ";

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

{

completed[i]=false;

cin>>arrivalTime[i];

}

cout<<"Enter burst time ";

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

{

cin>>burstTime[i];

}

int currentTime=arrivalTime[0];

queue<int>Q;

Q.push(0);

while(!Q.empty())

{

int job=Q.front();

Q.pop();

//process the job

if(currentTime< arrivalTime[job])

{

currentTime=arrivalTime[job];

}

currentTime+=burstTime[job];

completed[job]=true;

completeTime[job]=currentTime;

turnAroundTime[job]=completeTime[job]-arrivalTime[job];

waitTime[job]=turnAroundTime[job]-burstTime[job];

int minIndex=99999;

int minBurst=99999;

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

{

if(completed[i]==false && currentTime>=arrivalTime[i] && minBurst>burstTime[i])

{

minIndex=i;

minBurst=burstTime[i];

}

}

if(minIndex==99999)

{

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

{

if(!completed[i])

{

minIndex=i;

break;

}

}

}

if(minIndex==99999)

{

break;

}

Q.push(minIndex);

}

cout<<"wait Time ";

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

{

cout<<waitTime[i]<<" ";

}cout<<endl;

cout<<"turn around Time ";

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

{

cout<<turnAroundTime[i]<<" ";

}cout<<endl;

cout<<"completion Time ";

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

{

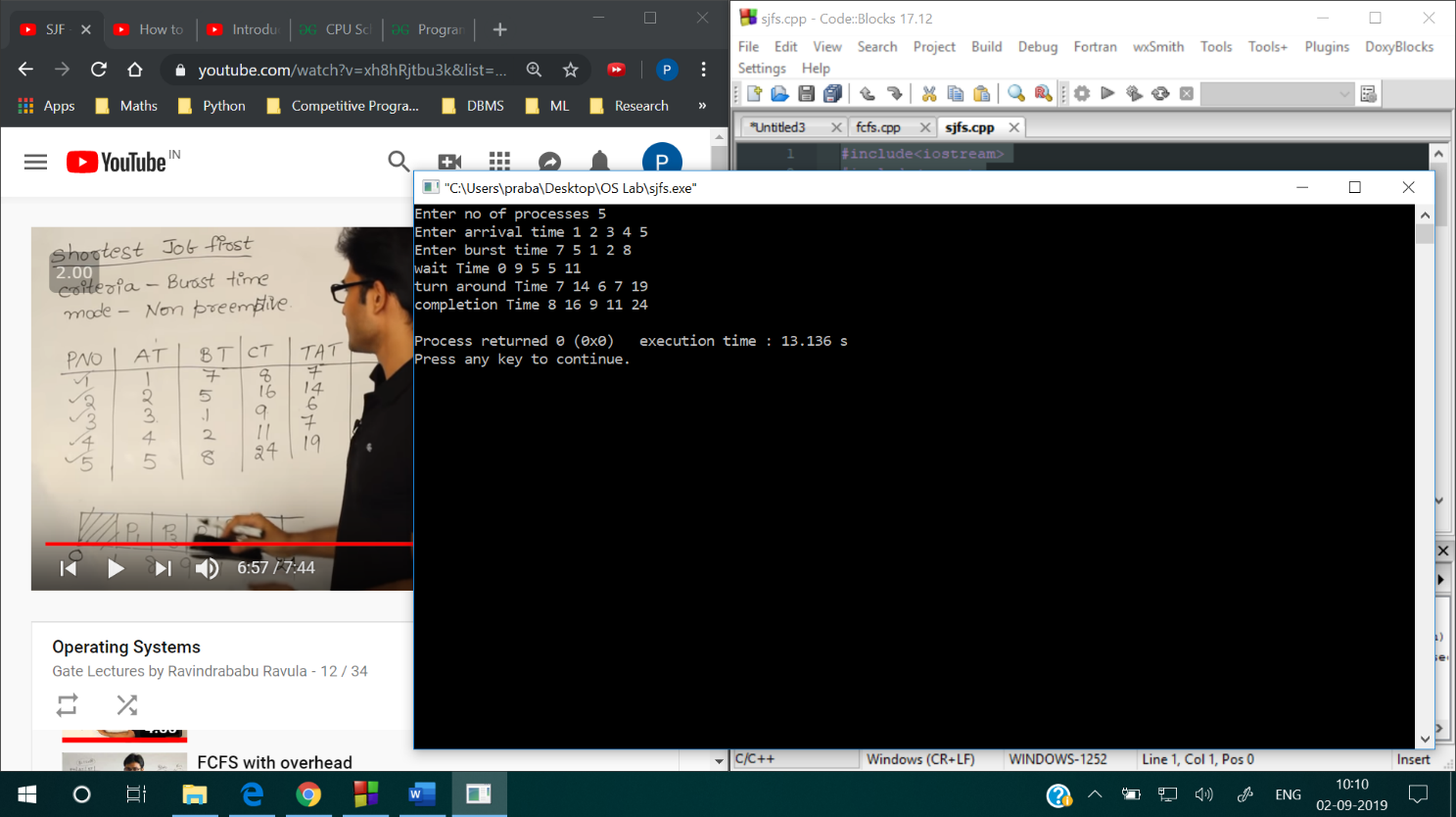
cout<<completeTime[i]<<" ";

}cout<<endl;

return 0;

}

**Output:**



**PRACTICAL-5**

**Aim:** Write a program for round robin CPU scheduling.

**Theory:**

Each process is assigned a fixed time(Time Quantum/Time Slice) in cyclic way.It is designed especially for the time-sharing system. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. To implement Round Robin scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

**Algorithm:**

1- Create an array rem\_bt[] to keep track of remaining

burst time of processes. This array is initially a

copy of bt[] (burst times array)

2- Create another array wt[] to store waiting times

of processes. Initialize this array as 0.

3- Initialize time : t = 0

4- Keep traversing the all processes while all processes

are not done. Do following for i'th process if it is

not done yet.

a- If rem\_bt[i] > quantum

(i) t = t + quantum

(ii) bt\_rem[i] -= quantum;

c- Else // Last cycle for this process

(i) t = t + bt\_rem[i];

(ii) wt[i] = t - bt[i]

(ii) bt\_rem[i] = 0; // This process is over

**Program:**

#include<iostream>

#include<queue>

using namespace std;

int main()

{

int n;

cout<<"Enter no of processes ";

cin>>n;

bool completed[n];

bool inQ[n];

int arrivalTime[n];

int burstTime[n],bT[n];

int completeTime[n]={0};

int turnAroundTime[n]={0};

int waitTime[n]={0};

cout<<"Enter arrival time ";

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

{

completed[i]=false;

inQ[i]=false;

cin>>arrivalTime[i];

}

cout<<"Enter burst time ";

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

{

cin>>burstTime[i];

bT[i]=burstTime[i];

}

int tq=2;

int currentTime=arrivalTime[0];

queue<int>Q;

Q.push(0);

inQ[0]=true;

while(!Q.empty())

{

int job=Q.front();

Q.pop();

//process the job

if(burstTime[job]>tq )

{

currentTime+=tq;

burstTime[job]-=tq;

}

else

{

currentTime+=burstTime[job];

completed[job]=true;

burstTime[job]=0;

}

if(burstTime[job]==0)

{

inQ[job]=false;

completed[job]=true;

completeTime[job]=currentTime;

turnAroundTime[job]=completeTime[job]-arrivalTime[job];

waitTime[job]=turnAroundTime[job]-bT[job];

}

bool check=false;

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

{

if(inQ[i]==false && completed[i]==false && arrivalTime[i] <= currentTime)

{

check=true;

Q.push(i);

inQ[i]=true;

}

}

if(check==false)

{

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

{

if(inQ[i]==false && completed[i]==false && arrivalTime[i] <= currentTime)

{

currentTime=arrivalTime[i];

check=true;

Q.push(i);

inQ[i]=true;

break;

}

}

}

if(burstTime[job]!=0)

{

inQ[job]=true;

Q.push(job);

}

}

cout<<"wait Time ";

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

{

cout<<waitTime[i]<<" ";

}cout<<endl;

cout<<"turn around Time ";

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

{

cout<<turnAroundTime[i]<<" ";

}cout<<endl;

cout<<"completion Time ";

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

{

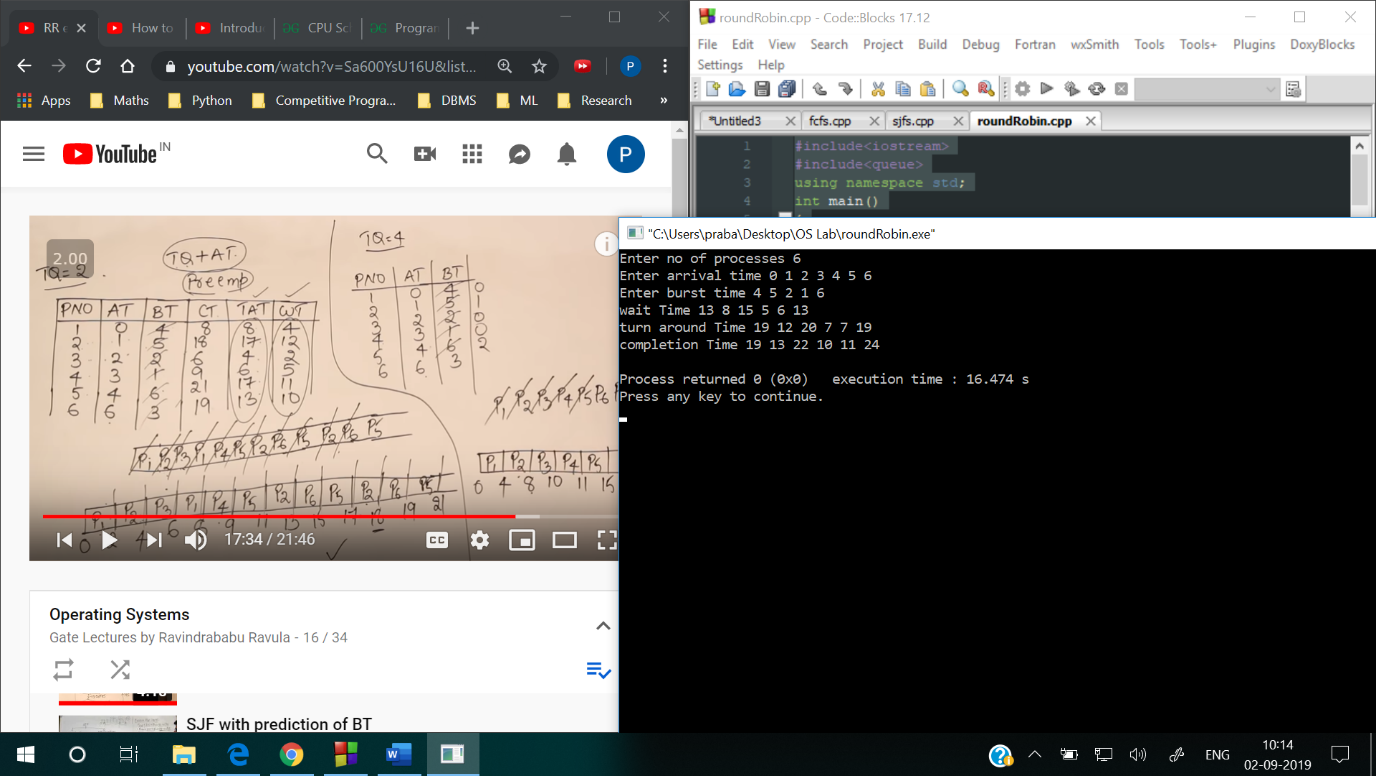
cout<<completeTime[i]<<" ";

}cout<<endl;

return 0;

}

**Output:**



**PRACTICAL-6**

**Aim:** Write a program for longest remaining job first CPU scheduling.

**Theory:**

It is preemptive mode of LJF algorithm in which we give priority to the process having largest burst time remaining.

**Algorithm:**

1. First, sort the processes in increasing order of their Arrival Time.
2. Choose the process having least arrival time but with most Burst Time. Then process it for 1 unit. Check if any other process arrives upto that time of execution or not.
3. Repeat the above both steps until execute all the processes.

**Program:**

#include<iostream>

#include<queue>

using namespace std;

int main()

{

int n;

cout<<"Enter no of processes ";

cin>>n;

bool completed[n];

int arrivalTime[n];

int burstTime[n];

int bt[n];

int completeTime[n]={0};

int turnAroundTime[n]={0};

int waitTime[n]={0};

cout<<"Enter arrival time ";

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

{

completed[i]=false;

cin>>arrivalTime[i];

}

cout<<"Enter burst time ";

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

{

cin>>burstTime[i];

bt[i]=burstTime[i];

}

int currentTime=arrivalTime[0];

queue<int>Q;

Q.push(0);

while(true)

{

int job=Q.front();

Q.pop();

currentTime++;

burstTime[job]--;

//process the job

if(burstTime[job]==0)

{

completed[job]=true;

completeTime[job]=currentTime;

turnAroundTime[job]=completeTime[job]-arrivalTime[job];

waitTime[job]=turnAroundTime[job]-bt[job];

}

// find the longest remaining

int mxIndex=-1,mxBurst=-1;

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

{

if(completed[i]==false && arrivalTime[i]<=currentTime && burstTime[i]>mxBurst)

{

mxBurst=burstTime[i];

mxIndex=i;

}

}

if(mxIndex==-1)

{

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

{

if(completed[i]==false)

{

currentTime=arrivalTime[i];

mxIndex=i;

break;

}

}

}

if(mxIndex==-1)

{

break;

}

Q.push(mxIndex);

}

cout<<"wait Time ";

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

{

cout<<waitTime[i]<<" ";

}cout<<endl;

cout<<"turn around Time ";

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

{

cout<<turnAroundTime[i]<<" ";

}cout<<endl;

cout<<"completion Time ";

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

{

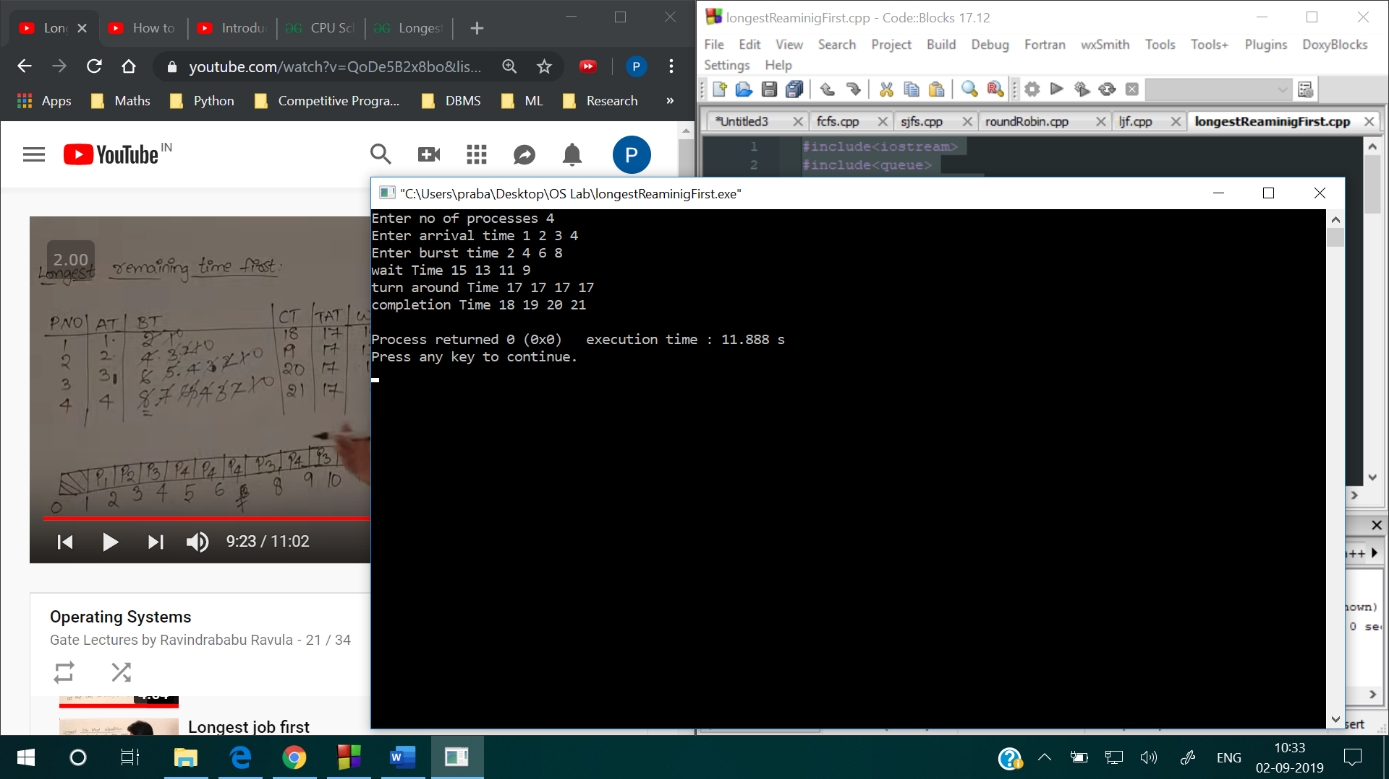
cout<<completeTime[i]<<" ";

}cout<<endl;

return 0;

}

**Output:**



**PRACTICAL-7**

**AIM:**Write as program to implement Priority based Non preemptive Scheduling.

**PROGRAM:**

#include<iostream>

using namespace std;

int main()

{

int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg\_wt,avg\_tat;

cout<<"Enter Total Number of Process:";

cin>>n;

cout<<"\nEnter Burst Time and Priority\n";

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

{

cout<<"\nP["<<i+1<<"]\n";

cout<<"Burst Time:";

cin>>bt[i];

cout<<"Priority:";

cin>>pr[i];

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

}

//sorting burst time, priority and process number in ascending order using selection sort

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

{

pos=i;

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

{

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

pos=j;

}

temp=pr[i];

pr[i]=pr[pos];

pr[pos]=temp;

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 is 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=total/n; //average waiting time

total=0;

cout<<"\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];

cout<<"\nP["<<p[i]<<"]\t\t "<<bt[i]<<"\t\t "<<wt[i]<<"\t\t\t"<<tat[i];

}

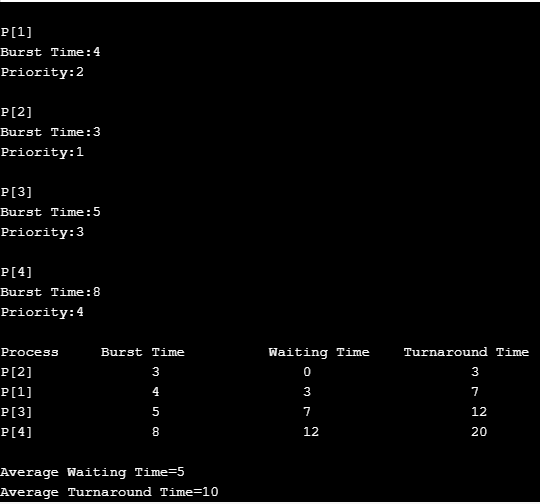
avg\_tat=total/n; //average turnaround time

cout<<"\n\nAverage Waiting Time="<<avg\_wt;

cout<<"\nAverage Turnaround Time="<<avg\_tat;

return 0;

}



**PRACTICAL-8**

Write a program to implement Priority based preemptive Scheduling.

#include <bits/stdc++.h>

using namespace std;

struct Process {

int processID;

int burstTime;

int tempburstTime;

int responsetime;

int arrivalTime;

int priority;

int outtime;

int intime;

};

void insert(Process Heap[], Process value, int\* heapsize,

int\* currentTime)

{

int start = \*heapsize, i;

Heap[\*heapsize] = value;

if (Heap[\*heapsize].intime == -1)

Heap[\*heapsize].intime = \*currentTime;

++(\*heapsize);

while (start != 0 && Heap[(start - 1) / 2].priority >

Heap[start].priority) {

Process temp = Heap[(start - 1) / 2];

Heap[(start - 1) / 2] = Heap[start];

Heap[start] = temp;

start = (start - 1) / 2;

}

}

void order(Process Heap[], int\* heapsize, int start)

{

int smallest = start;

int left = 2 \* start + 1;

int right = 2 \* start + 2;

if (left < \*heapsize && Heap[left].priority <

Heap[smallest].priority)

smallest = left;

if (right < \*heapsize && Heap[right].priority <

Heap[smallest].priority)

smallest = right;

if (smallest != start) {

Process temp = Heap[smallest];

Heap[smallest] = Heap[start];

Heap[start] = temp;

order(Heap, heapsize, smallest);

}

}

Process extractminimum(Process Heap[], int\* heapsize,

int\* currentTime)

{

Process min = Heap[0];

if (min.responsetime == -1)

min.responsetime = \*currentTime - min.arrivalTime;

--(\*heapsize);

if (\*heapsize >= 1) {

Heap[0] = Heap[\*heapsize];

order(Heap, heapsize, 0);

}

return min;

}

bool compare(Process p1, Process p2)

{

return (p1.arrivalTime < p2.arrivalTime);

}

void scheduling(Process Heap[], Process array[], int n,

int\* heapsize, int\* currentTime)

{

if (heapsize == 0)

return;

Process min = extractminimum(Heap, heapsize, currentTime);

min.outtime = \*currentTime + 1;

--min.burstTime;

printf("process id = %d current time = %d\n",

min.processID, \*currentTime);

if (min.burstTime > 0) {

insert(Heap, min, heapsize, currentTime);

return;

}

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

if (array[i].processID == min.processID) {

array[i] = min;

break;

}

}

void priority(Process array[], int n)

{

sort(array, array + n, compare);

int totalwaitingtime = 0, totalbursttime = 0,

totalturnaroundtime = 0, i, insertedprocess = 0,

heapsize = 0, currentTime = array[0].arrivalTime,

totalresponsetime = 0;

Process Heap[4 \* n];

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

totalbursttime += array[i].burstTime;

array[i].tempburstTime = array[i].burstTime;

}

do {

if (insertedprocess != n) {

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

if (array[i].arrivalTime == currentTime) {

++insertedprocess;

array[i].intime = -1;

array[i].responsetime = -1;

insert(Heap, array[i], &heapsize,&currentTime);

}

}

}

scheduling(Heap, array, n, &heapsize, &currentTime);

++currentTime;

if (heapsize == 0 && insertedprocess == n)

break;

} while (1);

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

totalresponsetime += array[i].responsetime;

totalwaitingtime += (array[i].outtime - array[i].intime -

array[i].tempburstTime);

totalbursttime += array[i].burstTime;

}

printf("Average waiting time = %f\n",

((float)totalwaitingtime / (float)n));

printf("Average response time =%f\n",

((float)totalresponsetime / (float)n));

printf("Average turn around time = %f\n",

((float)(totalwaitingtime + totalbursttime) / (float)n));

}

int main()

{

int n,i;

cout<<"enter number of processes : ";

cin>>n;

Process a[n];

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

{

cout<<"enter process names,arrival time and bus time : ";

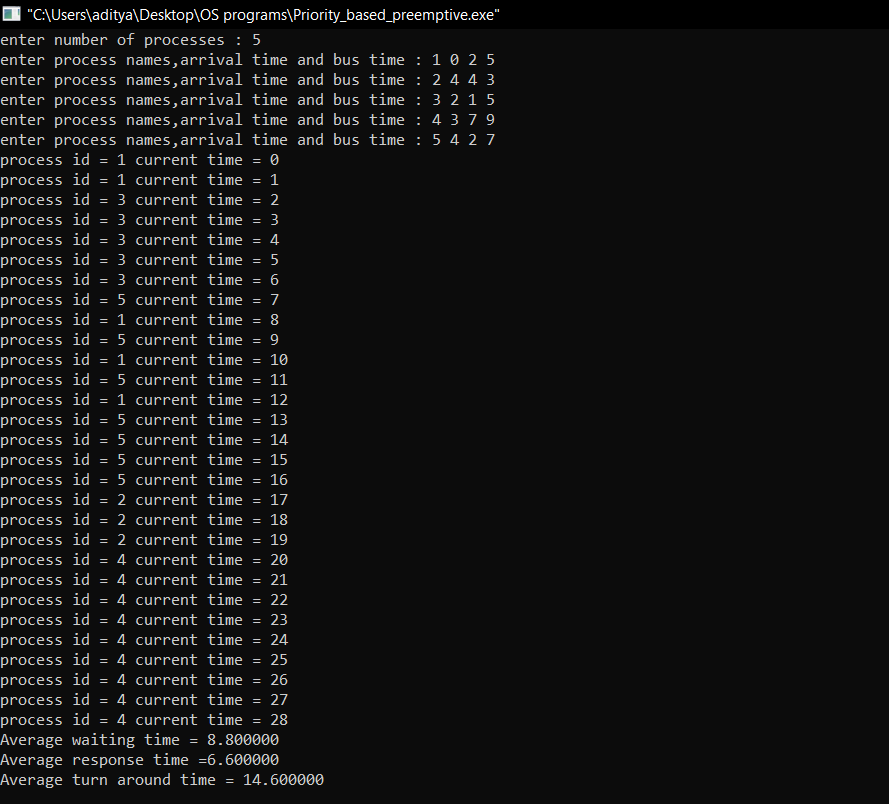
cin>>a[i].processID>>a[i].arrivalTime>>a[i].priority>>a[i].burstTime;

}

priority(a, 5);

return 0;

}



**PRACTICAL-9**

Write a program to implement Bankers Algorithm for Deadlock Prevention.

#include<iostream>

#include <stdbool.h>

using namespace std;

int main() {

int curr[5][5];

int max\_claim[5][5];

int avl[5];

int alloc[5] = {0, 0, 0, 0, 0};

int max\_res[5];

int running[5];

int i, j, exec, r, p;

int count = 0;

bool safe = false;

cout<<"\nEnter the number of resources: ";

cin>>r;

cout<<"\nEnter the number of processes: ";

cin>>p;

for (i = 0; i < p; i++) {

running[i] = 1;

count++;

}

cout<<"\nEnter Claim Vector: ";

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

cin>>max\_res[i];

cout<<"\nEnter Allocated Resource Table: ";

for (i = 0; i < p; i++) {

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

cin>>curr[i][j];

}

cout<<"\nEnter Maximum Claim table: ";

for (i = 0; i < p; i++) {

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

cin>>max\_claim[i][j];

}

cout<<"\nThe Claim Vector is: ";

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

cout<< max\_res[i];

cout<<"\nThe Allocated Resource Table:\n";

for (i = 0; i < p; i++) {

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

cout<<"\t"<< curr[i][j];

cout<<"\n";

}

cout<<"\nThe Maximum Claim Table:\n";

for (i = 0; i < p; i++) {

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

cout<<"\t"<< max\_claim[i][j];

cout<<"\n";

}

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

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

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

cout<<"\nAllocated resources: ";

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

cout<< alloc[i];

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

avl[i] = max\_res[i] - alloc[i];

cout<<"\nAvailable resources: ";

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

cout<< avl[i];

cout<<"\n";

while (count != 0) {

safe = false;

for (i = 0; i < p; i++) {

if (running[i]) {

exec = 1;

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

if (max\_claim[i][j] - curr[i][j] > avl[j]) {

exec = 0;

break;

}

}

if (exec) {

cout<<"\nProcess is executing.\n"<< i + 1;

running[i] = 0;

count--;

safe = true;

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

avl[j] += curr[i][j];

break;

}

}

}

if (!safe) {

cout<<"\nThe processes are in unsafe state.";

break;

}

if (safe)

cout<<"\nThe process is in safe state.";

cout<<"\nAvailable vector: ";

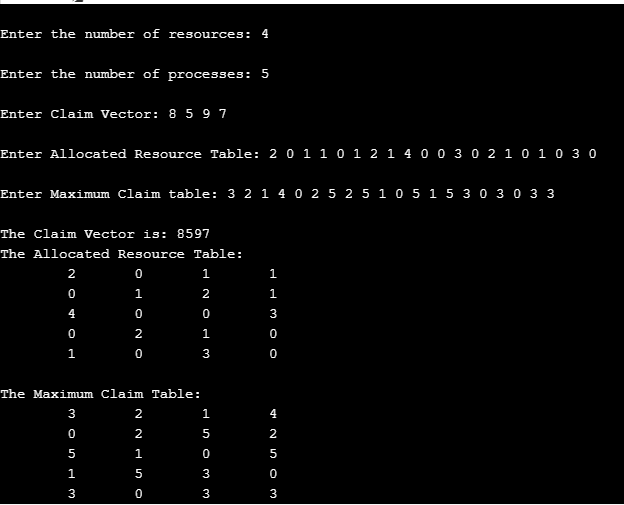
for (i = 0; i < r; i++)

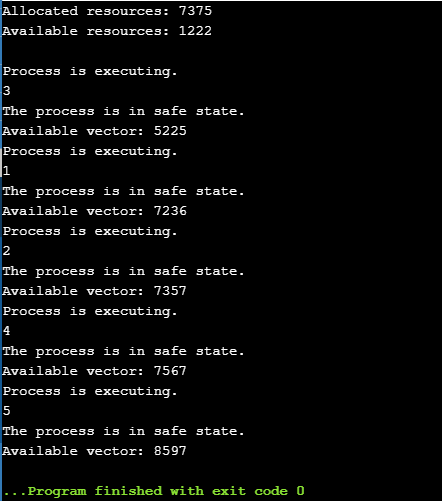
cout<<avl[i];

}

return 0;

}





**PRACTICAL-10**

Write a program to implement Producer Consumer problem With bounded Buffer.

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()

{

int n;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("\n1.Producer\n2.Consumer\n3.Exit");

while(1)

{

printf("\nEnter your choice:");

scanf("%d",&n);

switch(n)

{

case 1: if((mutex==1)&&(empty!=0))

producer();

else

printf("Buffer is full!!");

break;

case 2: if((mutex==1)&&(full!=0))

consumer();

else

printf("Buffer is empty!!");

break;

case 3:

exit(0);

break;

}

}

return 0;

}

int wait(int s)

{

return (--s);

}

int signal(int s)

{

return(++s);

}

void producer()

{

mutex=wait(mutex);

full=signal(full);

empty=wait(empty);

x++;

printf("\nProducer produces the item %d",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

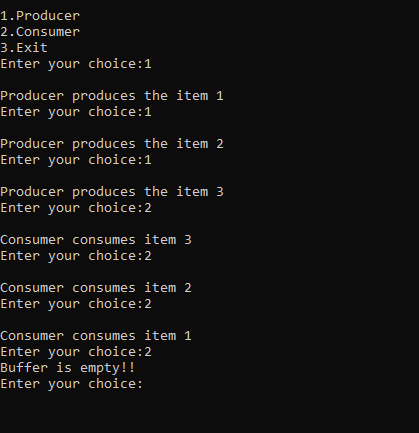
empty=signal(empty);

printf("\nConsumer consumes item %d",x);

x--;

mutex=signal(mutex);

}



**PRACTICAL-11**

. To implement Producer Consumer Problem with Unbounded Buffer.

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#define PROD\_DELAY 1000

#define CONS\_DELAY 1000

int buf\_items;

pthread\_mutex\_t mutex;

pthread\_cond\_t non\_zero\_item;

void\* producer(void\* arg)

{

while(1)

{

usleep(PROD\_DELAY);

pthread\_mutex\_lock(&mutex);

printf("Before producer production: %d\n", buf\_items);

++buf\_items;

printf("After producer production: %d\n", buf\_items);

pthread\_cond\_signal(&non\_zero\_item);

pthread\_mutex\_unlock(&mutex);

}

return NULL;

}

void\* consumer(void\* arg)

{

while(1)

{

usleep(CONS\_DELAY);

#ifdef BAD\_DESIGN

#endif

pthread\_mutex\_lock(&mutex);

while(buf\_items == 0)

pthread\_cond\_wait(&non\_zero\_item, &mutex);

printf("Before consumer consumption: %d\n",buf\_items);

--buf\_items;

printf("After consumer consumption: %d\n", buf\_items);

pthread\_mutex\_unlock(&mutex);

}

return NULL;

}

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

{

pthread\_t prod, cons;

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_create(&prod, &attr, producer, NULL);

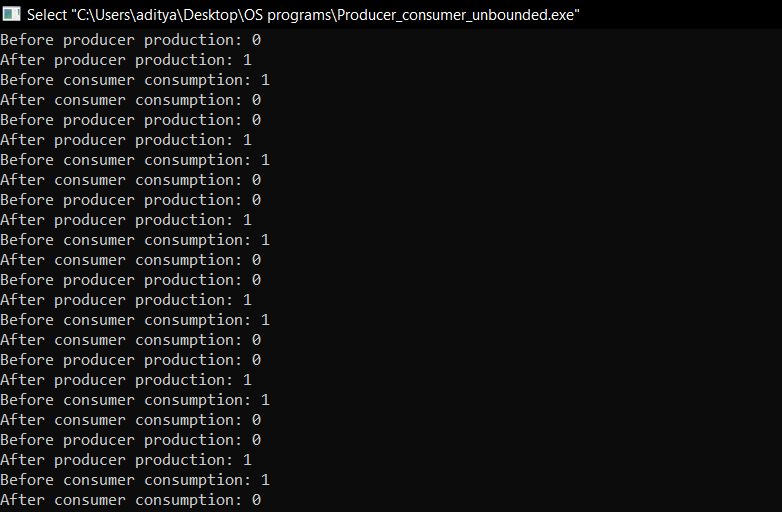
pthread\_create(&cons, &attr, consumer, NULL);

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

return 0;

}



**PRACTICAL-12**

To implement Reader Writer Problem.

#include "iostream"

#include <pthread.h>

#include <unistd.h>

using namespace std;

class monitor {

private:

int readers;

int writers;

int wait\_read;

int wait\_write;

pthread\_cond\_t can\_read;

pthread\_cond\_t canwrite;

pthread\_mutex\_t condlock;

public:

monitor() ///constructor

{

readers = 0;

writers = 0;

wait\_read = 0;

wait\_write = 0;

pthread\_cond\_init(&can\_read, NULL);

pthread\_cond\_init(&canwrite, NULL);

pthread\_mutex\_init(&condlock, NULL);

}

void beginread(int i)

{

pthread\_mutex\_lock(&condlock);

if (writers == 1 || wait\_write > 0) {

wait\_read++;

pthread\_cond\_wait(&can\_read, &condlock);

wait\_read--;

}

readers++;

cout << "reader " << i << " is reading\n";

pthread\_mutex\_unlock(&condlock);

pthread\_cond\_broadcast(&can\_read);

}

void endread(int i)

{

pthread\_mutex\_lock(&condlock);

if (--readers == 0)

pthread\_cond\_signal(&canwrite);

pthread\_mutex\_unlock(&condlock);

}

void beginwrite(int i)

{

pthread\_mutex\_lock(&condlock);

if (writers == 1 || readers > 0) {

++wait\_write;

pthread\_cond\_wait(&canwrite, &condlock);

--wait\_write;

}

writers = 1;

cout << "writer " << i << " is writing\n";

pthread\_mutex\_unlock(&condlock);

}

void endwrite(int i)

{

pthread\_mutex\_lock(&condlock);

writers = 0;

if (wait\_read > 0)

pthread\_cond\_signal(&can\_read);

else

pthread\_cond\_signal(&canwrite);

pthread\_mutex\_unlock(&condlock);

}

} M;

void\* reader(void\* id)

{

int c = 0;

int i = \*(int\*)id;

while (c < 3) {

usleep(1);

M.beginread(i);

M.endread(i);

c++;

}

}

void\* writer(void\* id)

{

int c = 0;

int i = \*(int\*)id;

while (c < 3) {

usleep(1000);

M.beginwrite(i);

M.endwrite(i);

c++;

}

}

int main()

{

pthread\_t r[3], w[3];

int id[3];

for (int i = 0; i < 3; i++) {

id[i] = i;

pthread\_create(&r[i], NULL, &reader, &id[i]);

pthread\_create(&w[i], NULL, &writer, &id[i]);

}

for (int i = 0; i < 3; i++) {

pthread\_join(r[i], NULL);

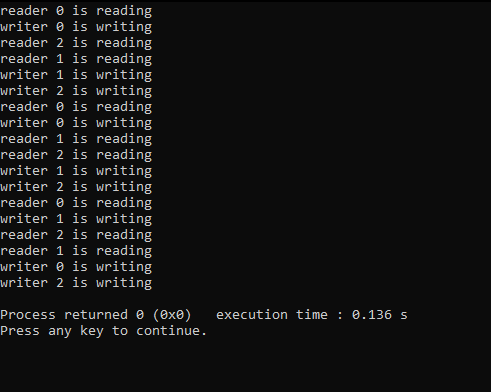
}

for (int i = 0; i < 3; i++) {

pthread\_join(w[i], NULL);

}

}



**PRACTICAL-13**

Write a program to implement Dinning Philosopher problem.

#include<iostream>

#define n 4

using namespace std;

int compltedPhilo = 0,i;

struct fork{

int taken;

}ForkAvil[n];

struct philosp{

int left;

int right;

}Philostatus[n];

void goForDinner(int philID){ //same like threads concept here cases implemented

if(Philostatus[philID].left==10 && Philostatus[philID].right==10)

cout<<"Philosopher "<<philID+1<<" completed his dinner\n";

//if already completed dinner

else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){

//if just taken two forks

cout<<"Philosopher "<<philID+1<<" completed his dinner\n";

Philostatus[philID].left = Philostatus[philID].right = 10; //remembering that he completed dinner by assigning value 10

int otherFork = philID-1;

if(otherFork== -1)

otherFork=(n-1);

ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0; //releasing forks

cout<<"Philosopher "<<philID+1<<" released fork "<<philID+1<<" and fork "<<otherFork+1<<"\n";

compltedPhilo++;

}

else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){ //left already taken, trying for right fork

if(philID==(n-1)){

if(ForkAvil[philID].taken==0){ //KEY POINT OF THIS PROBLEM, THAT LAST PHILOSOPHER TRYING IN reverse DIRECTION

ForkAvil[philID].taken = Philostatus[philID].right = 1;

cout<<"Fork "<<philID+1<<" taken by philosopher "<<philID+1<<"\n";

}else{

cout<<"Philosopher "<<philID+1<<" is waiting for fork "<<philID+1<<"\n";

}

}else{ //except last philosopher case

int dupphilID = philID;

philID-=1;

if(philID== -1)

philID=(n-1);

if(ForkAvil[philID].taken == 0){

ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;

cout<<"Fork "<<philID+1<<" taken by Philosopher "<<dupphilID+1<<"\n";

}else{

cout<<"Philosopher "<<dupphilID+1<<" is waiting for Fork "<<philID+1<<"\n";

}

}

}

else if(Philostatus[philID].left==0){ //nothing taken yet

if(philID==(n-1)){

if(ForkAvil[philID-1].taken==0){ //KEY POINT OF THIS PROBLEM, THAT LAST PHILOSOPHER TRYING IN reverse DIRECTION

ForkAvil[philID-1].taken = Philostatus[philID].left = 1;

cout<<"Fork "<<philID<<" taken by philosopher "<<philID+1<<"\n";

}else{

cout<<"Philosopher "<<philID+1<<" is waiting for fork "<<philID<<"\n";

}

}else{ //except last philosopher case

if(ForkAvil[philID].taken == 0){

ForkAvil[philID].taken = Philostatus[philID].left = 1;

cout<<"Fork "<<philID+1<<" taken by Philosopher "<<philID+1<<"\n";

}else{

cout<<"Philosopher "<<philID+1<<" is waiting for Fork "<<philID+1<<"\n";

}

}

}else{}

}

int main(){

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

ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n){

/\* Observe here carefully, while loop will run until all philosophers complete dinner

Actually problem of deadlock occur only thy try to take at same time

This for loop will say that they are trying at same time. And remaining status will print by go for dinner function

\*/

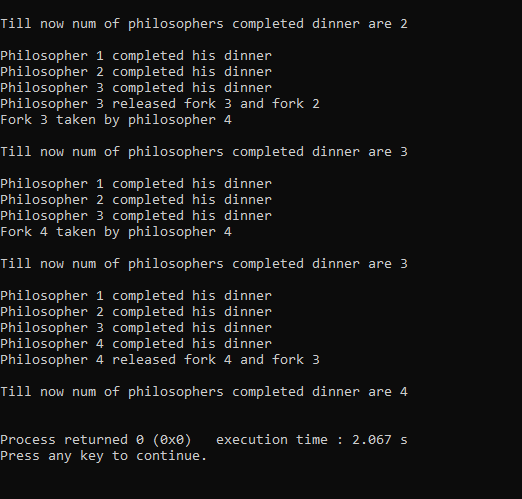
for(i=0;i<n;i++)

goForDinner(i);

cout<<"\nTill now num of philosophers completed dinner are "<<compltedPhilo<<"\n\n";

}

return 0;}



**PRACTICAL-14**

Write a program to implement reader writer problem using semaphores.

#include<semaphore.h>

#include<stdio.h>

#include<pthread.h>

# include<bits/stdc++.h>

using namespace std;

void \*reader(void \*);

void \*writer(void \*);

int readcount=0,writecount=0,sh\_var=5,bsize[5];

sem\_t x,y,z,rsem,wsem;

pthread\_t r[3],w[2];

void \*reader(void \*i)

{

cout << "\n-------------------------";

cout << "\n\n reader-" << i << " is reading";

sem\_wait(&z);

sem\_wait(&rsem);

sem\_wait(&x);

readcount++;

if(readcount==1)

sem\_wait(&wsem);

sem\_post(&x);

sem\_post(&rsem);

sem\_post(&z);

cout << "\nupdated value :" << sh\_var;

sem\_wait(&x);

readcount--;

if(readcount==0)

sem\_post(&wsem);

sem\_post(&x);

}

void \*writer(void \*i)

{

cout << "\n\n writer-" << i << "is writing";

sem\_wait(&y);

writecount++;

if(writecount==1)

sem\_wait(&rsem);

sem\_post(&y);

sem\_wait(&wsem);

sh\_var=sh\_var+5;

sem\_post(&wsem);

sem\_wait(&y);

writecount--;

if(writecount==0)

sem\_post(&rsem);

sem\_post(&y);}

int main()

{

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

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

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

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

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

pthread\_create(&r[0],NULL,(void \*)reader,(void \*)0);

pthread\_create(&w[0],NULL,(void \*)writer,(void \*)0);

pthread\_create(&r[1],NULL,(void \*)reader,(void \*)1);

pthread\_create(&r[2],NULL,(void \*)reader,(void \*)2);

pthread\_create(&r[3],NULL,(void \*)reader,(void \*)3);

pthread\_create(&w[1],NULL,(void \*)writer,(void \*)3);

pthread\_create(&r[4],NULL,(void \*)reader,(void \*)4);

pthread\_join(r[0],NULL);

pthread\_join(w[0],NULL);

pthread\_join(r[1],NULL);

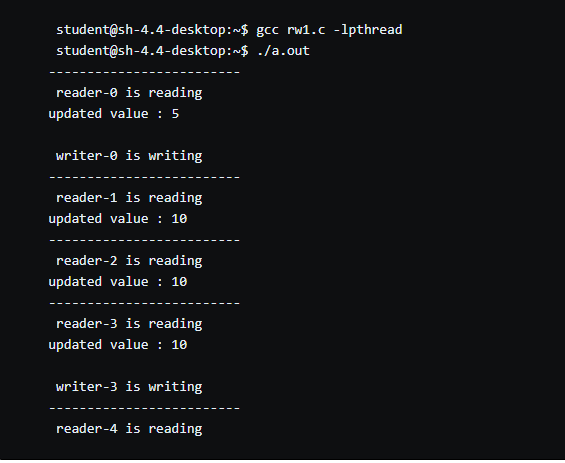
pthread\_join(r[2],NULL);

pthread\_join(r[3],NULL);

pthread\_join(w[1],NULL);

pthread\_join(r[4],NULL);

}



**PRACTICAL-15**

Write a program to implement Dinning Philosopher problem using semaphores.

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };

sem\_t mutex;

sem\_t S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY

&& state[LEFT] != EATING

&& state[RIGHT] != EATING) {

// state that eating

state[phnum] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n",

phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is Eating\n", phnum + 1);

// sem\_post(&S[phnum]) has no effect

// during takefork

// used to wake up hungry philosophers

// during putfork

sem\_post(&S[phnum]);

}

}

// take up chopsticks

void take\_fork(int phnum)

{

sem\_wait(&mutex);

// state that hungry

state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

// eat if neighbours are not eating

test(phnum);

sem\_post(&mutex);

// if unable to eat wait to be signalled

sem\_wait(&S[phnum]);

sleep(1);

}

// put down chopsticks

void put\_fork(int phnum)

{

sem\_wait(&mutex);

// state that thinking

state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n",

phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philospher(void\* num)

{ while (1) {

int\* i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

int main()

{ int i;

pthread\_t thread\_id[N];

// initialize the semaphores

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

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

sem\_init(&S[i], 0, 0);

for (i = 0; i < N; i++) {

pthread\_create(&thread\_id[i], NULL,

philospher, &phil[i]);

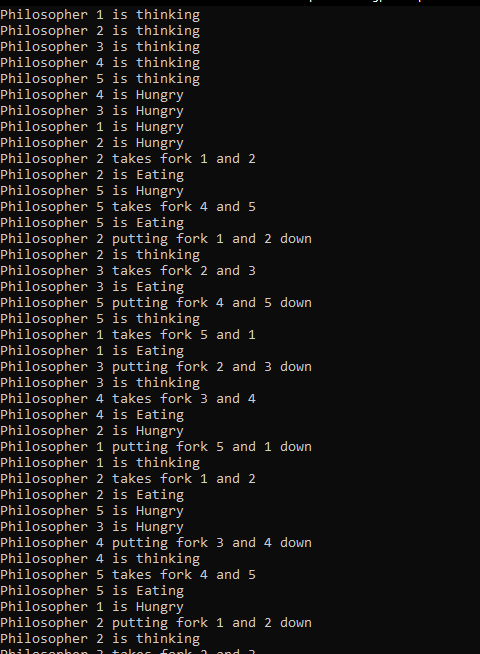
printf("Philosopher %d is thinking\n", i + 1);

}

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

pthread\_join(thread\_id[i], NULL);

}



**PRACTICAL**-**16**

**AIM:**

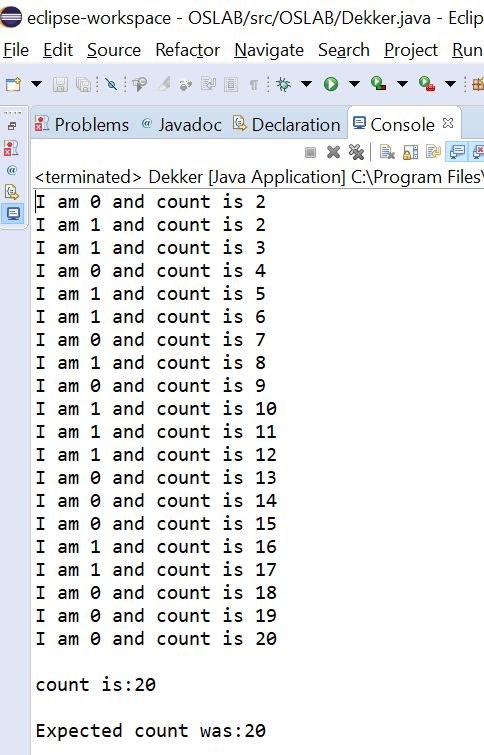
Implement Dekker’s algorithm for mutual exclusion in any programming Language

**Description:**

Dekker’s algorithm was the first provably-correct solution to the critical section problem. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

**Program: -**

**public class** Dekker **extends** Thread{  
  
 **public int thread\_id**;  
 **public static volatile int** *turn*;  
 **public static final int *countToThis***=10;  
 **public static final int *numberOfThreads***=2;  
 **public static volatile int** *count*=0;  
 **public static volatile boolean**[] *flag*=**new boolean**[***numberOfThreads***];  
  
  
 **public** Dekker(**int** id){  
 **thread\_id**=id;  
 }  
  
 **public void** run(){  
  
 **int** scale=10;  
 **for**(**int** i=0;i<***countToThis***;++i){  
 *count*++;  
 *flag*[**this**.**thread\_id**]=**true**;  
 **while**(*flag*[1-**this**.**thread\_id**]){  
 **if**(*turn*==1-**thread\_id**){  
 *flag*[**this**.**thread\_id**]=**false**;  
 **while**(*turn*==1-**thread\_id**){}  
 *flag*[**this**.**thread\_id**]=**true**;  
 }  
 }  
 System.***out***.println(**"I am "**+**thread\_id**+**" and count is "**+*count*);  
 *turn*=1-**thread\_id**;  
 *flag*[**this**.**thread\_id**]=**false**;  
 **try**{  
 *sleep*((**int**) (Math.*random*() \* scale));  
 }  
 **catch** (InterruptedException e){  
  
 }  
  
 }  
 }  
 **public static void** main(String[] args){  
 Dekker[] threads=**new** Dekker[***numberOfThreads***];  
 **for**(**int** i=0;i<threads.**length**;++i){  
 threads[i]=**new** Dekker(i);  
 threads[i].start();  
 }  
 **for**(**int** i=0;i<threads.**length**;++i){  
 **try**{  
 threads[i].join();  
 }  
 **catch** (InterruptedException e){  
 e.printStackTrace();  
 }  
 }  
 System.***out***.println(**"\ncount is:"**+*count*);  
 System.***out***.println(**"\nExpected count was:"**+(***countToThis***\****numberOfThreads***));  
 }  
}



**PRACTICAL-17**

**AIM:** Implement Lamport Bakery algorithm for mutual exclusion .

**Description: -**

The **Bakery algorithm** is one of the simplest known solutions to the mutual exclusion problem for the general case of N process. Bakery Algorithm is a critical section solution for **N** processes. The algorithm preserves the first come first serve property.

**PROGRAM :**

package OSLAB;

public class Bakery extends Thread {

public int thread\_id;

public static final int countToThis = 4;

public static final int numberOfThreads = 5;

public static volatile int count = 0;

private static volatile boolean[] choosing = new boolean[numberOfThreads];

private static volatile int[] ticket = new int[numberOfThreads];

public Bakery(int id) {

thread\_id = id;

}

public void run() {

int scale = 2;

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

lock(thread\_id);

count = count + 1;

System.out.println("I am " + thread\_id + " and count is: " + count);

try {

sleep((int) (Math.random() \* scale));

} catch (InterruptedException e) { }

unlock(thread\_id);

}

}

public void lock(int id) {

choosing[id] = true;

ticket[id] = findMax() + 1;

choosing[id] = false;

for (int j = 0; j < numberOfThreads; j++) {

if (j == id)

continue;

while (choosing[j]) { }

while (ticket[j] != 0 && (ticket[id] > ticket[j] || (ticket[id] == ticket[j] && id > j))) { /\* nothing \*/ }

}

}

private void unlock(int id) {

ticket[id] = 0;

}

private int findMax() {

int m = ticket[0];

for (int i = 1; i < ticket.length; i++) {

if (ticket[i] > m)

m = ticket[i];

}

return m;

}

public static void main(String[] args) {

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

choosing[i] = false;

ticket[i] = 0;

}

Bakery[] threads = new Bakery[numberOfThreads];

for (int i = 0; i < threads.length; i++) {

threads[i] = new Bakery(i);

threads[i].start();

}

for (int i = 0; i < threads.length; i++) {

try {

threads[i].join();

} catch (InterruptedException e) {

e.printStackTrace();

}

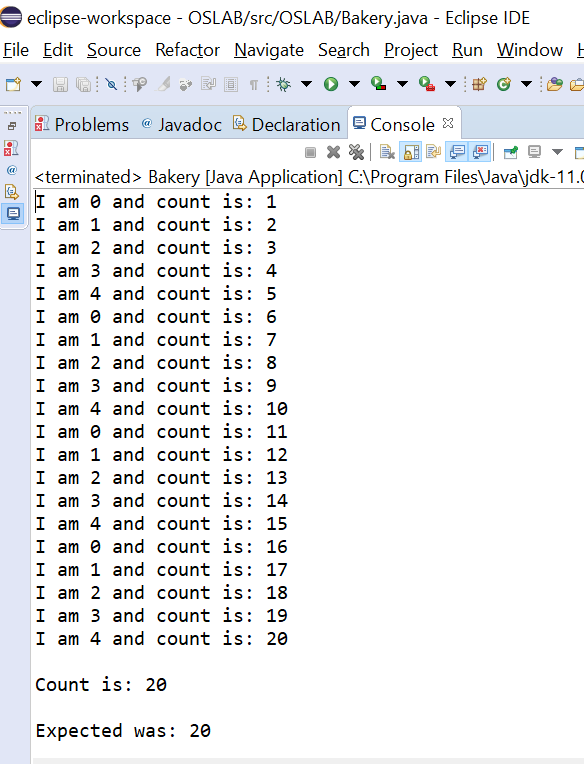
}

System.out.println("\nCount is: " + count);

System.out.println("\nExpected was: " + (countToThis \* numberOfThreads));

}

}



**PRACTICAL-18**

To implement FIFO page replacepment policy.

#include<iostream>

using namespace std;

class fifo\_alg

{

char ref[100];

int frame,fault,front,rear;

char \*cir\_que;

public:

fifo\_alg()

{

front=rear=-1;

fault=0;

}

void getdata();

void page\_fault();

};

void fifo\_alg::getdata()

{

cout<<"Enter Page reference string : ";

cin.getline(ref,50);

cout<<"Enter no. of frames : ";

cin>>frame;

}

void fifo\_alg::page\_fault()

{

cir\_que=new char[frame];

int flag=0;

for(int i=0;ref[i]!=0;i++)

{

if(ref[i]==' ')

continue;

if(front==-1)

{

cir\_que[0]=ref[i];

front=rear=0;

fault++;

}

else

{

for(int y=front%frame;y!=rear;y=(y+1)%frame)

if(cir\_que[y]==ref[i])

{

flag=1;

break;

}

if(cir\_que[rear]==ref[i])

flag=1;

if(flag==0)

{

if((rear+1)%frame==front)

front=(front+1)%frame;

rear=(rear+1)%frame;

cir\_que[rear]=ref[i];

fault++;

}

flag=0;

}

}

cout<<"Number of page faults : "<<fault;

}

int main()

{

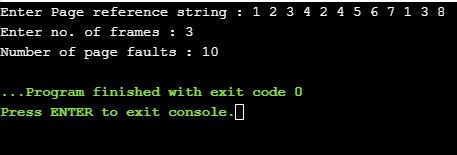
fifo\_alg page;

page.getdata();

page.page\_fault();

return 0;

}



**PRACTICAL-19**

To implement LRU page replacement policy.

#include<iostream>

using namespace std;

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;

cout<<"Enter number of frames: ";

cin>>no\_of\_frames;

cout<<"Enter number of pages: ";

cin>>no\_of\_pages;

cout<<"Enter reference string: ";

for(i = 0; i < no\_of\_pages; ++i){

cin>>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;

}

cout<<"\n";

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

cout<<"\t"<<frames[j];

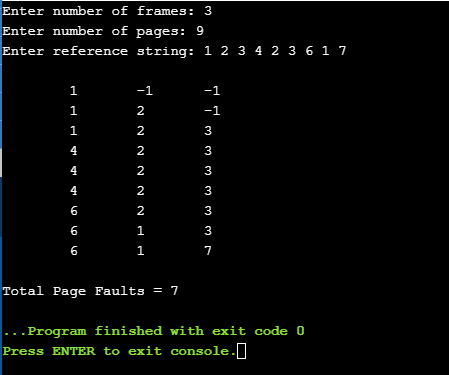
}

}

cout<<"\n\nTotal Page Faults = "<< faults;

return 0;

}



**PRACTICAL-20**

Write a program to implement optimal page replacement policy.

#include<iostream>

using namespace std;

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;

cout<<"Enter number of frames: ";

cin>>no\_of\_frames;

cout<<"Enter number of pages: ";

cin>>no\_of\_pages;

cout<<"Enter page reference string: ";

for(i = 0; i < no\_of\_pages; ++i){

cin>>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++;

}

cout<<"\n";

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

cout<<"\t"<< frames[j];

}

}

cout<<"\n\nTotal Page Faults = "<<faults;

return 0;}

