**DR B.R. AMBEDKAR NATIONAL INSTITUTE OF**

**TECHNOLOGY JALANDHAR,**

**PUNJAB, INDIA**



**Lab File**

**of**

**Advanced Operating System Laboratory**

**CSX-352**

**Session: Jan-May 2020**

**SUBMITTED BY-**

Name – Ankit Goyal

Roll no - 17103011

Group - G-1

Branch - CSE

**SUBMITTED TO-**

Dr. Prashant Kumar

Assistant Professor

CSE Department

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **PROGRAM NAME** | **PAGE No.** | **REMARKS** |
| 1. | LRU Page Replacement Algorithm | 3-4 |  |
| 2. | Optimal Page Replacement Policy | 5-6 |  |
| 3. | Producer Consumer Problem | 7-8 |  |
| 4. | Reader Writer Problem | 9-11 |  |
| 5. | Dining Philosopher Problem | 12-14 |  |
| 6. | Lamport’s Clock Algorithm | 15-16 |  |
| 7. | Vector Clock Algorithm | 17-19 |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Practical No. 1**

**Aim:** To implement LRU Page Replacement Algorithm.

**Description:** In this algorithm page will be replaced which is least recently used. LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too.

**Program:**

#include<bits/stdc++.h>

using namespace std;

int pageFaults(int pages[], int n, int capacity)

{

set<int> s;

map<int, int> indexes;

int page\_faults = 0;

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

{

if (s.size() < capacity)

{

if (s.find(pages[i])==s.end())

{

s.insert(pages[i]);

page\_faults++;

}

indexes[pages[i]] = i;

}

else

{

if (s.find(pages[i]) == s.end())

{

int lru = INT\_MAX, val;

set<int>::iterator it;

for ( it=s.begin(); it!=s.end(); it++)

{

if (indexes[\*it] < lru)

{

lru = indexes[\*it];

val = \*it;

}

}

s.erase(val);

s.insert(pages[i]);

page\_faults++;

}

indexes[pages[i]] = i;

}

}

return page\_faults;

}

int main()

{

int n;

cout<<"enter no. of pages: ";

cin>>n;

int pages[n];

cout<<"enter page numbers\n";

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

cin>>pages[i];

int capacity;

cout<<"enter capacity: ";

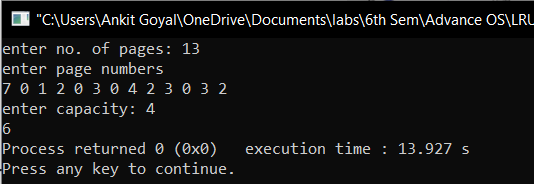
cin>>capacity;

cout << pageFaults(pages, n, capacity);

return 0;

}

**Output:**



**Practical no. 2**

**Aim:** To implement Optimal Page Replacement Policy.

**Description:** In this algorithm, when a page needs to be swapped in, the [operating system](https://en.wikipedia.org/wiki/Operating_system) swaps out the page whose next use will occur farthest in the future.

**Program:**

#include <bits/stdc++.h>

using namespace std;

bool search (int key, vector<int>& fr)

{

for (int i = 0; i < fr.size(); i++)

if (fr[i] == key)

return true;

return false;

}

int predict (int pg[], vector<int>& fr, int pn, int index)

{

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 (j == pn)

return i;

}

return (res == -1) ? 0 : res;

}

void optimalPage(int pg[], int pn, int fn)

{

vector<int> fr;

int hit = 0;

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

if (search(pg[i], fr)) {

hit++;

continue;

}

if (fr.size() < fn)

fr.push\_back(pg[i]);

else {

int j = predict(pg, fr, pn, i + 1);

fr[j] = pg[i];

}

}

cout << "No. of page faults = " << pn - hit << endl;

}

int main()

{

int n;

cout<<"enter no. of pages: ";

cin>>n;

int pages[n];

cout<<"enter page numbers\n";

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

cin>>pages[i];

int capacity;

cout<<"enter capacity: ";

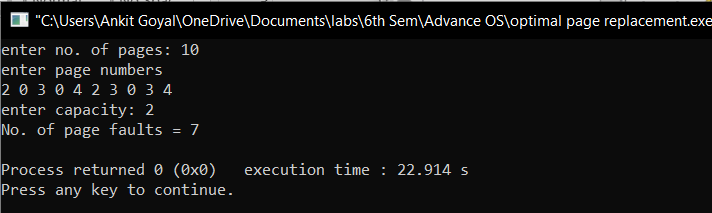
cin>>capacity;

optimalPage(pages, n, capacity);

return 0;

}

**Output:**



**Practical no. 3**

**Aim:** Write a program to illustrate solution for Producer Consumer Problem using two producer and two consumer processes which are sharing a common stack.

**Description:** In this problem we have a buffer of fixed size. Producer can produce an item and can place in the buffer. Consumer can pick items and can consume them. We need to ensure that when producer is placing an item in the buffer, then at the same time consumer should not consume any item.

**Program:**

#include<iostream>

#include<stdlib.h>

using namespace std;

int wait(int sem){

return --sem;

}

int signal(int sem){

return ++sem;

}

int no=0,empty=8,full=0,mutex=1;

void consumer(int n){

mutex=wait(mutex);

full=wait(full);

cout<<"Consumer "<<n<<" consumes item "<<no<<endl;

no--;

empty=signal(empty);

mutex=signal(mutex);

}

void producer(int n){

mutex=wait(mutex);

empty=wait(empty);

no++;

full=signal(full);

cout<<"Producer "<<n<<" produces the item "<<no<<endl;

mutex=signal(mutex);

}

int main(){

int n;

cout<<"press 1 for Producer 1\npress 2 for Producer 2\npress 3 for Consumer 1\npress 4 for Consumer 2\npress 5 for Exit\n Enter your choice\n ";

while(1){

cin>>n;

switch(n)

{

case 1:

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

producer(1);

else

cout<<"Buffer is full. you can not produce!!\n";

break;

case 2:

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

producer(2);

else

cout<<"Buffer is full. you can not produce!!\n";

break;

case 3:

if((mutex==1)&&(full!=0))

consumer(1);

else

cout<<"Buffer is empty. you can not consume!!\n";

break;

case 4:

if((mutex==1)&&(full!=0))

consumer(2);

else

cout<<"Buffer is empty. you can not consume!!\n";

break;

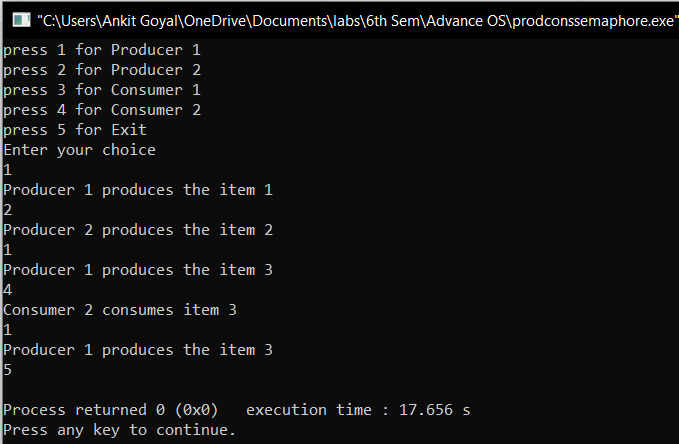
case 5:

exit(0);

}

}}

**Output:**



**Practical no. 4**

**Aim:** Write a program to illustrate monitor solution to implement the writer preference Reader Writer Problem.

**Description:** The readers-writer 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.

**Program:**

#include <iostream>

#include <pthread.h>

#include <unistd.h>

using namespace std;

class monitor {

private:

int rcnt, wcnt, waitr, waitw;

pthread\_cond\_t canread, canwrite;

pthread\_mutex\_t condlock;

public:

monitor()

{

rcnt = wcnt = waitr = waitw = 0;

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

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

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

}

void beginread(int i)

{

pthread\_mutex\_lock(&condlock);

if (wcnt == 1 || waitw > 0) {

cout<<"Reader "<<i<<" is waiting \n";

waitr++;

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

waitr--;

}

rcnt++;

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

pthread\_mutex\_unlock(&condlock);

pthread\_cond\_broadcast(&canread);

}

void endread(int i)

{

pthread\_mutex\_lock(&condlock);

if (--rcnt == 0)

pthread\_cond\_signal(&canwrite);

pthread\_mutex\_unlock(&condlock);

}

void beginwrite(int i)

{

pthread\_mutex\_lock(&condlock);

if (wcnt == 1 || rcnt > 0) {

cout<<"Writer "<<i<<" is waiting\n";

++waitw;

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

--waitw;

}

wcnt = 1;

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

pthread\_mutex\_unlock(&condlock);

}

void endwrite(int i){

pthread\_mutex\_lock(&condlock);

wcnt = 0;

if (waitw > 0)

pthread\_cond\_signal(&canwrite);

else

pthread\_cond\_signal(&canread);

pthread\_mutex\_unlock(&condlock);

}

} M;

void\* reader(void\* id)

{

int i = \*(int\*)id,c = 0;

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(1);

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++)

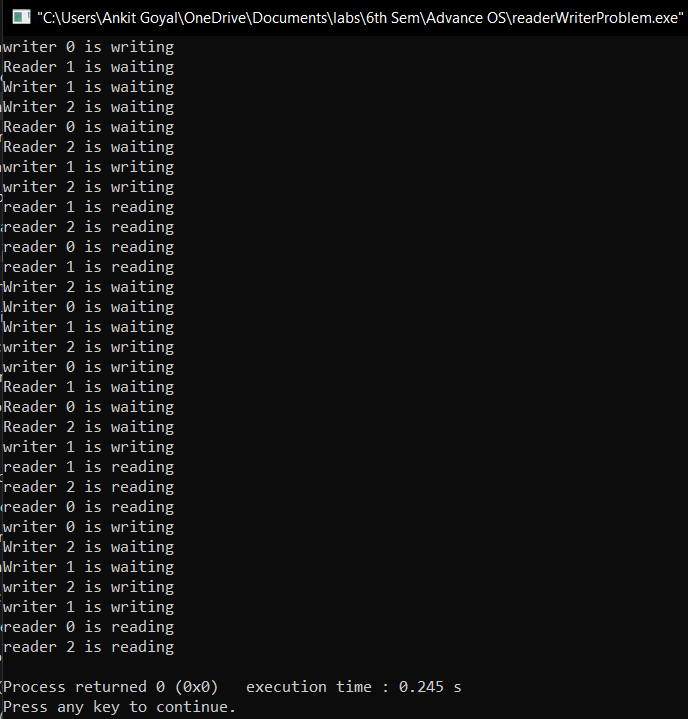
pthread\_join(r[i],NULL);

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

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

}

**Output:**



**Practical no. 5**

**Aim:** Write a program to illustrate solution for Dining Philosopher Problem.

**Description:** 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.

**Program:**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <bits/stdc++.h>

#include <windows.h>

#include <unistd.h>

#define N 3

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

using namespace std;

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

sem\_t mutex,S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)

{

state[phnum] = EATING;

usleep(2);

cout<<"Philosopher "<<phnum + 1<<" is eating\n";

sem\_post(&S[phnum]);

}

}

void take\_fork(int phnum)

{

sem\_wait(&mutex);

state[phnum] = HUNGRY;

cout<<"Philosopher "<<phnum + 1<<" is hungry\n";

test(phnum);

sem\_post(&mutex);

sem\_wait(&S[phnum]);

usleep(1);

}

void put\_fork(int phnum)

{

sem\_wait(&mutex);

state[phnum] = THINKING;

cout<<"Philosopher "<<phnum + 1<<" is thinking\n";

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philospher(void\* num)

{

int x=0;

while (x<3)

{

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

usleep(1);

take\_fork(\*i);

usleep(0);

put\_fork(\*i);

x++;

}

}

int main()

{

int i;

pthread\_t thread\_id[N];

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]);

cout<<"Philosopher "<<i+1<<" is thinking\n";

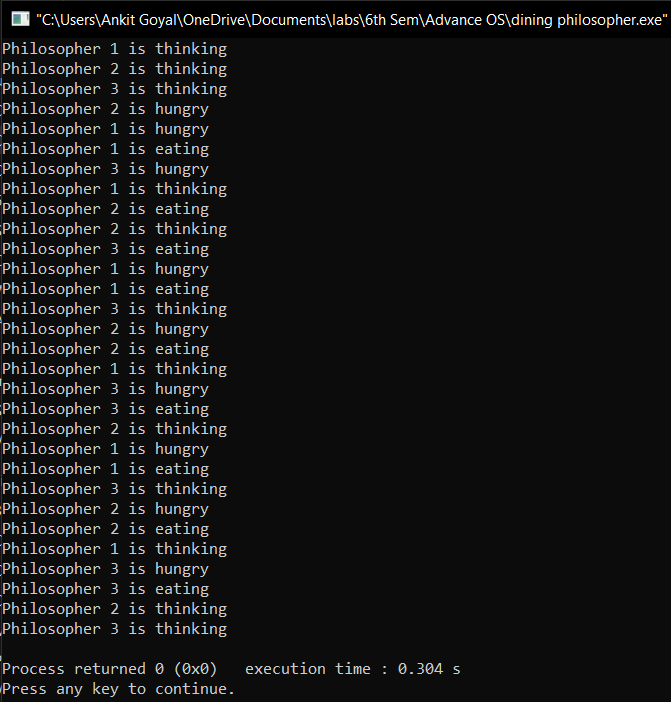
}

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

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

}

**Output:**



**Practical no. 6**

**Aim:** Write a program to illustrate Lamport’s Clock Algorithm.

**Description:** This algorithm is used to determine the order of events in a [distributed computer system](https://en.wikipedia.org/wiki/Distributed_computer_system). As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a [partial ordering](https://en.wikipedia.org/wiki/Partially_ordered_set) of events with minimal overhead

**Program:**

#include<bits/stdc++.h>

using namespace std;

int max1(int a, int b){

if (a>b)

return a;

return b;

}

int main(){

int i,j,k,p1[20],p2[20],e1,e2,dep[20][20];

cout<<"enter the events : ";

cin>>e1>>e2;

for(i=0;i<e1;i++)

p1[i]=i+1;

for(i=0;i<e2;i++)

p2[i]=i+1;

cout<<"enter the dependency matrix:"<<endl;

cout<<"\t enter 1 if e1->e2 \n\t enter -1, if e2->e1 \n\t else enter 0 \n"<<endl;

cout<<"\t";

for(i=0;i<e2;i++)

cout<<"e2"<<i+1<<"\t";

cout<<endl;

for(i=0;i<e1;i++){

cout<<"e1"<<i+1<<"\t";

for(j=0;j<e2;j++)

cin>>dep[i][j];

}

for(i=0;i<e1;i++){

for(j=0;j<e2;j++){

if(dep[i][j]==1){

p2[j]=max1(p2[j],p1[i]+1);

for(k=j;k<e2;k++)

p2[k+1]=p2[k]+1;

}

if(dep[i][j]==-1){

p1[i]=max1(p1[i],p2[j]+1);

for(k=i;k<e1;k++)

p2[k+1]=p1[k]+1;

}

}

}

cout<<"P1 : ";

for(i=0;i<e1;i++){

cout<<p1[i]<<" ";

}

cout<<endl;

cout<<"P2 : ";

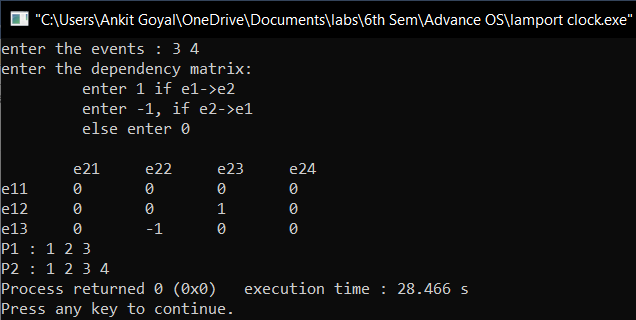
for(j=0;j<e2;j++)

cout<<p2[j]<<" ";

return 0 ;

}

**Output:**



**Practical no. 7**

**Aim:** Write a program to illustrate Vector Clock Algorithm.

**Description:** A vector clock is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for generating a [partial ordering](https://en.wikipedia.org/wiki/Partial_ordering) of events in a [distributed system](https://en.wikipedia.org/wiki/Distributed_system) and detecting [causality](https://en.wikipedia.org/wiki/Causality) violations. Just as in [Lamport timestamps](https://en.wikipedia.org/wiki/Lamport_timestamps" \o "Lamport timestamps), interprocess messages contain the state of the sending process's [logical clock](https://en.wikipedia.org/wiki/Logical_clock).

**Program:**

#include<iostream>

#include<conio.h>

#define SIZE 10

using namespace std;

class node {

public:

int data[SIZE];

node \*next;

node()

{

for(int p=0; p<SIZE; p++)

data[p] = 0;

next = NULL;

}

node(int v[], int n1)

{

for(int s = 0; s < n1; s++)

data[s] = v[s];

next = NULL;

}

friend class process;

}\*start=NULL;

int main()

{

int n, events, sent, receive, sentE, recE, commLines = 0;

node \*temp;

node \*proc[SIZE]; //array of processes

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

cin>>n;

int arr[n] = {0}; //representation of data

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

{ //number of processes

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

{

arr[v] = 0;

}

cout<<"Enter no. of events in process "<<i+1<<": ";

cin>>events;

for(int j = 1; j <= events; j++)

{

arr[i] = j;

node \*newnode = new node(arr,n);

if(start == NULL)

{

start = newnode;

temp = start;

}

else

{

temp->next = newnode;

temp = temp->next;

}

}

proc[i] = start;

start = NULL;

}

cout<<"\nEnter the number of communication lines: \n";

cin>>commLines;

node \*tempS, \*tempR;

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

{

cout<<"Enter sending process,sending event,receiving process,receiving event \n";

cin>>sent>>sentE>>receive>>recE;

tempS = proc[sent - 1];

tempR = proc[receive - 1];

for(int j = 1; j < sentE; j++)

tempS = tempS->next;

node \*preRecNode=NULL;

for(int k = 1; k < recE; k++)

{

preRecNode=tempR;

tempR = tempR->next;

}

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

tempR->data[j] = (tempR->data[j] < tempS->data[j]) ? tempS->data[j] : tempR->data[j];

while(tempR->next!=NULL)

{

preRecNode=tempR;

tempR=tempR->next;

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

{

if(preRecNode!=NULL)

tempR->data[j] = (tempR->data[j] < preRecNode->data[j]) ? preRecNode->data[j] : tempR->data[j];

}

}

}

cout<<"The resulting vectors are:\n";

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

{

cout<<"Process "<<k + 1<<": ";

node \*temp1 = proc[k];

while(temp1)

{

cout<<"(";

for(int f = 0; f < n - 1; f++)

cout<<temp1->data[f]<<",";

cout<<temp1->data[n-1]<<")";

temp1 = temp1->next;

}

cout<<endl;

}

return 0;

}

**Output:**

