**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 TO-**

Dr. Prashant Kumar
Assistant Professor
CSE Department

**SUBMITTED BY-**

Name – Ankit Goyal
Roll no - 17103011
Group - G-1
Branch - CSE

# INDEX

# Practical No. 1

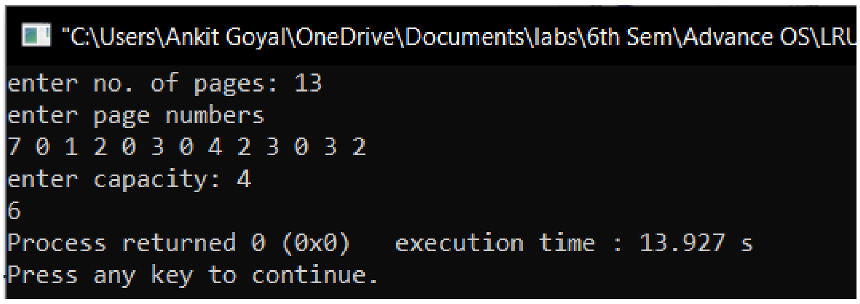
**Aim:** To implement LRU Page Replacement Algorithm.

**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:**



```
"C:\Users\Ankit Goyal\OneDrive\Documents\labs\6th Sem\Advance OS\LRU
enter no. of pages: 13
enter page numbers
7 0 1 2 0 3 0 4 2 3 0 3 2
enter capacity: 4
6
Process returned 0 (0x0)   execution time : 13.927 s
Press any key to continue.
```

# Practical no. 2

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

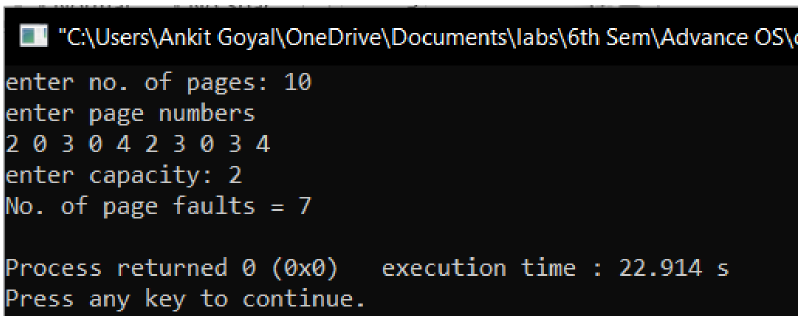
**Description:** In this algorithm, when a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future.

## 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:**



```
"C:\Users\Ankit Goyal\OneDrive\Documents\labs\6th Sem\Advance OS\
enter no. of pages: 10
enter page numbers
2 0 3 0 4 2 3 0 3 4
enter capacity: 2
No. of page faults = 7

Process returned 0 (0x0)   execution time : 22.914 s
Press any key to continue.
```

# 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.

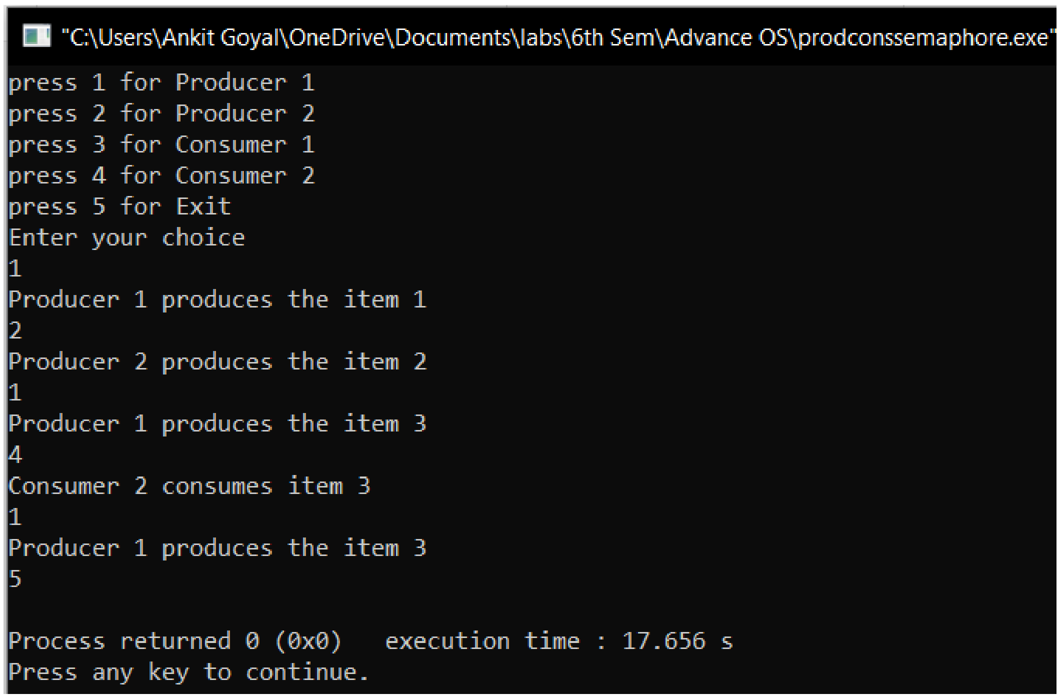
**Description:** In this problem we have a buffer of fixed size. Producer can produce an item and can place in the buffer. Consumer can pick items and can consume them. We need to ensure that when producer is placing an item in the buffer, then at the same time consumer should not consume any item.

## 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:**



"C:\Users\Ankit Goyal\OneDrive\Documents\labs\6th Sem\Advance OS\prodconssemaphore.exe"

```
press 1 for Producer 1
press 2 for Producer 2
press 3 for Consumer 1
press 4 for Consumer 2
press 5 for Exit
Enter your choice
1
Producer 1 produces the item 1
2
Producer 2 produces the item 2
1
Producer 1 produces the item 3
4
Consumer 2 consumes item 3
1
Producer 1 produces the item 3
5

Process returned 0 (0x0)   execution time : 17.656 s
Press any key to continue.
```

# Practical no. 4

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

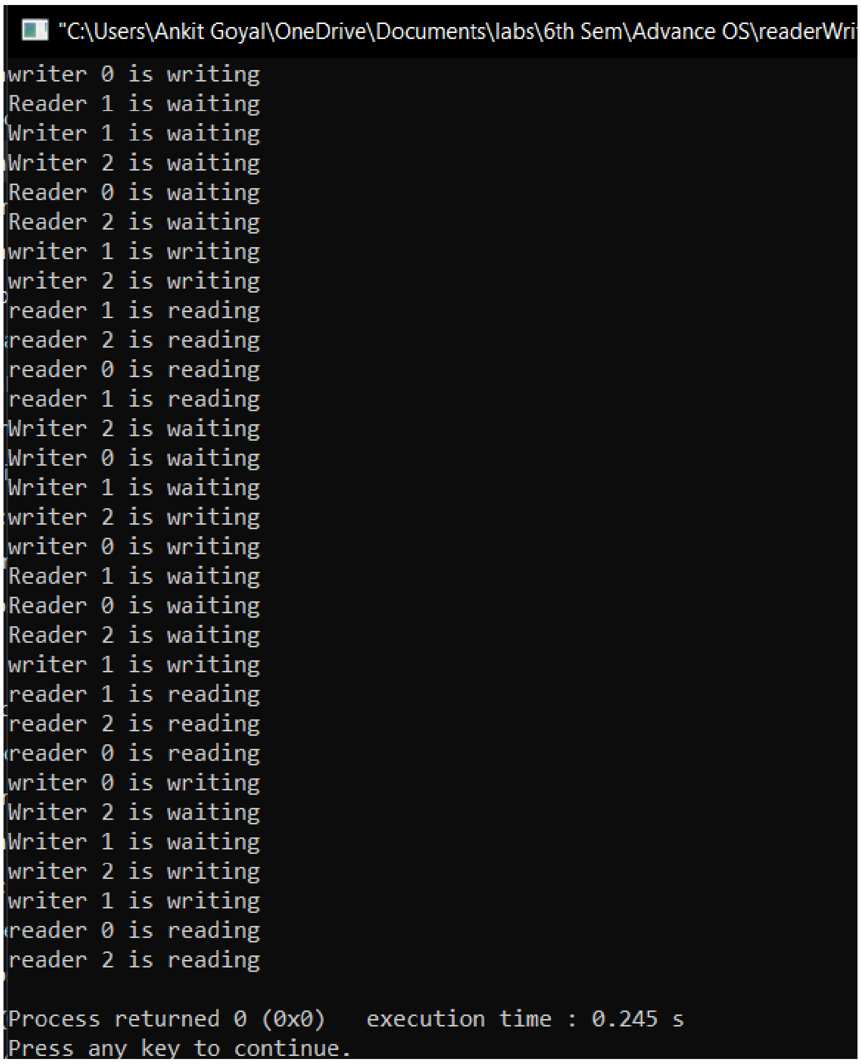
**Description:** The readers-writer problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

## 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:**



"C:\Users\Ankit Goyal\OneDrive\Documents\labs\6th Sem\Advance OS\readerWri

```
writer 0 is writing
Reader 1 is waiting
Writer 1 is waiting
Writer 2 is waiting
Reader 0 is waiting
Reader 2 is waiting
writer 1 is writing
writer 2 is writing
reader 1 is reading
reader 2 is reading
reader 0 is reading
reader 1 is reading
Writer 2 is waiting
Writer 0 is waiting
Writer 1 is waiting
writer 2 is writing
writer 0 is writing
Reader 1 is waiting
Reader 0 is waiting
Reader 2 is waiting
writer 1 is writing
reader 1 is reading
reader 2 is reading
reader 0 is reading
writer 0 is writing
Writer 2 is waiting
Writer 1 is waiting
writer 2 is writing
writer 1 is writing
reader 0 is reading
reader 2 is reading

Process returned 0 (0x0)   execution time : 0.245 s
Press any key to continue.
```

# Practical no. 5

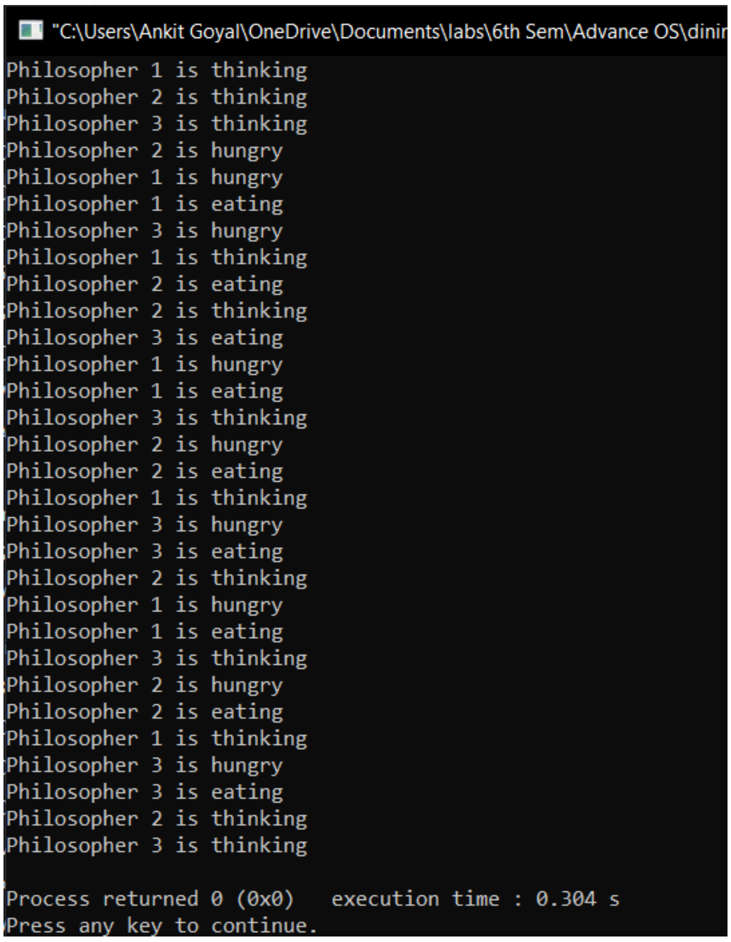
**Aim:** Write a program to illustrate solution for Dining Philosopher Problem.

**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:



"C:\Users\Ankit Goyal\OneDrive\Documents\labs\6th Sem\Advance OS\dinir

```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 2 is hungry
Philosopher 1 is hungry
Philosopher 1 is eating
Philosopher 3 is hungry
Philosopher 1 is thinking
Philosopher 2 is eating
Philosopher 2 is thinking
Philosopher 3 is eating
Philosopher 1 is hungry
Philosopher 1 is eating
Philosopher 3 is thinking
Philosopher 2 is hungry
Philosopher 2 is eating
Philosopher 1 is thinking
Philosopher 3 is hungry
Philosopher 3 is eating
Philosopher 2 is thinking
Philosopher 1 is hungry
Philosopher 1 is eating
Philosopher 3 is thinking
Philosopher 2 is hungry
Philosopher 2 is eating
Philosopher 1 is thinking
Philosopher 3 is hungry
Philosopher 3 is eating
Philosopher 2 is thinking
Philosopher 3 is thinking

Process returned 0 (0x0)   execution time : 0.304 s
Press any key to continue.
```

# Practical no. 6

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

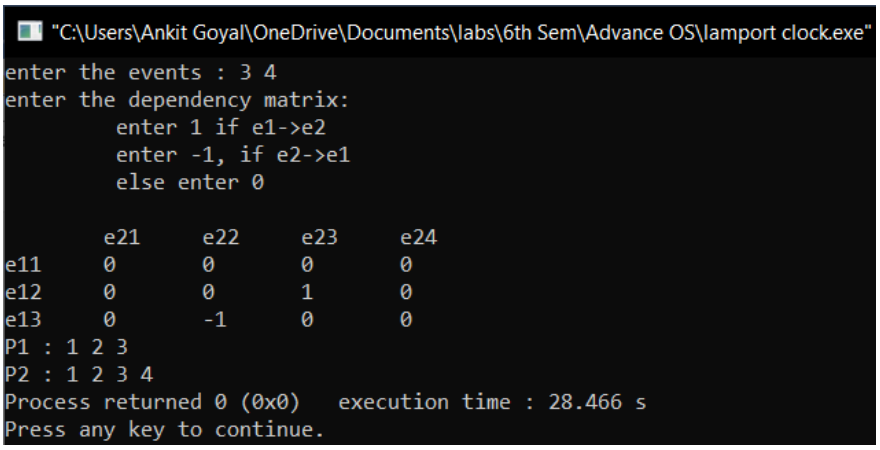
**Description:** This algorithm is used to determine the order of events in a distributed computer system. As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead

## 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:



```
"C:\Users\Ankit Goyal\OneDrive\Documents\labs\6th Sem\Advance OS\lamport clock.exe"
enter the events : 3 4
enter the dependency matrix:
         enter 1 if e1->e2
         enter -1, if e2->e1
         else enter 0

        e21     e22     e23     e24
e11     0       0       0       0
e12     0       0       1       0
e13     0       -1      0       0
P1 : 1 2 3
P2 : 1 2 3 4
Process returned 0 (0x0)   execution time : 28.466 s
Press any key to continue.
```

# Practical no. 7

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

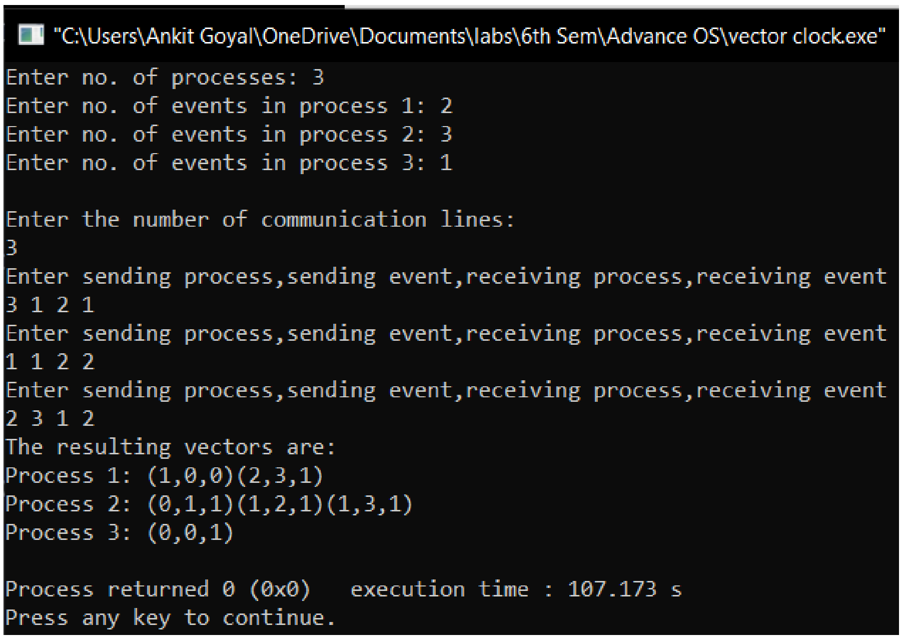
**Description:** A vector clock is an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations. Just as in Lamport timestamps, interprocess messages contain the state of the sending process's logical clock.

## 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:



```
"C:\Users\Ankit Goyal\OneDrive\Documents\labs\6th Sem\Advance OS\vector clock.exe"
Enter no. of processes: 3
Enter no. of events in process 1: 2
Enter no. of events in process 2: 3
Enter no. of events in process 3: 1

Enter the number of communication lines:
3
Enter sending process,sending event,receiving process,receiving event
3 1 2 1
Enter sending process,sending event,receiving process,receiving event
1 1 2 2
Enter sending process,sending event,receiving process,receiving event
2 3 1 2
The resulting vectors are:
Process 1: (1,0,0)(2,3,1)
Process 2: (0,1,1)(1,2,1)(1,3,1)
Process 3: (0,0,1)

Process returned 0 (0x0)   execution time : 107.173 s
Press any key to continue.
```