OPERATING SYSTEMS LAB

ETCS 352

Submitted To:

Dr. Navdeep Bohra

Submitted by:

Name: Shivangi Srivastava

Serial No: 27

Enrolment No: 05815002717

Class: CSE-2(B)

Maharaja Surajmal Institute of Technology,
C-4 Janak Puri, New Delhi 110058

INDEX

S NO.	CONTENT	PAGE NO.	DATE	REMARKS
1.	First come first serve	1-3	27-1-20	A 3/2/20
2.	Shortest Job First	4-6	3-2-20	A 24/2/20
3.	Priority scheduling	7-9		
4.	Round Robin algorithm	10-12		
5.	Page replacement algorithms			
	a) LRU	13-15		
	b) FIFO	16-17		
	c) Optimal	18-19		
6.	First fit, Best fit, Worst fit in memory management	20-25		
7.	Reader-writer problem using semaphore	26-29		
8.	Banker's algorithm for deadlock avoidance	30-32		

Experiment 1

AIM: Write a program to implement CPU scheduling for first come first serve.

THEORY: Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm.

First in, first out (FIFO), also known as first come, first served (FCFS),
is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.

In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.
Here we are considering that arrival time for all processes is 0.

How to compute below times in Round Robin using a program?

1. **Completion Time:** Time at which process completes its execution.
2. **Turn Around Time:** Time Difference between completion time and arrival time.
   Turn Around Time = Completion Time – Arrival Time
3. **Waiting Time(W.T):** Time Difference between turn around time and burst time.
   Waiting Time = Turn Around Time – Burst Time

CODE:

```
#include<iostream>

using namespace std;

void findWaitingTime(int processes[], int n, int
                     bt[], int wt[])
{
   wt[0] = 0;
   for (int  i = 1; i < n ; i++ )
      wt[i] =  bt[i-1] + wt[i-1]
      }


void findTurnAroundTime( int processes[], int n, int bt[],
              int wt[], int tat[])
{
    for (int  i = 0; i < n ; i++)
       tat[i] = bt[i] + wt[i];
}
```

```
void findavgTime( int processes[], int n, int bt[]){

    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(processes, n, bt, wt);
    findTurnAroundTime(processes, n, bt, wt, tat); cout <<
    "Processes  "<< " Burst time  "
         << " Waiting time  " << " Turn around time\n";
    for (int  i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << "    " << i+1 << "\t\t" << bt[i] <<"\t    "
            << wt[i] <<"\t\t  " << tat[i] <<endl;
    }
    cout << "Average waiting time = " << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
         << (float)total_tat / (float)n;
}
int main()
{
    int processes[] = { 1, 2, 3, 4, 5};
    int n = sizeof processes / sizeof processes[0];
    int  burst_time[] = {10, 5, 8, 7, 6};
    findavgTime(processes, n,  burst_time);
    return 0;
}
```

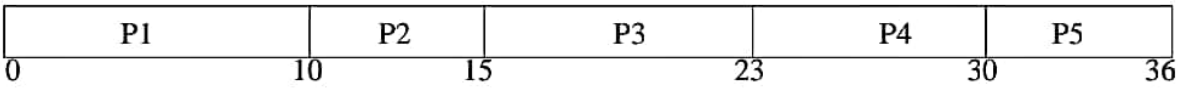
OUTPUT:

Processes	Burst time	Waiting time	Turn around time
1	10	0	10
2	5	10	15
3	8	15	23
4	7	23	30
5	6	30	36

Average waiting time = 15.6

Average turn around time = 22.8

GANTT CHART:



Experiment 2

AIM: Write a program to implement CPU scheduling for shortest job first.

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

How to compute below times in SJF using a program?

1. **Completion Time:** Time at which process completes its execution.
2. **Turn Around Time:** Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. **Waiting Time(W.T):** Time Difference between turn around time and burst time.
   Waiting Time = Turn Around Time – Burst Time

CODE:

```
#include <bits/stdc++.h>

using namespace std; struct

Process {

  int pid; // Process ID

  int bt; // Burst Time

  int art; // Arrival Time

};

void findTurnAroundTime(Process proc[], int n, int wt[], int tat[]) {

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

  tat[i] = proc[i].bt + wt[i];

}

void findWaitingTime(Process proc[], int n, int wt[]) {

  int rt[n];

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

  rt[i] = proc[i].bt;

  int complete = 0, t = 0, minm = INT_MAX;

  int shortest = 0, finish_time;

  bool check = false;
```

```
    while (complete != n) {
    for (int j = 0; j < n; j++) {
      if ((proc[j].art <= t) && (rt[j] < minm) && rt[j] > 0) {
            minm = rt[j];
            shortest = j;
            check = true;
      }
    }
    if (check == false) {
      t++;
      continue;
    }
    rt[shortest]--;
    minm = rt[shortest];
    if (minm == 0)
        minm = INT_MAX;
      if (rt[shortest] == 0) {
        complete++;
        check = false;
        finish_time = t + 1; wt[shortest] =
        finish_time - proc[shortest].bt -
        proc[shortest].art;
        if (wt[shortest] < 0)
              wt[shortest] = 0;
      }
      t++;
  }
}
void findavgTime(Process proc[], int n) {
  int wt[n], tat[n], total_wt = 0,
  total_tat = 0; findWaitingTime(proc, n, wt);
  findTurnAroundTime(proc, n, wt, tat);
  cout << "Processes " << " Burst time " << " Waiting time " << " Turn around
time\n";
```

```
   for (int i = 0; i < n; i++) {
          total_wt = total_wt + wt[i];
          total_tat = total_tat + tat[i];
      cout << " " << proc[i].pid << "\t\t"
      << proc[i].bt << "\t\t " << wt[i] <<
                        "\t\t "
<< tat[i] << endl;   }
   cout << "\nAverage waiting time = " << (float)total_wt / (float)n; cout <<
"\nAverage turn around time = " << (float)total_tat / (float)n;
}
int main() {
   Process proc[] = { { 1, 5, 0 }, { 2, 3, 1 }, { 3, 6, 2 }, { 4, 5, 3 } };
   int n = sizeof(proc) / sizeof(proc[0]);
   findavgTime(proc, n);
   return 0;   }
```

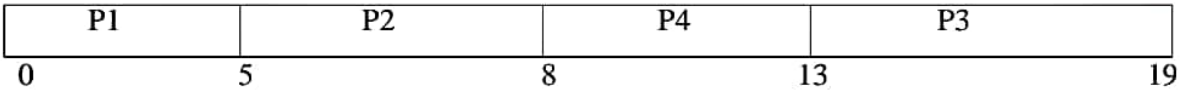
OUTPUT:

Processes	Burst time	Waiting time	Turn around time
1	5	0	5
2	3	4	7
3	6	10	17
4	5	5	10

Average waiting time = 4.75

Average turn around time = 9.5

GANTT CHART:



Experiment 3

AIM: Write a program to perform priority scheduling.

THEORY: Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned first arrival time (less arrival time process first) if two processes have same arrival time, then compare to priorities (highest process first). Also, if two processes have same priority then compare to process number (less process number first). This process is repeated while all process get executed.

1. First input the processes with their arrival time, burst time and priority.
2. Sort the processes, according to arrival time if two process arrival time is same then sort according process priority if two process priority are same then sort according to process number.
3. Now simply apply FCFS algorithm.

CODE:

```
#include<bits/stdc++.h>
using namespace std;
struct Process
{
        int pid; // Process ID
        int bt; // CPU Burst time required
        int priority; // Priority of this process
};
bool comparison(Process a, Process b)
{
        return (a.priority > b.priority);
}
void findWaitingTime(Process proc[], int n,
                                        int wt[]) {
        wt[0] = 0;
        for (int i = 1; i < n ; i++ )
                wt[i] = proc[i-1].bt + wt[i-1] ;
}
```

```
void findTurnAroundTime( Process proc[], int n,
                                                int wt[], int tat[])
{
        for (int i = 0; i < n ; i++)
                tat[i] = proc[i].bt + wt[i];
}
void findavgTime(Process proc[], int n)
{
        int wt[n], tat[n], total_wt = 0, total_tat = 0;
        findWaitingTime(proc, n, wt);
        findTurnAroundTime(proc, n, wt, tat);
        cout << "\nProcesses "<< " Burst time "
                << " Waiting time " << " Turn around time\n";
        for (int i=0; i<n; i++) {
                total_wt = total_wt + wt[i];
                total_tat = total_tat + tat[i];
                cout << " " << proc[i].pid << "\t\t"
                        << proc[i].bt << "\t " << wt[i]
                        << "\t\t " << tat[i] <<endl;
        }
        cout << "\nAverage waiting time = "
                << (float)total_wt / (float)n;
        cout << "\nAverage turn around time = "
                << (float)total_tat / (float)n;
}

void priorityScheduling(Process proc[], int n) {
        sort(proc, proc + n, comparison);
        cout<< "Order in which processes gets executed \n";
        for (int i = 0 ; i < n; i++)
                cout << proc[i].pid <<" " ;
        findavgTime(proc, n);
}
int main() {
```

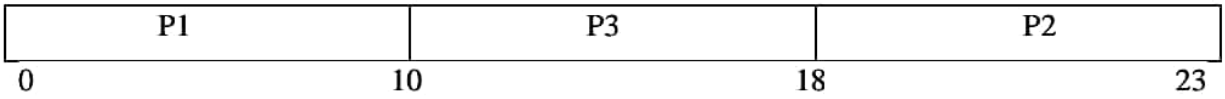
```
Process proc[] = {{1, 10, 2}, {2, 5, 0}, {3, 8, 1}};
int n = sizeof proc / sizeof proc[0];
priorityScheduling(proc, n);
return 0;  }
```

OUTPUT:

```
Order in which processes gets executed
1 3 2
Processes  Burst time  Waiting time  Turn around time
 1              10       0               10
 3              8        10              18
 2              5        18              23

Average waiting time = 9.33333
Average turn around time = 17
```

GANTT CHART:



Experiment 4

AIM: Write a program to implement CPU scheduling for Round robin.

THEORY: Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.

How to compute below times in Round Robin using a program?

1. **Completion Time:** Time at which process completes its execution.
2. **Turn Around Time:** Time Difference between completion time and arrival time.

   Turn Around Time = Completion Time – Arrival Time
3. **Waiting Time(W.T):** Time Difference between turn around time and burst time.

   Waiting Time = Turn Around Time – Burst Time

CODE:

```
#include<iostream>
using namespace std;
void findWaitingTime(int processes[], int n,
                        int bt[], int wt[], int quantum) {
    int rem_bt[n];
    for (int i = 0 ; i < n ; i++)
        rem_bt[i] = bt[i];
    int t = 0;
    while (1) {
        bool done = true;
        for (int i = 0 ; i < n; i++) {
            if (rem_bt[i] > 0) {
                done = false;
                if (rem_bt[i] > quantum) {
                    t += quantum;
                    rem_bt[i] -= quantum;
```

```
                        }

                        else{
                                t = t + rem_bt[i];
                                wt[i] = t - bt[i];
                                rem_bt[i] = 0;
                        }
                }
        }
        if (done == true)
        break;
    }
}
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}
void findavgTime(int processes[], int n, int bt[], int quantum) {
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(processes, n, bt, wt, quantum);
    findTurnAroundTime(processes, n, bt, wt, tat);
    cout << "Processes "<< " Burst time "
        << " Waiting time " << " Turn around time\n";
    for (int i=0; i<n; i++)  {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
            << wt[i] <<"\t\t " << tat[i] <<endl;
    }
    cout << "Average waiting time = "
        << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
        << (float)total_tat / (float)n;
}
```

```
int main() {
        int processes[] = { 1, 2, 3};
        int n = sizeof processes / sizeof processes[0];
        int burst_time[] = {10, 5, 8};

        int quantum = 2;
        findavgTime(processes, n, burst_time, quantum);
        return 0;
}
```

OUTPUT:

```
Processes  Burst time  Waiting time  Turn around time
 1           3       4            7
 2           4       5            9
 3           3       7            10
Average waiting time = 5.33333
Average turn around time = 8.66667
```

GANTT CHART:

P1	P2	P3	P1	P2	P3	P1	P2	P3	P2

0 10

Experiment 5

AIM: Write a program for page replacement policy using

a) LRU b) FIFO c) Optimal

THEORY: In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

LRU: In **L**east **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely to be used again anytime soon.

FIFO: This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

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

CODE (a):

```
#include<bits/stdc++.h>
using namespace std;
int pageFaults(int pages[], int n, int capacity) {
        unordered_set<int> s;
        unordered_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;
                for (auto 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 pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
```

```
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    int f = pageFaults(pages, n, capacity);
    cout <<"No of page faults: "<< f;
    cout<<"No of page hits: "<<(n-f);
    return 0;
}
```

OUTPUT:

No of page faults: 6

No of page hits: 7

CODE (b):

```
#include<bits/stdc++.h>
using namespace std;
int pageFaults(int pages[], int n, int capacity) {
        unordered_set<int> s;
        queue<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.push(pages[i]);
                        }
                }
                else{
                        if (s.find(pages[i]) == s.end()) {
                                int val = indexes.front();
                                indexes.pop();
                                s.erase(val);
                                s.insert(pages[i]);
                                indexes.push(pages[i]);
                                page_faults++;
                        }
                }
        }
        return page_faults;
}
int main() {
        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
        int n = sizeof(pages)/sizeof(pages[0]);
        int capacity = 4;
        int f = pageFaults(pages, n, capacity);
        cout <<"No of page faults: "<<f;
```

```
    cout<<"No of page hits: "<<(n-f);
    return 0; }
```

OUTPUT:

No of page faults: 7

No of page hits: 6

CODE (c):

```
#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 hits = " << hit << endl;
        cout << "No. of misses = " << pn - hit << endl;
}
int main()
{
        int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };
        int pn = sizeof(pg) / sizeof(pg[0]);
        int fn = 4;
        optimalPage(pg, pn, fn);
        return 0;
}
```

OUTPUT:

No of hits = 7

No of misses = 6

Experiment 6

AIM: Write a program to implement first fit, best fit and worst fit algorithm for memory management.

THEORY:

First-Fit Memory Allocation: This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. In this method, first job claims the first available memory with space more than or equal to it's size. The operating system doesn't search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.

Best-Fit Memory Allocation: This method keeps the free/busy list in order by size – smallest to largest. In this method, the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently. Here the jobs are in the order from smalest job to largest job.

Worst-Fit Memory Allocation: Worst Fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.

CODE (First-Fit):

```
#include<bits/stdc++.h>
using namespace std;
void firstFit(int blockSize[], int m,
                        int processSize[], int n) {
    int allocation[n];
    memset(allocation, -1, sizeof(allocation));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
```

```
            }
        }

        cout << "\nProcess No.\tProcess Size\tBlock no.\n";
        for (int i = 0; i < n; i++) {
            cout << " " << i+1 << "\t\t"<< processSize[i] << "\t\t";
            if (allocation[i] != -1)
                cout << allocation[i] + 1;
            else
                cout << "Not Allocated";
            cout << endl;
        }
}
int main() {
        int blockSize[] = {100, 500, 200, 300, 600};
        int processSize[] = {212, 417, 112, 426};
        int m = sizeof(blockSize) / sizeof(blockSize[0]);
        int n = sizeof(processSize) / sizeof(processSize[0]);
        firstFit(blockSize, m, processSize, n);
        return 0 ;
}
```

OUTPUT:

```
Process No.    Process Size    Block no.
   1            212             2
   2            417             5
   3            112             2
   4            426          Not Allocated
```

CODE (Best-Fit):

```
#include<bits/stdc++.h>
using namespace std;
void bestFit(int blockSize[], int m, int processSize[], int n) {
	int allocation[n];
	memset(allocation, -1, sizeof(allocation));
	for (int i=0; i<n; i++) {
		int bestIdx = -1;
		for (int j=0; j<m; j++) {
			if (blockSize[j] >= processSize[i]) {
				if (bestIdx == -1)
					bestIdx = j;
				else if (blockSize[bestIdx] > blockSize[j])
					bestIdx = j;
			}
		}
		if (bestIdx != -1) {
			allocation[i] = bestIdx;
			blockSize[bestIdx] -= processSize[i];
		}
	}
	cout << "\nProcess No.\tProcess Size\tBlock no.\n";
	for (int i = 0; i < n; i++) {
		cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
		if (allocation[i] != -1)
			cout << allocation[i] + 1;
		else
			cout << "Not Allocated";
		cout << endl;
	}
}
int main() {
	int blockSize[] = {100, 500, 200, 300, 600};
	int processSize[] = {212, 417, 112, 426};
```

```
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);
    bestFit(blockSize, m, processSize, n);
    return 0 ;
}
```

OUTPUT:

```
Process No.    Process Size    Block no.
 1              212             4
 2              417             2
 3              112             3
 4              426             5
```

CODE (Worst-Fit):

```
#include<bits/stdc++.h>
using namespace std;
void worstFit(int blockSize[], int m, int processSize[], int n) {
	int allocation[n];
	memset(allocation, -1, sizeof(allocation));
	for (int i=0; i<n; i++) {
		int wstIdx = -1;
		for (int j=0; j<m; j++) {
			if (blockSize[j] >= processSize[i]) {
				if (wstIdx == -1)
					wstIdx = j;
				else if (blockSize[wstIdx] < blockSize[j])
					wstIdx = j;
			}
		}
		if (wstIdx != -1) {
			allocation[i] = wstIdx;
			blockSize[wstIdx] -= processSize[i];
		}
	}
	cout << "\nProcess No.\tProcess Size\tBlock no.\n";
	for (int i = 0; i < n; i++) {
		cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
		if (allocation[i] != -1)
			cout << allocation[i] + 1;
		else
			cout << "Not Allocated";
		cout << endl;
	}
}

int main() {
	int blockSize[] = {100, 500, 200, 300, 600};
```

```
        int processSize[] = {212, 417, 112, 426};
        int m = sizeof(blockSize)/sizeof(blockSize[0]);
        int n = sizeof(processSize)/sizeof(processSize[0]);
        worstFit(blockSize, m, processSize, n);
        return 0 ;
}
```

OUTPUT:

```
Process No.    Process Size    Block no.
   1            212             5
   2            417             2
   3            112             5
   4            426           Not Allocated
```

Experiment 7

AIM: Write a program to implement reader/writer problem using semaphore.

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

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

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

Problem Parameters

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization. The wait operation decrements the value of its argument S, if it is positive.

CODE:

```
#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);
        return(0);
}
```

OUTPUT:

```
------------------------
    reader-0 is reading
   updated value : 5

    writer-0 is writing
   ------------------------
```

reader-1 is reading

updated value : 10

---

reader-2 is reading

updated value : 10

---

reader-3 is reading

updated value : 10

writer-3 is writing

---

reader-4 is reading

Experiment 8

AIM: Write a program to implement the Banker's algorithm for deadlock avoidance.

THEORY: The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Why Banker's algorithm is named so?

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

CODE:

```
#include <iostream>
using namespace std;
int main() {
        // P0, P1, P2, P3, P4 are the Process names here
        int n, m, i, j, k;
        n = 5; // Number of processes
        m = 3; // Number of resources
        int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                                                { 2, 0, 0 }, // P1
                                                { 3, 0, 2 }, // P2
                                                { 2, 1, 1 }, // P3
                                                { 0, 0, 2 } }; // P4


        int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
```

```
                    { 3, 2, 2 }, // P1
                    { 9, 0, 2 }, // P2
                    { 2, 2, 2 }, // P3
                    { 4, 3, 3 } }; // P4

int avail[3] = { 3, 3, 2 }; // Available Resources

int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
        f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
                need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
                if (f[i] == 0) {
                        int flag = 0;
                        for (j = 0; j < m; j++) {
                                if (need[i][j] > avail[j]){
                                        flag = 1;
                                        break;
                                }
                        }
                        if (flag == 0) {
                                ans[ind++] = i;
                                for (y = 0; y < m; y++)
                                        avail[y] += alloc[i][y];
                                f[i] = 1;
                        }
                }
```

```
            }
        }
        cout << "Following is the SAFE Sequence" << endl;
        for (i = 0; i < n - 1; i++)
            cout << " P" << ans[i] << " ->";
        cout << " P" << ans[n - 1] <<endl;
        return (0);
}
```

OUTPUT:

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```