EXPERIMENT 1

Aim: WAP to implement CPU scheduling for first come first serve.

Theory:

→ **First come first serve (FCFS) scheduling:-** FCFS is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. In this type of algorithm, processes which requests the CPU first get the CPU allocation first managed using ~~FF~~ FIFO queue.

→ **Characteristics of FCFS method:-**

1) It supports non-preemptive & preemptive scheduling algorithm.

2) Jobs are always executed on a first come first serve basis.

3) It is easy to implement & use.

Result: CPU scheduling on the basis of first come first serve has been performed and implemented successfully.

*

SOURCE 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};
	int n = sizeof processes / sizeof processes[0];

	int burst_time[] = {10, 5, 8};

	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
Average waiting time = 8.33333
Average turn around time = 16
--------------------------------
Process exited after 0.1102 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT 2

Aim: WAP to implement CPU scheduling for Shortest Job first.

Theory: Short Job first (SJF) is an algorithm in w/c the process having the smallest execution time is chosen for the next execution. This scheduling method can be both preemptive and non-preemptive. It helps in reducing the average waiting time for other process.

→ The two types of SJF scheduling :-
(i) Non-preemptive SJF (ii) Preemptive SJF

→ Characteristics of SJF scheduling :-
1) It is associated with each job as a unit of time to complete.
2) This algorithm method is helpful for batch type processing.
3) It increases process throughout.
4) It improves job output.

→ Terminologies :-
1) Completion time – Time at w/c process completes.
2) Turn around time – Time difference between completion and arrival time.
3) Waiting time – Time difference b/w turn around time and burst time.

Result: CPU scheduling on basis of Shortest Job first has been implemented successfully.

SOURCE CODE:

```
#include<iostream>
using namespace std;
int mat[10][6];

void swap(int *a, int *b)
{
	int temp = *a;
	*a = *b;
	*b = temp;
}

void arrangeArrival(int num, int mat[][6])
{
	for(int i=0; i<num; i++)
	{
		for(int j=0; j<num-i-1; j++)
		{
			if(mat[j][1] > mat[j+1][1])
			{
				for(int k=0; k<5; k++)
				{
					swap(mat[j][k], mat[j+1][k]);
				}
			}
		}
	}
}

void completionTime(int num, int mat[][6])
{
	int temp, val;
	mat[0][3] = mat[0][1] + mat[0][2];
	mat[0][5] = mat[0][3] - mat[0][1];
	mat[0][4] = mat[0][5] - mat[0][2];

	for(int i=1; i<num; i++)
	{
		temp = mat[i-1][3];
		int low = mat[i][2];
		for(int j=i; j<num; j++)
		{
			if(temp >= mat[j][1] && low >= mat[j][2])
			{
				low = mat[j][2];
				val = j;
			}
		}
		mat[val][3] = temp + mat[val][2];
		mat[val][5] = mat[val][3] - mat[val][1];
```

```
                mat[val][4] = mat[val][5] - mat[val][2];
                for(int k=0; k<6; k++)
                {
                        swap(mat[val][k], mat[i][k]);
                }
        }
}

int main()
{
        int num, temp;

        cout<<"Enter number of Process: ";
        cin>>num;

        cout<<"...Enter the process ID...\n";
        for(int i=0; i<num; i++)
        {
                cout<<"...Process "<<i+1<<"...\n";
                cout<<"Enter Process Id: ";
                cin>>mat[i][0];
                cout<<"Enter Arrival Time: ";
                cin>>mat[i][1];
                cout<<"Enter Burst Time: ";
                cin>>mat[i][2];
        }

        cout<<"Before Arrange...\n";
        cout<<"Process ID\tArrival Time\tBurst Time\n";
        for(int i=0; i<num; i++)
        {
                cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\n";
        }

        arrangeArrival(num, mat);
        completionTime(num, mat);
        cout<<"Final Result...\n";
        cout<<"Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n";
        for(int i=0; i<num; i++)
        {

cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\t\t"<<mat[i][4]<<"\t\t"<<mat[i]
[5]<<"\n";
        }
}
```

OUTPUT:

```
Enter number of Process: 4
...Enter the process ID...
...Process 1...
Enter Process Id: 1 2 3 4
Enter Arrival Time: Enter Burst Time: ...Process 2...
Enter Process Id: Enter Arrival Time: 4 5
Enter Burst Time: ...Process 3...
Enter Process Id: 6
Enter Arrival Time: 5
Enter Burst Time: 8
...Process 4...
Enter Process Id: 5
Enter Arrival Time: 2
Enter Burst Time: 5
Before Arrange...
Process ID      Arrival Time    Burst Time
1               2               3
4               4               5
6               5               8
5               2               5
Final Result...
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround Time
1               2               3               0               3
4               4               5               1               6
5               2               5               8               13
6               5               8               10              18

--------------------------------
Process exited after 68.46 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT 3

Aim: WAP to perform priority scheduling.

Theory: Priority scheduling is a method of scheduling processes that is based on priority. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a Round Robin or FCFS basis. Priorities are assigned on the basis of memory or time requirements, etc.

→ Types of priority scheduling:-
(i) Preemptive scheduling (ii) Non-Preemptive scheduling.

→ Characteristics of Priority Scheduling:-
1) CPU algorithm that schedule process based on priority.
2) It is used for batch processes.
3) For same priority jobs, FCFS is applied.
4) For priority each task is associated with a number.
5) Lower the number, higher the priority.

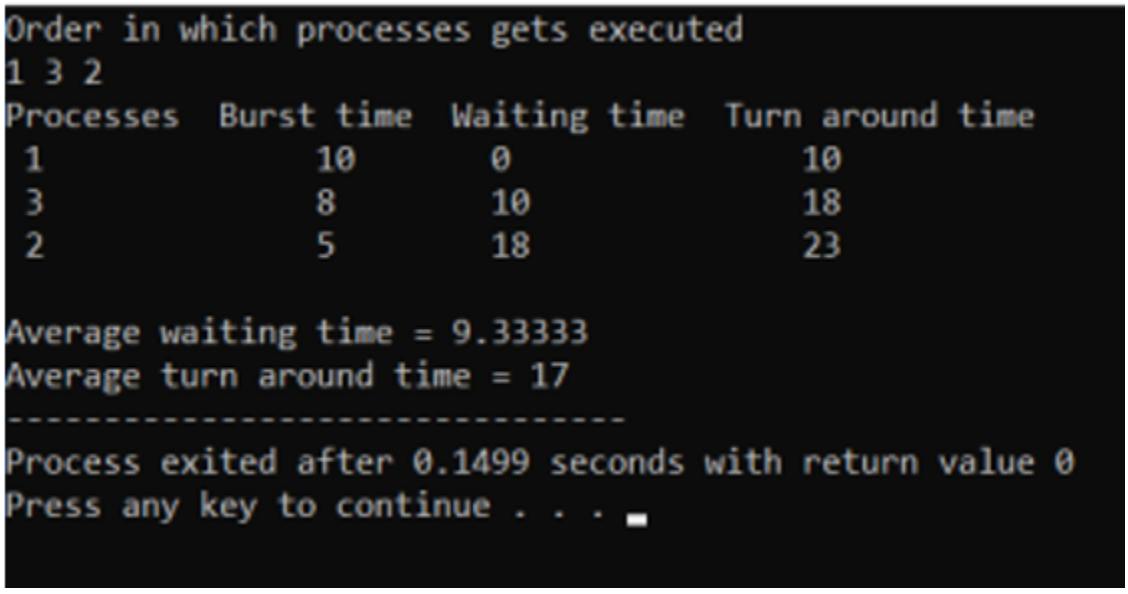
Result: Priority scheduling has been implemented successfully.

SOURCE 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:



DELTA Pg No. 4
Date / /

EXPERIMENT 4

Aim: Write a program to implement Round Robin for CPU scheduling.

Theory: The name of this algorithm come from the round-robin principle, where each person gets an equal share of something in turn. It is the oldest and simple scheduling algorithm and generally used for multitasking.

→ **Characteristics :-**

1) Round Robin is a preemptive algorithm.

2) CPU is shifted to the ~~next~~ next process after fixed time, interval time, called time quantum / slice.

3) Round robin is a hybrid model w/c is clock driven.

4) Time slice should be minimum.

Result: The CPU scheduling for ~~Round~~ Round Robin experiment was studied and performed successfully.

SOURCE 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;	// Current time

	while (1)	{
		bool done = true;

		for (int i = 0 ; i < n; i++)
		{
			if (rem_bt[i] > 0)
			{
				done = false;		// There is a pending process

				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              10         13             23
 2              5          10             15
 3              8          13             21
Average waiting time = 12
Average turn around time = 19.6667
--------------------------------
Process exited after 0.1045 seconds with return value 0
Press any key to continue . . .
```

DELTA Pg No. 52
Date / /

EXPERIMENT 5

Aim: Write a program for page replacement policy using:
(a) LRU
(b) FIFO
(c) Optimal Approach.

Theory: In an operating system that use paging for memory management, page replacement algorithm are needed to decide wh page needed to be replaced when new pages come in. Whenever a new page is found / ~~refrashed~~ referred and not present in memory page fault occurs and operating system replaces one of the existing pages with newly needed page. Different algorithms decide wh page to be replaced & to reduce no. of page faults.

Page fault: A page fault is a type of interrupt, raised by the hardware when a running prkess access a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Least Recently Used (LRU)
In this algorithm the least recently used page will be replaced.

Algorithm –
Let capacity be the no. of pages that memory can hold. Let set be the current set of pages in memory.

1) Start traversing the pages
If set holds less pages than capacity
Insert page into the set until the size of set reaches capacity or all page requests are processed.
Simultaneously maintain the recent occured index of each page in a map called indexes.
Increment page fault.
Else
If current page is present in set, do nothing.
Else
~~find the page in the set that~~ was least recently used. We basically need to replace the page with minimum index.
~~Increment pfno~~ Replace the found page with current page.
Increment page faults.
Update index of current page.
2) Return page faults.

First In First Out (FIFO)

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

Optimal Page Replacement

In this algorithm, pages are replaced ~~w~~ which are not used for the longest algorithm duration of the time in future.

Result: The experiment has been successfully performed & studied.

SOURCE CODE:

A) Program for page replacement policy using LRU:

```
#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;
        cout << pageFaults(pages, n, capacity);
        return 0;
}
```

OUTPUT:

```
6
--------------------------------
Process exited after 0.1252 seconds with return value 0
Press any key to continue . . .
```

B) Program for page replacement policy using FIFO:

```
#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;
        cout << pageFaults(pages, n, capacity);
        return 0;
}
```

OUTPUT:

```
7
--------------------------------
Process exited after 0.08578 seconds with return value 0
Press any key to continue . . .
```

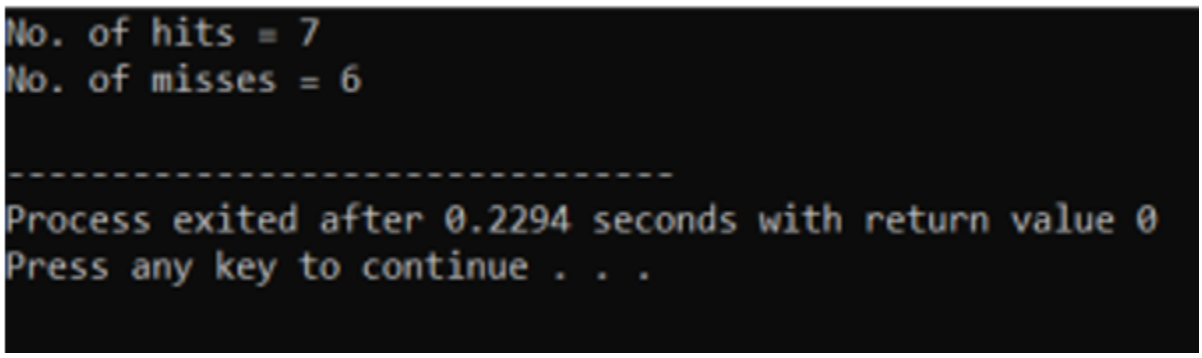
C) Program for page replacement policy using Optimal Approach:

```
#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:



EXPERIMENT 6

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

Theory: There comes three algorithms for memory management:

1) **Best fit:-** It deals with allocating the smallest free partition w/c meets the requirement of the requesting process. The algorithm first searches the entire list of free partitions and consider the smallest hole which is close to actual process size needed.

Algorithm -

1) Input memory blocks and processes with sizes.
2) Initialize all memory blocks as free.
3) Start by picking each process and find the minimum block size that can be assigned to the current process.
4) If not then leave that process & keep checking the further processes.

2) **First fit:-** In this approach, it allocates the first free partition or hole ~~too~~ large enough w/c can accomodate the process. It finishes the first suitable free partitions.

Algorithm -

1) Input memory block with size & processes with size.
2) Initialize all memory blocks ~~are~~ free.
3) Start by picking each process & check if it can be assigned to the current block.
4) If size_of_block >= size_of_process then assign & check for

next process.
5) If not then keep checking the further blocks.

3) **Worst fit**:- In this approach, it locates the largest available free portion so that the portion left will be big enough to be useful. It is reverse of best fit.

Algorithm -
1) Input memory blocks & processes.
2) Initialize all memory blocks as free.
3) Start by picking each process & find the maximum block size that can be assigned to current process.
4) If not leave the process & keep checking the further processes.

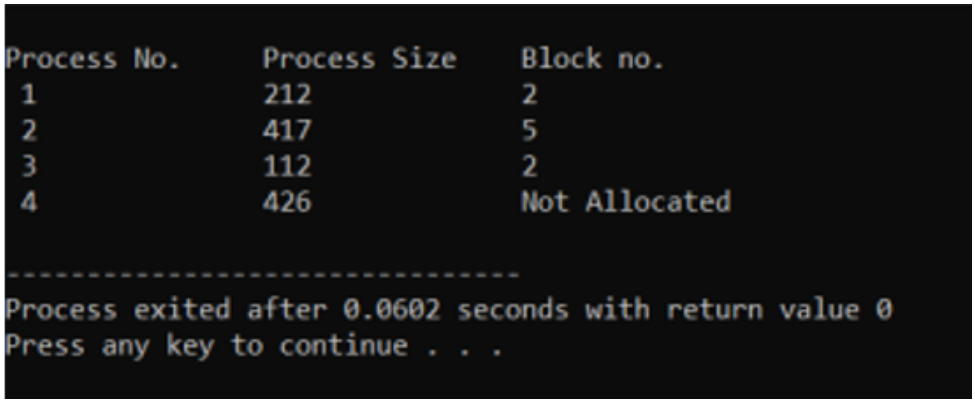
Result:- The experiment has been studied & implemented successfully.

SOURCE CODE:

A) Program implementing First Fit algorithm for memory management:

```
#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:



B) Program implementing Best Fit algorithm for memory management:

```
#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 ;
}
```

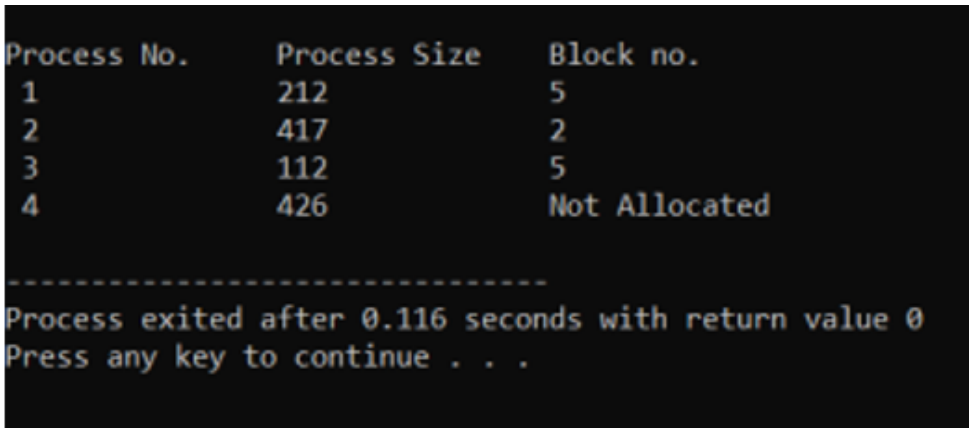
OUTPUT:

C) Program implementing Worst Fit algorithm for memory management:

```
#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:



DELTA Pg No. 10
Date / /

EXPERIMENT 7

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

Language used: C++

Theory: Parameters of reader/writer problems:

1) One set of data is shared among a no. of processes.
2) Once a writer is ready, it performs its write. Only one writer may write at a time.
3) If a process is writing, no other process can write or read it.
4) If atleast one reader is reading, no other process can write.
5) Readers may not write and only read.

Solution when reader has a priority over writer

Here priority means, no reader should wait if the share is currently opened for reading. Three variables are used: mutex, wrt, readent to implement various solutions:-

1) semaphore mutex, wrt; // semaphore mutex is used to ensure mutual exclusion when readent is updated i.e., when any reader enters or exit from the critical section and semaphore wrt is used by both readers & writers
2) int readent; // readent tells the no. of processes performing read in the critical section, initially 0.

Reader Process :-

1) It requests the entry to critical section

Pseudocode –

```
do {
    wait (mutex);   // The no. of reader increased by 1
    readent ++;
    if (readent == 1)
        wait (wrt);
    signal (mutex);
    wait (mutex);   // a reader wants to leave.
    readent --;
    if (readent == 0)
        signal (wrt);   // writers can enter
    signal (mutex);   // reader leaves.
}
while (true);
```

Writer Process :-

Pseudocode –

```
do {
    wait (wrt);   // Perform the write
    signal (wrt);   // leaves the critical section
}
while (true);
```

Result : The experiment has been successfully studied & implemented.

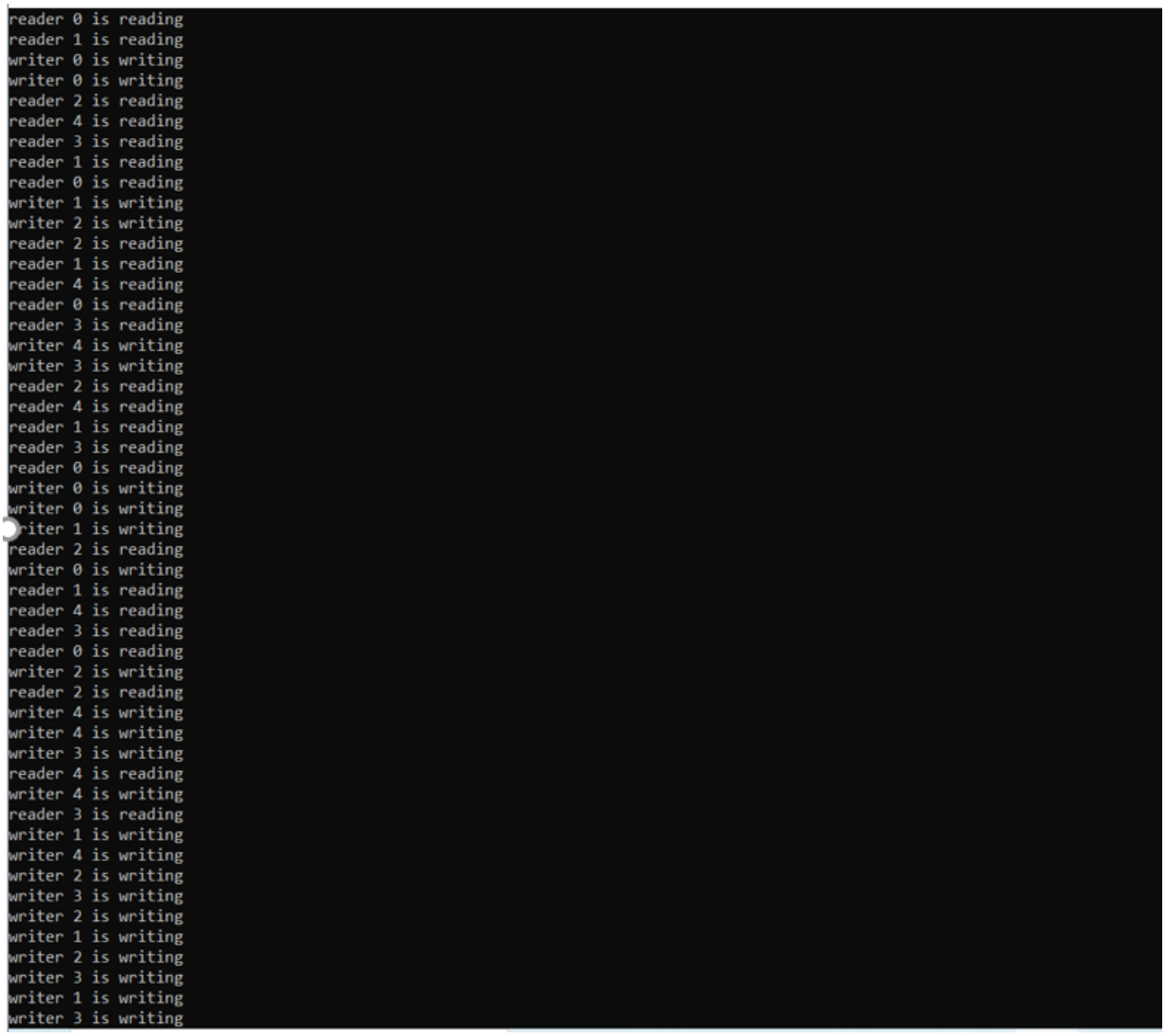
SOURCE CODE:

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
using namespace std;

class monitor {
private:
	int rcnt;
	int wcnt;
	int waitr;
	int waitw;
	pthread_cond_t canread;
	pthread_cond_t canwrite;
	pthread_mutex_t condlock;

public:
	monitor()
	{
		rcnt = 0;
		wcnt = 0;
		waitr = 0;
		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) {
			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) {
			++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 (waitr > 0)
			pthread_cond_signal(&canread);
		else
			pthread_cond_signal(&canwrite);
		pthread_mutex_unlock(&condlock);
	}
}

M;

void* reader(void* id)
{
	int c = 0;
	int i = *(int*)id;
	while (c < 5) {
		usleep(1);
		M.beginread(i);
		M.endread(i);
		c++;
	}
}

void* writer(void* id)
{
	int c = 0;
	int i = *(int*)id;
	while (c < 5) {
		usleep(1);
		M.beginwrite(i);
		M.endwrite(i);
		c++;
	}
}
```

```
int main()
{
	pthread_t r[5], w[5];
	int id[5];
	for (int i = 0; i < 5; 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 < 5; i++) {
		pthread_join(r[i], NULL);
	}
	for (int i = 0; i < 5; i++) {
		pthread_join(w[i], NULL);
	}
}
```

OUTPUT:



DELTA Pg No. 12
Date / /

EXPERIMENT 8

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

Theory: Banker's Algorithm is a deadlock avoidance algorithm. It is named so because it is used in banking system to determine whether a loan can be generated or not. It works in a similar way in computers. Whenever a new process is created, it must exactly specify the maximum instances of each resource types that it needs. Let us assume there are 'n' processes and 'm' resource types. Now, to describe the behaviour of system:

→ **Resource request algorithm:-**
This describes the behaviour of the system when a process makes a resource request in the form of a request matrix.

After this, we need to check if the system is in safe state by applying safety algorithm. If it is the safe state, proceed to allocate the requested resource. Else, the process has to wait longer.

→ **Safety Algorithm:-**
1) Let work & finish be vectors of length 'm' and 'n'.

Work = Available

Finish [i] = false for $i = 0, 1 \ldots, n-1$

2) find an index 'i' such that both
finish [i] = false
Need$_i$ <= Work
If there is no such 'i' present, then proceed to Step (4).

3) Perform the following:
Work = Work + Allocation$_i$
finish [i] = true;
Goto Step 2

4) If finish [i] == true, for all i, then the system is in safe state.

→ Example:

Input:

Total Resources		R1	R2	R3
		10	5	7

Process	Allocation			Max		
	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3
P2	2	0	0	3	2	2
P3	3	0	2	9	0	2
P4	2	1	1	2	2	2

Output: Safe sequences are: (4 total sequences)
P2 → P4 → P1 → P3
P4 → P4 → P3 → P1
P4 → P2 → P1 → P3
P4 → P2 → P3 → P1

Result: The experiment was successfully studied & implemented.

SOURCE CODE:

```
#include<iostream>
using namespace std;

const int P = 5;
const int R = 3;

void calculateNeed(int need[P][R], int maxm[P][R], int allot[P][R])
{
	for (int i = 0 ; i < P ; i++)
		for (int j = 0 ; j < R ; j++)
			need[i][j] = maxm[i][j] - allot[i][j];
}

bool isSafe(int processes[], int avail[], int maxm[][R], int allot[][R])
{
	int need[P][R];
	calculateNeed(need, maxm, allot);
	bool finish[P] = {0};
	int safeSeq[P];
	int work[R];
	for (int i = 0; i < R ; i++)
		work[i] = avail[i];
	int count = 0;
	while (count < P)
	{
		bool found = false;
		for (int p = 0; p < P; p++)
		{
			if (finish[p] == 0)
			{
				int j;
				for (j = 0; j < R; j++)
					if (need[p][j] > work[j])
						break;
				if (j == R)
				{
					for (int k = 0 ; k < R ; k++)
						work[k] += allot[p][k];
					safeSeq[count++] = p;
					finish[p] = 1;
					found = true;
				}
			}
		}

		if (found == false)
		{
			cout << "System is not in safe state";
			return false;
```

```
                }
        }

        cout << "System is in safe state.\nSafe"
                " sequence is: ";
        for (int i = 0; i < P ; i++)
                cout << safeSeq[i] << " ";
        return true;
}

int main()
{
        int processes[] = {0, 1, 2, 3, 4};
        int avail[] = {3, 3, 2};
        int maxm[][R] = {{7, 5, 3},
                                                {3, 2, 2},
                                                {9, 0, 2},
                                                {2, 2, 2},
                                                {4, 3, 3}};
        int allot[][R] = {{0, 1, 0},
                                                {2, 0, 0},
                                                {3, 0, 2},
                                                {2, 1, 1},
                                                {0, 0, 2}};

        isSafe(processes, avail, maxm, allot);
        return 0;
}
```

OUTPUT:

```
System is in safe state.
Safe sequence is: 1 3 4 0 2
--------------------------------
Process exited after 0.09811 seconds with return value 0
Press any key to continue . . .
```

CONTENT BEYOND SYLLABUS 1

Aim: Write a program to implement producer - consumer problem using semaphores.

Theory: We have a buffer of fixed size. A producer can produce an item & can place it in buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer then at the same time consumer should not consume any item.

In this we need two counting semaphores - full & empty. "full" keeps track of no. of items in the buffer at any given time & "empty" keeps track of no. of ~~unoccupied~~ unoccupied slots.

Initialization of semaphores:-

```
mutex = 1
full = 0        // Initially all slots are empty.
empty = n       // All slots are empty initially.
```

Solution for producer:-

When producer produces an item then the value of empty is reduced by 1 because 1 slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item & thus the value of full is increased by 1.

The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

Solution for consumer :-
As consumer is removing an item from buffer, the value of full is reduced by 1 and the value of mutex is also reduced by 1 & the value of mutex is also reduced so that the producer can't access the buffer at this time. Now, the consumer has consumed the item, thus increasing the value of Empty by 1. The value of mutex is also increased so that producer can access the buffer now.

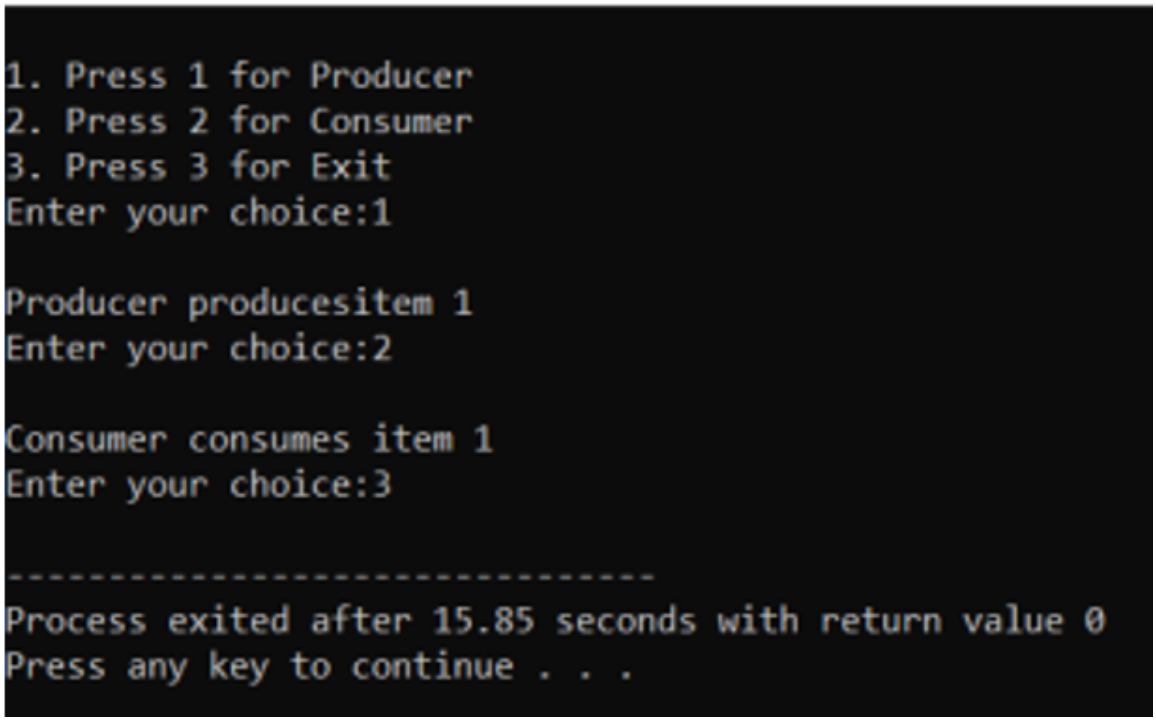
Result: The experiment has been studied & implemented successfully.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>

int mutex = 1;
int full = 0;
int empty = 10, x = 0;

void producer()
{
        --mutex;
        ++full;
        --empty;
        x++;
        printf("\nProducer produces" "item %d", x);
        ++mutex;
}

void consumer()
{
        --mutex;
        --full;
        ++empty;
        printf("\nConsumer consumes " "item %d", x);
        x--;
        ++mutex;
}

int main()
{
        int n, i;
        printf("\n1. Press 1 for Producer \n2. Press 2 for Consumer \n3. Press 3 for Exit");

#pragma omp critical

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

OUTPUT:



DELTA Pg No. 16
Date / /

CONTENT BEYOND SYLLABUS 2

Aim: Write a program to implement the concept of Dining - Philosophers problem.

Theory: The Dining Philosophers problem states that 'k' philosophers seated around a circular table with one chopstick between each pair of philosophers. A philosopher may eat if he can pick up the two chopsticks adjacent to them. One chopstick may be picked up by ~~one~~ any one of its adjacent followers but not both.

Pseudocode -

```
process P[i]
while true do {
        THINK;
        PICK UP [CHOPSTICK[i], CHOPSTICK[i+1 mod n]);
        EAT;
        PUT DOWN [CHOPSTICK[i], CHOPSTICK[i+1 mod n]);
}
```

There are 3 stages of philosophers: THINKING, HUNGRY & EATING. Here, there are two semaphores: Mutex & a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup & putdown at the same time. The array is used to control the behaviour of each philosopher. But, semaphores can result in deadlock.

Result: The experiment has been successfully studied & performed.

SOURCE CODE:

```
#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";

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

```
for(i=0;i<n;i++)
        goForDinner(i);
cout<<"\nTill now num of philosophers completed dinner are "<<compltedPhilo<<"\n\n";
}
return 0;
}
```

OUTPUT: