DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)
Shahabad Daulatpur, Bawana Road, Delhi- 110042

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS LAB FILE

CO-204

Submitted By:
Sandesh Shrestha
2K21/CO/417
A6 GROUP 2

Submitted To:
Prof. Minni Jain
Department of COE

INDEX: LIST OF PROGRAMS

11	Write a program to implement the following Memory Allocation Algorithms: a) First Fit b) Best Fit c) Worst Fit		
12	Write a program to implement First In First Out (FIFO) page replacement algorithm.		
13	Write a program to implement Least Recently Used (LRU) page replacement algorithm.		

Experiment 1

Program Objective: Write a program to implement First Come First Serve Job scheduling algorithm

Program theory:

First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which request the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve. As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue.

Example:

A real-life example of the FCFS method is buying a movie ticket on the ticket counter. In this scheduling algorithm, a person is served according to the queue manner. The person who arrives first in the queue first buys the ticket and then the next one. This will continue until the last person in the queue purchases the ticket. Using this algorithm, the CPU process works in a similar manner.

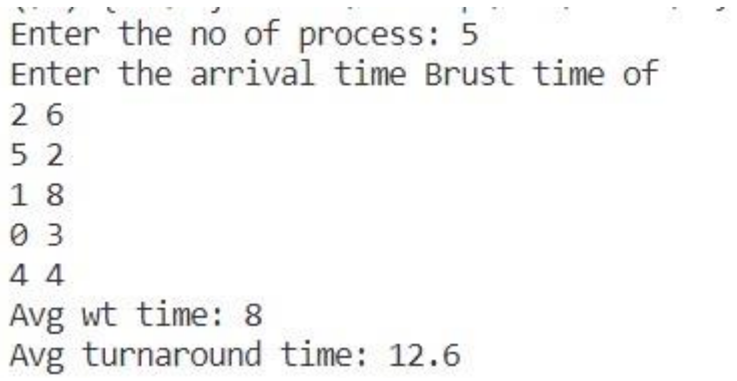
Algorithm

- Input the processes along with their burst time (bt).
- Find waiting time (wt) for all processes.
- As the first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.
- Find waiting time for all other processes i.e. for process i -> wt[i] = bt[i-1] + wt[i-1] .
- Find turnaround time = waiting_time + burst_time for all processes.
- Find average waiting time = total_waiting_time / no_of_processes.
- Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

Program Code:

```
#include <iostream>

#include <numeric>

#include <algorithm>

using namespace std;

struct Task{

int arr, burst, id; };
int main(){

    int n; cout<<"Enter the no of process: "; cin>>n; Task task[n];

    cout<<"Enter the arrival time Brust time of "; for (int i = 0;i<n;i++){
    cin>>task[i].arr>>task[i].burst; task[i].id = i+1;

    }

    sort(task, task+n, [&](Task a, Task b){return a.arr<b.arr;}); int st[n], en[n], wt[n], tot[n]; st[0] =
    0; en[0] = task[0].burst; wt[0] = 0; tot[0] = task[0].burst; for (int i = 1;i<n;i++){ st[i] =
    max(task[i].arr, en[i-1]); en[i] = st[i] + task[i].burst; wt[i] = st[i]-task[i].arr;

    tot[i] = en[i]-task[i].arr;

    }

    cout<<"Avg wt time: "<<accumulate(wt, wt+n, 0.0)/n<<endl;

    cout<<"Avg turnaround time: "<<accumulate(tot, tot+n,

    0.0)/n<<endl;

}
```

Output:



```
Enter the no of process: 5
Enter the arrival time Brust time of
2 6
5 2
1 8
0 3
4 4
Avg wt time: 8
Avg turnaround time: 12.6
```

Learning Outcome:

It is simple and easy to understand. The process with less execution time suffers i.e. waiting time is often quite long. Favors CPU Bound process then I/O bound process. Here, the first process will get the CPU first, other processes can get the CPU only after the current process has finished its execution. Now, suppose the first process has a large burst time, and other processes have less burst time, then the processes will have to wait more unnecessarily, this will result in *more average waiting time*, i.e., Convoy effect. This effect results in lower CPU and device utilization.

Experiment 2

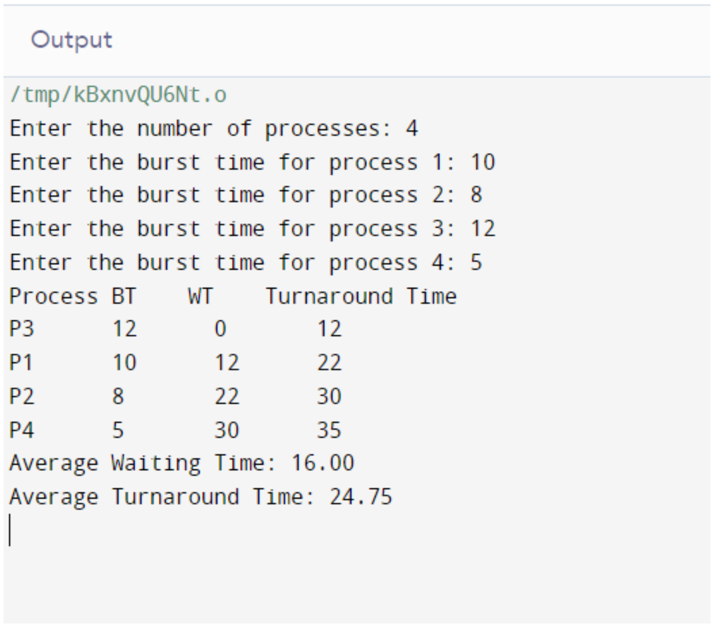
Program Objective: Write a program to implement Longest Job first scheduling algorithm
Program Theory:
The Longest Job First (LJF) scheduling algorithm is a non-preemptive scheduling algorithm where the process with the longest burst time is scheduled first. In this algorithm, the processes are sorted in descending order of their burst time, and the process with the highest burst time is executed first. This algorithm is suitable for situations where the processes have long execution times, as it reduces the average waiting time and turnaround time.

Algorithm:

- Input the number of processes and their burst times.
- Sort the processes in descending order of their burst times.
- Set the total waiting time and turnaround time to 0.
- For each process, do the following:
    - Execute the process.
    - Add the process's burst time to the total waiting time.
    - Calculate the process's turnaround time and add it to the total turnaround time.
- Calculate the average waiting time and average turnaround time.
- Print the results.

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

int main() {
    int n, i, j;
    float avg_wt = 0, avg_tat = 0;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int bt[n], wt[n], tat[n], p[n];
    for (i = 0; i < n; i++) {
        printf("Enter the burst time for process %d: ", i + 1);
        scanf("%d", &bt[i]);
        p[i] = i + 1;
    }
    // Sorting the processes in descending order of their burst times
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (bt[i] < bt[j]) {
                int temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;            }        }    }
    wt[0] = 0;
    tat[0] = bt[0];
    for (i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
        tat[i] = tat[i - 1] + bt[i];
    }
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
        avg_wt += wt[i];
        avg_tat += tat[i];    }
    avg_wt /= n;
    avg_tat /= n;    printf("Average Waiting Time: %.2f\n", avg_wt);    printf("Average Turnaround 
Time: %.2f\n", avg_tat);
    return 0;
}
```

Output:



Output

```
/tmp/kBxnvQU6Nt.o
Enter the number of processes: 4
Enter the burst time for process 1: 10
Enter the burst time for process 2: 8
Enter the burst time for process 3: 12
Enter the burst time for process 4: 5
Process BT    WT    Turnaround Time
P3      12      0       12
P1      10      12      22
P2      8       22      30
P4      5       30      35
Average Waiting Time: 16.00
Average Turnaround Time: 24.75
```

Learning Outcomes:

- How to implement the Longest Job First (LJF) scheduling algorithm.
- How to sort an array in descending order.
- How to calculate waiting time and turnaround time for each process.
- How to calculate the average waiting time and average turnaround time for all processes.

Experiment 3

Program Objective: Write a program to implement Shortest Job First Scheduling Algorithm

Program Theory:

Shortest job first is a scheduling algorithm in which the process with the smallest execution time is selected for execution next. Shortest job first can be either preemptive or non-preemptive. Owing to its simple nature, the shortest job first is considered optimal. It also reduces the average waiting time for other processes awaiting execution.

Shortest job first is also known as shortest job next (SJN) and shortest process next (SPN).

Example:

Online delivery apps always choose to deliver the nearest order first, then after delivering the first order, it searches for the next nearest delivery location.

Algorithm

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

Code

```
#include <iostream>

#include <algorithm>

#include <numeric>

using namespace std; const int INF = 1e9;

struct Task{ int arrival, burst, id; bool done; bool operator<(Task&t){

return burst<=t.burst;

} };

int main(){ int n; cout<<"Enter the no of process: "; cin>>n; Task task[n];

cout<<"Enter the arrival time, Brust time:\n"; for (int i = 0;i<n;i++){
cin>>task[i].arrival>>task[i].burst; task[i].id = i+1;

task[i].done = false; }

int en = 0; int wt = 0;
```

int tot = 0;

sort(task, task+n); cout<<"Gantt Chart: 0|";

for (int i = 0;i<n;i++){ // each iteration finish one task int first_arrived = INF;

int ind = 0; for (int j = 0;j<n;j++){ if (task[j].done) continue; if (task[j].arrival<=en){ ind = j;

break;

}

if (first_arrived>task[j].arrival){ ind = j;

first_arrived = task[j].arrival; }}

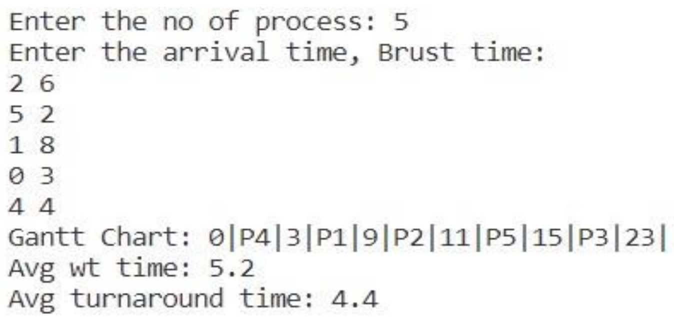
wt += max(en - task[ind].arrival, 0); en += task[ind].burst; tot = en - task[ind].arrival; task[ind].done = true;

cout<<"P"<<task[ind].id<<"|"<<en<<"|"; }

cout<<endl;

cout<<"Avg wt time: "<<(double)wt/n<<endl; cout<<"Avg turnaround time: "<<(double)tot/n<<endl; }

Output


```
Enter the no of process: 5
Enter the arrival time, Brust time:
2 6
5 2
1 8
0 3
4 4
Gantt Chart: 0|P4|3|P1|9|P2|11|P5|15|P3|23|
Avg wt time: 5.2
Avg turnaround time: 4.4
```

Learning Outcome

Shortest jobs are favored. It is probably optimal, in that it gives the minimum average waiting time for a given set of processes. SJF may cause starvation if shorter processes keep coming. This problem is solved by aging. It cannot be implemented at the level of short-term CPU scheduling.

Experiment 4

Program Objective: Write a program to implement SRTF Scheduling Algorithm (Preemptive version)

Program Theory

In the Shortest Remaining Time First (SRTF) scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

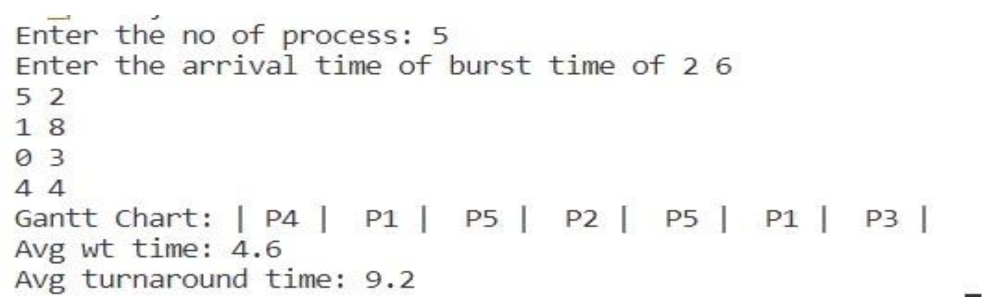
Algorithm

- Make a counter tim(variable) = from 0.
- For each tim, find the task with smallest burst time.
- Process the task and reduce the task’s burst time add the waiting time and turnaround time
- Mark task done if burst time is 0.
- Repeat the from 2 until all task are finished

Code

```
#include <iostream>

#include <algorithm>

#include <numeric>

#include <vector>


using namespace std; const int INF = 1e9;


struct Task{

int arrival, burst, id, wt, tot;

bool done;

};


int main(){ int n; cout<<"Enter the no of process: ";
cin>>n; Task task[n];

cout<<"Enter the arrival time of burst time of "; for (int i = 0;i<n;i++){
cin>>task[i].arrival>>task[i].burst; task[i].id = i+1; task[i].wt = task[i].tot = 0;

task[i].done = false;

}


int done = 0; vector<int> taskdone; for (int tim = 0;;tim++){ int mnburst = INF; int ind = 0; for (int
t = 0;t<n;t++){ if (task[t].done) continue;

if (task[t].arrival<=tim && mnburst>task[t].burst) mnburst = task[t].burst, ind = t;

}


task[ind].wt += tim - task[ind].arrival; task[ind].tot += tim - task[ind].arrival + 1; task[ind].burst--
; task[ind].arrival = tim + 1; taskdone.push_back(task[ind].id);

if (task[ind].burst==0) task[ind].done = true, done++;


if (done==n) break;

}

int tot_wt = 0, tot_tt = 0;

for (int i = 0;i<n;i++) tot_tt += task[i].tot, tot_wt += task[i].wt;

taskdone.erase(unique(taskdone.begin(), taskdone.end()), taskdone.end());


cout<<"Gantt Chart: |";

for (int x:taskdone) cout<<" P"<<x<<" | ";

cout<<endl;


cout<<"Avg wt time: "<<(double)tot_wt/n<<endl; cout<<"Avg turnaround time:
"<<(double)tot_tt/n<<endl;

}
```

Output



```
Enter the no of process: 5
Enter the arrival time of burst time of 2 6
5 2
1 8
0 3
4 4
Gantt Chart: | P4 |  P1 |  P5 |  P2 |  P5 |  P1 |  P3 |
Avg wt time: 4.6
Avg turnaround time: 9.2
```

Learning Outcome:

SRTF algorithm makes the processing of the jobs faster than SJN algorithm, given its overhead charges are not counted.

The context switch is done a lot more times in SRTF than in SJN, and consumes CPU's valuable time for processing. This adds up to its processing time and diminishes its advantage of fast processing.

Experiment 5

Program Objective: Write a program to implement Round Robin Scheduling Algorithm

Program Theory:

Round Robin scheduling algorithm is one of the most popular scheduling algorithms which can actually be implemented in most of the operating systems. This is the preemptive version of first come first serve scheduling. The Algorithm focuses on Time

Sharing. In this algorithm, every process gets executed in a cyclic way. A certain time

slice is defined in the system which is called time quantum. Each process present

in the ready queue is assigned the CPU for that time quantum, if the execution of the

process is completed during that time then the process will terminate else the process will

go back to the ready queue and waits for the next turn to complete the execution.

Algorithm

The algorithm fixes a time t

Each task according to arrival time is selected.

The selected task is executed for t time. And decrease the burst time by t. 4. If the task is not finished then pushed to end of queue

5. 2 to 4 is repeated until the queue is empty.

Code

```
#include <iostream>

#include <algorithm>

using namespace std;

 struct Process{

int id, arr, bru, wt, last; Process():wt(0){}

bool operator<(const Process& p){ return arr<p.arr;

} };

int main(){ int n; cout<<"Enter the no of process: "; cin>>n; int qntm; cout<<"Enter time quantum: ";cin>>qntm; Process p[n];

cout<<"Enter the arrival time, brust time:\n"; for (int i = 0;i<n;i++){ p[i].id = i+1; cin>>p[i].arr>>p[i].bru;
```

p[i].last = p[i].arr; }

sort(p, p+n); bool iscomplete = false; int tim = p[0].arr; while(!iscomplete){ bool all0 = true; for (int i = 0;i<n;i++){ if (p[i].bru) cout<<" | P"<<p[i].id; if (p[i].bru>qntm){ p[i].wt += tim-p[i].last; p[i].bru -= qntm; tim+=qntm; p[i].last = tim; }else if (p[i].bru!=0) { p[i].wt += tim-p[i].last; tim += p[i].bru; p[i].bru = 0;

p[i].last = tim; }

if (p[i].bru!=0) all0 = false; }

iscomplete = all0; }

cout<<" |\n"; int totwait = 0, totalturn = 0; for (int i = 0;i<n;i++){ totalturn += p[i].last-p[i].arr; totwait += p[i].wt;

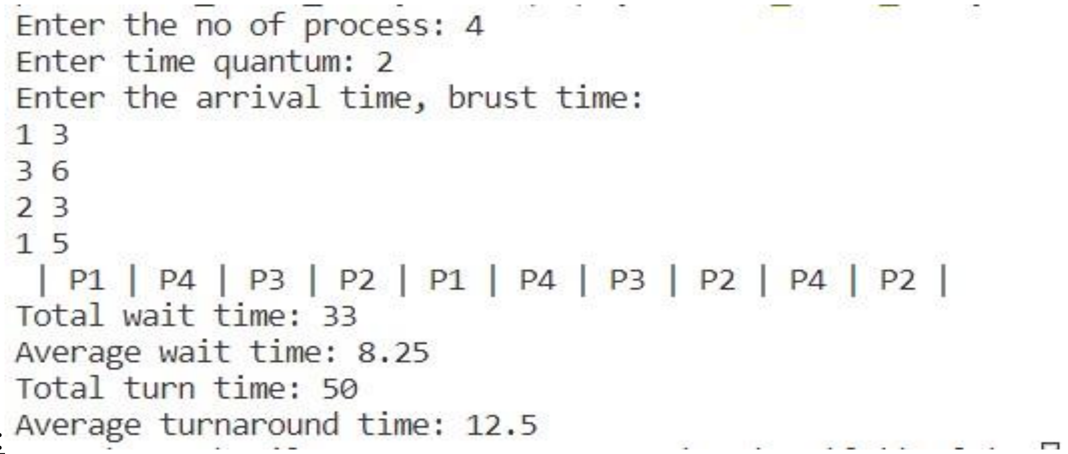
}

cout<<"Total wait time: "<<totwait<<endl; cout<<"Average wait time: "<<(float)totwait/n<<endl;

cout<<"Total turn time: "<<totalturn<<endl; cout<<"Average turnaround time:

"<<(float)totalturn/n<<endl; }

Output:



```
Enter the no of process: 4
Enter time quantum: 2
Enter the arrival time, brust time:
1 3
3 6
2 3
1 5
 | P1 | P4 | P3 | P2 | P1 | P4 | P3 | P2 | P4 | P2 |
Total wait time: 33
Average wait time: 8.25
Total turn time: 50
Average turnaround time: 12.5
```

Learning Outcome

Every process gets an equal share of the CPU. RR is cyclic in nature, so there is no starvation.

Setting the quantum too short increases the overhead and lowers the CPU efficiency, but setting it too long may cause a poor response to short processes. The average waiting time under the RR policy is often long. If the time quantum is very high then RR degrades to FCFS.

Experiment 6

Program Objective: Write a program to implement Priority Scheduling Algorithm

Program Theory

Priority Scheduling is a process scheduling algorithm based on priority where the scheduler selects tasks according to priority. Thus, processes with higher priority execute first followed by processes with lower priorities. If two jobs have the same priorities then the process that should execute first is chosen on the basis of roundrobin or FCFS. Which process should have what priority depends on a process' memory requirements, time requirements, the ratio of I/O burst to CPU burst, etc.

Algorithm

Sort the tasks according to priority and then by arrival time.

For each timestamp, tim(variable) selects the task with highest priority whose arrival time is less than tim (i.e select first unfinished task with arrival time not more than tim).

Do this task for a unit amount of time and reduce the burst time of this task. If remaining burst time is zero mark task as finished Repeat 2 until all tasks are finished.

Example:

Consider two task

Arrival time | Burst time | priority t1 0 | 2 | 3 t2 1 | 1 | 0

If we sort according to priority, then task will be like

Arrival time | Burst time | priority

t2 1 | 1 | 0 t1 0 | 2 | 3

Now at tim = 0, t1 arrival time <= tim so t1 will be done for 1 unit time

Now

Arrival time | Burst time | priority

t2 1 | 1 | 0

t1 0 | 1 | 3

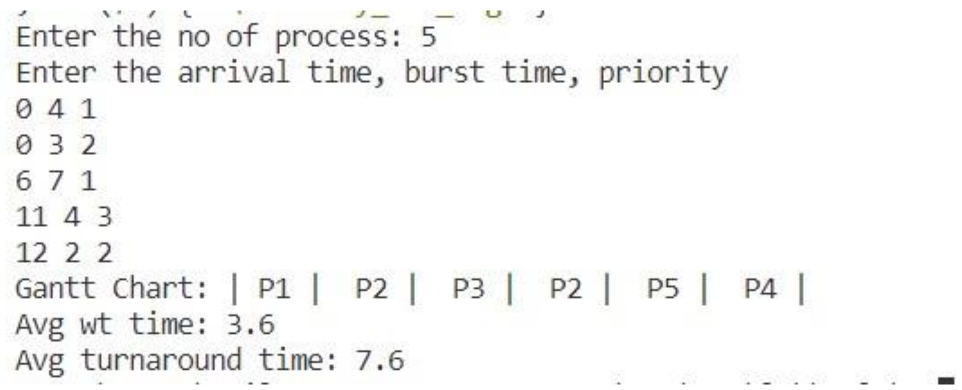
At tim = 1, t2 has arrived with higher priority so t2 will be done for one second.

Then at tim = 2, again t2 has finished, so t1 will be done. The gantt chart will be like t1   t2   t1 0 1   2   3

Code

```
#include <iostream>

#include <algorithm>

#include <numeric>

#include <vector>


using namespace std; const int INF = 1e9;


struct Task{ int arrival, burst, pri, id, wt, tot; bool done; bool operator<(Task&t){ if (pri!=t.pri)
return pri<=t.pri; return arrival<=t.arrival;

} };


int main(){ int n; cout<<"Enter the no of process: ";

cin>>n; Task task[n];

cout<<"Enter the arrival time, burst time, priority \n"; for (int i = 0;i<n;i++){

cin>>task[i].arrival>>task[i].burst>>task[i].pri; task[i].done = false; task[i].id = i+1; task[i].wt =
0;

task[i].tot = 0;

}


sort(task, task+n); int done = 0; vector<int> taskdone; for (int tim = 0;;tim++){ for (int t =
0;t<n;t++){ if (task[t].done) continue; if (task[t].arrival<=tim){ task[t].wt += tim - task[t].arrival;
task[t].tot += tim - task[t].arrival + 1; task[t].burst--;

taskdone.push_back(task[t].id); if (task[t].burst==0) task[t].done = true, done++; break;

}

}
if (done==n) break;

}

int tot_wt = 0, tot_tt = 0;

for (int i = 0;i<n;i++) tot_tt += task[i].tot, tot_wt += task[i].wt;

taskdone.erase(unique(taskdone.begin(),      taskdone.end()), taskdone.end());


cout<<"Gantt Chart: |";

for (int x:taskdone) cout<<" P"<<x<<" | "; cout<<endl;


cout<<"Avg wt time: "<<(double)tot_wt/n<<endl; cout<<"Avg turnaround time: "<<(double)tot_tt/n<<endl;

}
```

Output



```
Enter the no of process: 5
Enter the arrival time, burst time, priority
0 4 1
0 3 2
6 7 1
11 4 3
12 2 2
Gantt Chart: | P1 |  P2 |  P3 |  P2 |  P5 |  P4 |
Avg wt time: 3.6
Avg turnaround time: 7.6
```

Learning Outcome

This provides a good mechanism where the relative importance of each process may be precisely defined.

If high-priority processes use up a lot of CPU time, lower-priority processes may starve and be postponed indefinitely. The situation where a process never gets scheduled to run is called starvation.

Another problem is deciding which process gets which priority level assigned to it.

Experiment 7

Program Objective: Write a program to implement Producer-Consumer Problem
Program theory:
The Producer-Consumer problem is a classic synchronization problem where there are two processes, a producer and a consumer, that share a fixed-size buffer. The producer's job is to produce data and put it into the buffer, while the consumer's job is to take data out of the buffer and consume it. The challenge is to ensure that the producer and consumer don't access the buffer at the same time, which can lead to data corruption or deadlock.

Algorithm:

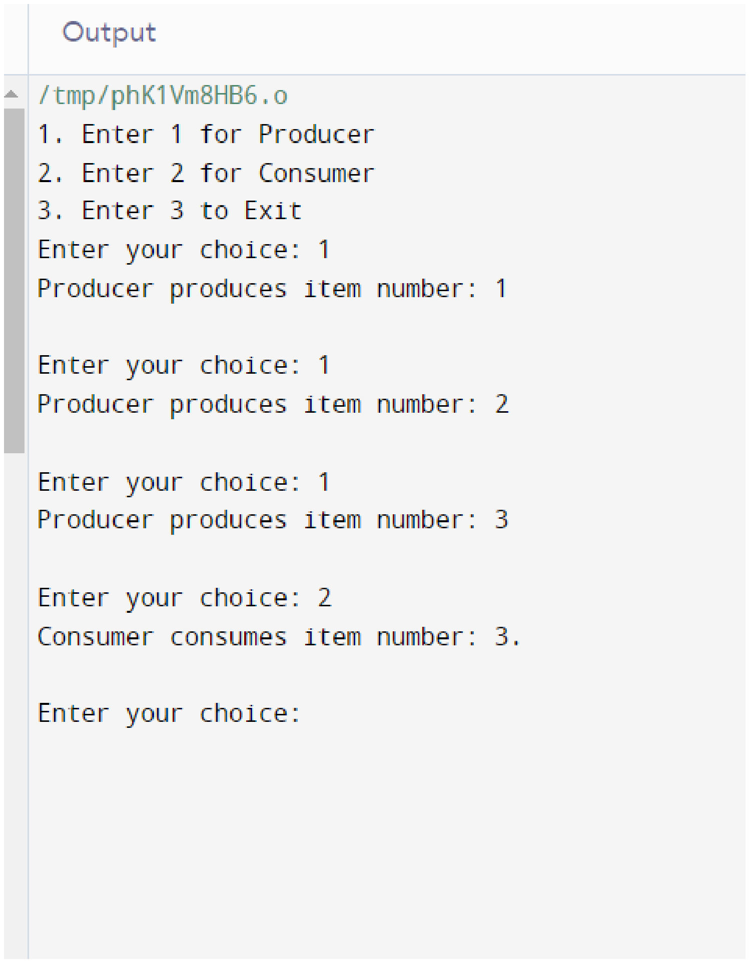
- Initialize buffer, mutex, and condition variables.
- Create producer and consumer threads.
- In the producer thread, while(true):
- Wait for the mutex to be available.
- If buffer is full, wait for the consumer to consume some data.
- Produce data and put it in the buffer.
- Notify the consumer that data is available.
- Release the mutex.
- In the consumer thread, while(true):
- Wait for the mutex to be available.
- If buffer is empty, wait for the producer to produce some data.
- Consume data from the buffer.
- Notify the producer that the buffer has space available.
- Release the mutex.

Code:

```
#include<iostream>

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

void producer(){
--mutex;
++full;
--empty;
data++;
printf("\nProducer produces item number: %d\n", data);
```

```
++mutex;
}

void consumer()
{
--mutex;
--full;
++empty;
printf("\nConsumer consumes item number: %d.\n", data);
data--;
++mutex;
}

int main()
{
int n, i;
printf("\n1. Enter 1 for Producer"
"\n2. Enter 2 for Consumer"
"\n3. Enter 3 to Exit");
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("The Buffer is full. New data cannot be produced!");
        }
        break;

    case 2:
        if ((mutex == 1) && (full != 0))
        {
            consumer();
        }
        else
        {
            printf("The Buffer is empty! New data cannot be consumed!");
        }
        break;

    case 3:
        exit(0);
        break;
    }
}
```

Output :



```
/tmp/phK1Vm8HB6.o
1. Enter 1 for Producer
2. Enter 2 for Consumer
3. Enter 3 to Exit
Enter your choice: 1
Producer produces item number: 1

Enter your choice: 1
Producer produces item number: 2

Enter your choice: 1
Producer produces item number: 3

Enter your choice: 2
Consumer consumes item number: 3.

Enter your choice:
```

Learning Outcomes :

The implementation of the producer-consumer problem provides an understanding of the classic synchronization problem in computer science. By completing this program, learners can learn about the challenges of multi-threading and how to use synchronization constructs to avoid race conditions and ensure thread safety. Additionally, this program can help learners to understand the different approaches for implementing producer-consumer problem and choose the one that suits the problem requirements best. Finally, learners can also learn about the importance of mutual exclusion and synchronization when multiple threads are accessing the shared buffer.

Experiment 8

Program Objective: Write a program to implement Dining Philosopher's problem

Program Theory:

The Dining Philosopher's problem is a classic synchronization problem in computer science. The problem involves a group of philosophers sitting at a table and trying to acquire chopsticks to eat their meal. There are n philosophers and n chopsticks, with each philosopher sitting between two chopsticks. To eat, a philosopher needs two chopsticks, one to their left and one to their right. The challenge is to design a solution to prevent deadlocks, where every philosopher holds a chopstick but is unable to acquire the other one, causing all of them to be stuck indefinitely.

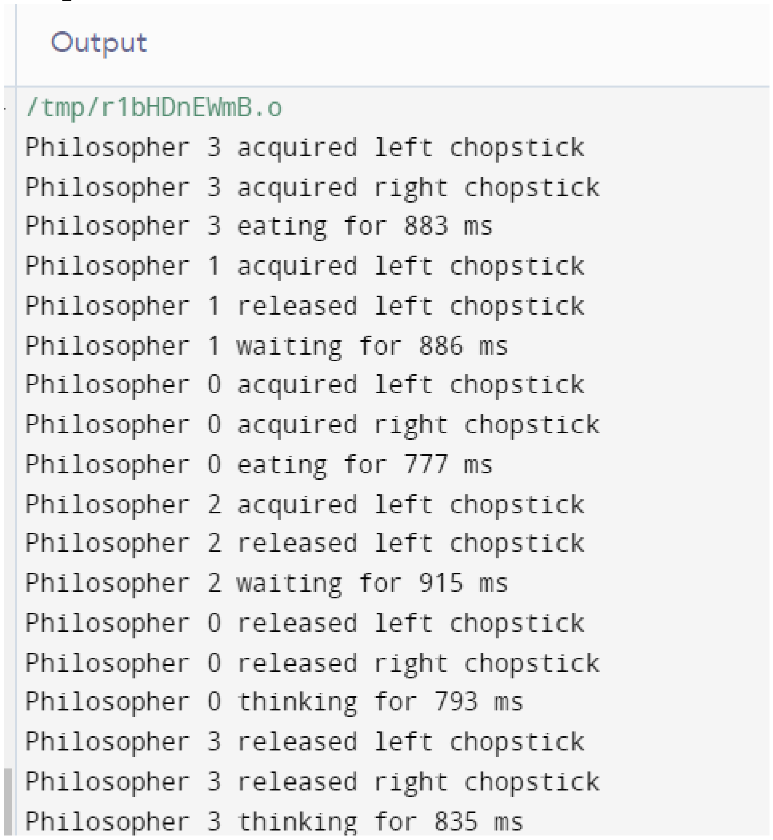
Algorithm:

- Create n philosophers and n chopsticks.
- Each philosopher repeatedly tries to acquire the chopsticks to their left and right.
- If a philosopher can acquire both chopsticks, they eat for a random amount of time and then release the chopsticks.
- If a philosopher cannot acquire both chopsticks, they release any chopsticks they are currently holding and wait for a random amount of time before trying again.

Code:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
#include <vector>

using namespace std;

const int num_philosophers = 5;
vector<mutex> chopsticks(num_philosophers);

void philosopher(int id) {
    while (true) {
        int left = id;
        int right = (id + 1) % num_philosophers;

        // Try to acquire left chopstick
        chopsticks[left].lock();
        cout << "Philosopher " << id << " acquired left chopstick" << endl;
        // Try to acquire right chopstick
        if (chopsticks[right].try_lock()) {
            cout << "Philosopher " << id << " acquired right chopstick" << endl;

            // Eat for a random amount of time
            int eat_time = rand() % 500 + 500;
            cout << "Philosopher " << id << " eating for " << eat_time << " ms" << endl;
            this_thread::sleep_for(chrono::milliseconds(eat_time));

            // Release chopsticks
            chopsticks[left].unlock();
            cout << "Philosopher " << id << " released left chopstick" << endl;
            chopsticks[right].unlock();
            cout << "Philosopher " << id << " released right chopstick" << endl;

            // Think for a random amount of time
            int think_time = rand() % 500 + 500;
            cout << "Philosopher " << id << " thinking for " << think_time << " ms" << endl;
            this_thread::sleep_for(chrono::milliseconds(think_time));
        } else {
            chopsticks[left].unlock();
            cout << "Philosopher " << id << " released left chopstick" << endl;

            // Wait for a random amount of time before trying again
            int wait_time = rand() % 500 + 500;
            cout << "Philosopher " << id << " waiting for " << wait_time << " ms" << endl;
            this_thread::sleep_for(chrono::milliseconds(wait_time));
        }
    }
}

int main() {
    vector<thread> philosophers;

    // Start the philosophers
    for (int i = 0; i < num_philosophers; i++) {
        philosophers.emplace_back(philosopher, i);
    }
    // Wait for the philosophers to finish
    for (auto& p : philosophers) {
        p.join();
    }

    return 0;
}
```

Output :



```
Output
/tmp/r1bHDnEWmB.o
Philosopher 3 acquired left chopstick
Philosopher 3 acquired right chopstick
Philosopher 3 eating for 883 ms
Philosopher 1 acquired left chopstick
Philosopher 1 released left chopstick
Philosopher 1 waiting for 886 ms
Philosopher 0 acquired left chopstick
Philosopher 0 acquired right chopstick
Philosopher 0 eating for 777 ms
Philosopher 2 acquired left chopstick
Philosopher 2 released left chopstick
Philosopher 2 waiting for 915 ms
Philosopher 0 released left chopstick
Philosopher 0 released right chopstick
Philosopher 0 thinking for 793 ms
Philosopher 3 released left chopstick
Philosopher 3 released right chopstick
Philosopher 3 thinking for 835 ms
```

Learning Outcomes:

The Dining Philosopher's problem is a classic synchronization problem in computer science that illustrates the challenges of coordinating multiple processes. Implementing a solution requires a deep understanding of concurrency, deadlock prevention, and mutual exclusion. By studying and implementing a solution to this problem, one can gain practical experience in designing efficient synchronization algorithms that prevent deadlocks and starvation, and learn to identify and resolve synchronization issues in concurrent systems. Additionally, this problem helps to improve problem-solving and analytical skills by presenting a complex problem that requires creative and innovative solutions.

Experiment 9

Program Objective: Write a program to create a child process implementing fork() system calls

Program Theory:

The fork() system call is used to create a new process, which is called the child process. The child process is an exact copy of the parent process, except for a few attributes, such as the process ID and the parent process ID. Both the parent and child processes run independently of each other.

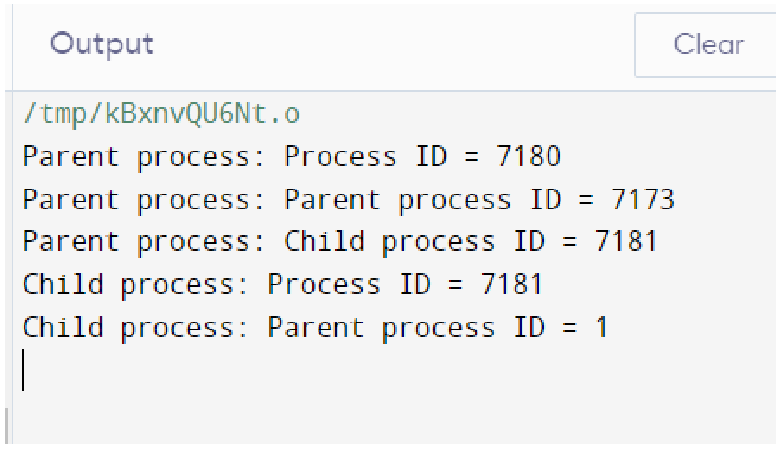
Algorithm:

- Call the fork() system call to create a child process.
- Check if the child process was created successfully.
- If the child process was created, the fork() system call returns 0 in the child process.
- In the child process, execute the child code.
- If the child process was not created, the fork() system call returns the process ID of the child in the parent process.
- In the parent process, execute the parent code.

Code:

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

int main() {
    pid_t pid;
    pid = fork();
    if (pid == -1) {
        printf("Error: Failed to fork a new process.\n");
        exit(1);
    }
    if (pid == 0) {
        printf("Child process: Process ID = %d\n", getpid());
        printf("Child process: Parent process ID = %d\n", getppid());
        // Child code
    } else {
        printf("Parent process: Process ID = %d\n", getpid());
        printf("Parent process: Parent process ID = %d\n", getppid());
        printf("Parent process: Child process ID = %d\n", pid);
        // Parent code
    }
    return 0;
}
```

Output:



```
/tmp/kBxnvQU6Nt.o
Parent process: Process ID = 7180
Parent process: Parent process ID = 7173
Parent process: Child process ID = 7181
Child process: Process ID = 7181
Child process: Parent process ID = 1
```

Learning Outcomes:

- How to create a child process using the fork() system call.
- How to check if the child process was created successfully.
- How to execute different code in the parent and child processes.
- How to use process IDs to identify the parent and child processes.

Experiment 10

Program Objective: Write a program to Implement Bankers Algorithm
Program Theory:
The Banker's algorithm is a resource allocation and deadlock avoidance algorithm that is used in operating systems. It is used to determine whether a request for resources can be safely granted to a process without causing a deadlock. The algorithm is based on the idea of a banker who only loans money to customers if the bank has enough money to cover all outstanding loans. Similarly, the operating system only grants resources to a process if there are enough resources available to fulfill the process's request.

Algorithm:

- Initialize the available resources, maximum resources, and allocation matrix.
- Calculate the need matrix by subtracting the allocation matrix from the maximum matrix.
- Initialize the finish array to false for all processes.
- Loop until all processes are finished or a deadlock is detected.
    - For each process, check if the need can be satisfied with the available resources.
    - If the need can be satisfied, grant the resources and mark the process as finished.
    - If the need cannot be satisfied, move to the next process.
    - If all processes cannot be finished, a deadlock is detected.

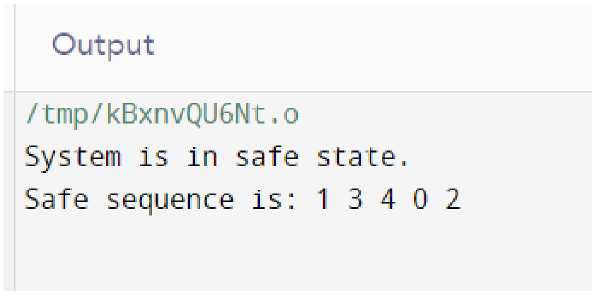
Code:
```
#include<iostream>
using namespace std;

// Number of processes
const int P = 5;

// Number of resources
const int R = 3;

// Function to find the need of each process
void calculateNeed(int need[P][R], int maxm[P][R],
                   int allot[P][R])
{
  // Calculating Need of each P
  for (int i = 0 ; i < P ; i++)
      for (int j = 0 ; j < R ; j++)

          // Need of instance = maxm instance -
```

```
            //                                        allocated instance
            need[i][j] = maxm[i][j] - allot[i][j];
}

// Function to find the system is in safe state or not
bool isSafe(int processes[], int avail[], int maxm[][R],
            int allot[][R])
{
  int need[P][R];

  // Function to calculate need matrix
  calculateNeed(need, maxm, allot);

  // Mark all processes as infinish
  bool finish[P] = {0};

  // To store safe sequence
  int safeSeq[P];

  // Make a copy of available resources
  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++)
      {
      // First check if a process is finished,
            // if no, go for next condition
            if (finish[p] == 0)
            {
                      int j;
                      for (j = 0; j < R; j++)
                                if (need[p][j] > work[j])
                                          break;

                      // If all needs of p were satisfied.
```

```
                    if (j == R)
                    {
                            // Add the allocated resources of
                            // current P to the available/work
                            // resources i.e.free the resources
                            for (int k = 0 ; k < R ; k++)
                                    work[k] += allot[p][k];

                            // Add this process to safe sequence.
                            safeSeq[count++] = p;

                            // Mark this p as finished
                            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;
}

// Driver code
int main()
{
    int processes[] = {0, 1, 2, 3, 4};

    // Available instances of resources
    int avail[] = {3, 3, 2};
```

```
  // Maximum R that can be allocated
  // to processes
  int maxm[][R] = {{7, 5, 3},
                          {3, 2, 2},
                          {9, 0, 2},
                          {2, 2, 2},
                          {4, 3, 3}};

  // Resources allocated to processes
  int allot[][R] = {{0, 1, 0},
                          {2, 0, 0},
                          {3, 0, 2},
                          {2, 1, 1},
                          {0, 0, 2}};

  // Check system is in safe state or not
  isSafe(processes, avail, maxm, allot);

  return 0;
}
```

Output:



Output

```
/tmp/kBxnvQU6Nt.o
System is in safe state.
Safe sequence is: 1 3 4 0 2
```

Learning Outcomes :

By implementing the Banker's Algorithm, the code provides a practical example of a solution to the problem of resource allocation and deadlock avoidance. It demonstrates how to use a systematic approach to ensure that resources are allocated safely and efficiently, and that the system avoids deadlocks. As such, this code is a useful tool for understanding the Banker's Algorithm and its application in operating systems.

Experiment 11

Program Objective: Write a program to implement the following Memory Allocation Algorithms:

a)First Fit b) Best Fit c) Worst Fit

First fit

Program Theory

First-Fit Allocation is a memory allocation technique used in operating systems to allocate memory to a process. In First-Fit, the operating system searches through the list of free blocks of memory, starting from the beginning of the list, until it finds a block that is large enough to accommodate the memory request from the process. Once a suitable block is found, the operating system splits the block into two parts: the portion that will be allocated to the process, and the remaining free block.

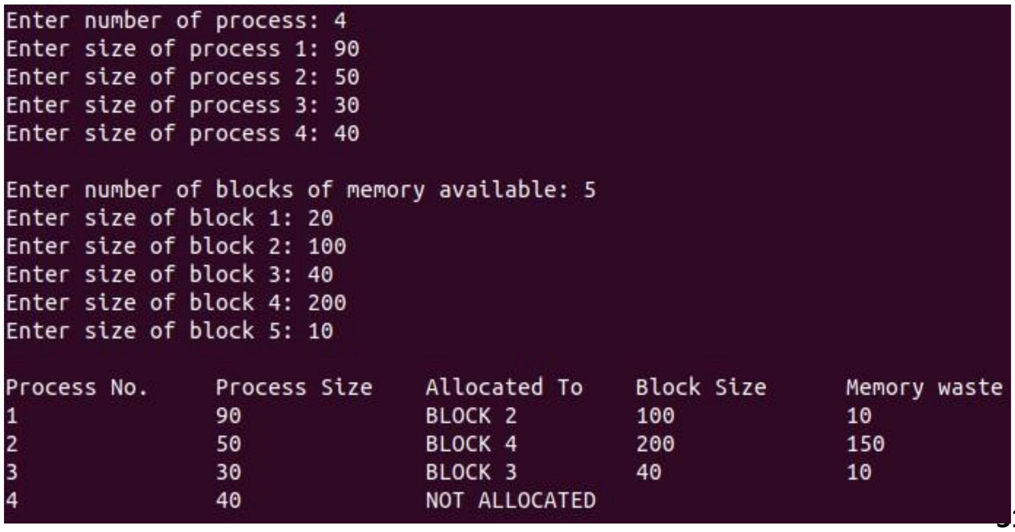
Algorithm

- For process 1, check for the block which fits the size of process 1. i.e size[process1] <= size[blockX] ; X is a random block.
- If the block is not occupied by any processes, then allocate that block to the process.
- Repeat the same for all the processes.
- If a certain process doesn't find any block, it will be unallocated.

Code

```
#include<iostream>

#include<vector>

using namespace std;

int main(){ int processes, blocks;

cout << "Enter number of process: "; cin >> processes; int process_size[processes]; for(int i = 0; i
< processes; i++){

cout << "Enter size of process " << i+1 << ": "; cin >> process_size[i];

}

cout << "\n";
cout << "Enter number of blocks of memory available: "; cin >> blocks; int block_size[blocks];
for(int i = 0; i < blocks; i++){

cout << "Enter size of block " << i+1 << ": "; cin >> block_size[i];

}

vector<int> allocated(processes, -1); vector<int> occupied(blocks, 0); vector<int>
waste_mem(processes);

for(int p = 0; p < processes; p++){ for(int b = 0; b < blocks; b++){ if(!occupied[b] &&
block_size[b] >= process_size[p]){ occupied[b] = 1; allocated[p] = b+1;

waste_mem[p] = block_size[b]-process_size[p]; break;

}

}

}

cout << "\nProcess No.\tProcess Size\tAllocated To\tBlock Size\tMemory waste\n"; for(int i = 0; i
< processes; i++){

cout << i+1 << "\t\t" << process_size[i] << "\t\t"; if(allocated[i] != -1){

cout << "BLOCK " << allocated[i] << "\t\t"; cout << block_size[allocated[i]-1] << "\t\t";

cout << waste_mem[i] << "\n";

}

else{

cout << "NOT ALLOCATED\n";

}

}

return 0;

}
```

Output



```
Enter number of process: 4
Enter size of process 1: 90
Enter size of process 2: 50
Enter size of process 3: 30
Enter size of process 4: 40

Enter number of blocks of memory available: 5
Enter size of block 1: 20
Enter size of block 2: 100
Enter size of block 3: 40
Enter size of block 4: 200
Enter size of block 5: 10

Process No.     Process Size    Allocated To    Block Size      Memory waste
1               90              BLOCK 2         100             10
2               50              BLOCK 4         200             150
3               30              BLOCK 3         40              10
4               40              NOT ALLOCATED
```

Learning Outcome

Simple and efficient search algorithm. Minimizes memory fragmentation. Fast allocation of memory. Poor performance in highly fragmented memory. May lead to poor memory utilization. May allocate larger blocks of memory than required.

Best Fit:

Program Theory

Best-Fit Allocation is a memory allocation technique used in operating systems to allocate memory to a process. In Best-Fit, the operating system searches through the

list of free blocks of memory to find the block that is closest in size to the memory request from the process. Once a suitable block is found, the operating system splits the block into two parts: the portion that will be allocated to the process, and the remaining free block.

Algorithm

- For process 1, find a block of memory satisfying the condition : size[blockX] >= size[process1]; X is a random block.
- Find the memory waste of that block i.e. memory waste = size[blockX]size[process1].
- Find memory waste for all the blocks.
- Assign the minimum waste memory block to the process 1.
- Similarly, repeat this for all the processes.
- If a process doesn't find any block, that process is marked unallocated.

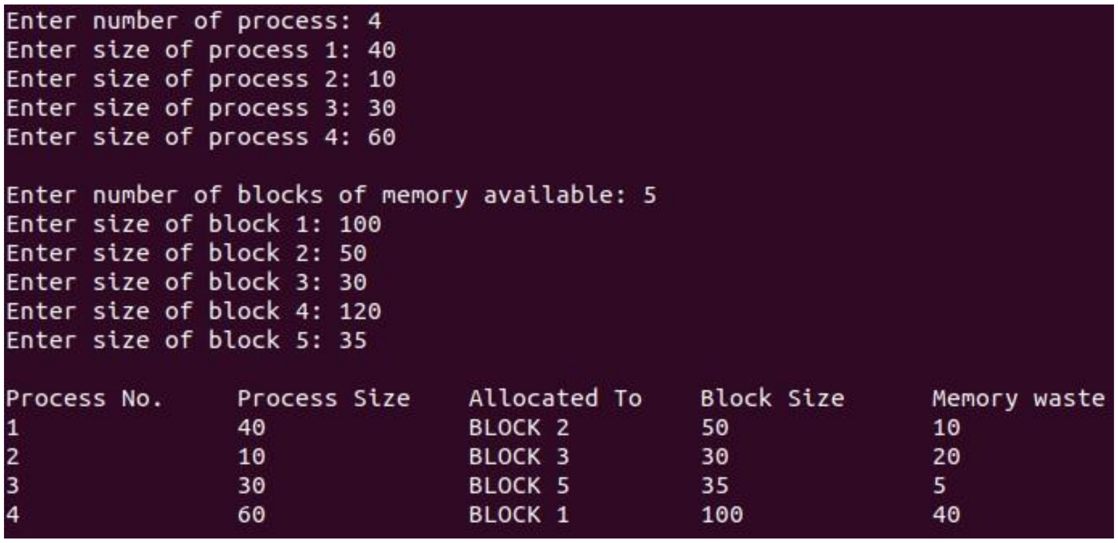
Code

```
#include<iostream>

#include<climits> #include<vector> using namespace std;


int main(){ int processes, blocks;

cout << "Enter number of process: "; cin >> processes; int process_size[processes]; for(int i = 0; i < processes; i++){

cout << "Enter size of process " << i+1 << ": "; cin >> process_size[i];
```

```
}

cout << "\n";

cout << "Enter number of blocks of memory available: "; cin >> blocks; int block_size[blocks];
for(int i = 0; i < blocks; i++){

cout << "Enter size of block " << i+1 << ": "; cin >> block_size[i];

}


vector<int> allocated(processes, -1); vector<int> occupied(blocks, 0); vector<int>
waste_mem(processes);


for(int p = 0; p < processes; p++){ int curr_min_waste = INT_MAX; int min_waste_block = 0;
for(int b = 0; b < blocks; b++){

if(!occupied[b] && block_size[b] >= process_size[p]){ if(block_size[b] - process_size[p] <
curr_min_waste){ curr_min_waste = block_size[b]-process_size[p]; min_waste_block = b;

}

}

}

if(!occupied[min_waste_block]){ occupied[min_waste_block] = 1; allocated[p] =
min_waste_block+1;

waste_mem[p] = block_size[min_waste_block]-process_size[p];

}

}

cout << "\nProcess No.\tProcess Size\tAllocated To\tBlock Size\tMemory waste\n"; for(int i = 0; i
< processes; i++){

cout << i+1 << "\t\t" << process_size[i] << "\t\t"; if(allocated[i] != -1){

cout << "BLOCK " << allocated[i] << "\t\t"; cout << block_size[allocated[i]-1] << "\t\t";

cout << waste_mem[i] << "\n";

}

else{
```

```
cout << "NOT ALLOCATED\n";
}
}
return 0;
}
```

Output



```
Enter number of process: 4
Enter size of process 1: 40
Enter size of process 2: 10
Enter size of process 3: 30
Enter size of process 4: 60

Enter number of blocks of memory available: 5
Enter size of block 1: 100
Enter size of block 2: 50
Enter size of block 3: 30
Enter size of block 4: 120
Enter size of block 5: 35

Process No.     Process Size    Allocated To    Block Size      Memory waste
1               40              BLOCK 2         50              10
2               10              BLOCK 3         30              20
3               30              BLOCK 5         35              5
4               60              BLOCK 1         100             40
```

Learning Outcome

Memory Efficient. The operating system allocates the job minimum possible space in the memory, making memory management very efficient. To save memory from getting wasted, it is the best method. Improved memory utilization. Reduced memory fragmentation. Minimizes external fragmentation It is a Slow Process. Checking the whole memory for each job makes the working of the operating system very slow. It takes a lot of time to complete the work. Increased computational overhead. May lead to increased internal fragmentation. Can result in slow memory allocation times.

Worst Fit

Program Theory

In this allocation technique, the process traverses the whole memory and always searches for the largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search the largest hole.

Algorithm

- For process 1, find a block of memory satisfying the condition : size[blockX] >= size[process1]; X is a random block.

- Find the memory waste of that block i.e. memory waste = size[blockX]size[process1].
- Find memory waste for all the blocks.
- Assign the maximum waste memory block to the process 1.
- Similarly, repeat this for all the processes.
- If a process doesn’t find any block, that process is marked unallocated.

Code

```
#include<iostream>

#include<climits> #include<vector> using namespace std;


int main(){ int processes, blocks;

cout << "Enter number of process: "; cin >> processes; int process_size[processes]; for(int i = 0; i
< processes; i++){

cout << "Enter size of process " << i+1 << ": ";

cin >> process_size[i];

}

cout << "\n";

cout << "Enter number of blocks of memory available: "; cin >> blocks; int block_size[blocks];
for(int i = 0; i < blocks; i++){

cout << "Enter size of block " << i+1 << ": "; cin >> block_size[i];

}


vector<int> allocated(processes, -1); vector<int> occupied(blocks, 0); vector<int>
waste_mem(processes);


for(int p = 0; p < processes; p++){ int curr_max_waste = INT_MIN; int max_waste_block = 0;
for(int b = 0; b < blocks; b++){ if(!occupied[b] && block_size[b] >= process_size[p]){
if(block_size[b] - process_size[p] > curr_max_waste){ curr_max_waste = block_size[b]-
process_size[p]; max_waste_block = b;

} }}
```

if(!occupied[max_waste_block]){ occupied[max_waste_block] = 1; allocated[p] = max_waste_block+1;

waste_mem[p] = block_size[max_waste_block]-process_size[p]; }}

cout << "\nProcess No.\tProcess Size\tAllocated To\tBlock Size\tMemory waste\n"; for(int i = 0; i < processes; i++){

cout << i+1 << "\t\t" << process_size[i] << "\t\t"; if(allocated[i] != -1){

cout << "BLOCK " << allocated[i] << "\t\t"; cout << block_size[allocated[i]-1] << "\t\t";

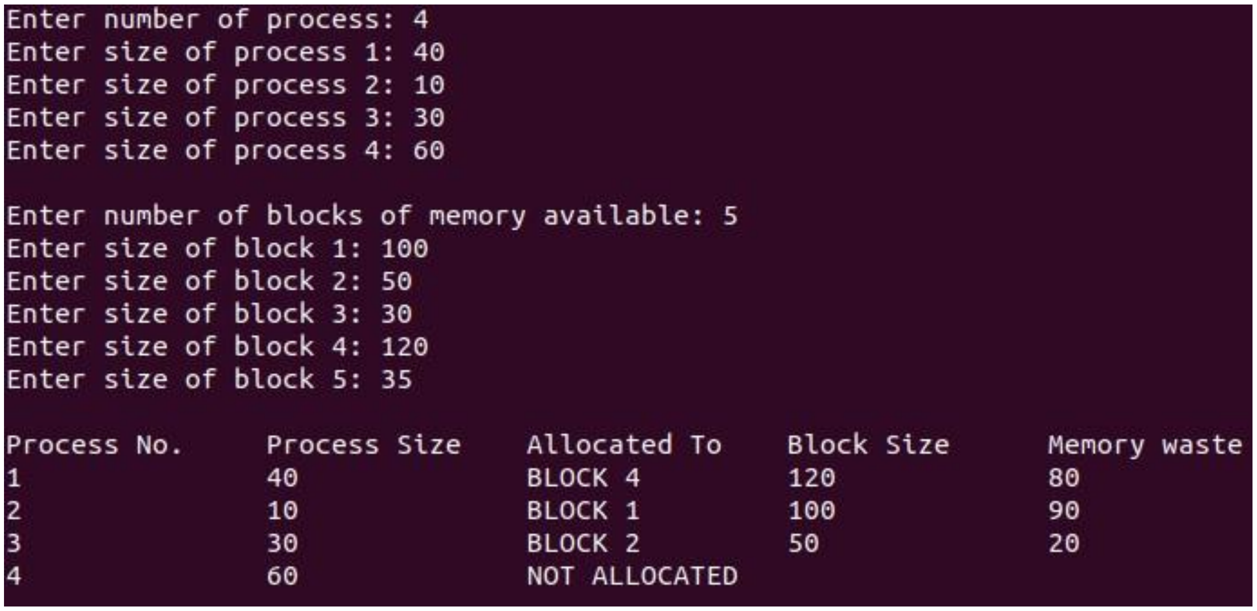
cout << waste_mem[i] << "\n"; }

else{

cout << "NOT ALLOCATED\n"; }}

return 0; }

Output



```
Enter number of process: 4
Enter size of process 1: 40
Enter size of process 2: 10
Enter size of process 3: 30
Enter size of process 4: 60

Enter number of blocks of memory available: 5
Enter size of block 1: 100
Enter size of block 2: 50
Enter size of block 3: 30
Enter size of block 4: 120
Enter size of block 5: 35

Process No.     Process Size    Allocated To    Block Size      Memory waste
1               40              BLOCK 4         120             80
2               10              BLOCK 1         100             90
3               30              BLOCK 2         50              20
4               60              NOT ALLOCATED
```

Learning Outcome

Since this process chooses the largest hole/partition, therefore there will be large internal fragmentation. Now, this internal fragmentation will be quite big so that other small processes can also be placed in that leftover partition. It is a slow process because it traverses all the partitions in the memory and then selects the largest partition among all the partitions, which is a time-consuming process.

Experiment 12

Program Objective: Write a program to implement First In First Out (FIFO) page replacement algorithm.

Program theory:

The First In First Out (FIFO) page replacement algorithm is the simplest page replacement algorithm. In this algorithm, the operating system maintains a list of all pages in memory, with the oldest page at the front of the list and the newest page at the end of the list. Whenever a page fault occurs, the operating system replaces the oldest page (i.e., the page at the front of the list) with the new page. This algorithm is simple and easy to implement, but it may not be the most efficient algorithm in terms of page fault rate.

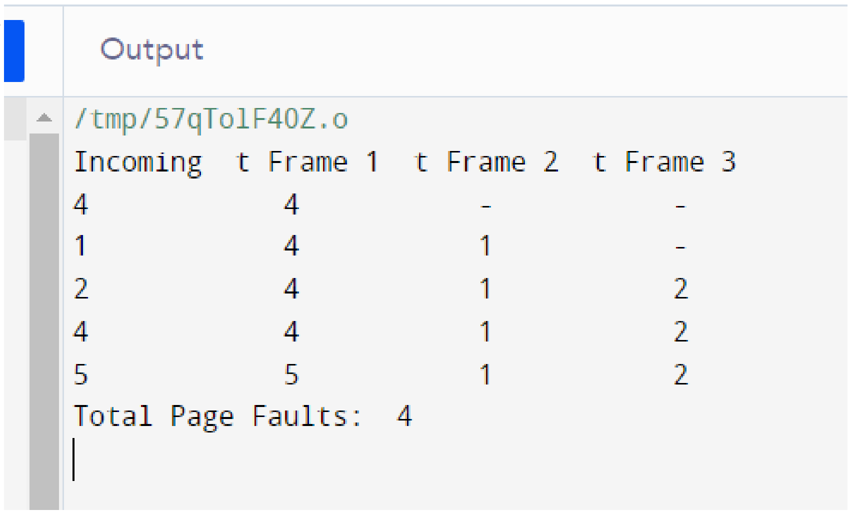
Algorithm:

- Initialize an empty list to represent the page frame.
- For each page reference, do the following:
    - If the page is not in the page frame, add the page to the end of the page frame list.
    - If the page is already in the page frame, do nothing.
    - If the page frame list is full, remove the first page from the list (i.e., the oldest page) and add the new page to the end of the list.
- Repeat step 2 for all page references.
- Count the number of page faults.

Code :

```
#include < stdio.h >
int main()
{
   int incomingStream[] = {4 , 1 , 2 , 4 , 5};
   int pageFaults = 0;
   int frames = 3;
   int m, n, s, pages;
   pages = sizeof(incomingStream)/sizeof(incomingStream[0]);
   printf(" Incoming \ t Frame 1 \ t Frame 2 \ t Frame 3 ");
   int temp[ frames ];
   for(m = 0; m < frames; m++)
   {
      temp[m] = -1;
   }
   for(m = 0; m < pages; m++)
   {
      s = 0;
      for(n = 0; n < frames; n++)
      {
        if(incomingStream[m] == temp[n])
        {
          s++;
          pageFaults--;           }         }
      pageFaults++;
      if((pageFaults <= frames) && (s == 0))
      {
        temp[m] = incomingStream[m];
      }
      else if(s == 0)
      {
        temp[(pageFaults - 1) % frames] = incomingStream[m];          }
      printf("\n");
      printf("%d\t\t\t",incomingStream[m]);
      for(n = 0; n < frames; n++)          {
        if(temp[n] != -1)
          printf(" %d\t\t\t", temp[n]);
        else
          printf(" - \t\t\t");          }      }
   printf("\nTotal Page Faults
}
```

Output:



```
/tmp/57qTolF40Z.o
Incoming  t Frame 1  t Frame 2  t Frame 3
4           4           -           -
1           4           1           -
2           4           1           2
4           4           1           2
5           5           1           2
Total Page Faults:  4
```

Learning Outcomes:

This program can help to understand how the FIFO algorithm works, how to implement it in code, and how to test its performance using different page reference sequences. Through this program, learners can also gain insights into operating system concepts like page faults, page hits, and the working set of a process. The program can help learners to develop their skills in problem-solving, programming, and algorithm design, as well as to improve their understanding of operating systems and computer architecture.

Experiment 13

Program Objective: Write a program to implement Least Recently Used (LRU) Page replacement Algorithm

Program theory:

The Least Recently Used (LRU) algorithm is a page replacement algorithm in operating systems that replaces the least recently used page from the memory when the page table is full. It works on the principle of locality of reference, which states that the pages that are recently used are more likely to be used again in the near future. In this algorithm, each page is assigned a counter that keeps track of the time of the last access. When a page needs to be replaced, the page with the lowest counter value (least recently used) is chosen for replacement.

Algorithm:

- Initialize an empty list (queue) to keep track of the pages in the memory.
- For each page request, do the following:
    - If the page is already in the memory, mark it as accessed and update its access time.
    - If the page is not in the memory and there is an empty slot, add the page to the memory and mark its access time.
    - If the page is not in the memory and there is no empty slot, remove the least recently used page from the memory and add the new page to the memory.

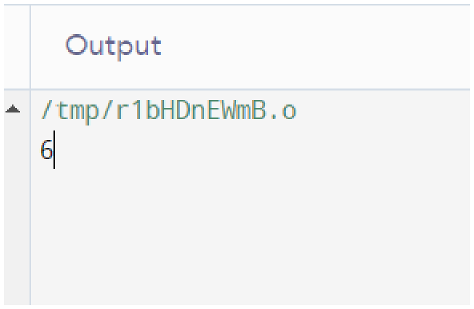
Code:

```
//C++ implementation of above algorithm
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using indexes
int pageFaults(int pages[], int n, int capacity)
{
  // To represent set of current pages. We use
  // an unordered_set so that we quickly check
  // if a page is present in set or not
  unordered_set<int> s;

  // To store least recently used indexes
  // of pages.
  unordered_map<int, int> indexes;

  // Start from initial page
```

```
int page_faults = 0;
for (int i=0; i<n; i++)
{
    // Check if the set can hold more pages
    if (s.size() < capacity)
    {
        // Insert it into set if not present
        // already which represents page fault
        if (s.find(pages[i])==s.end())
        {
            s.insert(pages[i]);

            // increment page fault
            page_faults++;
        }

        // Store the recently used index of
        // each page
        indexes[pages[i]] = i;
    }

    // If the set is full then need to perform lru
    // i.e. remove the least recently used page
    // and insert the current page
    else
    {
        // Check if current page is not already
        // present in the set
        if (s.find(pages[i]) == s.end())
        {
            // Find the least recently used pages
            // that is present in the set
            int lru = INT_MAX, val;
            for (auto it=s.begin(); it!=s.end(); it++)
            {
                if (indexes[*it] < lru)
                {
                    lru = indexes[*it];
                    val = *it;
                }
```

```
                    }

                    // Remove the indexes page
                    s.erase(val);

                    // insert the current page
                    s.insert(pages[i]);

                    // Increment page faults
                    page_faults++;
               }

               // Update the current page index
               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 :



Output

```
/tmp/r1bHDnEWmB.o
6
```

Learning Outcomes:

After completing this program, you should have gained a better understanding of how the Least Recently Used (LRU) Page Replacement Algorithm works and how it can be implemented in C++. You should also have improved your knowledge of vector manipulation and searching algorithms in C++. Additionally, you may have gained experience in analyzing and optimizing algorithms for better performance.