OS Simulator

23.04.2021

—

191CS231 Keerthana Patil
191CS232 P Shiva
191CS233 Pooja G
191CS234 Pranav DV
191CS235 Praneeth G
191CS236 Priyanka G Pai
191CS237 Jithendra Puppala
191CS238 Radhika Suradkar
191CS239 Rahul Kumar
191CS240 Rajat Partani

Overview

The following reports summarizes the work done for building the OS simulator that served as a mini-project as part of course CS255- Operating Systems Lab.

Goals

The major goal of this simulator is to put together various functions of an Operating Systems learnt throughout the duration of this semester and to demonstrate how an Operating System functions behind the scenes.

Specifications

The entire project was coded in C++. The user is presented with a menu displaying all the functions being implemented.

Topics Covered

1. Scheduling
2. Synchronization
3. Deadlock Avoidance
4. Memory Management
5. Page Replacement
6. Disk Scheduling

Contribution

S No	Member Name	Contribution
1	Keerthana Patil	-
2	P Shiva	HRRN (CPU Scheduling), Paging
3	Pooja G	MFT
4	Pranav DV	Page Replacement
5	Praneeth G	CPU Scheduling
6	Priyanka G Pai	Banker's Algorithm, MFT
7	Jithendra Puppala	-
8	Radhika Suradkar	MVT
9	Rahul Kumar	Disk Scheduling Algorithms
10	Rajat Partani	Process Synchronization, Paging

Content

CPU Scheduling

CPU Scheduling is the process of determining to which the CPU will be allocated while another process is on hold. The main idea behind CPU scheduling is to make sure the CPU never remains idle. There should always be at least one process always in the ready queue which can be served to the CPU for execution.

CPU Scheduling aims to attains two major goals:

- Maximize CPU Utilization and the number of processes being executed per unit time. In other words, keeping the CPU busy at all times is of utmost importance. CPU remaining idle is nothing but a waste of available resources.
- Minimize the waiting time and response time. This is to provide a smooth user experience. No process in the ready queue should have to wait a long time to be allocated the CPU and the response time should be as low as possible.

Terminologies involved:

1. Process: Program which has been brought into the ready queue
2. Burst Time: Time required by a process to finish execution.
3. Arrival time: The time at which a process enters it's ready state.
4. Finish time: The time a process finishes its execution.
5. Waiting time: The time a process has to wait in the ready queue.
6. Turnaround time: The time taken to execute a specific process.
7. Response time: The time in which a process produces its first response.

Types of CPU Scheduling Algorithm

1. FCFS
2. Shortest Job First
3. Longest Job First
4. Shortest Remaining Time First
5. Round Robin
6. High Response Ratio Next

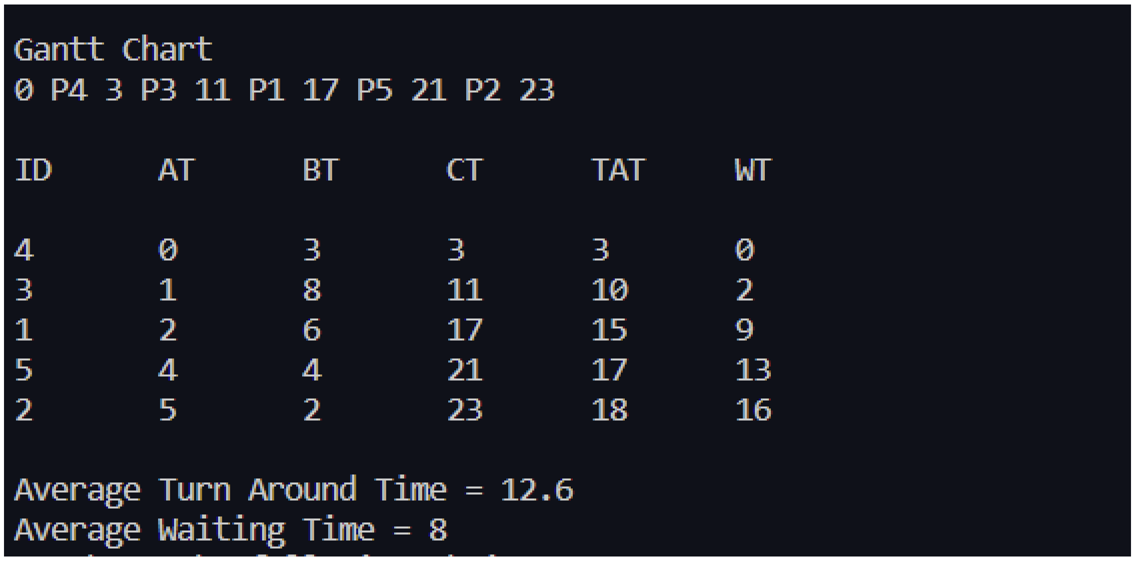
FCFS (First Come First Serve)

This CPU scheduling algorithm automatically executes queued requests and processes them in order of their arrival. In this type of algorithm, the processes which request the algorithm first get allocation of the CPU first and subsequent processes are executed. This order is maintained using a FIFO queue.

Consider the following process table:

Process	Arrival time	Burst time
P1	2	6
P2	5	2
P3	1	8
P4	0	3
P5	4	4

When the processes are executed in FIFO order, following is the final status of the queue.



```
Gantt Chart
0 P4 3 P3 11 P1 17 P5 21 P2 23

ID      AT      BT      CT      TAT     WT

4       0       3       3       3       0
3       1       8       11      10      2
1       2       6       17      15      9
5       4       4       21      17      13
2       5       2       23      18      16

Average Turn Around Time = 12.6
Average Waiting Time = 8
```

Shortest Job First

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution.

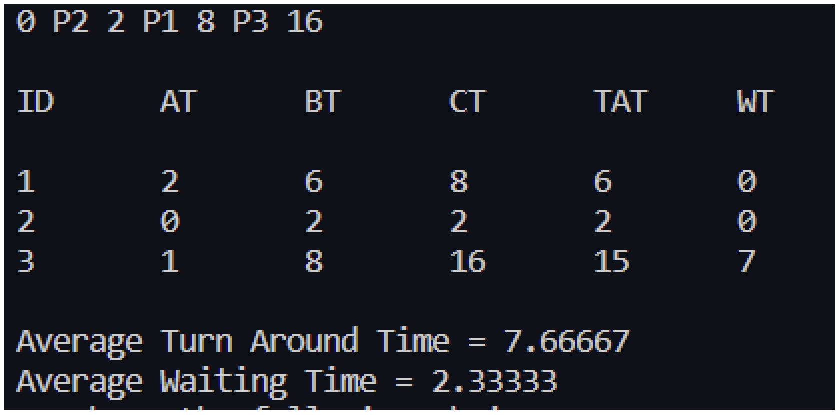
There are two types of SJF- Preemptive and Nonpreemtive.

We will first look at the non preemptive one.

In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.

Consider the following process table:

Process	Arrival time	Burst time
P1	2	6
P2	0	2
P3	1	8

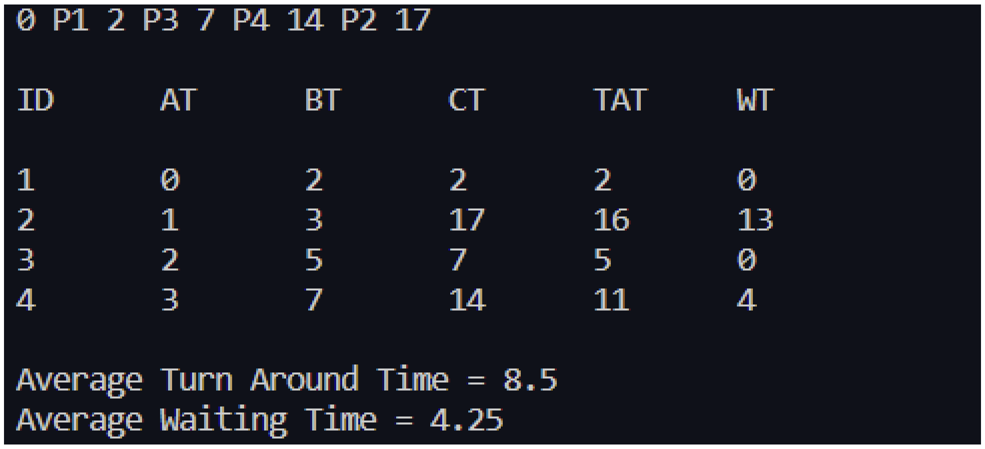


```
0 P2 2 P1 8 P3 16

ID      AT      BT      CT      TAT     WT

1       2       6       8       6       0
2       0       2       2       2       0
3       1       8       16      15      7

Average Turn Around Time = 7.66667
Average Waiting Time = 2.33333
```

Longest Job First

This scheduling is similar to the SJF scheduling algorithm. But, in this scheduling algorithm, the priority is given to the process having the longest burst time.

Process	Arrival Time	Burst Time
P1	0	2
P2	1	3
P3	2	5
P4	3	7

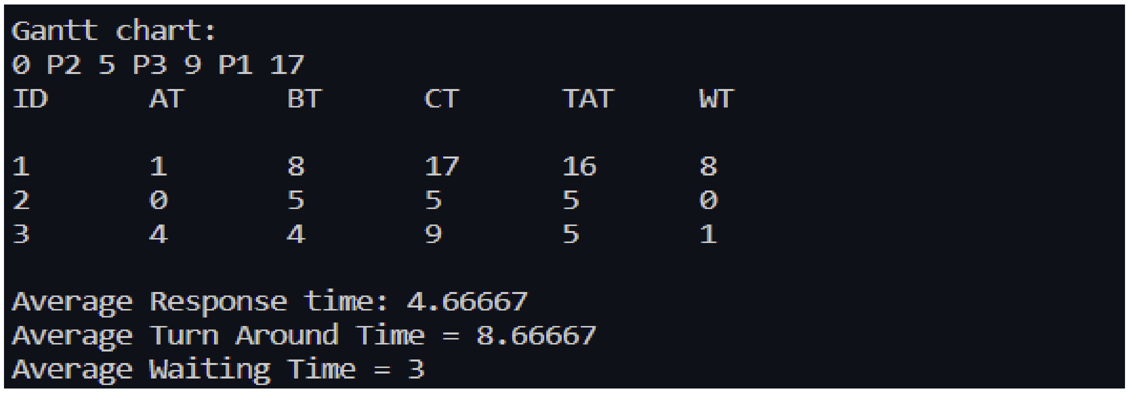


```
0 P1 2 P3 7 P4 14 P2 17

ID      AT      BT      CT      TAT     WT

1       0       2       2       2       0
2       1       3       17      16      13
3       2       5       7       5       0
4       3       7       14      11      4

Average Turn Around Time = 8.5
Average Waiting Time = 4.25
```

Shortest Remaining Time First

This is the preemptive version of the SJF algorithm. Jobs are put into the ready queue as they come. A process with the shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated to the CPU cycle.

Process	Arrival time	Burst time
P1	1	8
P2	0	5
P3	4	4



```
Gantt chart:
0 P2 5 P3 9 P1 17
ID      AT      BT      CT      TAT     WT

1       1       8       17      16      8
2       0       5       5       5       0
3       4       4       9       5       1

Average Response time: 4.66667
Average Turn Around Time = 8.66667
Average Waiting Time = 3
```

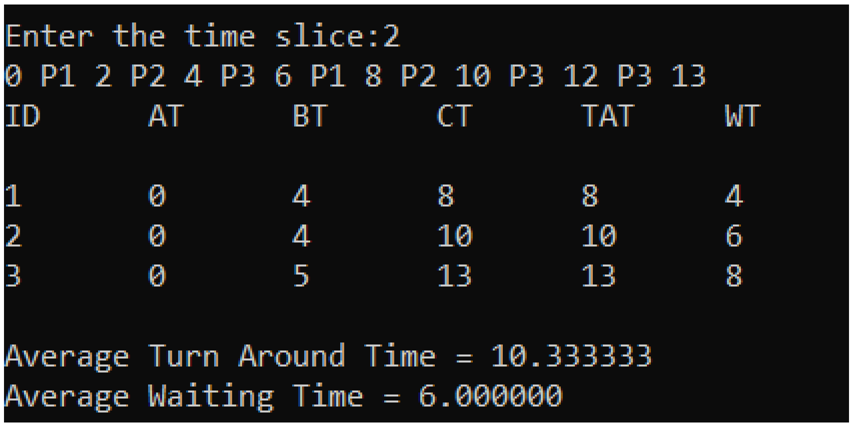
Round Robin

The motivation behind this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. This is most prevalent in multitasking scenarios.

In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. This algorithm also offers starvation free execution of processes.

Consider the following processes with a time slice of 2 units.

Process	Burst Time
P1	4
P2	4
P3	5



```
Enter the time slice:2
0 P1 2 P2 4 P3 6 P1 8 P2 10 P3 12 P3 13
ID      AT      BT      CT      TAT     WT

1       0       4       8       8       4
2       0       4       10      10      6
3       0       5       13      13      8

Average Turn Around Time = 10.333333
Average Waiting Time = 6.000000
```

Highest Response Ratio Next Algorithm

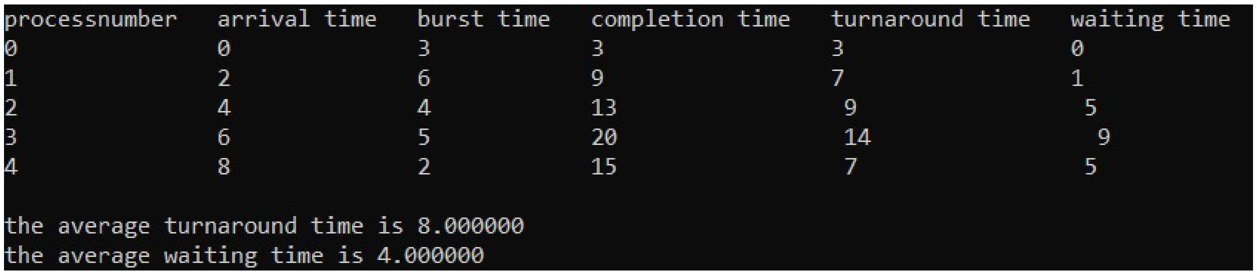
This is a non preemptive CPU scheduling algorithm wherein the processes are allocated CPU based on their response ratio.

This response ratio for a process is calculated as follows:

Response Ratio = (W+B)/B

where W is the waiting time and B is the burst time.
For the following processes, we formulate the Gantt chart:

Process	Arrival time	Burst time
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2



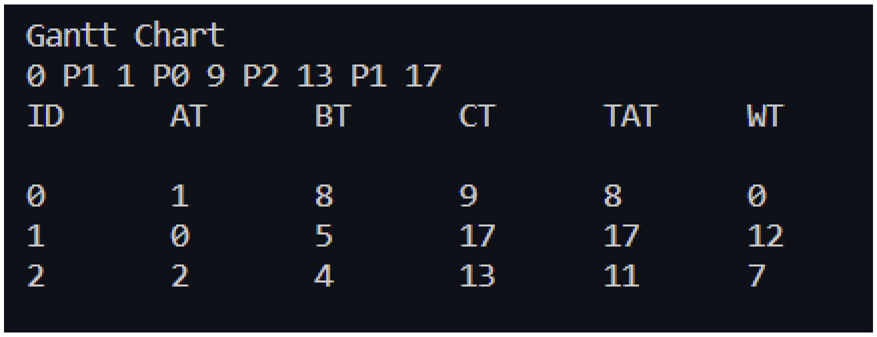
```
processnumber    arrival time    burst time    completion time    turnaround time    waiting time
0                0               3             3                  3                  0
1                2               6             9                  7                  1
2                4               4             13                  9                  5
3                6               5             20                  14                  9
4                8               2             15                  7                  5

the average turnaround time is 8.000000
the average waiting time is 4.000000
```

Priority Scheduling

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.
We describe preemptive scheduling here.

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

Process	Arrival time	Burst time	Priority
P0	1	8	1
P1	0	5	3
P2	2	4	2



```
Gantt Chart
0 P1 1 P0 9 P2 13 P1 17
ID      AT      BT      CT      TAT     WT

0       1       8       9       8       0
1       0       5       17      17      12
2       2       4       13      11      7
```

Process Synchronization

In Operating Systems, process synchronization is the task of co-ordinating the execution of processes in a manner which prevents two processes from having access to same shared resources.

Process Synchronization is mainly needed in a multi-process system when multiple processes are running concurrently, and more than one processes try to gain access to the same shared resource or any data at the same time.

Whenever multiple processes are accessing and manipulating the same data concurrently, it may so happen that the outcome depends on the order in which the two concurrent processes execute. This condition happens inside the critical section.

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section.

In order to solve this critical section problem, the algorithm must satisfy three conditions:

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

To further discuss process synchronization, we describe few classic problems in process synchronization:

Producer Consumer Problem

The producer consumer problem (or bounded buffer problem) is described as below:

There is a buffer of **n** slots each slot capable of storing a unit of data. We have two processes namely **producer** and **consumer**, both of which operate on the buffer.'

The producer inserts a unit of data into the buffer (an empty slot of the buffer) and the consumer tries to remove a unit of data from a filled slot in the buffer. In order to produce the expected result we have to make sure the two processes do not run concurrently as in:

The producer should not produce items when the consumer is consuming from the buffer and vice versa. The buffer should only be accessible to either the producer or consumer at a time.

One of the solutions to the producer consumer problem is using semaphores namely mutex, empty and full.

The mutex semaphore ensures mutual exclusion while the empty and full semaphores count the number of empty and full spaces in the shared buffer.

The producer process works as follows:

- In the producer code, the semaphores work as follows:
- After the item is produced, wait operation is carried out on empty. This indicates that the empty space in the buffer has decreased by 1. Then wait operation is carried out on mutex so that the consumer process cannot interfere.
- After the item is put in the buffer, signal operation is carried out on mutex and full. The former indicates that the consumer process can now act and the latter shows that the buffer is full by 1.

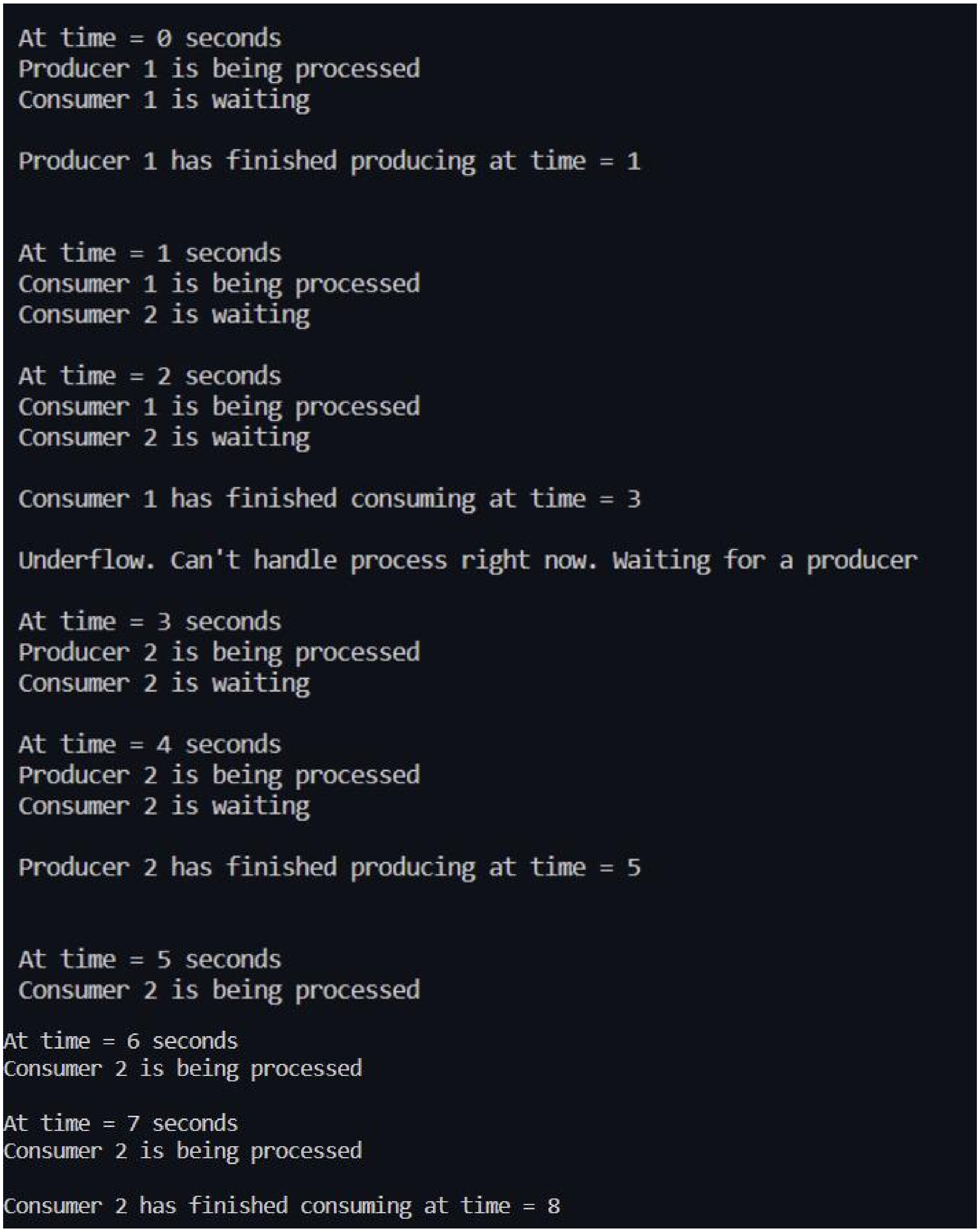
Meanwhile, the consumer works as:

- The wait operation is carried out on full. This indicates that items in the buffer have decreased by 1. Then, the wait operation is carried out on mutex so that the producer process cannot interfere.
- Then the item is removed from the buffer. After that, signal operation is carried out on mutex and empty. The former indicates that the consumer process can now act and the latter shows that the empty space in the buffer has increased by 1.

We see an example output for the following scenario:

Process Name	Arrival Time	Burst Time
Producer 1	0	1
Producer 2	3	2
Consumer 1	0	2
Consumer 2	1	3

Output:



```
At time = 0 seconds
Producer 1 is being processed
Consumer 1 is waiting

Producer 1 has finished producing at time = 1


At time = 1 seconds
Consumer 1 is being processed
Consumer 2 is waiting

At time = 2 seconds
Consumer 1 is being processed
Consumer 2 is waiting

Consumer 1 has finished consuming at time = 3

Underflow. Can't handle process right now. Waiting for a producer

At time = 3 seconds
Producer 2 is being processed
Consumer 2 is waiting

At time = 4 seconds
Producer 2 is being processed
Consumer 2 is waiting

Producer 2 has finished producing at time = 5


At time = 5 seconds
Consumer 2 is being processed

At time = 6 seconds
Consumer 2 is being processed

At time = 7 seconds
Consumer 2 is being processed

Consumer 2 has finished consuming at time = 8
```

Reader Writer Problem

The readers-writer problem is another classical problem in process synchronization. In this, a data set or a file is shared between multiple processes at a time. The two types of processes in this case are:

1. Readers- They can only read the file but can't perform write operations on the file.
2. Writers- These can perform both read and write on the file.

Synchronization is needed between the reader and writer so as to prevent any inconsistency which may arise as a result of concurrent read and writes.

Inconsistencies may arise in the following scenarios:

- If a writer A is writing to a file and another writer process B starts writing on the same file at the exact same time, the system will go into an inconsistent state.
- If a reader is reading from the file and the writer is writing at the same time, this may lead to dirty read because the writer may change the value of the file but the reader would have had read the old value of the file.

The above two scenarios should be avoided which we do using semaphores namely **writer** and **mutex**. A readCount variable is used which keeps track of all the readers reading the file concurrently.

The writer semaphore ensures mutual exclusion property. It ensures that no other process should enter the critical section at that instant of time.

This semaphore is used to achieve mutual exclusion during changing the variable that is storing the count of the processes that are reading a particular file.

The writer process works as follows:

- The wait(writer) function is called so that it achieves the mutual exclusion. The wait() function will reduce the writer value to "0" and this will block other processes to enter into the critical section.
- The write operation will be carried and finally, the signal(writer) function will be called and the value of the writer will be again set to "1" and now other processes will be allowed to enter into the critical section.

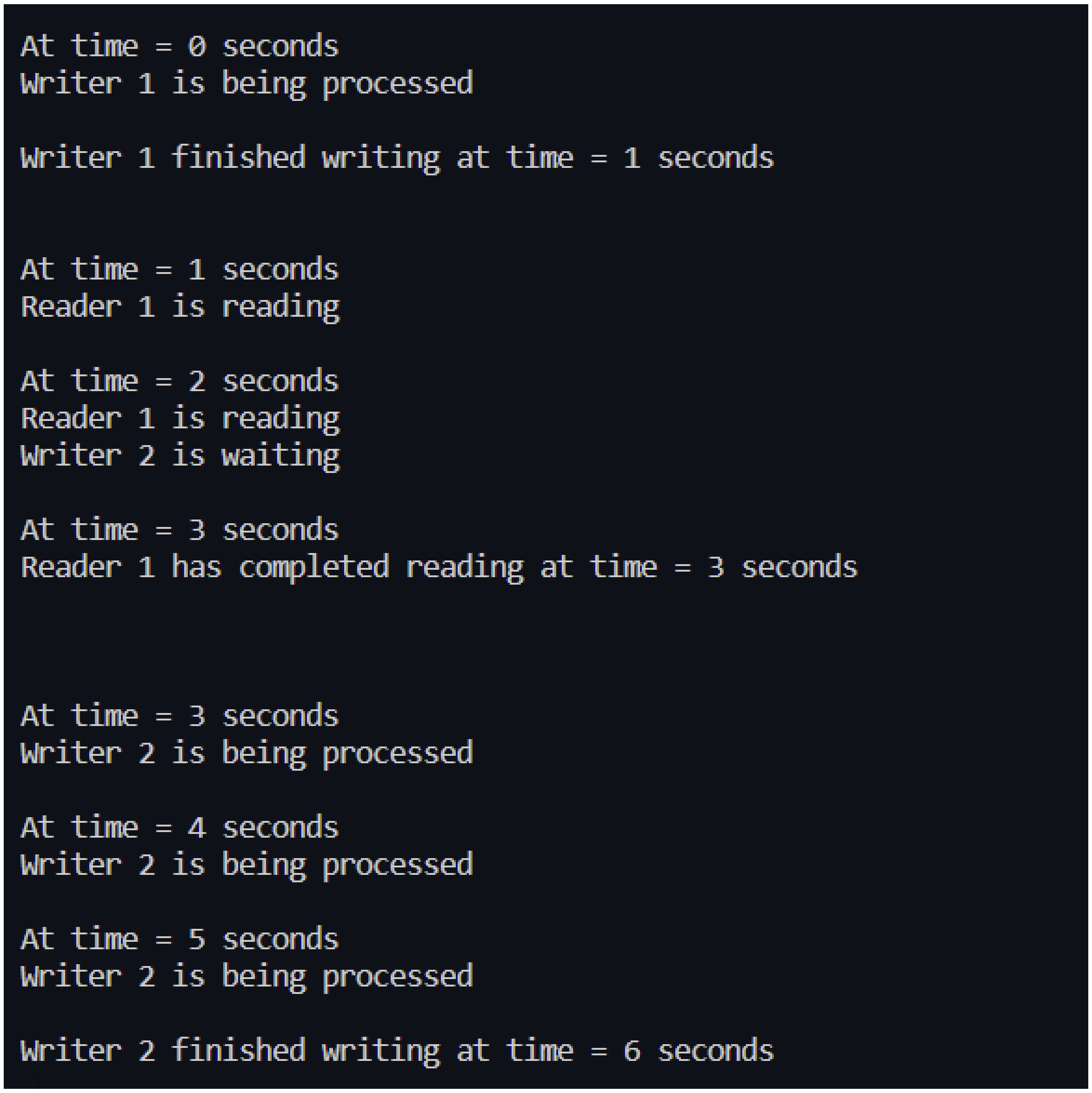
The reader process works as follows:

- The moment a reader process executes, the variable readCount becomes 1, wait operation is used to write semaphore which decreases the value by one. This means that a writer is not allowed how to access the file anymore. On completion of the read operation, readcount is decremented by one. When readCount becomes 0, the signal operation which is used to write permits a writer to access the file.

  As an example, we consider the following scenario:

Process Type	Arrival Time	Burst Time
Writer 1	0	1
Writer 2	2	3
Reader 1	1	2

The output for the above scenario we get is:



```
At time = 0 seconds
Writer 1 is being processed

Writer 1 finished writing at time = 1 seconds


At time = 1 seconds
Reader 1 is reading

At time = 2 seconds
Reader 1 is reading
Writer 2 is waiting

At time = 3 seconds
Reader 1 has completed reading at time = 3 seconds



At time = 3 seconds
Writer 2 is being processed

At time = 4 seconds
Writer 2 is being processed

At time = 5 seconds
Writer 2 is being processed

Writer 2 finished writing at time = 6 seconds
```

Dining Philosophers Problem

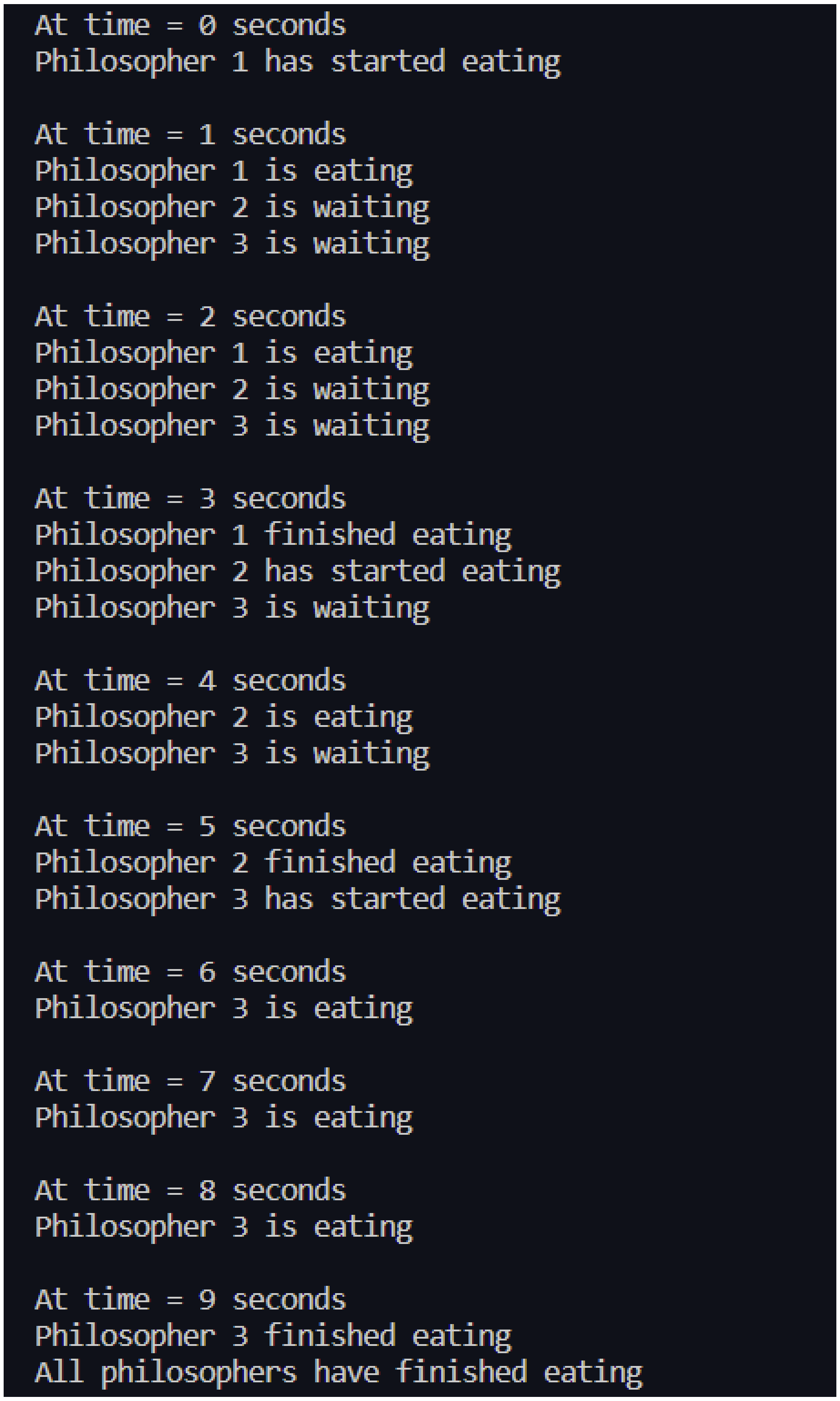
According to the dining philosopher's problem, there are 5 philosophers that sit around a circular table. All of them eat and think alternatively. For each of them philosophers, there is a bowl of rice to eat and a total of 5 chopsticks. A philosopher can eat only when they have both the chopsticks(the one to their immediate right and left). If not, the philosopher has to put down the single chopstick they have and start thinking again. This is a classical problem of synchronization as it represents the importance of synchronization in a lot of concurrency control problems.

Consider a scenario when all five of the philosophers want to eat. Thus, each of them pick up the chopstick. This leads to a deadlock and they will now be waiting for the other chopstick forever.

Consider a simple instance of the dining philosopher's problem with 3 philosophers:

Philosopher	Arrival Time	Burst Time
1	0	3
2	1	2
3	1	4

The output which we get is as follows:



```
At time = 0 seconds
Philosopher 1 has started eating

At time = 1 seconds
Philosopher 1 is eating
Philosopher 2 is waiting
Philosopher 3 is waiting

At time = 2 seconds
Philosopher 1 is eating
Philosopher 2 is waiting
Philosopher 3 is waiting

At time = 3 seconds
Philosopher 1 finished eating
Philosopher 2 has started eating
Philosopher 3 is waiting

At time = 4 seconds
Philosopher 2 is eating
Philosopher 3 is waiting

At time = 5 seconds
Philosopher 2 finished eating
Philosopher 3 has started eating

At time = 6 seconds
Philosopher 3 is eating

At time = 7 seconds
Philosopher 3 is eating

At time = 8 seconds
Philosopher 3 is eating

At time = 9 seconds
Philosopher 3 finished eating
All philosophers have finished eating
```

Sleeping Barber Problem

Consider a hypothetical barber shop. There is one barber, one barber chair and **n** chairs. Each chair is waiting for customers to sit on.

Here is how the scenario plays out:

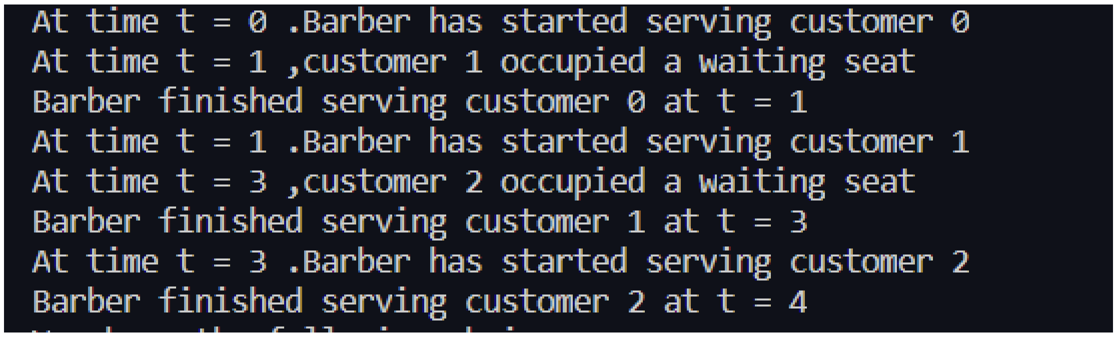
If there is no customer, the barber sleeps on his own chair. Whenever a customer arrives and the barber is sleeping, the barber has to be woken up by him. If there are multiple customers and the barber is busy cutting another customer's hair, the remaining customers have two choices- either wait on one of the empty chairs(if any) or leave.

A solution to this problem involves use of three semaphores. One of the semaphores keeps a count of the customers present. The second being a binary semaphore tells us whether the barber is idle or busy and the third is a mutex which provides mutual exclusion.

When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up.When a customer arrives, the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex.When the haircut is over, the customer exits the procedure and if there are no customers in the waiting room the barber sleeps.

Consider an instance of the sleeping barber problem with 3 empty chairs as follows

Customer	Arrival Time	Burst Time
1	0	1
2	1	2
3	2	1



```
At time t = 0 .Barber has started serving customer 0
At time t = 1 ,customer 1 occupied a waiting seat
Barber finished serving customer 0 at t = 1
At time t = 1 .Barber has started serving customer 1
At time t = 3 ,customer 2 occupied a waiting seat
Barber finished serving customer 1 at t = 3
At time t = 3 .Barber has started serving customer 2
Barber finished serving customer 2 at t = 4
```

Deadlock Avoidance:

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

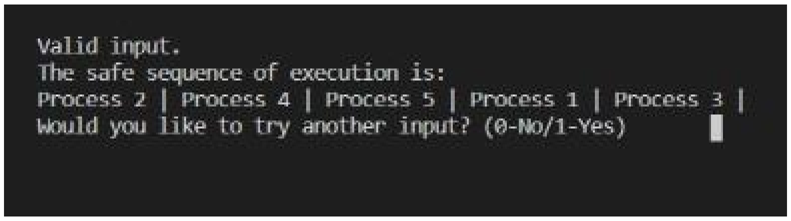
1. **Mutual exclusion:** At least one resource must be held in a non sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Banker’s Algorithm

Banker's Algorithm is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not. This algorithm is used to test for safely simulating the allocation for determining the maximum amount available for all resources. It also checks for all the possible activities before determining whether allocation should be continued or not.

Consider the following input instance for the banker's algorithm:

Process	Allocation			Max		
Resource->	A	B	C	A	B	C
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
P5	0	0	2	4	3	3



```
Valid input.
The safe sequence of execution is:
Process 2 | Process 4 | Process 5 | Process 1 | Process 3 |
Would you like to try another input? (0-No/1-Yes)
```

Page Replacement Algorithms

Page replacement is the technique used by most modern operating systems in order to increase the degree of multiprogramming. Whenever a process needs to be loaded onto memory, the system first checks if there are enough free frames to accommodate it in the free frame pool. In case it is not sufficient, the system must now select a victim in order to swap out of the memory.

Earlier, operating systems would remove all the pages belonging to a particular process when we need to load another one onto memory. However this is an expensive operation and limits the degree of multiprogramming. Instead, if the system chose to simply terminate the process that is requesting for frames, it could lead to underutilization of the CPU.

The approach adopted by most modern operating systems is to swap only some pages and keep track of them using a page replacement algorithm. This is a more efficient option since it is unlikely that a process will be using all its frames at once and it is safe to replace some of its infrequently used frames with ones for a new process. A page replacement algorithm is used to determine which of the frames need to be replaced.

We cover the following replacement algorithms in the code :

1. First In First Out (FIFO)
2. Optimal
3. Least Recently Used (LRU)
4. Most Recently Used (MRU)

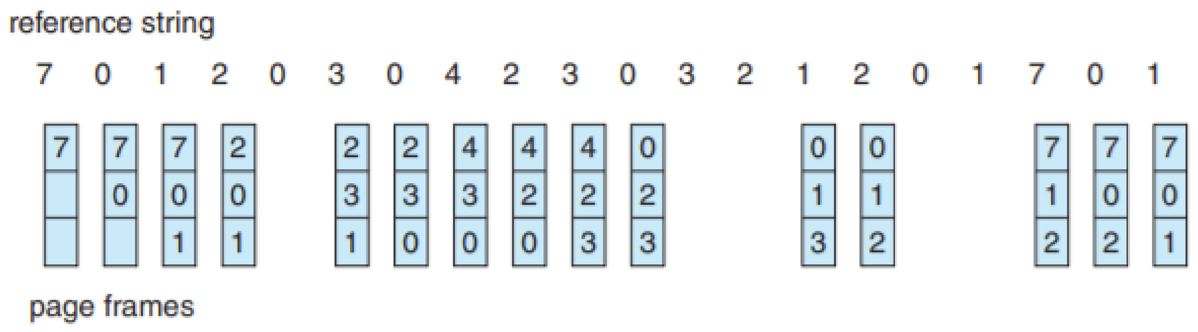
In order to illustrate and compare the working of each of these algorithms, we use the following sequence of page references :

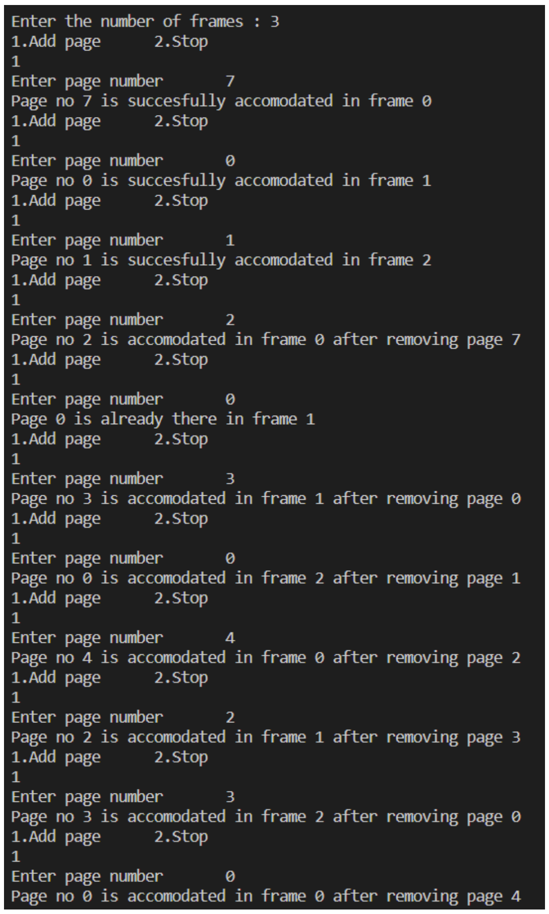
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Where each number in the sequence is a request for that page index.

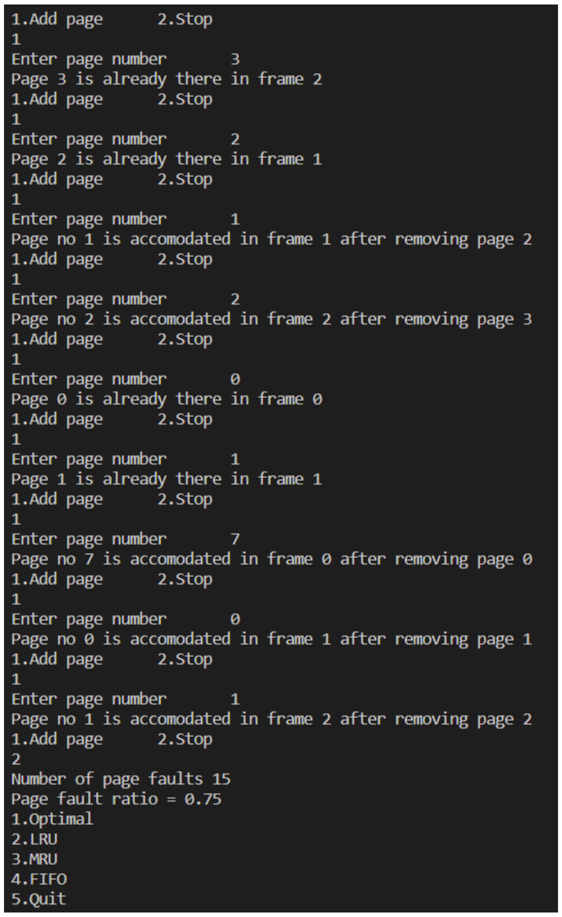
First In First Out (FIFO)

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.





```
Enter the number of frames : 3
1.Add page      2.Stop
1
Enter page number       7
Page no 7 is succesfully accomodated in frame 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is succesfully accomodated in frame 1
1.Add page      2.Stop
1
Enter page number       1
Page no 1 is succesfully accomodated in frame 2
1.Add page      2.Stop
1
Enter page number       2
Page no 2 is accomodated in frame 0 after removing page 7
1.Add page      2.Stop
1
Enter page number       0
Page 0 is already there in frame 1
1.Add page      2.Stop
1
Enter page number       3
Page no 3 is accomodated in frame 1 after removing page 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 2 after removing page 1
1.Add page      2.Stop
1
Enter page number       4
Page no 4 is accomodated in frame 0 after removing page 2
1.Add page      2.Stop
1
Enter page number       2
Page no 2 is accomodated in frame 1 after removing page 3
1.Add page      2.Stop
1
Enter page number       3
Page no 3 is accomodated in frame 2 after removing page 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 0 after removing page 4
```



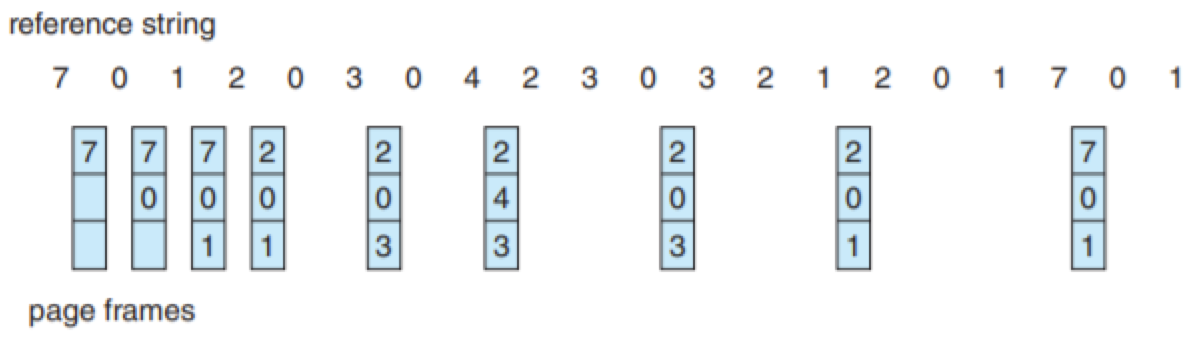
```
1.Add page      2.Stop
1
Enter page number       3
Page 3 is already there in frame 2
1.Add page      2.Stop
1
Enter page number       2
Page 2 is already there in frame 1
1.Add page      2.Stop
1
Enter page number       1
Page no 1 is accomodated in frame 1 after removing page 2
1.Add page      2.Stop
1
Enter page number       2
Page no 2 is accomodated in frame 2 after removing page 3
1.Add page      2.Stop
1
Enter page number       0
Page 0 is already there in frame 0
1.Add page      2.Stop
1
Enter page number       1
Page 1 is already there in frame 1
1.Add page      2.Stop
1
Enter page number       7
Page no 7 is accomodated in frame 0 after removing page 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 1 after removing page 1
1.Add page      2.Stop
1
Enter page number       1
Page no 1 is accomodated in frame 2 after removing page 2
1.Add page      2.Stop
2
Number of page faults 15
Page fault ratio = 0.75
1.Optimal
2.LRU
3.MRU
4.FIFO
5.Quit
```

For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

Optimal

The optimal page replacement algorithm uses the following strategy : Replace the page that will not be used for the longest period of time.

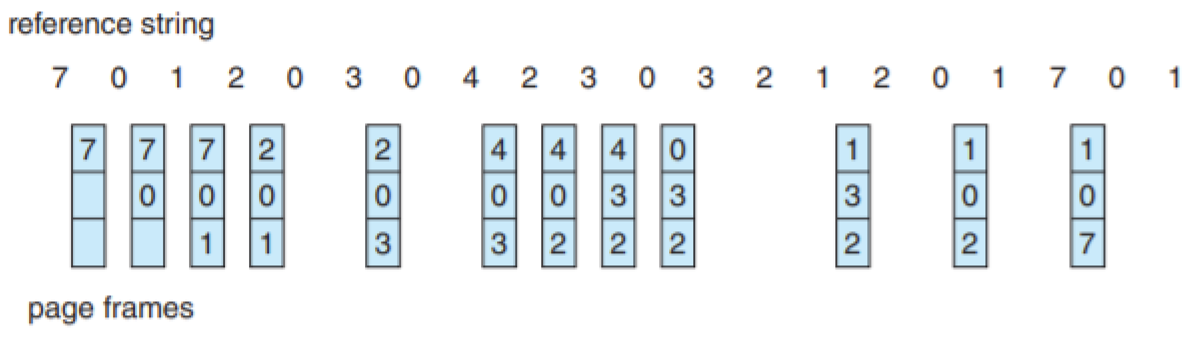
Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.



Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

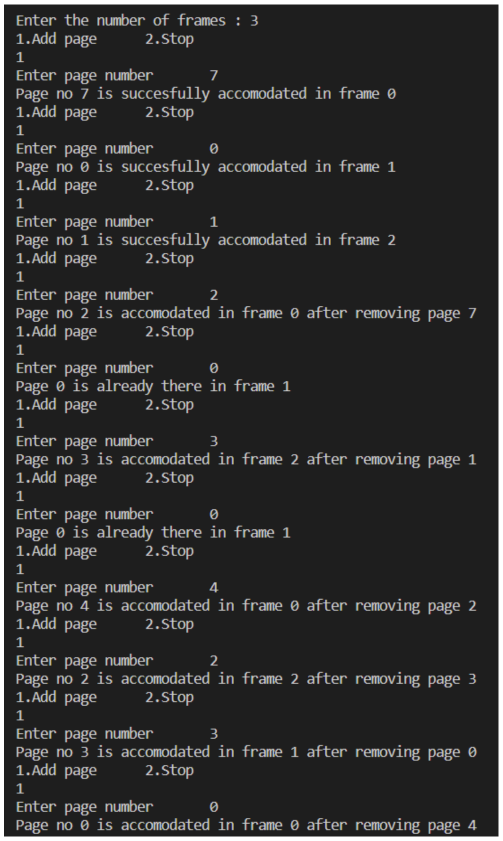
Least Recently Used (LRU)

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let SR be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on SR. Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on SR.)

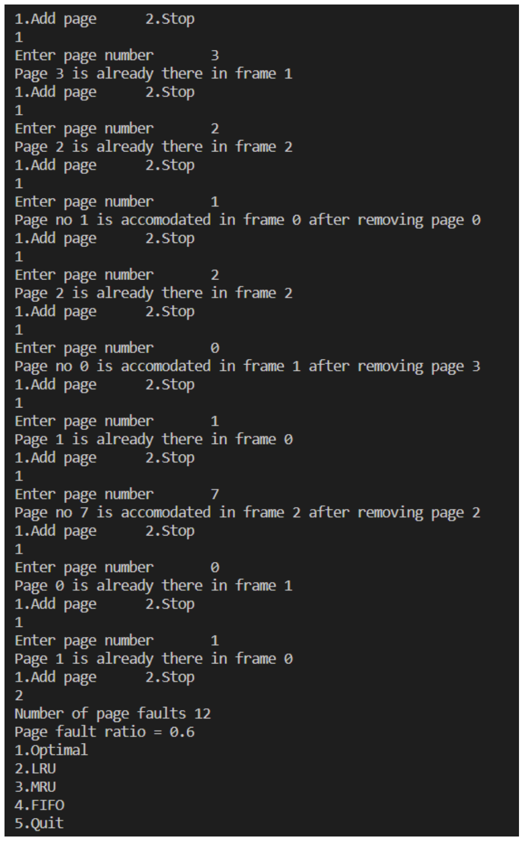


The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

- Counters
- Stack



```
Enter the number of frames : 3
1.Add page      2.Stop
1
Enter page number       7
Page no 7 is succesfully accomodated in frame 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is succesfully accomodated in frame 1
1.Add page      2.Stop
1
Enter page number       1
Page no 1 is succesfully accomodated in frame 2
1.Add page      2.Stop
1
Enter page number       2
Page no 2 is accomodated in frame 0 after removing page 7
1.Add page      2.Stop
1
Enter page number       0
Page 0 is already there in frame 1
1.Add page      2.Stop
1
Enter page number       3
Page no 3 is accomodated in frame 2 after removing page 1
1.Add page      2.Stop
1
Enter page number       0
Page 0 is already there in frame 1
1.Add page      2.Stop
1
Enter page number       4
Page no 4 is accomodated in frame 0 after removing page 2
1.Add page      2.Stop
1
Enter page number       2
Page no 2 is accomodated in frame 2 after removing page 3
1.Add page      2.Stop
1
Enter page number       3
Page no 3 is accomodated in frame 1 after removing page 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 0 after removing page 4
```

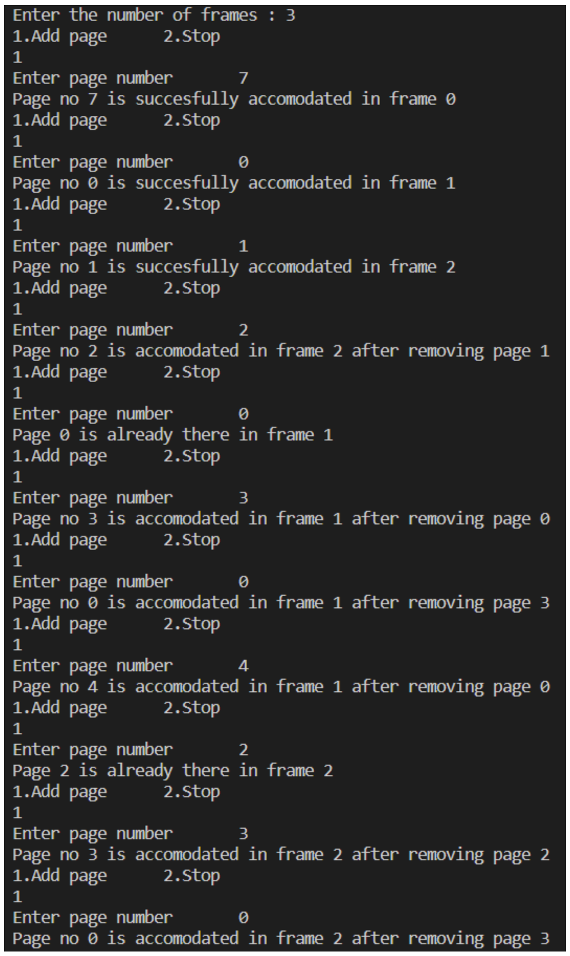


```
1.Add page      2.Stop
1
Enter page number       3
Page 3 is already there in frame 1
1.Add page      2.Stop
1
Enter page number       2
Page 2 is already there in frame 2
1.Add page      2.Stop
1
Enter page number       1
Page no 1 is accomodated in frame 0 after removing page 0
1.Add page      2.Stop
1
Enter page number       2
Page 2 is already there in frame 2
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 1 after removing page 3
1.Add page      2.Stop
1
Enter page number       1
Page 1 is already there in frame 0
1.Add page      2.Stop
1
Enter page number       7
Page no 7 is accomodated in frame 2 after removing page 2
1.Add page      2.Stop
1
Enter page number       0
Page 0 is already there in frame 1
1.Add page      2.Stop
1
Enter page number       1
Page 1 is already there in frame 0
1.Add page      2.Stop
2
Number of page faults 12
Page fault ratio = 0.6
1.Optimal
2.LRU
3.MRU
4.FIFO
5.Quit
```

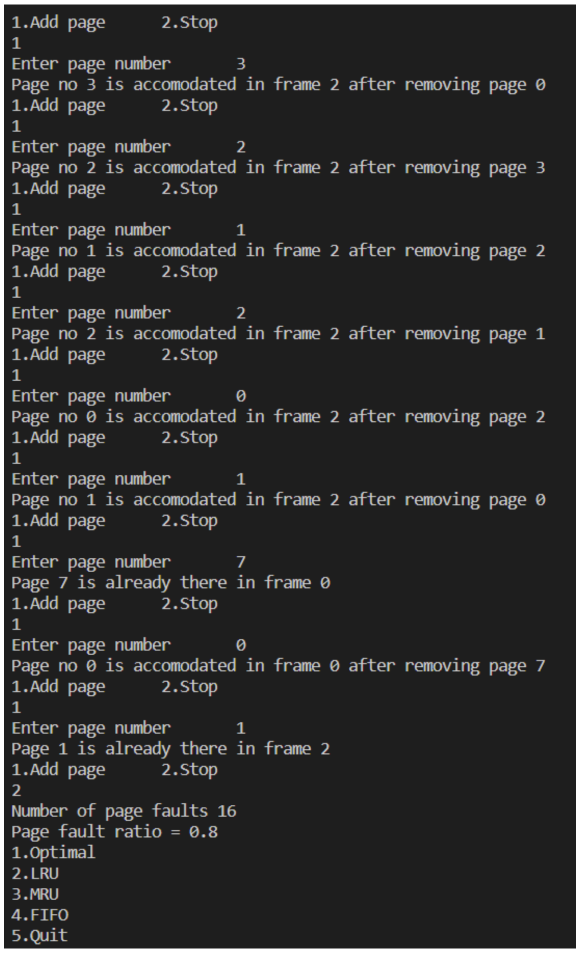
Most Recently Used (MRU)

Similar to LRU, MRU replaces the page with the most recently used page. It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.



```
Enter the number of frames : 3
1.Add page      2.Stop
1
Enter page number       7
Page no 7 is succesfully accomodated in frame 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is succesfully accomodated in frame 1
1.Add page      2.Stop
1
Enter page number       1
Page no 1 is succesfully accomodated in frame 2
1.Add page      2.Stop
1
Enter page number       2
Page no 2 is accomodated in frame 2 after removing page 1
1.Add page      2.Stop
1
Enter page number       0
Page 0 is already there in frame 1
1.Add page      2.Stop
1
Enter page number       3
Page no 3 is accomodated in frame 1 after removing page 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 1 after removing page 3
1.Add page      2.Stop
1
Enter page number       4
Page no 4 is accomodated in frame 1 after removing page 0
1.Add page      2.Stop
1
Enter page number       2
Page 2 is already there in frame 2
1.Add page      2.Stop
1
Enter page number       3
Page no 3 is accomodated in frame 2 after removing page 2
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 2 after removing page 3
```



```
1.Add page      2.Stop
1
Enter page number       3
Page no 3 is accomodated in frame 2 after removing page 0
1.Add page      2.Stop
1
Enter page number       2
Page no 2 is accomodated in frame 2 after removing page 3
1.Add page      2.Stop
1
Enter page number       1
Page no 1 is accomodated in frame 2 after removing page 2
1.Add page      2.Stop
1
Enter page number       2
Page no 2 is accomodated in frame 2 after removing page 1
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 2 after removing page 2
1.Add page      2.Stop
1
Enter page number       1
Page no 1 is accomodated in frame 2 after removing page 0
1.Add page      2.Stop
1
Enter page number       7
Page 7 is already there in frame 0
1.Add page      2.Stop
1
Enter page number       0
Page no 0 is accomodated in frame 0 after removing page 7
1.Add page      2.Stop
1
Enter page number       1
Page 1 is already there in frame 2
1.Add page      2.Stop
2
Number of page faults 16
Page fault ratio = 0.8
1.Optimal
2.LRU
3.MRU
4.FIFO
5.Quit
```

Disc Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For HDDs, meeting this responsibility entails minimizing access time and maximizing data transfer bandwidth.

For HDDs and other mechanical storage devices that use platters, access time has two major components. The seek time is the time for the device arm to move the heads to the cylinder containing the desired sector, and the rotational latency is the additional time for the platter to rotate the desired sector to the head.

The device bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by managing the order in which storage I/O requests are serviced.

We implement the following scheduling algorithms :

1. First Come First Serve (FCFS)
2. SCAN
3. LOOK
4. C-SCAN
5. C-LOOK
6. Shortest Seek Time First (SSTF)

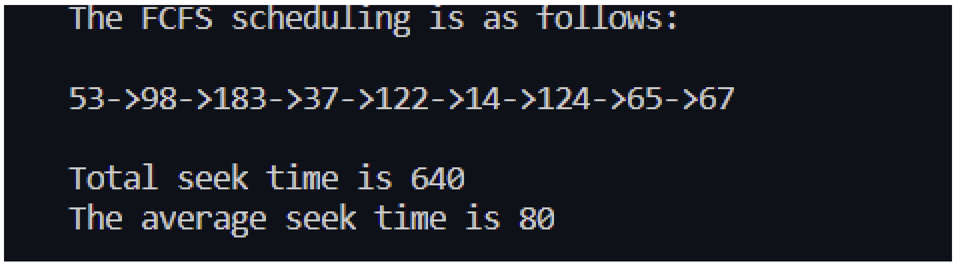
First Come First Serve (FCFS)

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm (or FIFO). This algorithm is intrinsically fair, but it generally does not provide the fastest service. FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

For example, consider the following queue of disc accesses :

98, 183, 37, 122, 14, 124, 65, 67,

Assuming that the head was initially at track 53, the sequence of accesses are :

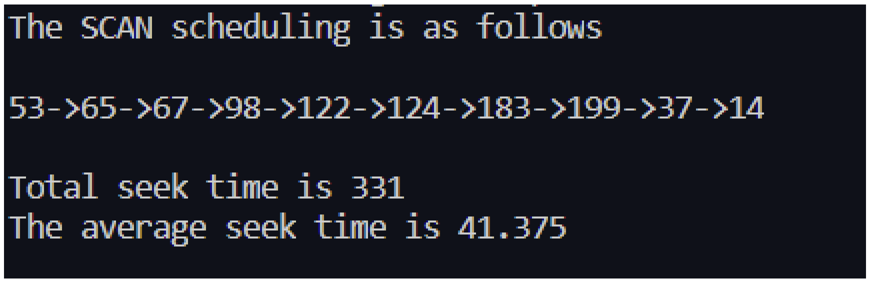


```
The FCFS scheduling is as follows:

53->98->183->37->122->14->124->65->67

Total seek time is 640
The average seek time is 80
```

SCAN

In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

For the same sequence of disc accesses and initial position of read write head, SCAN is executed as follows:

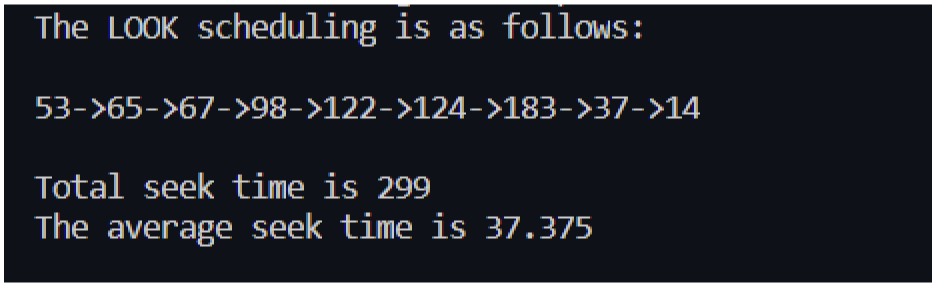


```
The SCAN scheduling is as follows

53->65->67->98->122->124->183->199->37->14

Total seek time is 331
The average seek time is 41.375
```

LOOK

This algorithm works similar to SCAN. It also processes requests by moving in only one direction first and then reversing it. Unlike SCAN however, LOOK does not go all the way to the last or first track unless there is a request for those tracks. Instead it stops at the last request in the direction it is moving in and then reverses direction.

The execution of LOOK for the above sequence is :

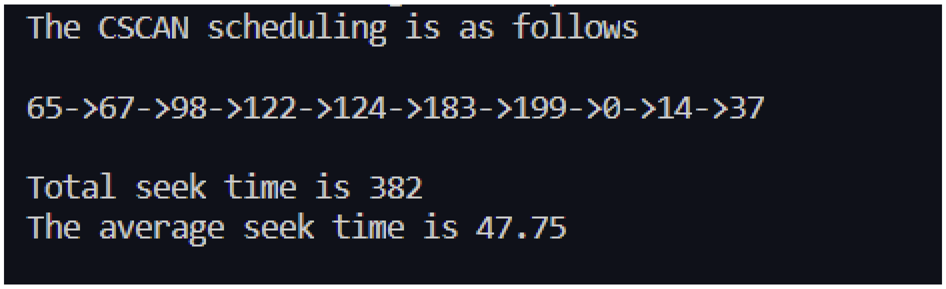


```
The LOOK scheduling is as follows:

53->65->67->98->122->124->183->37->14

Total seek time is 299
The average seek time is 37.375
```

C-SCAN

C-SCAN also services requests by moving in one direction along the disc but once it reaches the end, instead of simply reversing direction like SCAN, it moves to track 0 without servicing any requests. From here it fulfills requests by once again moving along the same direction. Due to the circular movement of the read write head, it is called circular SCAN.

The execution of C-SCAN on the above sequence is :

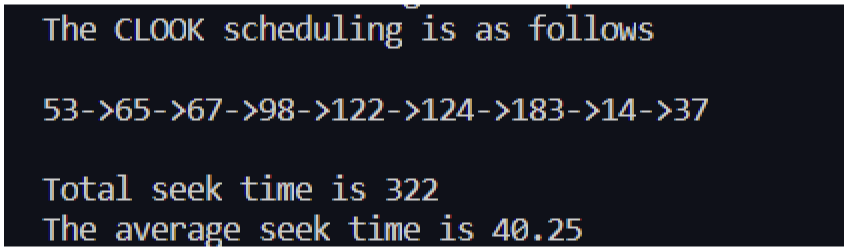


```
The CSCAN scheduling is as follows

65->67->98->122->124->183->199->0->14->37

Total seek time is 382
The average seek time is 47.75
```

C-LOOK

C-LOOK is similar to LOOK the same way CSCAN is similar to SCAN. Once the last request along one direction has been serviced, the read write head immediately moves to the first unserviced request (if it moved towards right initially) or the last unserviced request (if it moved to the left initially).

The execution of C-LOOK on the above sequence is :

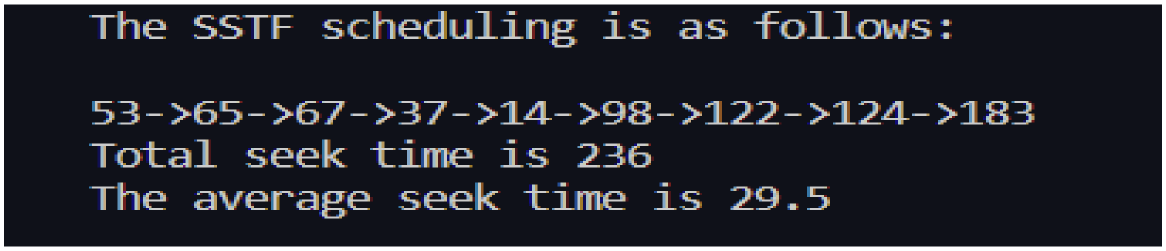


```
The CLOOK scheduling is as follows

53->65->67->98->122->124->183->14->37

Total seek time is 322
The average seek time is 40.25
```

Shortest Seek Time First (SSTF)

In this algorithm, requests that have the shortest seek time from the current position of the read write head are serviced first. This is an improvement over the simple FCFS algorithm since it decreases average response time and increases throughput of the system. However, for this to work efficiently, the seek times for each of the requests from the initial position of the read write head must be calculated in advance.

The execution of SSTF for the above sequence is :



```
The SSTF scheduling is as follows:

53->65->67->37->14->98->122->124->183
Total seek time is 236
The average seek time is 29.5
```

Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general dynamic storage-allocation problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

- **First fit:** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit:** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit:** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Internal Fragmentation:

When there is a difference between required memory space vs allocated memory space, the problem is termed as Internal Fragmentation. It occurs when allotted memory blocks are of fixed size. Best Fit Block Search is the solution for internal fragmentation

External Fragmentation:

When there are small and non-contiguous memory blocks which cannot be assigned to any process, the problem is termed as External Fragmentation.It occurs when a process is removed from the main memory. Compaction is the solution for external fragmentation.

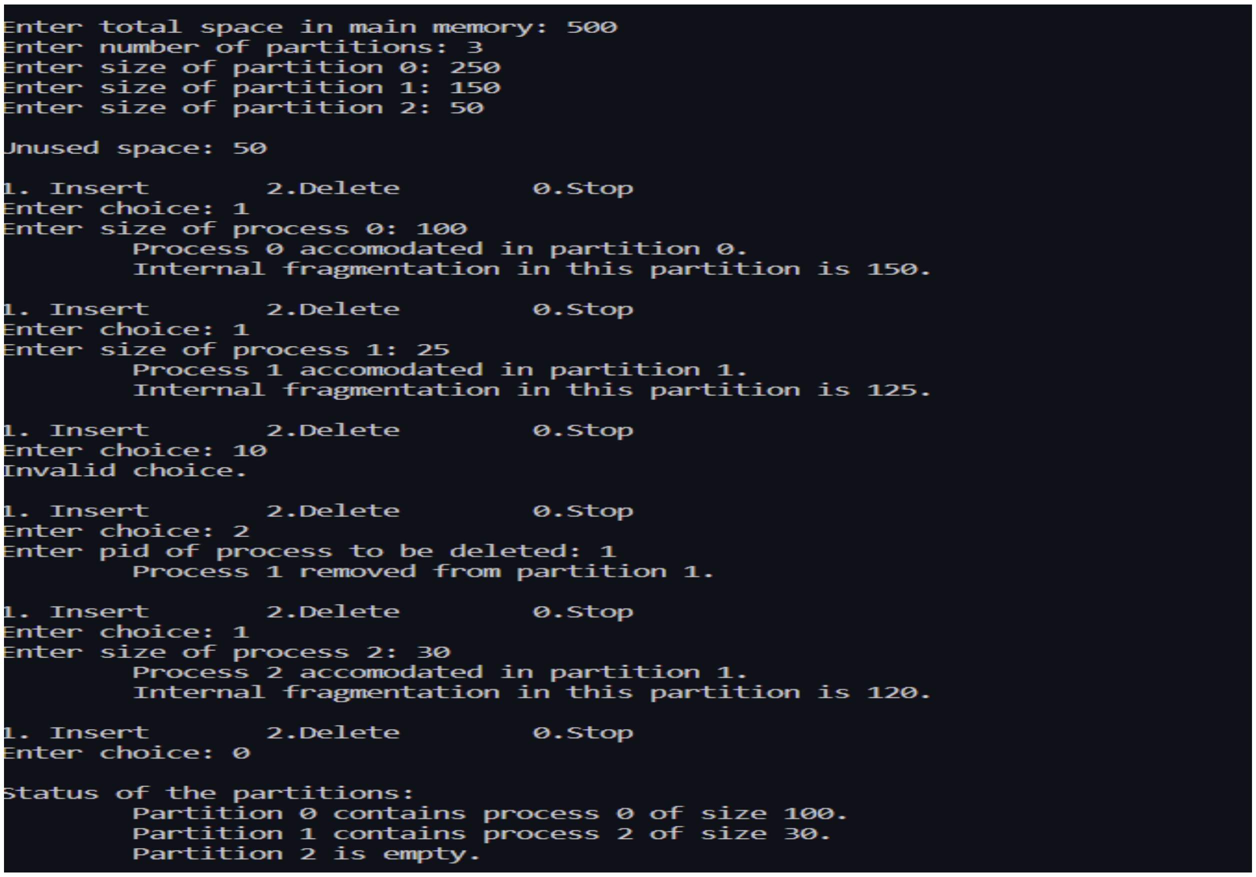
There are two types of Memory allocation techniques:

Multiprogramming with fixed number of partitions (MFT):

It is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change.

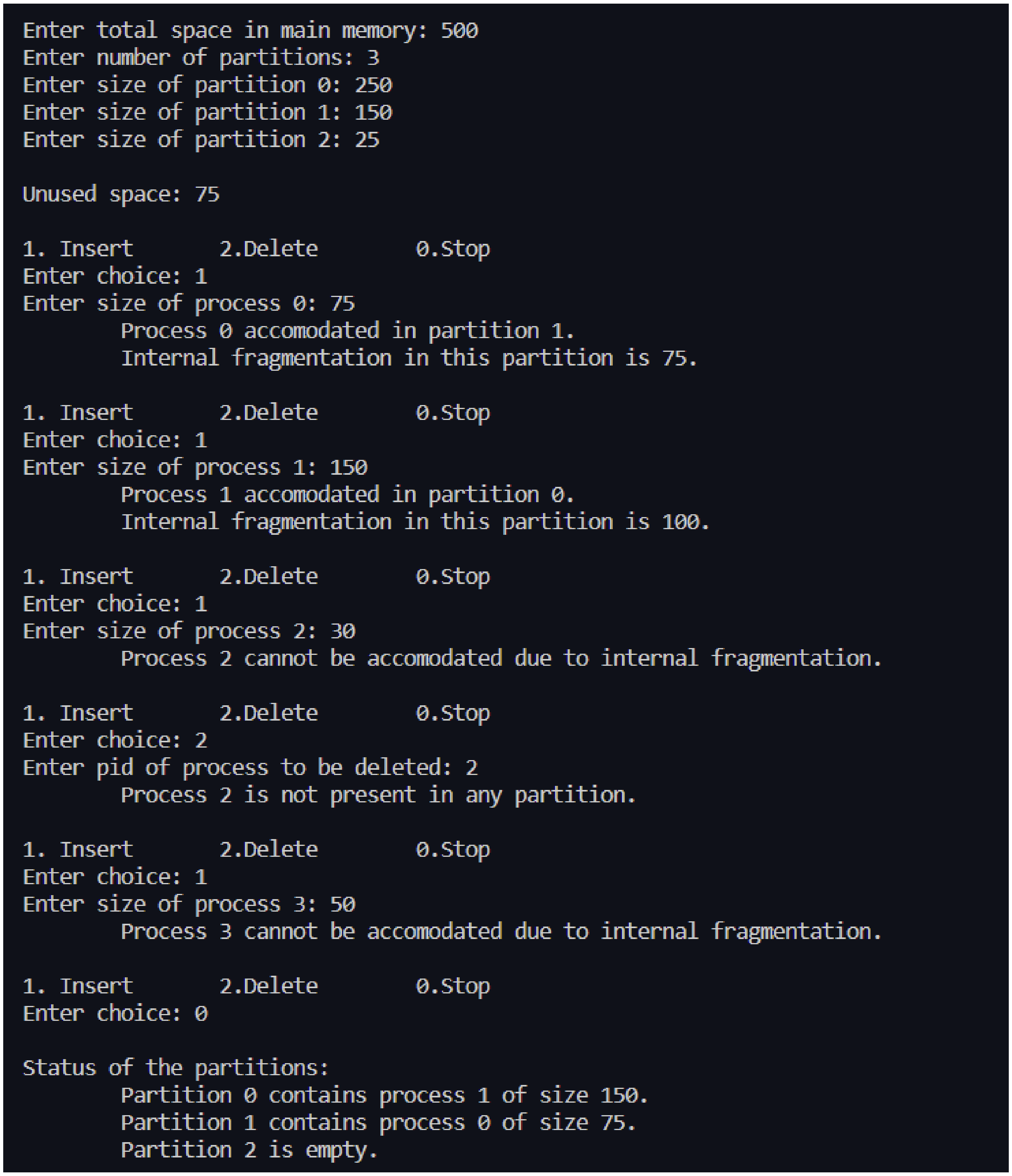
Following are screenshots of output from our implementation:

First Fit:



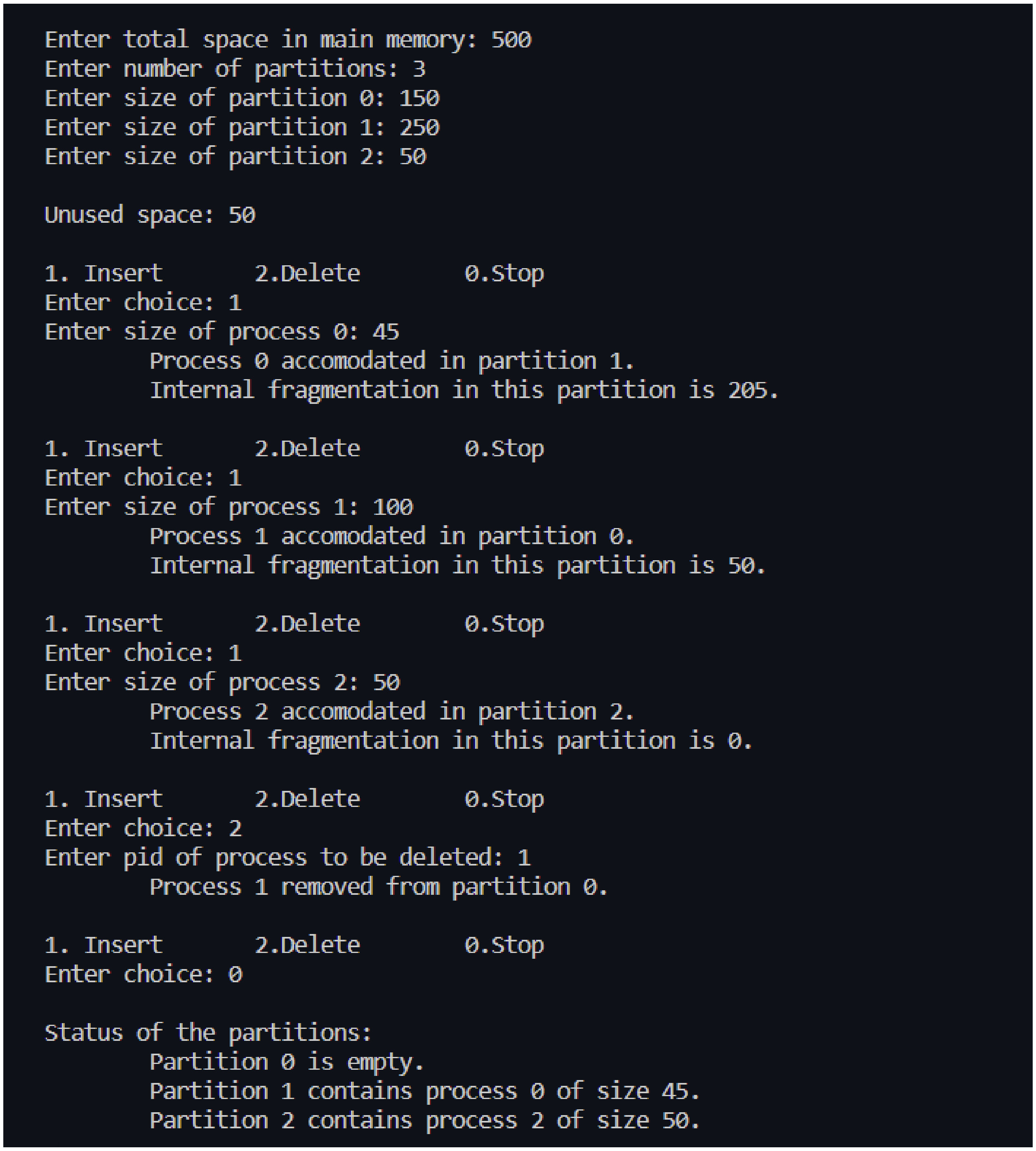
```
Enter total space in main memory: 500
Enter number of partitions: 3
Enter size of partition 0: 250
Enter size of partition 1: 150
Enter size of partition 2: 50

Unused space: 50

1. Insert        2.Delete        0.Stop
Enter choice: 1
Enter size of process 0: 100
        Process 0 accomodated in partition 0.
        Internal fragmentation in this partition is 150.

1. Insert        2.Delete        0.Stop
Enter choice: 1
Enter size of process 1: 25
        Process 1 accomodated in partition 1.
        Internal fragmentation in this partition is 125.

1. Insert        2.Delete        0.Stop
Enter choice: 10
Invalid choice.

1. Insert        2.Delete        0.Stop
Enter choice: 2
Enter pid of process to be deleted: 1
        Process 1 removed from partition 1.

1. Insert        2.Delete        0.Stop
Enter choice: 1
Enter size of process 2: 30
        Process 2 accomodated in partition 1.
        Internal fragmentation in this partition is 120.

1. Insert        2.Delete        0.Stop
Enter choice: 0

Status of the partitions:
        Partition 0 contains process 0 of size 100.
        Partition 1 contains process 2 of size 30.
        Partition 2 is empty.
```

Best Fit:



```
Enter total space in main memory: 500
Enter number of partitions: 3
Enter size of partition 0: 250
Enter size of partition 1: 150
Enter size of partition 2: 25

Unused space: 75

1. Insert       2.Delete        0.Stop
Enter choice: 1
Enter size of process 0: 75
        Process 0 accomodated in partition 1.
        Internal fragmentation in this partition is 75.

1. Insert       2.Delete        0.Stop
Enter choice: 1
Enter size of process 1: 150
        Process 1 accomodated in partition 0.
        Internal fragmentation in this partition is 100.

1. Insert       2.Delete        0.Stop
Enter choice: 1
Enter size of process 2: 30
        Process 2 cannot be accomodated due to internal fragmentation.

1. Insert       2.Delete        0.Stop
Enter choice: 2
Enter pid of process to be deleted: 2
        Process 2 is not present in any partition.

1. Insert       2.Delete        0.Stop
Enter choice: 1
Enter size of process 3: 50
        Process 3 cannot be accomodated due to internal fragmentation.

1. Insert       2.Delete        0.Stop
Enter choice: 0

Status of the partitions:
        Partition 0 contains process 1 of size 150.
        Partition 1 contains process 0 of size 75.
        Partition 2 is empty.
```

Worst Fit



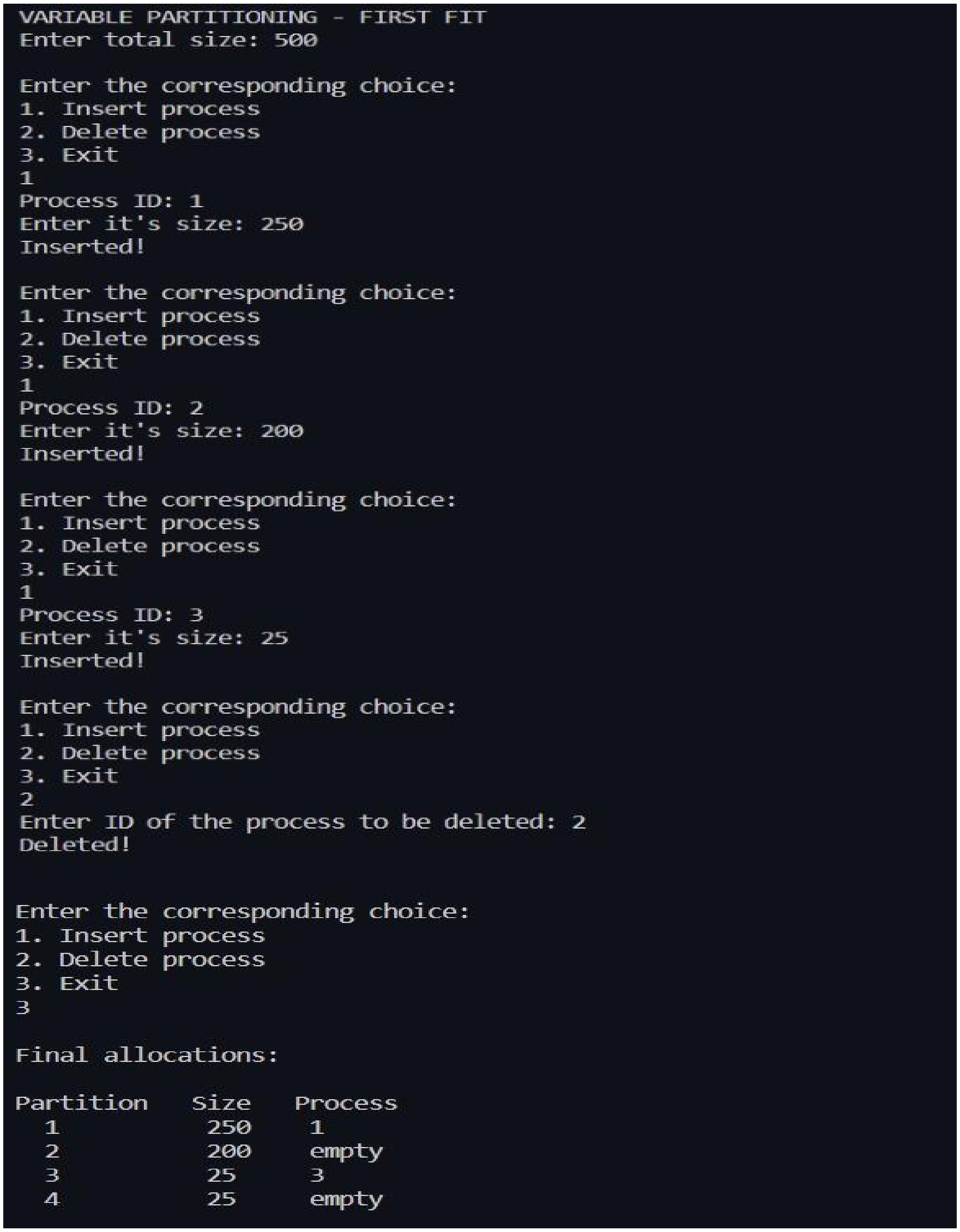
```
Enter total space in main memory: 500
Enter number of partitions: 3
Enter size of partition 0: 150
Enter size of partition 1: 250
Enter size of partition 2: 50

Unused space: 50

1. Insert       2.Delete        0.Stop
Enter choice: 1
Enter size of process 0: 45
        Process 0 accomodated in partition 1.
        Internal fragmentation in this partition is 205.

1. Insert       2.Delete        0.Stop
Enter choice: 1
Enter size of process 1: 100
        Process 1 accomodated in partition 0.
        Internal fragmentation in this partition is 50.

1. Insert       2.Delete        0.Stop
Enter choice: 1
Enter size of process 2: 50
        Process 2 accomodated in partition 2.
        Internal fragmentation in this partition is 0.

1. Insert       2.Delete        0.Stop
Enter choice: 2
Enter pid of process to be deleted: 1
        Process 1 removed from partition 0.

1. Insert       2.Delete        0.Stop
Enter choice: 0

Status of the partitions:
        Partition 0 is empty.
        Partition 1 contains process 0 of size 45.
        Partition 2 contains process 2 of size 50.
```

Multiprogramming with variable number of partitions (MVT):

is the memory management technique in which each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more “efficient” user of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

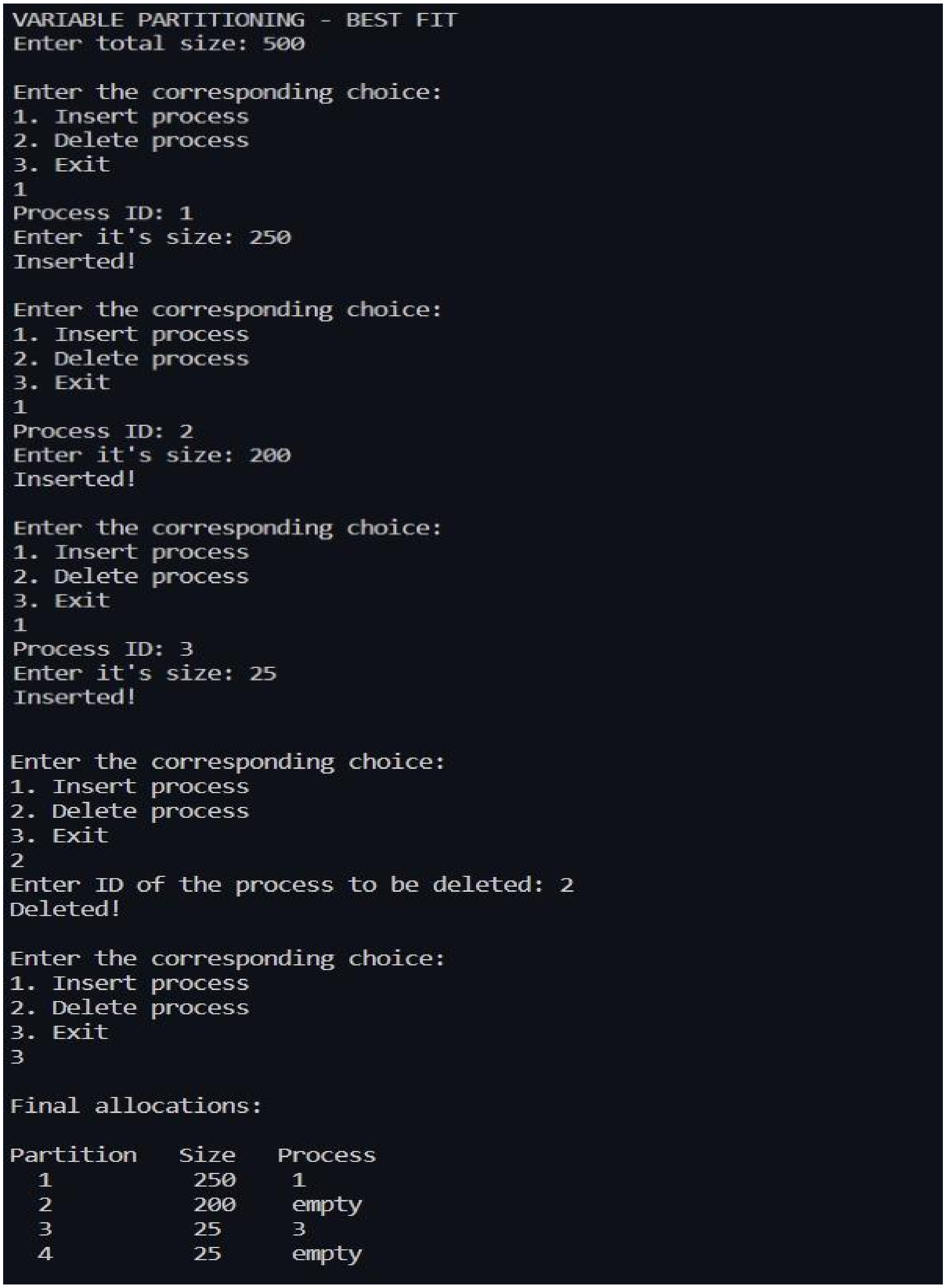
Following are screenshots for the MVT implementation:

First Fit:



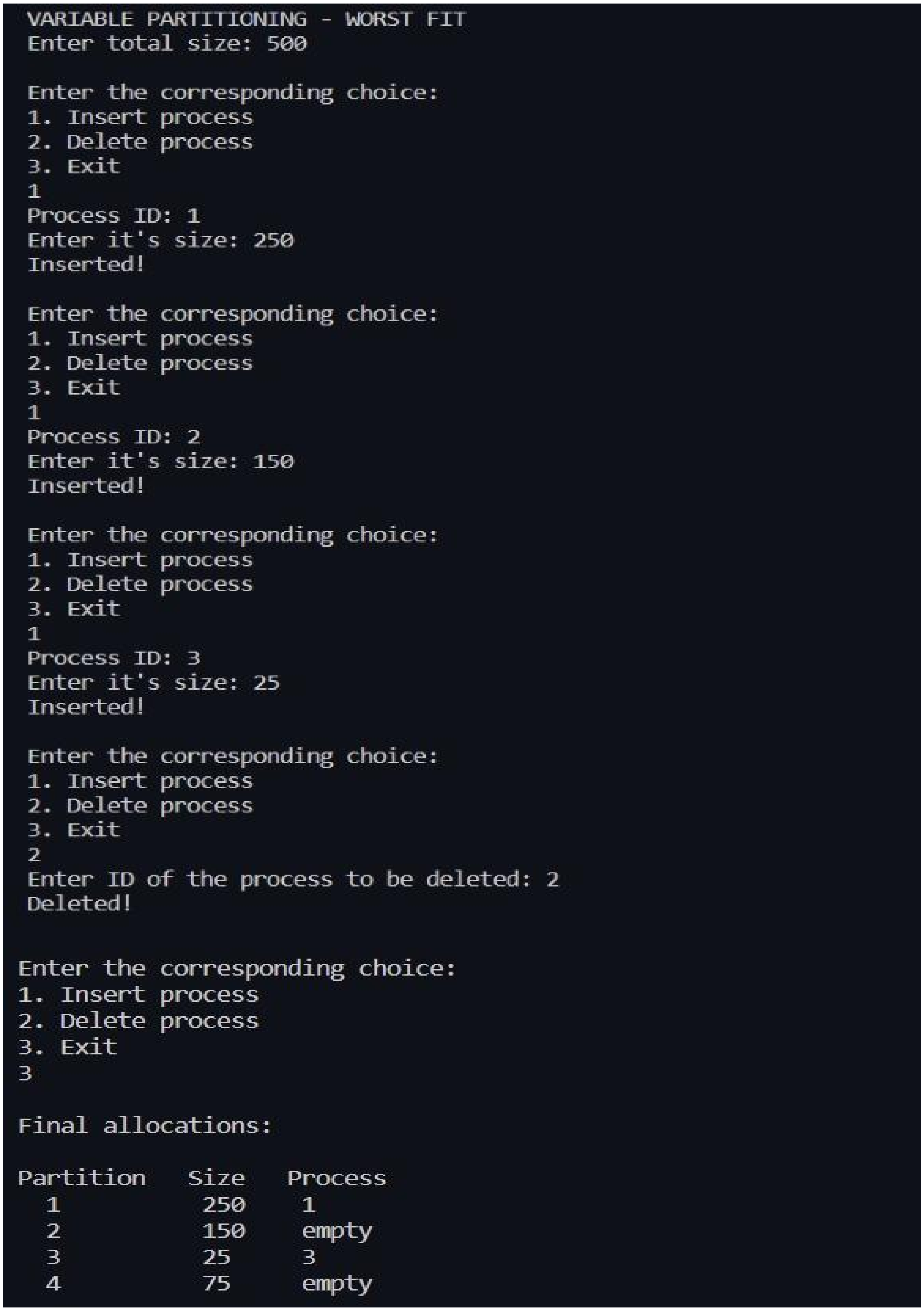
```
VARIABLE PARTITIONING - FIRST FIT
Enter total size: 500

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 1
Enter it's size: 250
Inserted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 2
Enter it's size: 200
Inserted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 3
Enter it's size: 25
Inserted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
2
Enter ID of the process to be deleted: 2
Deleted!


Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
3

Final allocations:

Partition    Size    Process
  1           250     1
  2           200     empty
  3           25      3
  4           25      empty
```

Best Fit



```
VARIABLE PARTITIONING - BEST FIT
Enter total size: 500

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 1
Enter it's size: 250
Inserted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 2
Enter it's size: 200
Inserted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 3
Enter it's size: 25
Inserted!


Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
2
Enter ID of the process to be deleted: 2
Deleted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
3

Final allocations:

Partition   Size   Process
  1          250    1
  2          200    empty
  3          25     3
  4          25     empty
```

Worst Fit



```
VARIABLE PARTITIONING - WORST FIT
Enter total size: 500

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 1
Enter it's size: 250
Inserted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 2
Enter it's size: 150
Inserted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
1
Process ID: 3
Enter it's size: 25
Inserted!

Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
2
Enter ID of the process to be deleted: 2
Deleted!


Enter the corresponding choice:
1. Insert process
2. Delete process
3. Exit
3

Final allocations:

Partition    Size    Process
  1           250     1
  2           150     empty
  3           25      3
  4           75      empty
```

Conclusion

This simulator is the culmination of all the concepts we learnt in the Operating Systems course taught by respected Professor Shashidhar G. Koolagudi (HOD Of CSE Department). We tried our level best to incorporate all functionalities into a single code which efficiently works as a simulator. The efforts put by the entire team into this code construction has resulted in a simulator that closely mimics the basic functionalities of an Operating System. We also want to express our gratitude towards all the TAs and professors who have helped us during the entirety of the course.

THANK YOU!