Lab Report On

Course Title: Operating System Lab

Course Code: CSEL-3104

Submitted To,

Md. Aminul Islam Ph.D

Associate Professor

Department of Computer Science & Engineering

Jagannath University, Dhaka

Submitted By,

Mst. Mahfuza Akter

(B210305002)

Department of Computer Science & Engineering

Jagannath University, Dhaka

Date of Submission: 04/09/2025

1.First Come First Served (FCFS)

Problem analysis:

The FCFS problem is one of the simplest CPU scheduling approaches, where processes are executed in the exact order they arrive, following the **First In First Out (FIFO)** principle. It is **non-preemptive**, meaning once a process starts, it runs until completion before the next process begins. The main parameters are Arrival Time (AT), Burst Time (BT), Completion Time (CT), Turnaround Time (TAT = CT – AT), and Waiting Time (WT = TAT – BT). FCFS is **easy to implement** and ensures fairness, but it may suffer from the **convoy effect**, where long processes delay shorter ones, reducing overall system efficiency.

Code:

```
echo "INPUT"
echo "Enter the number of processes--"
read n
declare -a pid
declare -a bt
declare -a wt
declare -a tat
for ((i = 0; i < n; i++)); do
   pid[$i]=$((i))
   echo "Enter burst time for Process ${pid[$i]}--"
   read bt[$i]
```

```
done
wt[0]=0
for ((i = 1; i < n; i++)); do
   wt[$i]=$((wt[$i-1] + bt[$i-1]))
done
for ((i = 0; i < n; i++)); do
   tat[$i]=$((wt[$i] + bt[$i]))
done
echo "OUTPUT"
echo -e "\nProcess\tBurst Time\tWaiting Time\tTurnaround Time"
total_wt=0
total_tat=0
for ((i = 0; i < n; i++)); do
   echo -e "P${pid[$i]}\t${bt[$i]}\t\t${wt[$i]}\t\t${tat[$i]}"
   total_wt=$((total_wt + wt[$i]))
   total_tat=$((total_tat + tat[$i]))
done
avg_wt=$(echo "scale=6; $total_wt / $n" | bc)
avg_tat=$(echo "scale=6; $total_tat / $n" | bc)
echo -e "\nAverage Waiting Time-- $avg_wt"
echo "Average Turnaround Time-- $avg_tat"
```

Input output:

```
INPUT
Enter the number of processes--
3
Enter burst time for Process 0--
24
Enter burst time for Process 1--
3
Enter burst time for Process 2--
3
OUTPUT

Process Burst Time      Waiting Time    Turnaround Time
P0      24              0               24
P1      3               24              27
P2      3               27              30

Average Waiting Time-- 17.000000
Average Turnaround Time-- 27.000000
```

2.Shortest Job First (SJF)

Problem analysis:

The SJF problem is a CPU scheduling method where the process with the **smallest burst time** is executed first. It can be **non-preemptive** (once started, the process runs until completion) or **preemptive** (Shortest Remaining Time First, SRTF). SJF is efficient because it reduces the **average waiting time** compared to FCFS. However, it requires knowing the burst time of each process in advance, which is not always possible. Another drawback is the risk of **starvation**, where longer processes may wait indefinitely if shorter ones keep arriving.

Code:

```
echo "INPUT"

echo -n "Enter number of processes: "

read n

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

do

   echo -n "Enter burst time for process $((i))-- "

   read bt[$i]

   pid[$i]=$i

done

for (( i=0; i<n-1; i++ ))

do

   for (( j=0; j<n-i-1; j++ ))

   do

      if [ ${bt[j]} -gt ${bt[j+1]} ]

      then

         # Swap burst time

         temp=${bt[j]}

         bt[j]=${bt[j+1]}

         bt[j+1]=$temp


         # Swap process ID
```

```
        temp=${pid[j]}
        pid[j]=${pid[j+1]}
        pid[j+1]=$temp
      fi
  done
done
wt[0]=0
tat[0]=${bt[0]}
total_wt=0
total_tat=0

for (( i=1; i<n; i++ ))
do
  wt[$i]=$(( wt[i-1] + bt[i-1] ))
  tat[$i]=$(( wt[i] + bt[i] ))
done

echo "OUTPUT"
echo -e "\nProcess\tBurst Time\tWaiting Time\tTurnaround Time"
for (( i=0; i<n; i++ ))
do
  echo -e "p${pid[$i]}\t${bt[i]}\t\t${wt[i]}\t\t${tat[i]}"
```

```
   total_wt=$(( total_wt + wt[i] ))
   total_tat=$(( total_tat + tat[i] ))
done
avg_wt=$(echo "scale=6; $total_wt / $n" | bc)
avg_tat=$(echo "scale=6; $total_tat / $n" | bc)
echo -e "\nAverage Waiting Time-- $avg_wt"
echo -e "Average Turnaround Time-- $avg_tat"
```

Input output:

```
INPUT
Enter number of processes: 4
Enter burst time for process 0-- 6
Enter burst time for process 1-- 8
Enter burst time for process 2-- 7
Enter burst time for process 3-- 3
OUTPUT

Process Burst Time      Waiting Time    Turnaround Time
p3      3               0               3
p0      6               3               9
p2      7               9               16
p1      8               16              24

Average Waiting Time-- 7.000000
Average Turnaround Time-- 13.000000
```

3.Priority Scheduling

Problem analysis:

Priority Scheduling is a CPU scheduling method where each process is assigned a **priority value**, and the CPU is given to the process with the **highest priority**. It can be **preemptive** (a higher-priority process interrupts a running lower-priority one) or **non-preemptive** (the running process continues until it finishes). This method is flexible since priorities can be based on importance, resource needs, or deadlines. However, it may cause **starvation**, where low-priority processes wait too long. To solve this, **aging** is used, which gradually increases the priority of waiting processes to ensure fairness.

Code:

```
echo "INTPUT"

echo  "Enter the number of processes -- "

read n

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

do

   echo -n "Enter the Burst Time & Priority of Process $i --- "

   read bt[$i] pr[$i]

   pid[$i]=$i

done
```

```
for (( i=0; i<n-1; i++ ))
do
  for (( j=0; j<n-i-1; j++ ))
  do
    if [ ${pr[j]} -gt ${pr[j+1]} ]
    then
      temp=${pr[j]}
      pr[j]=${pr[j+1]}
      pr[j+1]=$temp
      temp=${bt[j]}
      bt[j]=${bt[j+1]}
      bt[j+1]=$temp
      temp=${pid[j]}
      pid[j]=${pid[j+1]}
      pid[j+1]=$temp
    fi
  done
done
wt[0]=0
tat[0]=${bt[0]}
total_wt=0
total_tat=0
```

```
for (( i=1; i<n; i++ ))
do
   wt[$i]=$(( wt[i-1] + bt[i-1] ))
   tat[$i]=$(( wt[i] + bt[i] ))
done
echo -e "\nOUTPUT"
printf "%-10s %-10s %-12s %-15s %-10s\n" "PROCESS" "PRIORITY" "BURST TIME" "WAITING TIME" "TURNAROUND TIME"
for (( i=0; i<n; i++ ))
do
   printf "%-10s %-10s %-12s %-15s %-10s\n" "$((pid[i]))" "${pr[i]}" "${bt[i]}" "${wt[i]}" "${tat[i]}"
   total_wt=$(( total_wt + wt[i] ))
   total_tat=$(( total_tat + tat[i] ))
done
# Averages

avg_wt=$(echo "scale=6; $total_wt / $n" | bc)
avg_tat=$(echo "scale=6; $total_tat / $n" | bc)
echo -e "\nAverage Waiting Time is --- $avg_wt"
echo -e "Average Turnaround Time is --- $avg_tat"
```

Input output:

```
INTPUT
Enter the number of processes --
5
Enter the Burst Time & Priority of Process 0 --- 10 3
Enter the Burst Time & Priority of Process 1 --- 1 1
Enter the Burst Time & Priority of Process 2 --- 2 4
Enter the Burst Time & Priority of Process 3 --- 1 5
Enter the Burst Time & Priority of Process 4 --- 5 2

OUTPUT
PROCESS     PRIORITY    BURST TIME    WAITING TIME    TURNAROUND TIME
1           1           1             0               1
4           2           5             1               6
0           3           10            6               16
2           4           2             16              18
3           5           1             18              19

Average Waiting Time is --- 8.200000
Average Turnaround Time is --- 12.000000
```

4. Round Robin Scheduling

Problem analysis:

Round Robin is a **preemptive CPU scheduling** technique mainly used in **time-sharing systems**. Every process gets a fixed time unit called a **time quantum**. When the time ends, the process is moved to the back of the ready queue, and the CPU is given to the next process. This ensures **fairness**, as all processes get equal CPU time. If the quantum is too large, waiting time increases, and if it is too small, frequent **context switching** reduces efficiency.

Code:

```
echo "INPUT:"

echo  "Enter the number of processes-- "

read n

declare -a pid bt rt wt tat completed

# Input burst times

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

do

   pid[i]=$((i+1))

   echo -n "Enter Burst Time for process $((i+1)) -- "

   read bt[i]

   rt[i]=${bt[i]}

   wt[i]=0

   tat[i]=0

   completed[i]=0

done

echo -n "Enter the size of time slice -- "

read tq

time=0

remain=$n

while (( remain > 0 ))

do
```

```
  for ((i=0; i<n; i++))
  do
    if (( rt[i] > 0 ))
    then
      if (( rt[i] > tq ))
      then
        time=$((time + tq))
        rt[i]=$((rt[i] - tq))
      else
        time=$((time + rt[i]))
        wt[i]=$((time - bt[i]))
        rt[i]=0
        tat[i]=$((bt[i] + wt[i]))
        completed[i]=1
        remain=$((remain - 1))
      fi
    fi
  done
done
echo -e "\nOUTPUT:"
echo -e "PROCESS\tBURST TIME\tWAITING TIME\tTURNAROUND TIME"
total_wt=0
```

```
total_tat=0

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

do

    echo -e "${pid[i]}\t${bt[i]}\t\t${wt[i]}\t\t${tat[i]}"

    total_wt=$((total_wt + wt[i]))

    total_tat=$((total_tat + tat[i]))

done

avg_wt=$(echo "scale=6; $total_wt / $n" | bc)

avg_tat=$(echo "scale=6; $total_tat / $n" | bc)

echo -e "\nThe Average Turnaround time is-- $avg_tat"

echo -e "Average Waiting time is----------- $avg_wt"
```

Input output:

```
INPUT:
Enter the number of processes--
3
Enter Burst Time for process 1 -- 24
Enter Burst Time for process 2 -- 3
Enter Burst Time for process 3 -- 3
Enter the size of time slice -- 3

OUTPUT:
PROCESS BURST TIME      WAITING TIME    TURNAROUND TIME
1       24              6               30
2       3               3               6
3       3               6               9

The Average Turnaround time is-- 15.000000
Average Waiting time is----------- 5.000000
```

5. Producer-Consumer Problem

Problem Analysis:

The Producer-Consumer problem manages a fixed-size buffer where the producer adds items and the consumer removes them. It prevents overflow and underflow, ensuring smooth data flow. Performance depends on buffer size; small buffers fill quickly, while large buffers can store more items before being full.

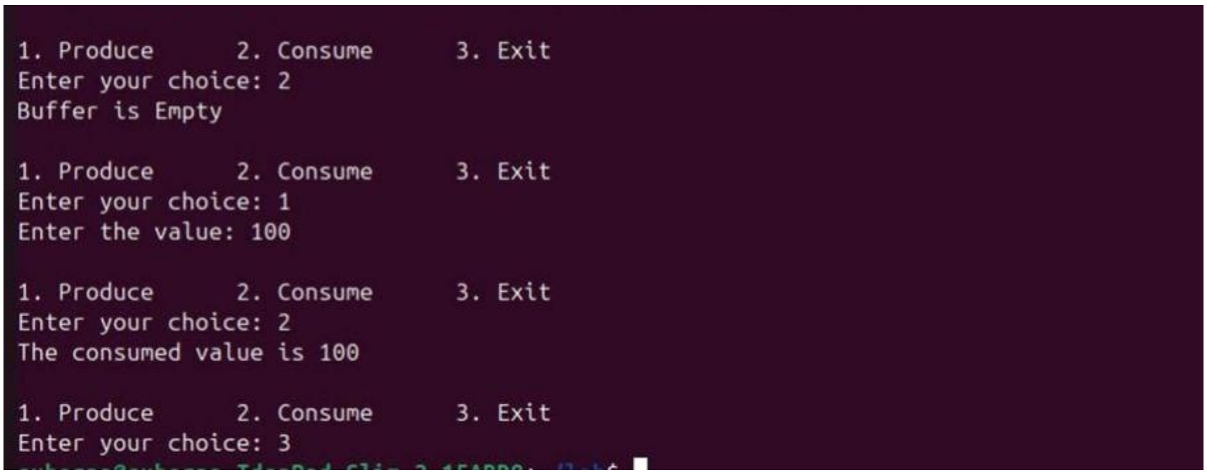
Code:

```
#include <stdio.h>

#include <stdlib.h>

int buffer = 0;

int full = 0;

void produce() {

    if (full == 1) {

        printf("Buffer is Full\n");

    } else {

        printf("Enter the value: ");

        scanf("%d", &buffer);

        full = 1;

    }

}
```

```c
void consume() {
    if (full == 0) {
        printf("Buffer is Empty\n");
    } else {
        printf("The consumed value is %d\n", buffer);
        buffer = 0;
        full = 0;
    }
}
int main() {
    int choice;
    while (1) {
        printf("\n1. Produce\t2. Consume\t3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: produce(); break;
            case 2: consume(); break;
            case 3: exit(0);
            default: printf("Invalid choice\n");
```

```
      }
   }
   return 0;
}
```

Input Output:



6.Dining-Philosophers Problem

Problem Analysis:

The Dining Philosophers problem is a classic example of **synchronization** in concurrent systems. It models five philosophers sitting around a table with a bowl of rice and five chopsticks. Each philosopher alternates between **thinking** and **eating**, but to eat, a philosopher needs **two chopsticks**. If all philosophers pick up one chopstick at the same time, it can cause a **deadlock**. The problem highlights the need for proper **resource allocation** to avoid deadlock and starvation while allowing all philosophers to eat.

Code:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

   int n, h, choice;

   int hungry[20];

   int i, j, found;

   printf("DINING PHILOSOPHER PROBLEM\n");

   printf("Enter the total no. of philosophers: ");

   scanf("%d", &n);

   printf("How many are hungry : ");

   scanf("%d", &h);

   for (i = 0; i < h; i++) {

      printf("Enter philosopher %d position: ", i + 1);

      scanf("%d", &hungry[i]);

   }

   while (1) {

      printf("\nOUTPUT\n");

      printf("1. One can eat at a time\n");

      printf("2. Two can eat at a time\n");

      printf("3. Exit\n");

      printf("Enter your choice: ")
```

```
    scanf("%d", &choice);
     if (choice == 1) {
        printf("\nAllow one philosopher to eat at any time\n");
        for (i = 0; i < h; i++) {
           printf("P %d is granted to eat\n", hungry[i]);
           for (j = 0; j < h; j++) {
              if (i != j)
                 printf("P %d is waiting\n", hungry[j]);
           }
        }
     }

     else if (choice == 2) {
        printf("\nAllow two philosophers to eat at the same time\n");
        found = 0;
        for (i = 0; i < h; i++) {
           for (j = i + 1; j < h; j++) {
               if (abs(hungry[i] - hungry[j]) != 1 &&
                 abs(hungry[i] - hungry[j]) != (n - 1))
                 printf("P %d and P %d are granted to eat\n", hungry[i],
hungry[j]);
                 for (int k = 0; k < h; k++) {
                    if (k != i && k != j)
```

```
                printf("P %d is waiting\n", hungry[k]);
            }
            found = 1;
            break;
          }
        }
        if (found) break;
      }
      if (!found) {
        printf("No two philosophers can eat together (all are neighbours)\n");
      }
    }else if (choice == 3) {
      printf("Exiting...\n");
      exit(0);
    }else {
      printf("Invalid choice!\n");
    }
  }
  return 0;
}
```

Input Output:

```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry : 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 5

OUTPUT
1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 1

Allow one philosopher to eat at any time
P 2 is granted to eat
P 4 is waiting
P 5 is waiting
P 4 is granted to eat
P 2 is waiting
P 5 is waiting
P 5 is granted to eat
P 2 is waiting
P 4 is waiting

OUTPUT
1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 3
Exiting...
```

7.MFT

Problem Analysis:

MFT (Multiprogramming with Fixed Partitions) allocates fixed-size memory blocks to processes. A process is loaded if it fits, otherwise it waits. Internal fragmentation occurs when blocks are partially used, and external fragmentation is leftover memory outside the blocks. Performance depends on block size and process memory requirements.

Code:

```
read -p "Enter the total memory available (in Bytes) -- " total_mem
read -p "Enter the block size (in Bytes) -- " block_size
read -p "Enter the number of processes -- " num_proc
declare -a processes
for ((i=1; i<=num_proc; i++))
do
   read -p "Enter memory required for process $i (in Bytes) -- " p
   processes[$i]=$p
done
num_blocks=$((total_mem / block_size))
echo -e "\nNo. of Blocks available in memory -- $num_blocks\n"
allocated=0
```

```
internal_frag=0
external_frag=0
echo -e "OUTPUT"
echo -e "PROCESS\tMEMORY REQUIRED\tALLOCATED\tINTERNAL FRAGMENTATION"
for ((i=1; i<=num_proc; i++))
do
  p=${processes[$i]}
  if [ $allocated -lt $num_blocks ]; then
    if [ $p -le $block_size ]; then
      allocated=$((allocated + 1))
      frag=$((block_size - p))
      internal_frag=$((internal_frag + frag))
      echo -e "$i\t$p\t\tYES\t\t$frag"
    else
      echo -e "$i\t$p\t\tNO\t\t-----"
    fi
  else
    echo -e "$i\t$p\t\tNO\t\t-----"
    external_frag=$((external_frag + p))
  fi
done
```

echo -e "\nMemory is Full, Remaining Processes cannot be accommodated"

echo "Total Internal Fragmentation is $internal_frag"

echo "Total External Fragmentation is $external_frag"

Input Output:

```
Enter the total memory available (in Bytes) -- 1000
Enter the block size (in Bytes) -- 300
Enter the number of processes -- 5
Enter memory required for process 1 (in Bytes) -- 275
Enter memory required for process 2 (in Bytes) -- 400
Enter memory required for process 3 (in Bytes) -- 290
Enter memory required for process 4 (in Bytes) -- 293
Enter memory required for process 5 (in Bytes) -- 100

No. of Blocks available in memory -- 3

OUTPUT
PROCESS MEMORY REQUIRED ALLOCATED       INTERNAL FRAGMENTATION
1       275             YES             25
2       400             NO              -----
3       290             YES             10
4       293             YES             7
5       100             NO              -----

Memory is Full, Remaining Processes cannot be accommodated
Total Internal Fragmentation is 42
Total External Fragmentation is 100
```

8.MVT

Problem analysis:

The **MVT** (Multiprogramming with Variable Tasks) method allocates memory to processes based on their requested size. A process gets memory only if enough is available, and leftover memory may cause external fragmentation. MVT helps use memory efficiently, but performance depends on process sizes and order—large or uneven requests can leave gaps in memory.

Code:

```
# MVT (Multiprogramming with Variable Tasks) Algorithm Simulation
echo "Enter total memory available (in Bytes)--"
read ms
total=$ms
allocated=0
declare -a process
declare -a mem
i=0
while [ $ms -gt 0 ]
do
  echo "Enter memory required for process $((i+1)) (in Bytes)--"
  read req
  if [ $req -le $ms ]
  then
    process[$i]=$((i+1))
    mem[$i]=$req
    ms=$((ms - req))
    allocated=$((allocated + req))
    echo "Memory is allocated for Process $((i+1))"
```

```
  else
    echo "Memory is Full"
    break
  fi
  echo "Do you want to continue (y/n)--"
  read choice
  if [ "$choice" = "n" ]
  then
    break
  fi
  i=$((i+1))
done

echo
echo "Total Memory Available = $total"
echo
echo "PROCESS   MEMORY ALLOCATED"
for j in $(seq 0 $i)
do
  echo "   ${process[$j]}        ${mem[$j]}"
done
echo
```

echo "Total Memory Allocated = $allocated"

extfrag=$((total - allocated))

echo "Total External Fragmentation = $extfrag"

Input Output:

```
Enter total memory available (in Bytes)--
1000
Enter memory required for process 1 (in Bytes)--
400
Memory is allocated for Process 1
Do you want to continue (y/n)--
y
Enter memory required for process 2 (in Bytes)--
275
Memory is allocated for Process 2
Do you want to continue (y/n)--
y
Enter memory required for process 3 (in Bytes)--
550
Memory is Full

Total Memory Available = 1000

PROCESS   MEMORY ALLOCATED
   1          400
   2          275


Total Memory Allocated = 675
Total External Fragmentation = 325
```