Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

Experiment Title: Process Scheduling

Aim/Objective:

The objective is to efficiently utilize the available system resources and ensure fair and timely execution of processes. Process scheduling involves determining which process should be allocated to the CPU (Central Processing Unit) at any given time, considering factors such as priority, fairness, and efficiency.

Description:

The primary objective of process scheduling is to make the best possible use of the CPU resources available in the system. The scheduling algorithm aims to keep the CPU busy by constantly assigning it to a process for execution. By minimizing idle time and maximizing CPU utilization, the system can achieve optimal performance and throughput.

Pre-Requisites:

- Knowledge of simple system calls and process scheduling

Pre-Lab:

SCHEDULING ALGORITHMS	FUNCTIONALITY
First Come First Scheduling	Executes processes in arrival order, no preemption.
Shortest Job First Scheduling	Executes the process with the shortest burst time.

Shortest Remaining Time First Scheduling	Executes the process with the least remaining burst time.
Round Robin Scheduling	Allocates equal time slices, rotates through processes cyclically.
Priority Scheduling	Executes based on process priority, preemptive or non-preemptive.

In-Lab:

1. Write a C program to implement the FCFS process scheduling algorithm.

```
#include <stdio.h>

struct Process {
   int processID;
   int arrivalTime;
   int burstTime;
};

int main() {
   int n;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   struct Process processes[n];

   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;

      printf("Enter arrival time for process P%d: ", i + 1);
      scanf("%d", &processes[i].arrivalTime);

      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);
   }

   int completionTime = 0;
   float totalWaitingTime = 0;

   printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\n");

   for (int i = 0; i < n; i++) {
      if (processes[i].arrivalTime > completionTime) {
         completionTime = processes[i].arrivalTime;
      }

      completionTime += processes[i].burstTime;
```

```
        float waitingTime = completionTime - processes[i].arrivalTime - processes[i].burstTime;
        totalWaitingTime += waitingTime;

        printf("P%d\t%d\t\t%d\t\t%d\t\t%.2f\n",
            processes[i].processID,
            processes[i].arrivalTime,
            processes[i].burstTime,
            completionTime,
            waitingTime
        );
    }

    float averageWaitingTime = totalWaitingTime / n;

    printf("\nAverage Waiting Time: %.2f\n", averageWaitingTime);

    return 0;
}
```

2. Write a C program to implement the SJF process scheduling algorithm.

```
#include <stdio.h>

struct Process {
   int processID;
   int arrivalTime;
   int burstTime;
   int waitingTime;
   int turnaroundTime;
   int completed;
};

int main() {
   int n;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   struct Process processes[n];

   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;

      printf("Enter arrival time for process P%d: ", i + 1);
      scanf("%d", &processes[i].arrivalTime);

      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);

      processes[i].completed = 0;
   }

   int currentTime = 0;
   int completedProcesses = 0;

   printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

   while (completedProcesses < n) {
      int shortestJob = -1;
      int shortestBurstTime = 9999;
```

```
    for (int i = 0; i < n; i++) {
      if (!processes[i].completed &&
        processes[i].arrivalTime <= currentTime &&
        processes[i].burstTime < shortestBurstTime) {
        shortestJob = i;
        shortestBurstTime = processes[i].burstTime;
      }
    }

    if (shortestJob == -1) {
      currentTime++;
    } else {
      int p = shortestJob;

      processes[p].waitingTime = currentTime - processes[p].arrivalTime;
      processes[p].turnaroundTime = processes[p].waitingTime + processes[p].burstTime;

      currentTime += processes[p].burstTime;
      processes[p].completed = 1;
      completedProcesses++;

      printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n",
        processes[p].processID,
        processes[p].arrivalTime,
        processes[p].burstTime,
        processes[p].waitingTime,
        processes[p].turnaroundTime
      );
    }
  }

  return 0;
}
```

3. Write a C program to implement the Round Robin process scheduling algorithm.

```
#include <stdio.h>

struct Process {
   int processID;
   int burstTime;
   int remainingTime;
   int waitingTime;
};

int main() {
   int n, timeQuantum;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   printf("Enter the time quantum: ");
   scanf("%d", &timeQuantum);

   struct Process processes[n];

   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;

      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);

      processes[i].remainingTime = processes[i].burstTime;
      processes[i].waitingTime = 0;
   }

   int currentTime = 0;
   int completedProcesses = 0;

   printf("\nProcess\tBurst Time\tWaiting Time\n");

   while (completedProcesses < n) {
      for (int i = 0; i < n; i++) {

         if (processes[i].remainingTime > 0) {
```

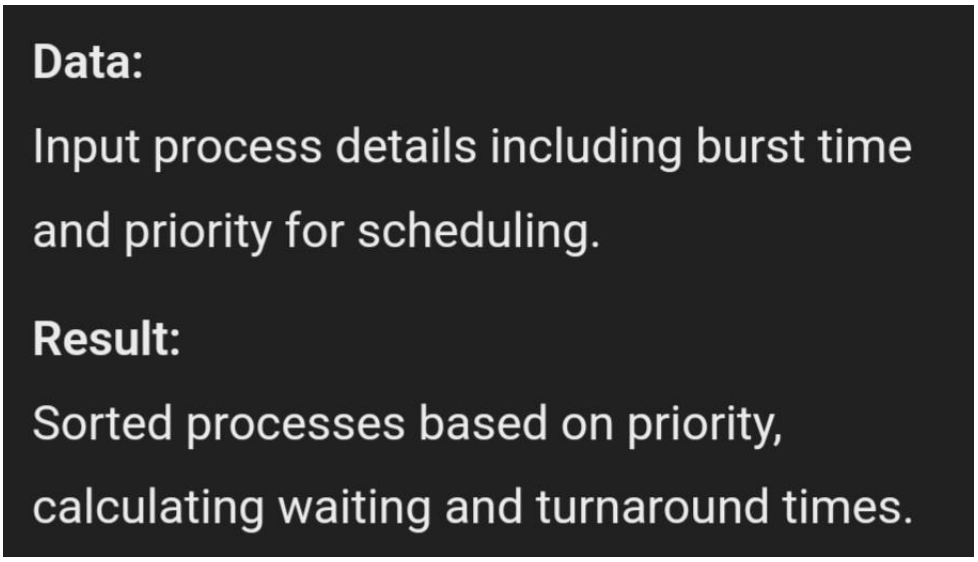
```
            if (processes[i].remainingTime <= timeQuantum) {
              currentTime += processes[i].remainingTime;

              processes[i].waitingTime += currentTime - processes[i].burstTime;
              processes[i].remainingTime = 0;
              completedProcesses++;
            } else {
              currentTime += timeQuantum;
              processes[i].remainingTime -= timeQuantum;
            }
          }
        }
      }

      for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\n",
          processes[i].processID,
          processes[i].burstTime,
          processes[i].waitingTime
        );
      }

      return 0;
}
```

4. Write a C program to implement the Priority process scheduling algorithm.

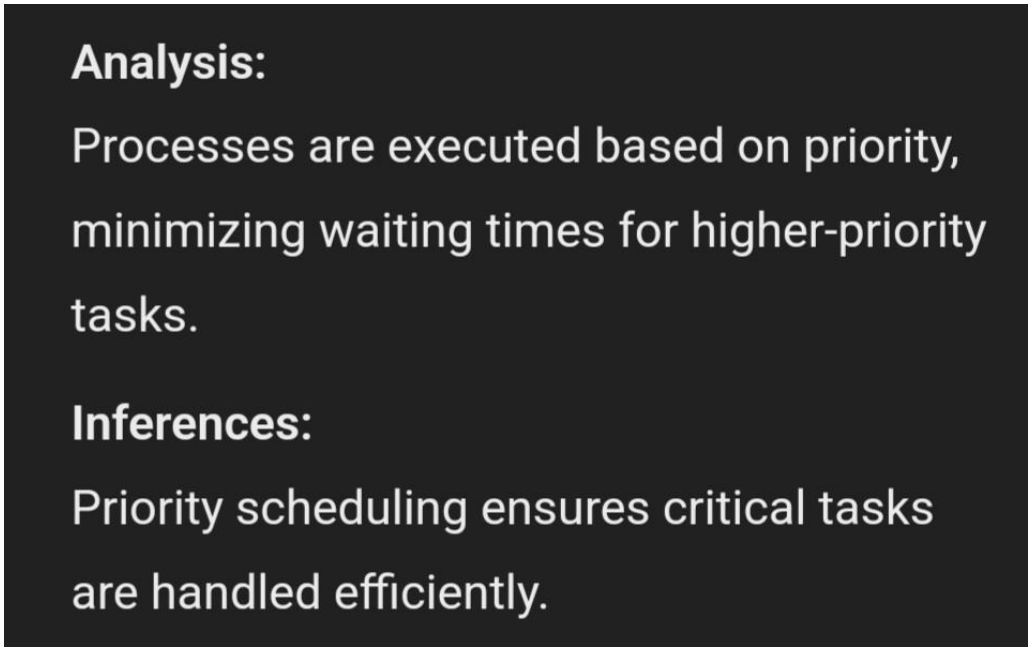
```
#include <stdio.h>

struct Process {
   int processID;
   int burstTime;
   int priority;
   int waitingTime;
   int turnaroundTime;
};

void sortProcesses(struct Process processes[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (processes[j].priority > processes[j + 1].priority) {
            struct Process temp = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = temp;
         }
      }
   }
}

int main() {
   int n;


    printf("Enter the number of processes: ");
   scanf("%d", &n);

   struct Process processes[n];

   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;

      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);

      printf("Enter priority for process P%d: ", i + 1);
```

```
    scanf("%d", &processes[i].priority);


    processes[i].waitingTime = 0;
    processes[i].turnaroundTime = 0;
  }

  sortProcesses(processes, n);

  int currentTime = 0;

  printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
  for (int i = 0; i < n; i++) {
    processes[i].waitingTime = currentTime;
    processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;
    currentTime += processes[i].burstTime;

    printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n",
      processes[i].processID,
      processes[i].burstTime,
      processes[i].priority,
      processes[i].waitingTime,
      processes[i].turnaroundTime
    );
  }

  return 0;
}
```

- **Data and Result:**



- **Analysis and Inferences:**



1. Write C Program to simulate Multi-Level Feedback Queue CPU Scheduling algorithm.

- **Procedure/Program:**

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

struct Process {
   int processID;
   int burstTime;
   int priority;
};

int main() {
   int n, quantum1, quantum2;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   printf("Enter time quantum for first queue: ");
   scanf("%d", &quantum1);
   printf("Enter time quantum for second queue: ");
   scanf("%d", &quantem2);

   struct Process processes[n];

   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;
      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);
      processes[i].priority = 1;
   }

   int currentTime = 0;
   int completedProcesses = 0;

   while (completedProcesses < n) {
      for (int i = 0; i < n; i++) {
         if (processes[i].burstTime > 0) {
            if (processes[i].burstTime <= quantum1) {
               currentTime += processes[i].burstTime;
               processes[i].burstTime = 0;
               completedProcesses++;
```

```
            } else {
               currentTime += quantum1;
               processes[i].burstTime -= quantum1;
               processes[i].priority = 2;
            }
            printf("P%d (Queue %d) completed at time %d\n", processes[i].processID,
processes[i].priority,

currentTime);
         }
      }

      for (int i = 0; i < n; i++) {
         if (processes[i].priority == 2 && processes[i].burstTime > 0) {
            if (processes[i].burstTime <= quantum2) {
               currentTime += processes[i].burstTime;
               processes[i].burstTime = 0;
               completedProcesses++;
            } else {
               currentTime += quantum2;
               processes[i].burstTime -= quantum2;
            }
            printf("P%d (Queue %d) completed at time %d\n", processes[i].processID,
processes[i].priority, currentTime);
         }
      }
   }

   return 0;
}
```

- **Data and Results:**

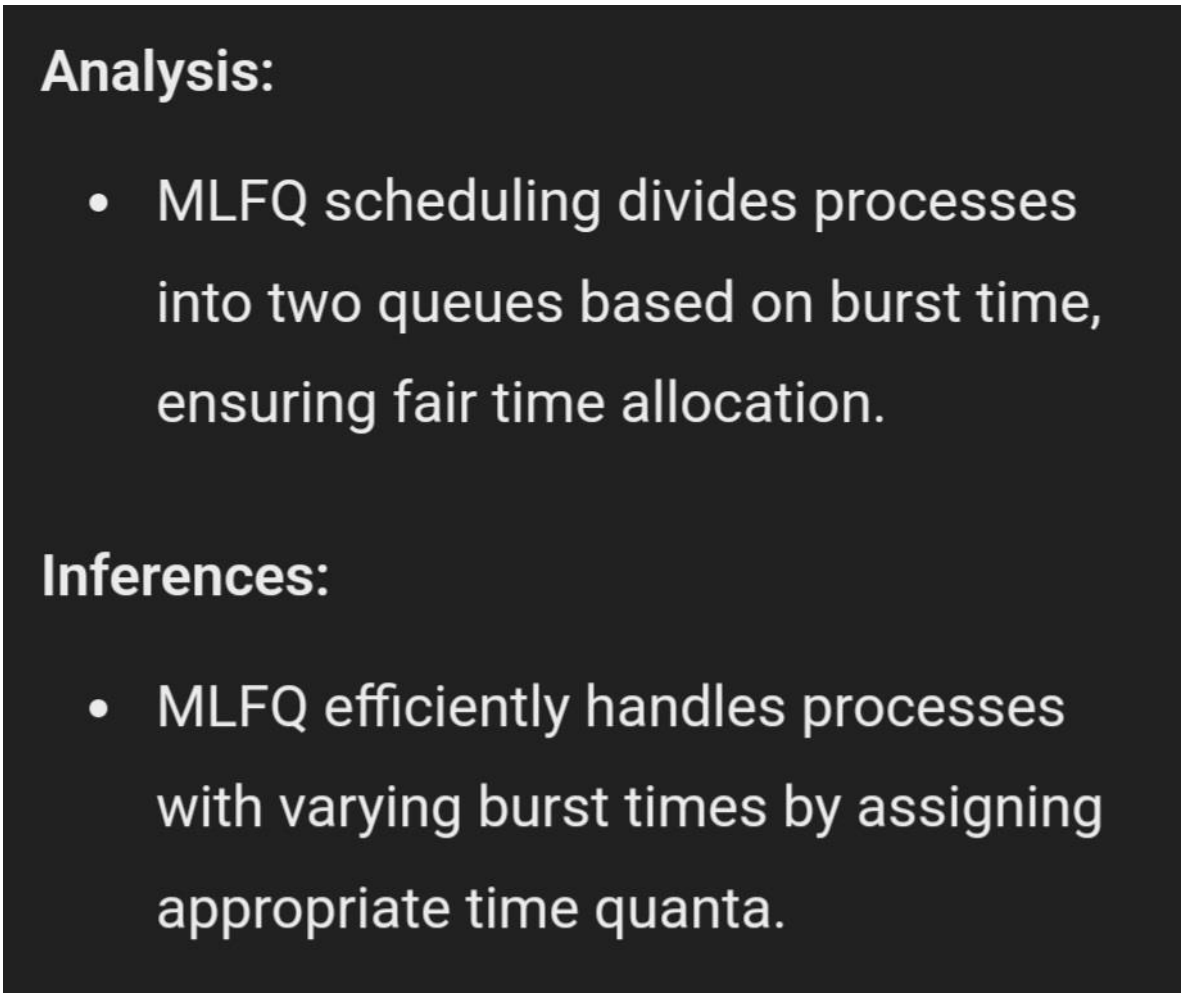
Data:

- Number of processes, burst time, time quantum values are input for MLFQ scheduling simulation.

Result:

- The completion time for each process in both queues is printed during simulation.

- **Analysis and Inferences:**



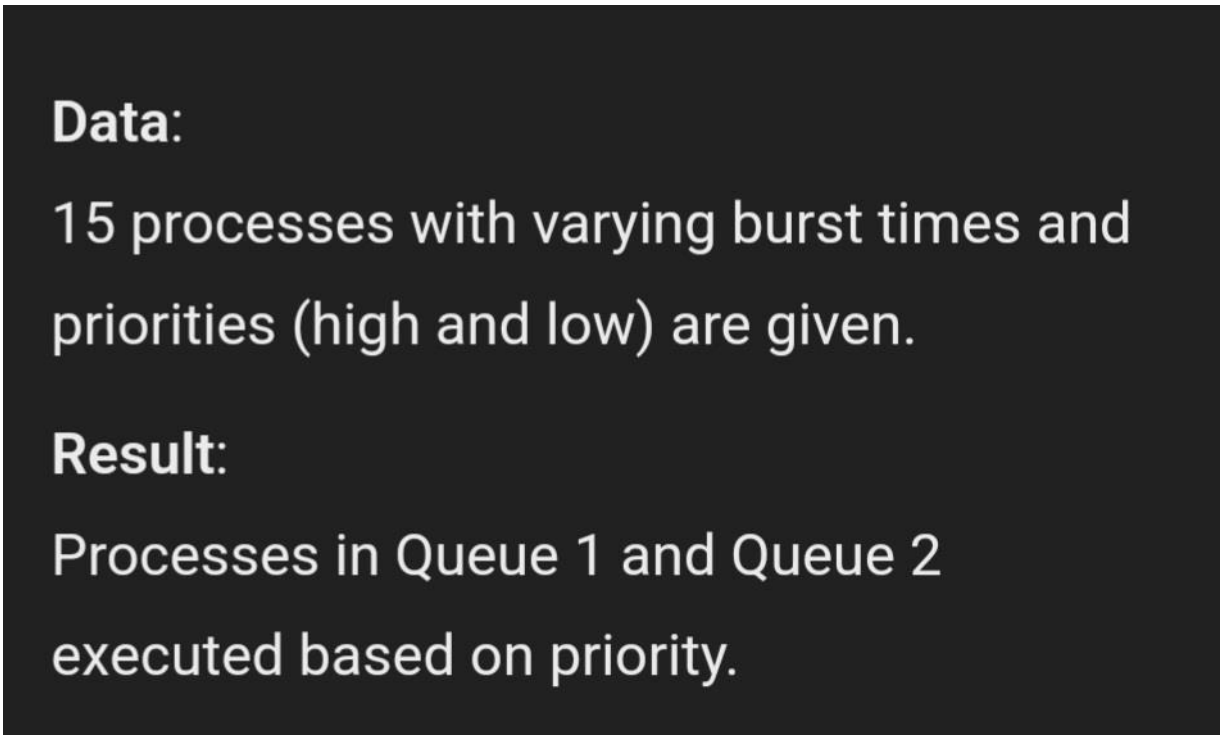
2. Write a C Program to simulate Multi Level Queue CPU Scheduling algorithm.

- **Procedure/Program:**

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

struct Process {
   int processID;
   int burstTime;
   int priority;
};

int main() {
   int n;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   struct Process processes[n];
   struct Process queue1[n];
   struct Process queue2[n];
   int front1 = -1, rear1 = -1;
   int front2 = -1, rear2 = -1;

   for (int i = 0; i < n; i++) {
      processes[i].processID = i + 1;
      printf("Enter burst time for process P%d: ", i + 1);
      scanf("%d", &processes[i].burstTime);
      printf("Enter priority for process P%d (1 for high, 2 for low): ", i + 1);
      scanf("%d", &processes[i].priority);

      if (processes[i].priority == 1) {
         if (front1 == -1) {
            front1 = 0;
         }
         rear1++;
         queue1[rear1] = processes[i];
      } else {
         if (front2 == -1) {
            front2 = 0;
         }
         rear2++;
         queue2[rear2] = processes[i];
```

```
    }
   }

   printf("\nExecuting processes in Queue 1:\n");
   for (int i = 0; i <= rear1; i++) {

     printf("Process P%d (Priority 1) completed.\n", queue1[i].processID);
   }

   printf("\nExecuting processes in Queue 2:\n");
   for (int i = 0; i <= rear2; i++) {
      printf("Process P%d (Priority 2) completed.\n", queue2[i].processID);
   }

   return 0;
}
```

- **Data and Results:**



- **Analysis and Inferences:**

Analysis:

Queue 1 processes executed first, then Queue 2 processes based on priority levels.

Inferences:

Higher priority processes are executed first, ensuring efficient task management in the system.

3. Write a C Program to simulate Multi Process Scheduling algorithm.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_CORES 4
#define NUM_PROCESSES 10

pthread_t cores[NUM_CORES];
pthread_mutex_t mutex;
void* process(void* processID) {
   int id = *((int*)processID);
   printf("Process P%d is running on Core %d\n", id, id % NUM_CORES);
   sleep(1);
   printf("Process P%d completed on Core %d\n", id, id % NUM_CORES);
   return NULL;
}
int main() {
   int processIDs[NUM_PROCESSES];
   for (int i = 0; i < NUM_PROCESSES; i++) {
      processIDs[i] = i + 1;
   }
   pthread_mutex_init(&mutex, NULL);
   for (int i = 0; i < NUM_CORES; i++) {
      pthread_create(&cores[i], NULL, process, &processIDs[i]);
   }
   for (int i = 0; i < NUM_CORES; i++) {
      pthread_join(cores[i], NULL);
   }
   pthread_mutex_destroy(&mutex);
   return 0;
}
```

- **Data and Results:**

Data:

- Number of cores: 4
- Number of processes: 10
- Each process is assigned to a core based on modulo operation.

Result:

- Processes executed on respective cores with output indicating start and completion time.

- **Analysis and Inferences:**

Analysis:

- The code demonstrates the parallel execution of processes across multiple cores with synchronization.

Inferences:

- Each core executes processes independently, ensuring efficient process management with mutex synchronization.

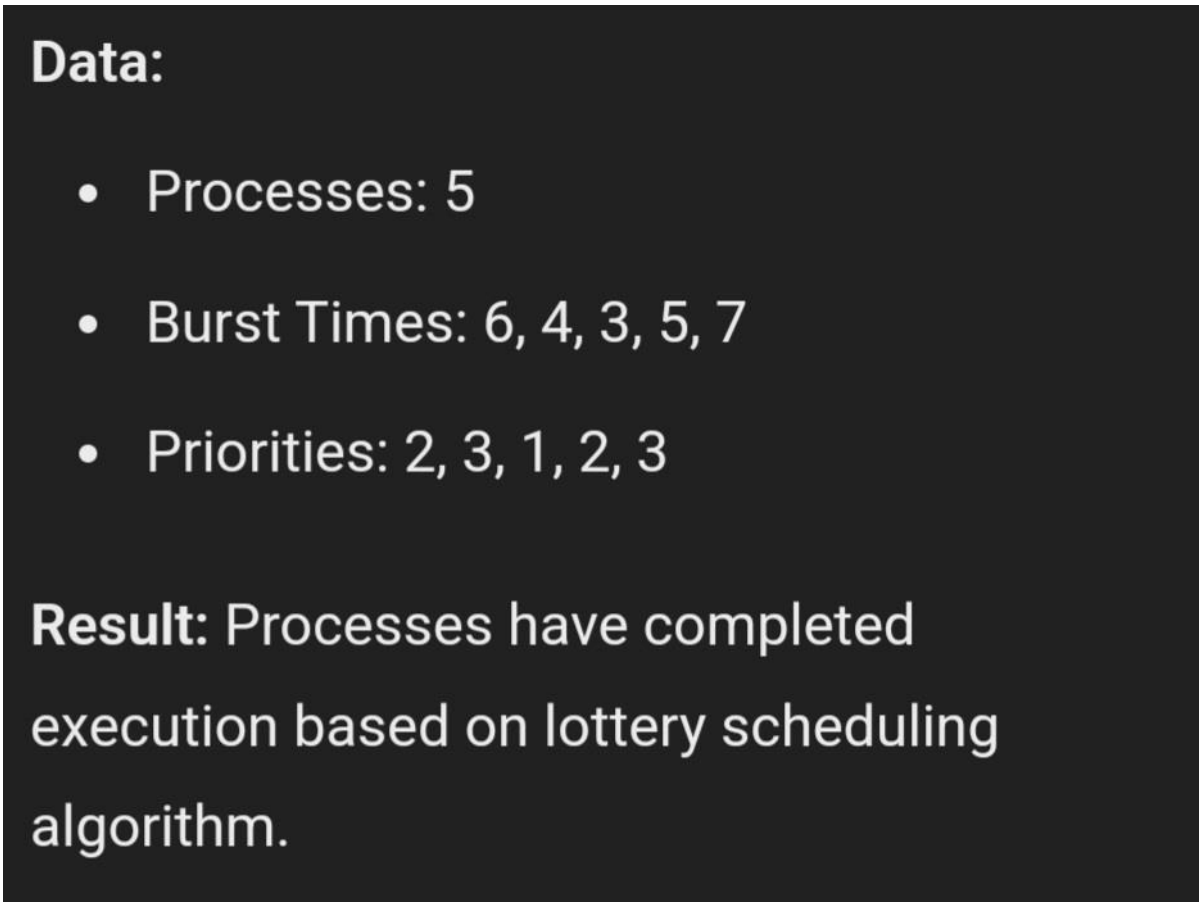
4. Write a C program to simulate the Lottery Process Scheduling algorithm.

- **Procedure/Program:**

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

struct Process {
   char name;
   int burst_time;
   int priority;
   int lottery_tickets;
};

int generate_random_number(int max) {
   srand(time(NULL));
   return (rand() % max) + 1;
}

void assign_lottery_tickets(struct Process *processes, int n) {
   int total_tickets = 0;
   for (int i = 0; i < n; i++) {
      total_tickets += processes[i].priority;
   }
   for (int i = 0; i < n; i++) {
      processes[i].lottery_tickets = (processes[i].priority / total_tickets) * 100;
   }
}

struct Process *select_next_process(struct Process *processes, int n) {
   int random_ticket = generate_random_number(100);
   int current_ticket = 0;
   for (int i = 0; i < n; i++) {
      current_ticket += processes[i].lottery_tickets;
      if (random_ticket <= current_ticket) {
         return &processes[i];
      }
   }
   return NULL;
}
```

```
void simulate_lottery_process_scheduling(struct Process *processes, int n) {
   int time = 0;
   while (1) {
      struct Process *next_process = select_next_process(processes, n);
      if (next_process == NULL) {
         break;

      }
      next_process->burst_time--;
      time++;
      if (next_process->burst_time == 0) {
         for (int i = 0; i < n; i++) {
            if (processes[i].name == next_process->name) {
               processes[i] = processes[n - 1];
               n--;
               break;
            }
         }
      }
   }
}

int main() {
   int n;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   struct Process processes[n];
   for (int i = 0; i < n; i++) {
      printf("Enter the name of process %d: ", i + 1);
      scanf(" %c", &processes[i].name);
      printf("Enter the burst time of process %d: ", i + 1);
      scanf("%d", &processes[i].burst_time);
      printf("Enter the priority of process %d: ", i + 1);
      scanf("%d", &processes[i].priority);
   }
   assign_lottery_tickets(processes, n);
   simulate_lottery_process_scheduling(processes, n);
   printf("The processes have finished executing.\n");
   return 0;
}
```

- **Data and Results:**



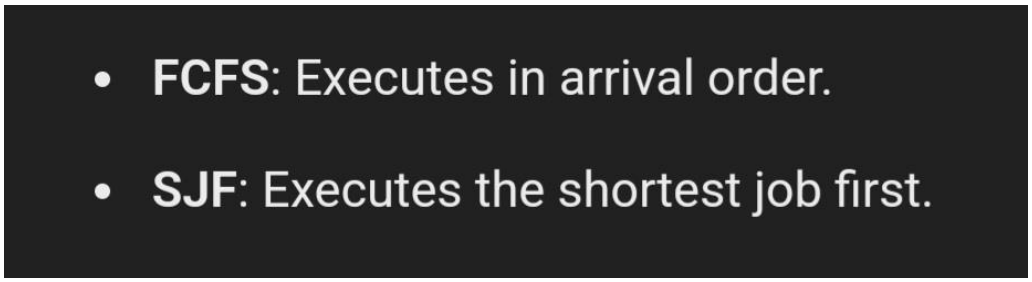
- **Analysis and Inferences:**

Analysis: The lottery scheduling allocates CPU time based on process priority and lottery tickets assigned.

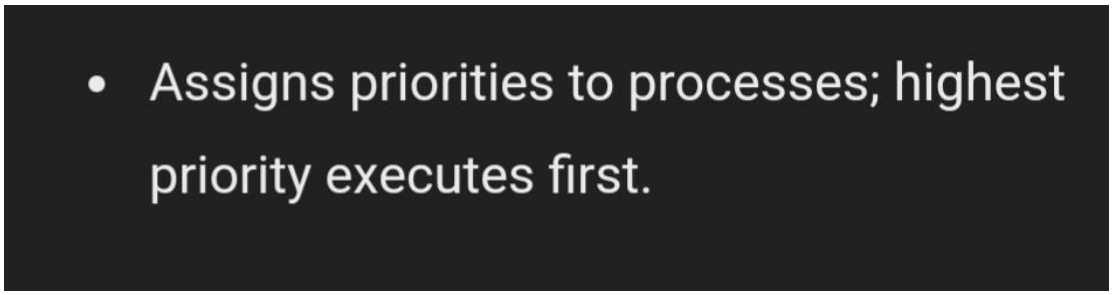
Inferences: Higher priority processes have more chances to be selected and executed first.

Sample VIVA-VOCE Questions (In-Lab):

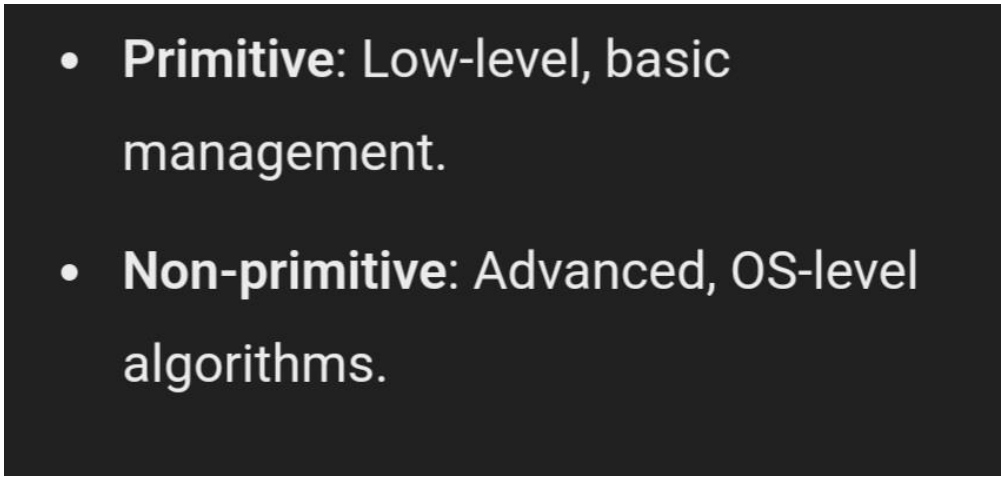
1. **What is the difference between FCFS and SJF Scheduling Algorithms**



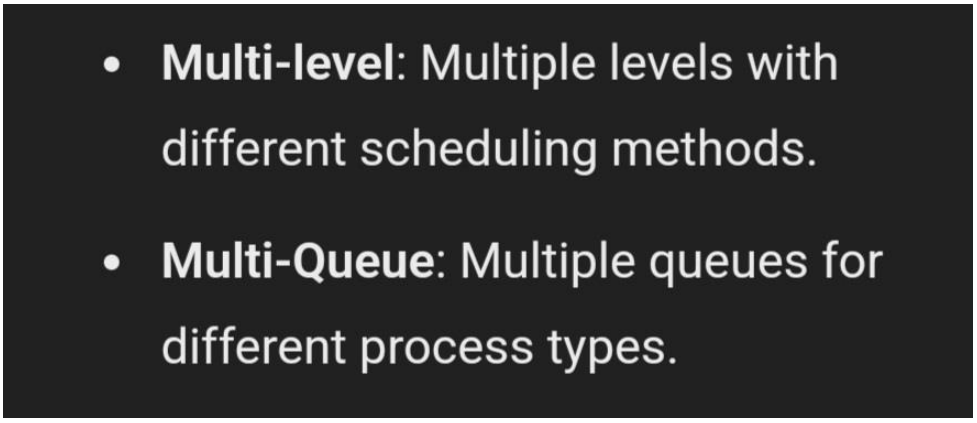
2. **What is a priority scheduling algorithm?**



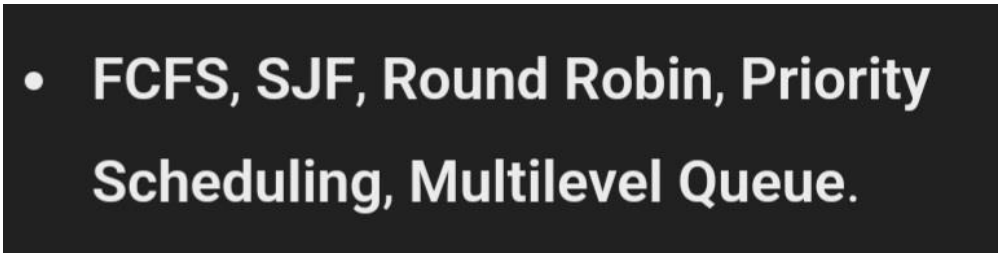
3. **What are the differences between primitive and non-primitive scheduling algorithms?**



4. Explain the concept of multi-level and Multi-Queue scheduling.



5. What are the different types of CPU Scheduling Algorithms?



Evaluator Remark (if any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce before signing and posting marks for each experiment.