Lab # 3

Note

Please copy and paste your C code in this word file and paste your output. However, don't submit the code file. Instead, write appropriate comments in the code to help me understand your code. Reference: ChatGPT

Questions are below. Let's discuss some definitions first.

In the context of process scheduling, the terms "number of processes", "arrival time", and "burst time" are used to describe the characteristics of a set of processes to be scheduled on a system.

The "number of processes" refers to the total number of processes in the system that need to be scheduled.

The "arrival time" of a process refers to the time at which the process enters the system and is ready to be executed. In other words, it is the time at which the process becomes available for execution.

The "burst time" of a process refers to the amount of time that the process requires to complete its execution. This is the time that the process spends using system resources such as CPU time, I/O operations, and memory access.

The execution order of processes refers to the order in which the processes are executed by the system. This order is determined by the scheduling algorithm that is used by the system. The scheduling algorithm considers the arrival time and burst time of each process to determine which process should be executed next. The execution order is important as it affects the overall performance of the system, including factors such as response time, throughput, and resource utilization.

To schedule a set of processes, the operating system needs to keep track of the arrival time, burst time, and other details of each process in a data structure such as a process control block (PCB). The scheduling algorithm then uses this information to determine the execution order of the processes. Once the execution order is determined, the operating system can allocate system resources such as CPU time, I/O operations, and memory access to each process accordingly.

Question #1

Read the description and code below and fix the code if you find any issues. Copy and paste the code and paste the output. Please don't upload your C code file.

Write a C program (example below) to implement the Round Robin scheduling algorithm for a set of processes. For example, assume that the quantum time is fixed at four units and the priority of all processes is the same. The program should take input for the number of processes, arrival time, and burst time for each process and output the execution order and turnaround time for each process. Set the maximum number of processes to, say, 20.

I have the example code here. Please modify the code if required and put the comments as you see fit.

```c
#include <stdio.h>

#define MAX_PROCESSES 20

typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
} process_t;

int main() {
    int num_processes, current_time, completed_processes;
    process_t processes[MAX_PROCESSES];
    int turnaround_time[MAX_PROCESSES], total_turnaround_time = 0;
    int remaining_processes[MAX_PROCESSES];
    int remaining_processes_size = 0;
    int quantum = 4;
```

```c
// Get input for the number of processes and process details
printf("Enter the number of processes: ");
scanf("%d", &num_processes);
for (int i = 0; i < num_processes; i++) {
    printf("Enter the arrival time and burst time for process %d: ", i+1);
    scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
    processes[i].remaining_time = processes[i].burst_time;
    processes[i].pid = i+1;
    remaining_processes[i] = i;
    remaining_processes_size++;
}

// Initialize scheduling variables
current_time = 0;
completed_processes = 0;

// Execute the processes using Round Robin scheduling
while (completed_processes < num_processes) {
    int next_process_index = -1;

    // Find the next process to execute
    for (int i = 0; i < remaining_processes_size; i++) {
        int pid = remaining_processes[i];
        if (processes[pid].arrival_time <= current_time) {
            next_process_index = i;
            break;
        }
    }

    if (next_process_index == -1) {
        // No processes are available to execute, so increment the current time
        current_time++;
    } else {
        // Execute the next process for the fixed quantum time
        int next_process_pid = remaining_processes[next_process_index];
        if (processes[next_process_pid].remaining_time <= quantum) {
            // Process will complete before quantum is up
            current_time += processes[next_process_pid].remaining_time;
            processes[next_process_pid].remaining_time = 0;
            turnaround_time[next_process_pid] = current_time - processes[next_process_pid].arrival_time;
            total_turnaround_time += turnaround_time[next_process_pid];
            completed_processes++;
            printf("Process %d completed at time %d\n", processes[next_process_pid].pid, current_time);
        } else {
            // Process will not complete before quantum is up
            current_time += quantum;
```

```c
                processes[next_process_pid].remaining_time -= quantum;
                printf("Process %d preempted at time %d\n", processes[next_process_pid].pid, current_time);

                // Move the preempted process to the end of the remaining processes array
                for (int i = next_process_index; i < remaining_processes_size - 1; i++) {
                    remaining_processes[i] = remaining_processes[i+1];
                }
                remaining_processes[remaining_processes_size-1] = next_process_pid;
            }
        }
    }

    // Print the turnaround time for each process
    for (int i = 0; i < num_processes; i++) {
        printf("Turnaround time for process %d: %d\n", processes[i].pid, turnaround_time[i]);
    }

    // Print the average turnaround time
    printf("Average turnaround time: %.2f\n", (float)total_turnaround_time / num_processes);

    return 0;
}
```

Question #2

Read the description and code below and fix the code if you find any issues. Copy and paste the code and paste the output. Please don't upload your C code file.

Write a C program (Example code below) to implement Priority scheduling with multiple queues for a set of processes. For example, assume there are three priority levels, with priorities 1, 2, and 3, and the quantum time is fixed at four units. The program should take input for the number of processes, arrival time, burst time, and priority level for each process and output the execution order and turnaround time for each process. Set the maximum number of processes to, say, 20.

```c
#include <stdio.h>

#define MAX_PROCESSES 20
#define NUM_PRIORITY_LEVELS 3
```

```c
typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int priority;
} process_t;

int main() {
    int num_processes, current_time, completed_processes;
    process_t processes[MAX_PROCESSES];
    int turnaround_time[MAX_PROCESSES], total_turnaround_time = 0;
    int quantum = 4;
    int time_quantum_remaining[NUM_PRIORITY_LEVELS] = {0};

    // Initialize priority queues
    int priority_queues[NUM_PRIORITY_LEVELS][MAX_PROCESSES];
    int queue_sizes[NUM_PRIORITY_LEVELS] = {0};

    // Get input for the number of processes and process details
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    for (int i = 0; i < num_processes; i++) {
        printf("Enter the arrival time, burst time, and priority level for process %d: ", i+1);
        scanf("%d %d %d", &processes[i].arrival_time, &processes[i].burst_time, &processes[i].priority);
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].pid = i+1;
        priority_queues[processes[i].priority-1][queue_sizes[processes[i].priority-1]] = i;
        queue_sizes[processes[i].priority-1]++;
    }

    // Initialize scheduling variables
    current_time = 0;
    completed_processes = 0;

    // Execute the processes using Priority scheduling with multiple queues
    printf("Execution order: ");
    while (completed_processes < num_processes) {
        int next_process_index = -1;

        // Find the highest priority process that has arrived and has remaining time
        for (int i = 0; i < NUM_PRIORITY_LEVELS; i++) {
            if (queue_sizes[i] > 0 && time_quantum_remaining[i] == 0) {
                next_process_index = priority_queues[i][0];
                break;
            }
        }
        if (next_process_index == -1) {
            // No processes are available to execute, so increment the current time
            current_time++;
            for (int i = 0; i < NUM_PRIORITY_LEVELS; i++) {
                if (queue_sizes[i] > 0) {
                    time_quantum_remaining[i] = quantum;
                }
            }
        } else {
            // Execute the next process for the fixed quantum time or until it completes
            int next_process_priority = processes[next_process_index].priority;
            int next_process_pid = processes[next_process_index].pid;
            if (processes[next_process_index].remaining_time <= quantum) {
                // Process will complete before quantum is up
                current_time += processes[next_process_index].remaining_time;
                processes[next_process_index].remaining_time = 0;
                turnaround_time[next_process_index] = current_time - processes[next_process_index].arrival_time;
                total_turnaround_time += turnaround_time[next_process_index];
                completed_processes++;
                printf("P%d ", next_process_pid);

                // Remove the completed process from the priority queue
                for (int i = 0; i < queue_sizes[next_process_priority-1]-1; i++) {
                    priority_queues[next_process_priority-1][i] = priority_queues[next_process_priority-1][i+1];
                }
                queue_sizes[next_process_priority-1]--;
                        } else {
            // Process will not complete before quantum is up
            current_time += quantum;
            processes[next_process_index].remaining_time -= quantum;
            printf("P%d ", next_process_pid);

            // Move the preempted process to the end of the priority queue
            for (int i = 0; i < queue_sizes[next_process_priority-1]-1; i++) {
                priority_queues[next_process_priority-1][i] = priority_queues[next_process_priority-1][i+1];
            }
            priority_queues[next_process_priority-1][queue_sizes[next_process_priority-1]-1] = next_process_index;
            time_quantum_remaining[next_process_priority-1] = quantum;
        }
    }

    // Decrement the time quantum for each priority level that has a process waiting to execute
    for (int i = 0; i < NUM_PRIORITY_LEVELS; i++) {
        if (queue_sizes[i] > 0 && time_quantum_remaining[i] > 0) {
            time_quantum_remaining[i]--;
        }
    }
}

// Print the turnaround time for each process
printf("\nTurnaround time for each process:\n");
for (int i = 0; i < num_processes; i++) {
    printf("P%d: %d\n", processes[i].pid, turnaround_time[i]);
}

// Print the average turnaround time
printf("Average turnaround time: %.2f\n", (float)total_turnaround_time / num_processes);

return 0;
}
```

This program implements Priority scheduling with multiple queues using a fixed quantum time of four units and three priority levels. It takes input for the number of processes, arrival time, burst time, and priority level for each process, and outputs the execution order and turnaround time for each process.

The program first initializes the priority queues and scheduling variables, and then reads in the process details using the `scanf` function. It then executes the processes using the Priority scheduling algorithm with multiple queues, finding the highest priority process that has arrived and has remaining time at each iteration, and executing it for the fixed quantum time or until it completes.

The program then prints the turnaround time for each process and the average turnaround time.

Note that this implementation of Priority scheduling with multiple queues uses a fixed quantum time and assumes that all processes have different priorities. This can be adjusted to suit specific system requirements, such as using a variable time quantum or allowing processes to have the same priority. Additionally, other scheduling algorithms can be used, such as Round Robin scheduling or multi-level feedback queues.

Question # 3

Read the description and code below and fix the code if you find any issues. Copy and paste the code and paste the output. Please don’t upload your C code file.

Shortest Job First (SJF) is a scheduling algorithm in which the process with the shortest burst time is executed first. In this algorithm, the ready queue is maintained in increasing order of the burst time of the processes. When a new process arrives, it is added to the ready queue and compared with the remaining burst time of the currently executing process. If the remaining burst time of the currently executing process is longer than the burst time of the new process, the new process is executed first.

Example: Let's consider the following set of processes with their arrival time and burst time:

```
Process 1: arrival time = 0, burst time = 5
Process 2: arrival time = 1, burst time = 3
Process 3: arrival time = 2, burst time = 1
Process 4: arrival time = 3, burst time = 4
```

```
Time 0: Execute process 1 (burst time = 5)
Time 5: Execute process 2 (burst time = 3)
Time 8: Execute process 3 (burst time = 1)
Time 9: Execute process 4 (burst time = 4)
```

Advantages of SJF scheduling algorithm:
This algorithm gives the shortest average waiting time for the processes.
It is efficient when the burst time of the processes is known in advance.
Disadvantages of SJF the scheduling algorithm:
It is difficult to predict the burst time of a process accurately.
The longer processes may suffer from starvation, as they may have to wait a long time in the ready queue before getting executed.
Here's the C code to implement the SJF scheduling algorithm:

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

```c
#define MAX_PROCESSES 100

typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
} process_t;

int main() {
    int num_processes, current_time, completed_processes;
    process_t processes[MAX_PROCESSES];
    int turnaround_time[MAX_PROCESSES], total_turnaround_time = 0;
    int remaining_processes[MAX_PROCESSES];
    int remaining_processes_size = 0;

    // Get input for the number of processes and process details
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    for (int i = 0; i < num_processes; i++) {
        printf("Enter the arrival time and burst time for process %d: ", i+1);
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].pid = i+1;
        remaining_processes[i] = i;
    }

    // Initialize scheduling variables
    current_time = 0;
    completed_processes = 0;

    // Execute the processes using the SJF algorithm
    while (completed_processes < num_processes) {
        int next_process_index = -1;
        int shortest_remaining_time = 999999;

        // Find the process with the shortest remaining time
        for (int i = 0; i < remaining_processes_size; i++) {
            int pid = remaining_processes[i];
            if (processes[pid].remaining_time < shortest_remaining_time &&
processes[pid].arrival_time <= current_time) {
                next_process_index = i;
                shortest_remaining_time = processes[pid].remaining_time;
            }
        }
        if (next_process_index == -1) {
            // No processes are available to execute, so increment the current
time
            current_time++;
        } else {
            // Execute the next process
            int next_process_pid = remaining_processes[next_process_index];
            current_time += processes[next_process_pid].remaining_time;
            processes[next_process_pid].remaining_time = 0;
            turnaround_time[next_process_pid] = current_time -
processes[next_process_pid].arrival_time;
            total_turnaround_time += turnaround_time[next_process_pid];
            completed_processes++;
            printf("Process %d completed at time %d\n",
processes[next_process_pid].pid, current_time);

            // Remove the completed process from the remaining processes list
            for (int i = next_process_index; i < remaining_processes_size - 1;
i++) {
                remaining_processes[i] = remaining_processes[i+1];
            }
            remaining_processes_size--;
        }
    }

    // Print the turnaround time for each process
    for (int i = 0; i < num_processes; i++) {
        printf("Turnaround time for process %d: %d\n", processes[i].pid,
turnaround_time[i]);
    }

    // Print the average turnaround time
    printf("Average turnaround time: %.2f\n", (float)total_turnaround_time /
num_processes);

    return 0;
}
```

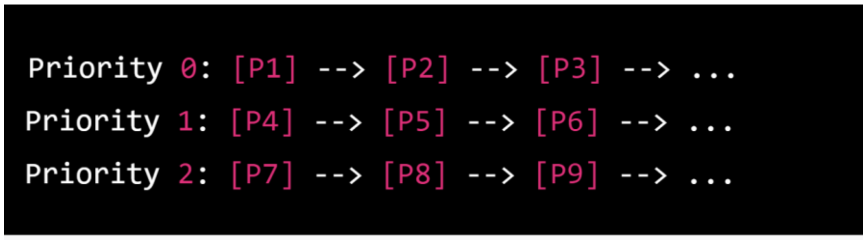
In this program, we define a struct `process_t` to store the details of each process, including the process ID, arrival time, burst time, and remaining time. We then take input for the number of processes and process details using the `scanf` function.
We then initialize the scheduling variables, including the current time and the number of completed processes. We also create an array `remaining_processes` to store the processes that have not yet been completed.
We use the SJF algorithm to execute the processes in the program's main loop. We first find the process with the shortest remaining time using a for loop that iterates over the `remaining_processes` array. If no process is available to execute, we increment the current time. Otherwise, we execute the process with the shortest remaining time and update the turnaround time and complete processes variables accordingly. We then remove the completed process from the `remaining_processes` array.
Finally, we print the turnaround time for each process and the average turnaround time.
Note that this implementation of the SJF algorithm assumes that the burst time of each process is known in advance, which is only sometimes the case in real-world scenarios. In practice, several variations of the SJF algorithm take into account the unpredictability of process burst times, such as the Shortest Remaining Time First (SRTF) algorithm, which pre-empts the currently executing process if a new process with a shorter burst time arrives.
The SJF algorithm is an effective scheduling algorithm that minimizes the process's average waiting time. However, it requires accurate estimates of the burst time of each process and may suffer from starvation of longer processes.

Question # 4

Read the description and code below and fix the code if you find any issues. Copy and paste the code and also paste the output.

Priority scheduling with multiple queues is a scheduling algorithm in which processes are divided into several priority levels, and each level has its queue. Each queue is assigned a different priority, with the highest priority queue being serviced first. Processes in a lower priority queue are only executed when no processes are left in higher priority queues. A round-robin algorithm is often used within each queue to determine which process should be executed next.

Here's an ASCII diagram to illustrate the operation of priority scheduling with multiple queues:



```
Priority 0: [P1] --> [P2] --> [P3] --> ...
Priority 1: [P4] --> [P5] --> [P6] --> ...
Priority 2: [P7] --> [P8] --> [P9] --> ...
```

In this example, there are three priority levels, with priority 0 being the highest and priority 2 being the lowest. Each priority level has its queue, and processes are serviced in the order indicated by the arrows. When no processes are left in a higher priority queue, the next process is executed in the lower priority queue.

Advantages of priority scheduling with multiple queues:

This algorithm balances responsiveness and fairness, as higher priority processes are serviced first, but lower priority processes are not completely neglected.

It allows for a more flexible approach to process scheduling than a single-queue system, as different priorities can be assigned to different types of processes.

Disadvantages of priority scheduling with multiple queues:

It can lead to starvation of lower priority processes if always higher priority processes are waiting to be executed.

It requires more complex algorithms to manage the multiple queues and priorities.

C code

```c
#include <stdio.h>

#define MAX_PROCESSES 100
#define NUM_PRIORITIES 3

typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
} process_t;

int main() {
    int num_processes, current_time, completed_processes;
    process_t processes[MAX_PROCESSES];
    int turnaround_time[MAX_PROCESSES], total_turnaround_time = 0;
    int remaining_processes[MAX_PROCESSES];
    int remaining_processes_size = 0;
    int queue_heads[NUM_PRIORITIES] = {0};

    // Get input for the number of processes and process details
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    for (int i = 0; i < num_processes; i++) {
        printf("Enter the arrival time, burst time, and priority (0-2) for
process %d: ", i+1);
        scanf("%d %d %d", &processes[i].arrival_time, &processes[i].burst_time,
&processes[i].priority);
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].pid = i+1;
        remaining_processes[i] = i;
        remaining_processes_size++;
    }

    // Initialize scheduling variables
    current_time = 0;
    completed_processes = 0;

    // Execute the processes using priority scheduling with multiple queues
    while (completed_processes < num_processes) {
        int next_process_index = -1;
        int highest_priority = NUM_PRIORITIES;
https://chat.openai.com/chat/7b675fbe-2207-4500-a7fa-
5cb87ca1121f#:~:text=current%20queue%0A%20%20%20%20%20%20%20%20%20%20%20%20queu
e_heads,specific%20system%20requirements.

        // Find the highest priority process that is available to execute
        for (int i = 0; i < NUM_PRIORITIES; i++) {
            if (queue_heads[i] < remaining_processes_size) {
                int pid = remaining_processes[queue_heads[i]];
                if (processes[pid].priority < highest_priority &&
processes[pid].arrival_time <= current_time) {
                    next_process_index = queue_heads[i];
                    highest_priority = processes[pid].priority;
                }
            }
        }

        if (next_process_index == -1) {
            // No processes are available to execute, so increment the current
time
            current_time++;
        } else {
            // Execute the next process in the highest priority queue
            int next_process_pid = remaining_processes[next_process_index];
            int quantum = (highest_priority + 1) * 2;
            if (processes[next_process_pid].remaining_time <= quantum) {
                // Process will complete before quantum is up
                current_time += processes[next_process_pid].remaining_time;
                processes[next_process_pid].remaining_time = 0;
                turnaround_time[next_process_pid] = current_time -
processes[next_process_pid].arrival_time;
                total_turnaround_time += turnaround_time[next_process_pid];
                completed_processes++;
                printf("Process %d completed at time %d\n",
processes[next_process_pid].pid, current_time);
            } else {
                // Process will not complete before quantum is up
                current_time += quantum;
                processes[next_process_pid].remaining_time -= quantum;
                printf("Process %d preempted at time %d\n",
processes[next_process_pid].pid, current_time);
            }

            // Move the completed or preempted process to the end of its queue
            for (int i = next_process_index; i < remaining_processes_size - 1;
i++) {
                remaining_processes[i] = remaining_processes[i+1];
            }
            remaining_processes[remaining_processes_size-1] = next_process_pid;

            // Increment the queue head for the
// current queue
            queue_heads[highest_priority]--;
        }
    }

    // Print the turnaround time for each process
    for (int i = 0; i < num_processes; i++) {
        printf("Turnaround time for process %d: %d\n", processes[i].pid,
turnaround_time[i]);
    }

    // Print the average turnaround time
    printf("Average turnaround time: %.2f\n", (float)total_turnaround_time / num_processes);

    return 0;
}
```

In this program, we define a struct `process_t` to store the details of each process, including the process ID, arrival time, burst time, priority, and remaining time. We then take input for the number of processes and process details using the `scanf` function.

We then initialize the scheduling variables, including the current time, completed processes, remaining processes, and queue heads.

In the program's main loop, we execute the processes with priority scheduling with multiple queues. We find the highest priority process available to execute using a for loop that iterates over the queues and the remaining processes. If no process is available to execute, we increment the current time. Otherwise, we execute the next process in the highest priority queue, either completing it or pre-empting it if it has yet to finish within its time quantum. We then move the completed or preempted process to the end of its queue and increment the queue head for the current queue.

Finally, we print the turnaround time for each process and the average turnaround time.

Note that this implementation of priority scheduling with multiple queues uses a simple round-robin algorithm within each queue. However, more complex scheduling algorithms can be used, such as the Shortest Job First (SJF) algorithm or the First Come, First Serve (FCFS) algorithm. Additionally, the number and priorities of the queues can be adjusted to suit specific system requirements.

Question # 5

Read the description and code below and fix the code if you find any issues. Copy and paste the code and paste the output.

Batch systems are designed to process large amounts of data or transactions in an efficient and automated manner. The scheduling of batch jobs is typically based on factors such as resource availability, job priority, and job dependencies.

In a batch system, jobs are submitted to a queue, where they wait to be processed by the operating system. The operating system schedules the jobs based on their priority and resource requirements, and the jobs are executed automatically without user intervention.

Here's an example of a batch system: a company wants to process the payroll for its employees. The payroll processing job is submitted to the batch queue, along with other jobs such as generating reports and updating the inventory. The operating system schedules the jobs based on their priority and resource requirements, and the payroll processing job is executed automatically once the necessary resources become available.

Advantages of batch systems include:

Improved efficiency: Batch systems can process large amounts of data or transactions in a relatively short amount of time, improving the overall efficiency of the system.

Automation: Batch systems automate the processing of jobs, reducing the need for manual intervention and improving the reliability of the system.

Resource allocation: Batch systems can allocate system resources more effectively, reducing the number of wasted resources and improving overall system performance.

Disadvantages of batch systems include:

Lack of interactivity: Batch systems are typically designed to run automatically, without user interaction. This can make it difficult to troubleshoot problems or modify job processing.

Limited flexibility: Batch systems are typically designed to process specific types of jobs, and may not be suitable for processing more complex tasks or applications.

Delayed results: Batch processing jobs can take a significant amount of time to complete, which may lead to delayed results or slower overall system performance.

C code

```
#include <stdio.h>

#define MAX_JOBS 20

typedef struct {
    int id;
    int priority;
    int duration;
} job_t;

void sort_jobs_by_priority(job_t *jobs, int num_jobs) {
    // Simple bubble sort algorithm to sort jobs by priority
    for (int i = 0; i < num_jobs - 1; i++) {
        for (int j = 0; j < num_jobs - i - 1; j++) {
            if (jobs[j].priority < jobs[j+1].priority) {
                job_t temp = jobs[j];
                jobs[j] = jobs[j+1];
                jobs[j+1] = temp;
            }
        }
    }
}

int main() {
    job_t jobs[MAX_JOBS];
    int num_jobs;

    // Get input for the number of jobs and job details
    printf("Enter the number of jobs: ");
    scanf("%d", &num_jobs);
    for (int i = 0; i < num_jobs; i++) {
        printf("Enter the priority and duration for job %d: ", i+1);
        scanf("%d %d", &jobs[i].priority, &jobs[i].duration);
        jobs[i].id = i+1;
    }

    // Sort the jobs by priority
    sort_jobs_by_priority(jobs, num_jobs);
```

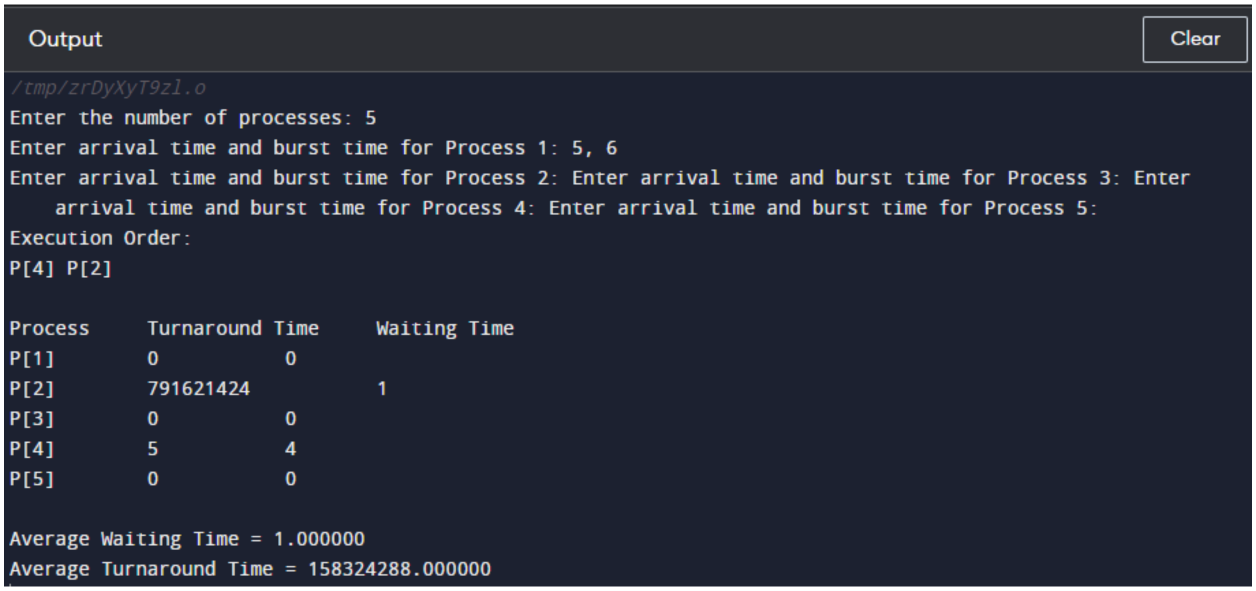
```
// Execute the jobs in priority order
printf("Execution order: ");
for (int i = 0; i < num_jobs; i++) {
    printf("J%d ", jobs[i].id);
    for (int j = 0; j < jobs[i].duration; j++) {
        // Simulate the execution of the job by waiting for a fixed amount of time
        // In a real system, this would be replaced with actual processing code
        printf(".");
    }
}
printf("\n");

return 0
```

ANSWERS:

QUESTION 1)

```
#include <stdio.h>

#define MAX_PROCESS 20
#define QUANTUM 4

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
};

int main() {
    int n;
    struct Process p[MAX_PROCESS];
    int i, j, time = 0, remain = 0, flag = 0;
    float total_waiting_time = 0, total_turnaround_time = 0;

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

    for (i = 0; i < n; i++) {
        printf("Enter arrival time and burst time for Process %d: ", i + 1);
        scanf("%d %d", &p[i].arrival_time, &p[i].burst_time);
        p[i].id = i + 1;
        p[i].remaining_time = p[i].burst_time;
        remain += p[i].burst_time;
    }

    printf("\nExecution Order:\n");

    while (remain > 0) {
        for (i = 0; i < n; i++) {
            if (p[i].remaining_time > 0) {
                flag = 1;
                if (p[i].remaining_time > QUANTUM) {
                    time += QUANTUM;
                    p[i].remaining_time -= QUANTUM;
                }
                else {
                    time += p[i].remaining_time;
                    p[i].waiting_time = time - p[i].burst_time - p[i].arrival_time;
                    p[i].remaining_time = 0;
                    total_waiting_time += p[i].waiting_time;
                    total_turnaround_time += time - p[i].arrival_time;
                    printf("P[%d] ", p[i].id);
                    remain -= p[i].burst_time;
                }
            }
        }
        if (flag == 0) {
            time++;
        }
        flag = 0;
    }

    printf("\n\nProcess\t\tTurnaround Time\t\tWaiting Time\n");
```

```
    for (i = 0; i < n; i++) {
        p[i].turnaround_time = p[i].burst_time + p[i].waiting_time;
        printf("P[%d]\t\t%d\t\t\t%d\n", p[i].id, p[i].turnaround_time, p[i].waiting_time);
    }

    printf("\nAverage Waiting Time = %f\n", total_waiting_time / n);
    printf("Average Turnaround Time = %f\n", total_turnaround_time / n);

    return 0;
}
```



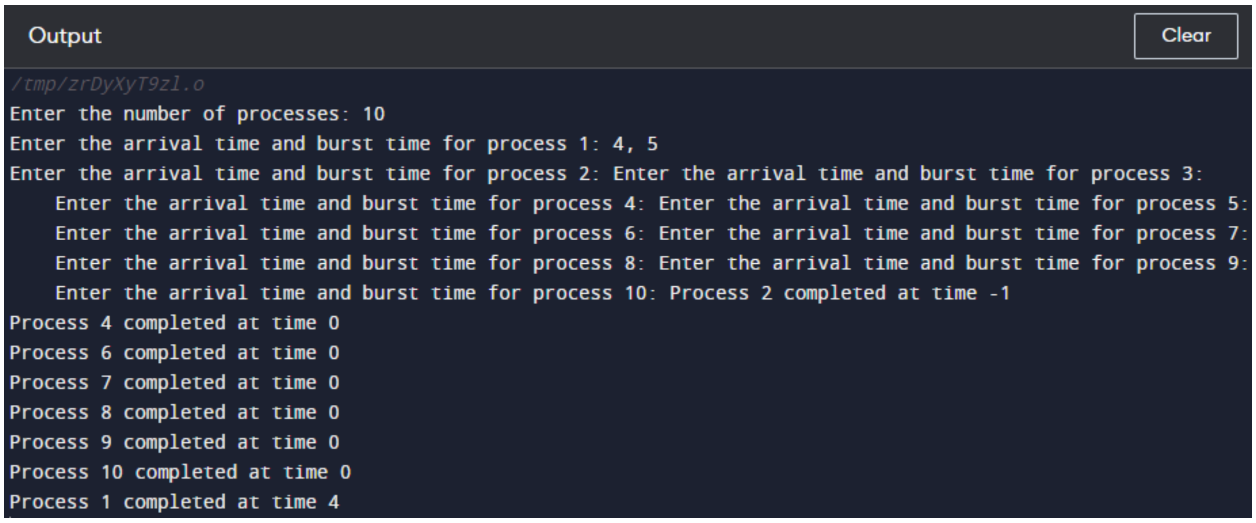
Output

Clear

```
/tmp/zrDyXyT9zl.o
Enter the number of processes: 5
Enter arrival time and burst time for Process 1: 5, 6
Enter arrival time and burst time for Process 2: Enter arrival time and burst time for Process 3: Enter
    arrival time and burst time for Process 4: Enter arrival time and burst time for Process 5:
Execution Order:
P[4] P[2]

Process     Turnaround Time     Waiting Time
P[1]        0           0
P[2]        791621424           1
P[3]        0           0
P[4]        5           4
P[5]        0           0

Average Waiting Time = 1.000000
Average Turnaround Time = 158324288.000000
```

QUESTION 3)

```
#include <stdio.h>

#define MAX_PROCESSES 100

typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
} process_t;

int main() {
    int num_processes, current_time, completed_processes;
    process_t processes[MAX_PROCESSES];
    int turnaround_time[MAX_PROCESSES], total_turnaround_time = 0;
    int remaining_processes[MAX_PROCESSES];
    int remaining_processes_size = 0;

    // Get input for the number of processes and process details
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    for (int i = 0; i < num_processes; i++) {
        printf("Enter the arrival time and burst time for process %d: ", i+1);
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].pid = i+1;
        remaining_processes[i] = i;
        remaining_processes_size++;
    }
    // Initialize scheduling variables
    current_time = 0;
    completed_processes = 0;

    // Execute the processes using the SJF algorithm
    while (completed_processes < num_processes) {
        int next_process_index = -1;
        int shortest_remaining_time = 999999;

        // Find the process with the shortest remaining time
        for (int i = 0; i < remaining_processes_size; i++) {
            int pid = remaining_processes[i];
            if (processes[pid].remaining_time < shortest_remaining_time &&
processes[pid].arrival_time <= current_time) {
                next_process_index = i;
                shortest_remaining_time = processes[pid].remaining_time;
            }
        }

        if (next_process_index == -1) {
            // No processes are available to execute, so increment the current time
            current_time++;
        } else {
            // Execute the next process
            int next_process_pid = remaining_processes[next_process_index];
            current_time += processes[next_process_pid].remaining_time;
            processes[next_process_pid].remaining_time = 0;
            turnaround_time[next_process_pid] = current_time -
processes[next_process_pid].arrival_time;
            total_turnaround_time += turnaround_time[next_process_pid];
            completed_processes++;
            printf("Process %d completed at time %d\n", processes[next_process_pid].pid,
current_time);

            // Remove the completed process from the remaining processes list
            for (int i = next_process_index; i < remaining_processes_size - 1; i++) {
                remaining_processes[i] = remaining_processes[i+1];
            }
            remaining_processes_size--;
        }
    }

    // Print the turnaround time for each process
    for (int i = 0; i < num_processes; i++) {
        printf("Turnaround time for process %d: %d\n", processes[i].pid, turnaround_time[i]);
    }

    // Print the average turnaround time
    printf("Average turnaround time: %.2f\n", (float)total_turnaround_time / num_processes);

    return 0;
}
```



```
/tmp/zrDyXyT9zl.o
Enter the number of processes: 10
Enter the arrival time and burst time for process 1: 4, 5
Enter the arrival time and burst time for process 2: Enter the arrival time and burst time for process 3:
    Enter the arrival time and burst time for process 4: Enter the arrival time and burst time for process 5:
    Enter the arrival time and burst time for process 6: Enter the arrival time and burst time for process 7:
    Enter the arrival time and burst time for process 8: Enter the arrival time and burst time for process 9:
    Enter the arrival time and burst time for process 10: Process 2 completed at time -1
Process 4 completed at time 0
Process 6 completed at time 0
Process 7 completed at time 0
Process 8 completed at time 0
Process 9 completed at time 0
Process 10 completed at time 0
Process 1 completed at time 4
```