EE6253 Operating Systems and Network Programming

Take-Home Software Assignment

Name: Lelwala L.G.S.R

Reg. No: EG/2020/4047

Tasks:

1. **Implement the FCFS scheduling algorithm.**

    1. Processes are executed in the order they arrive.
    2. When a new process arrives, it is added to the end of the ready queue.
    3. The CPU is allocated to the process at the front of the ready queue.
    4. The process executes until it finishes or is interrupted by a higher-priority process (in non-preemptive scheduling).
    5. Once the current process completes execution, it is removed from the ready queue.
    6. The CPU is then allocated to the next process in the ready queue.
    7. The cycle repeats, with the CPU executing processes in the order they arrive in the ready queue.
    8. Processes are not prioritized based on their burst time (CPU time required) or any other factor.
    9. Calculates the completion times, turn-around times, waiting times and average waiting time and print them.

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

// Define a structure to store process information
typedef struct fcfs {
    int process; // Process Number
    int burst;   // Burst Time
    int arrival; // Arrival Time
    int tat;     // Turn Around Time
    int wt;      // Waiting Time
} fcfs;

// Function prototype for sorting processes based on arrival time
int sort(fcfs[], int);

int main() {
    int n, i, temp = 0;
    float AvWt = 0;

    n = 4;       // No. of Processes
    fcfs arr[n];   // Array of type fcfs
    int tct[n];    // Array to store completion time of each process

    // Initialize process data
    int process_data[][3] = {
        // Process,  Arrival Time, Brust Time
        {1, 0, 10},
        {2, 6, 8},
        {3, 7, 4},
        {4, 9, 5}
    };

    // Assign process data to fcfs array
```

```
    for (i = 0; i < n; i++) {
        arr[i].process = process_data[i][0];
        arr[i].arrival = process_data[i][1];
        arr[i].burst = process_data[i][2];
    }

    // Sorting the processes according to their arrival time
    sort(arr, n);

    // Print the header
    printf("Process\t\tArrival Time\tBurst Time\tTurn Around Time\tWaiting Time\n");

    // Calculate and print process details
    for (i = 0; i < n; i++) {
        // Calculate the completion time for the current process
        tct[i] = temp + arr[i].burst;

        // Update the temporary completion time
        temp = tct[i];

        // Calculate the turn around time and waiting time
        arr[i].tat = tct[i] - arr[i].arrival;
        arr[i].wt = arr[i].tat - arr[i].burst;

        // Update the average waiting time
        AvWt = AvWt + arr[i].wt;

        // Print the process details
        printf("%5d\t%15d\t\t%9d\t%12d\t%12d\n", arr[i].process, arr[i].arrival, arr[i].burst, arr[i].tat,
arr[i].wt);
    }

    // Print the average waiting time
    printf("Average Waiting Time: %.2f\n", AvWt / n);

    return 0;
}

// Bubble Sort function to sort processes based on arrival time
int sort(fcfs arr[], int n) {
    int i, j;
    fcfs k;

    // Outer loop for the number of passes
    for (i = 0; i < n - 1; i++) {
        // Inner loop for comparisons and swapping
        for (j = i + 1; j < n; j++) {
            // Sorting the processes according to their arrival time
            if (arr[i].arrival > arr[j].arrival) {
                // Swap the processes
                k = arr[i];
                arr[i] = arr[j];
                arr[j] = k;
            }
        }
    }

    return 0;
}
```

2. Implement the SJF scheduling algorithm.

1. Processes are sorted based on their arrival time.
2. At any given time, the process with the shortest burst time (CPU time required) among the arrived processes is selected for execution.
3. The selected process is executed until completion (non-preemptive).
4. After completing a process, the algorithm checks for any new arrivals.
5. If new processes have arrived, it selects the next shortest process from the arrived processes.
6. If no new processes have arrived, it selects the next shortest process from the remaining processes.
7. The selected process is executed until completion.
8. The cycle repeats until all processes are executed.
9. Calculates the completion times, turn-around times, waiting times and average waiting time and print them.

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

// Define a structure to store process information
typedef struct sjf {
    int process;     // Process ID
    int burst;       // Burst time (CPU time required)
    int arrival;     // Arrival time
    int tat;         // Turnaround time
    int wt;          // Waiting time (Turn around time - Burst time)
} sjf;

// Function prototype for sorting processes based on arrival time
void sort(sjf[], int);

int main() {
    int n, i, j, TCT, count_process = 0, count = 0, minBurst, pos;
    float AvWT = 0;

    n = 4;  // No. of Processes

    sjf arr[n]; // Array of type sjf

    // Initialize process data
    int process_data[][3] = {
        // Process,  Arrival Time, Brust Time
        {1, 0, 10},
        {2, 6, 8},
        {3, 7, 4},
        {4, 9, 5}
    };

    // Assign process data to sjf array
    for (i = 0; i < n; i++) {
        arr[i].process = process_data[i][0];
        arr[i].arrival = process_data[i][1];
        arr[i].burst = process_data[i][2];
    }

    // Sort processes based on arrival time
    sort(arr, n);

    // Initialize variables for the first process
```

```
    TCT = arr[0].tat = arr[0].burst;
    arr[0].wt = arr[0].tat - arr[0].burst;
    arr[0].arrival = -1; // Mark the first process as completed

    // Sort processes again based on arrival time
    sort(arr, n);
    count_process = 1;

    // Main loop for executing processes
    while (count_process < n) {
        minBurst = 999; // Initialize with a large value
        count = 0;
        i = count_process;

        // Count the number of processes arrived by the current time
        while (TCT >= arr[i].arrival && i < n) {
            count++;
            i++;
        }

        // Find the process with the minimum burst time among the arrived processes
        for (j = i - count; count != 0 && j < n; j++, count--) {
            if (arr[j].burst < minBurst) {
                minBurst = arr[j].burst;
                pos = j;
            }
        }

        // Execute the process with the minimum burst time
        TCT = TCT + arr[pos].burst;
        arr[pos].tat = TCT - arr[pos].arrival;
        arr[pos].wt = arr[pos].tat - arr[pos].burst;
        arr[pos].arrival = -1; // Mark the process as completed

        // Sort processes again based on arrival time
        sort(arr, n);
        count_process++;
    }

    // Print the processes with their details
    printf("Process\t\tBurst Time\tTurn Around Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        AvWT = AvWT + arr[i].wt;
        printf("%5d\t%15d\t\t%9d\t%12d\n", arr[i].process, arr[i].burst, arr[i].tat, arr[i].wt);
    }

    printf("Average Waiting Time: %.2f\n", AvWT / n);
    return 0;
}

// Function to sort processes based on arrival time
void sort(sjf arr[], int n) {
    int i, j;
    sjf temp;

    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++) {
            // Sorting the processes according to their arrival time
            if (arr[i].arrival > arr[j].arrival) {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
}
```

3. Implement the RR scheduling algorithm.

1. Processes are taken as input with their arrival time and burst time (CPU time required).
2. A time quantum (fixed time slice) is set for the algorithm.
3. Processes are added to a ready queue based on their arrival time.
4. The process at the front of the ready queue is selected for execution.
5. The selected process is executed for the time quantum or its remaining burst time, whichever is smaller.
6. If the process completes execution within the time quantum, it is removed from the ready queue.
7. If the process does not complete within the time quantum, it is added back to the end of the ready queue with its remaining burst time.
8. The next process in the ready queue is selected for execution, and the cycle continues.
9. If no process is available for execution, the algorithm waits for the next process to arrive.
10. The algorithm continues until all processes are executed.
11. For each process, it calculates the completion time, turnaround time, and waiting time
12. Finally, it calculates and prints the average waiting time for all processes.

```c
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

// Structure to store process details
typedef struct RR {
    int process;     // Process ID
    int arrival;     // Arrival time
    int burst;       // Burst time
    int start[20];   // Array to store the start times
    int wt;          // Waiting time
    int complete;    // Completion time
    int tat;         // Turnaround time
} RR;

int quantum; // Quantum for Round Robin scheduling

int main() {
    int n, i, j;

    // Taking Input
    n = 4; // No. of Processes
    RR p[n];
    quantum = 2; // Time Quantum

    // Initialize process data
    int process_data[][3] = {
        // Process,  Arrival Time, Burst Time
        {1, 0, 10},
        {2, 6, 8},
        {3, 7, 4},
        {4, 9, 5}
    };

    // Assign process data to arr array
```

```
    for (i = 0; i < n; i++) {
        p[i].process = process_data[i][0];
        p[i].arrival = process_data[i][1];
        p[i].burst = process_data[i][2];
        p[i].complete = 0; // Completion time is initially set to 0
    }

    // Declaring variables
    int c = n, s[n][20];
    float time = 0, mini = INT_MAX, b[n], a[n];

    // Initializing burst and arrival time arrays
    int index = -1;
    for (i = 0; i < n; i++) {
        b[i] = p[i].burst;
        a[i] = p[i].arrival;
        for (j = 0; j < 20; j++) {
            s[i][j] = -1; // Initialize start time array with -1
        }
    }

    int tot_wt, tot_tat;
    tot_wt = 0;
    tot_tat = 0;
    bool flag = false;

    // Round Robin Scheduling Algorithm
    while (c != 0) {
        mini = INT_MAX;
        flag = false;

        // Find the process with minimum arrival time
        for (i = 0; i < n; i++) {
            float p = time + 0.1;
            if (a[i] <= p && mini > a[i] && b[i] > 0) {
                index = i;
                mini = a[i];
                flag = true;
            }
        }

        // If no process is available, increment time
        if (!flag) {
            time++;
            continue;
        }

        // Calculate start time
        j = 0;
        while (s[index][j] != -1) {
            j++;
        }

        if (s[index][j] == -1) {
            s[index][j] = time;
            p[index].start[j] = time;
        }

        // Execute the process for quantum time or remaining burst time
        if (b[index] <= quantum) {
            time += b[index];
            b[index] = 0;
        } else {
            time += quantum;
            b[index] -= quantum;
        }

        // If process is not completed, update arrival time
        if (b[index] > 0) {
            a[index] = time + 0.1;
        }

        // If process is completed, update completion time, waiting time, and turnaround time
        if (b[index] == 0) {
            c--;
```

```
            p[index].complete = time;
            p[index].wt = p[index].complete - p[index].arrival - p[index].burst;
            tot_wt += p[index].wt;
            p[index].tat = p[index].burst + p[index].wt;
            tot_tat += p[index].tat;
        }
    } // end of while loop

    // Printing output
    printf("Process\t\tArrival Time\tBurst Time\tCompletion time\t\tTurn Around Time\tWaiting Time\n");

    // Print process details
    for (i = 0; i < n; i++) {
        printf("%5d\t%12d\t\t%8d\t\t%d\t\t\t%12d\t%12d\n", p[i].process, p[i].arrival, p[i].burst,
p[i].complete, p[i].tat, p[i].wt);
    }

    // Calculating average wait time
    double avg_wt, avg_tat;
    avg_wt = tot_wt / (float)n;

    // Printing average wait time and turnaround time
    printf("Average Waiting Time: %.2f\n", avg_wt);

    return 0;
}
```

4. **Simulate the execution of the provided processes using each scheduling algorithm.**

• FCFS Algorithm Results

```
[Running] cd "c:\Users\sahan\Desktop\MYProjects\CPU-SchedulingAlgorithms-Imple
Process     Arrival Time    Burst Time  Turn Around Time  Waiting Time
    1               0            10             10              0
    2               6             8             12              4
    3               7             4             15             11
    4               9             5             18             13
Average Waiting Time: 7.00

[Done] exited with code=0 in 1.468 seconds
```

• SJF Algorithm Results

```
[Running] cd "c:\Users\sahan\Desktop\MYProjects\CPU-SchedulingAlgorith
Process     Burst Time  Turn Around Time  Waiting Time
    1              10             10              0
    3               4              7              3
    4               5             10              5
    2               8             21             13
Average Waiting Time: 5.25

[Done] exited with code=0 in 1.09 seconds
```

• RR Algorithm Results (Quantum = 2)

```
[Running] cd "c:\Users\sahan\Desktop\MYProjects\CPU-SchedulingAlgorithms-Implementation\" && gcc Roun
"c:\Users\sahan\Desktop\MYProjects\CPU-SchedulingAlgorithms-Implementation\"Round-Robin
Process     Arrival Time    Burst Time  Completion time     Turn Around Time    Waiting Time
   1             0             10       18                  18                   8
   2             6              8       26                  20                  12
   3             7              4       20                  13                   9
   4             9              5       27                  18                  13
Average Waiting Time: 10.50

[Done] exited with code=0 in 2.17 seconds
```

5. **Compute and compare the average waiting time for each algorithm.**

   After running the above codes, we should get the average waiting times as follows,

FCFS Average Waiting Time	7.00
SJF Average Waiting Time	5.25
RR (Quantum = 2) Average Waiting Time	10.50

- FCFS

  The FCFS algorithm has an average waiting time of 7.00, which is higher than SJF but lower than RR. In FCFS, processes are executed in the order they arrive, without considering their burst times. This can lead to longer waiting times for shorter processes that arrive after longer processes.

- SJF

  The SJF algorithm has the lowest average waiting time of 5.25 among the three algorithms. This is because SJF prioritizes the execution of shorter processes first, minimizing the waiting time for most processes. However, it can lead to starvation for longer processes if a steady stream of shorter processes keeps arriving.

- RR

  The RR algorithm has the highest average waiting time of 10.50 among the three algorithms, given a time quantum of 2. RR allocates CPU time in a circular order, switching between processes after a fixed time quantum. While it provides fairness and prevents starvation, it can lead to higher waiting times due to frequent context switches and the possibility of processes being preempted before completion.

  In summary, the average waiting times follow this order,

  $$SJF < FCFS < RR$$

  The SJF algorithm provides the lowest average waiting time by prioritizing shorter processes, while the RR algorithm with a time quantum of 2 has the highest average waiting time due to frequent context switches and process preemptions.

6. **Write a brief analysis comparing the performance of the three scheduling algorithms based on the average waiting time.**

   From the results, SJF has the lowest average waiting time, followed by FCFS, and RR has the highest average waiting time. This is expected as SJF minimizes the waiting time by executing the shortest jobs first, while RR provides fair allocation but can have higher waiting times due to context switching.

   Choosing the right scheduling method is important. SJF gives the lowest wait times but can skip over big jobs. FCFS is straightforward but less ideal with different job sizes. RR treats all jobs fairly but has extra work from switching often. Knowing the pros and cons helps pick the best method for each situation.