CPU Scheduling Simulator Design Document

Nathan Posniak

Overview

This project simulates CPU process scheduling algorithms using the C programming language. It demonstrates how an operating system decides which process to run next based on different scheduling strategies.

The simulator supports the following algorithms:

- First-Come, First-Served
- Shortest Job First
- Shortest Remaining Time First
- Round Robin

For each algorithm the simulator displays:

- A process table with timing data for each process
- Average turnaround time, average waiting time, and average response time for the dataset
- A Gantt chart showing execution order

Objectives

The objective for this project was to;

- Implement multiple different CPU scheduling algorithms in the c language
- Display the timing outcome of these algorithms in both a table and Gantt chart
- Measure and compare their average turnaround times, waiting times, and response times
- Use different datasets that play into and against the algorithm's inherent advantages and record how they perform

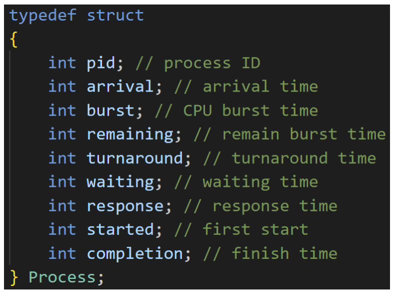
Code Structure

The program is divided among three C language files

- scheduler.h
  - The header file for the scheduler program, defines the constants and functions for the main file
- scheduler.c
  - Contains the implementations for the scheduling algorithms and results displaying algorithms
- experimental.c
  - Defines and creates the datasets and test cases, and runs the experimental schedules with the algorithms created in scheduler.c

Implementation

‘Process’ Data structure

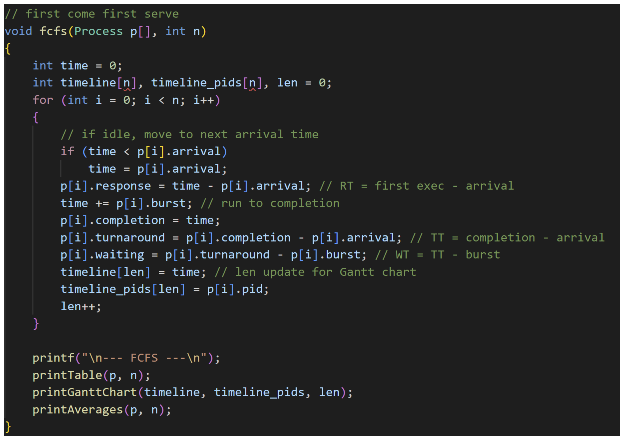


```c
typedef struct
{
    int pid; // process ID
    int arrival; // arrival time
    int burst; // CPU burst time
    int remaining; // remain burst time
    int turnaround; // turnaround time
    int waiting; // waiting time
    int response; // response time
    int started; // first start
    int completion; // finish time
} Process;
```

The Process data structure, used to keep track of individual processes within an algorithm, defined within the header file.

Scheduling Algorithms

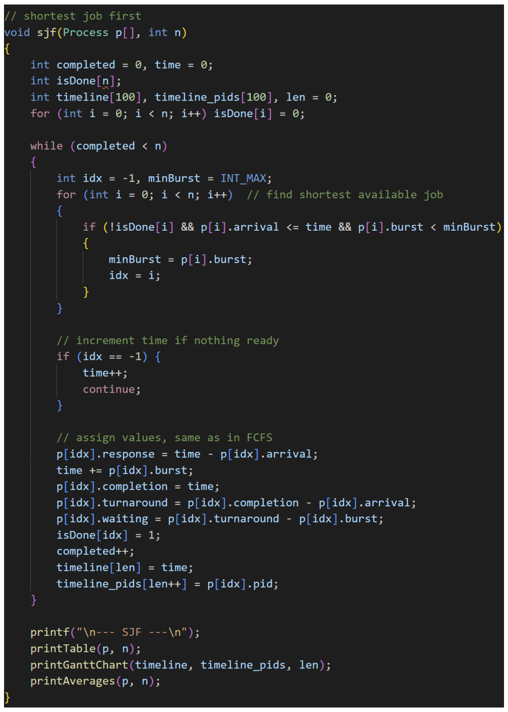
FCFS



```c
// first come first serve
void fcfs(Process p[], int n)
{
    int time = 0;
    int timeline[n], timeline_pids[n], len = 0;
    for (int i = 0; i < n; i++)
    {
        // if idle, move to next arrival time
        if (time < p[i].arrival)
            time = p[i].arrival;
        p[i].response = time - p[i].arrival; // RT = first exec - arrival
        time += p[i].burst; // run to completion
        p[i].completion = time;
        p[i].turnaround = p[i].completion - p[i].arrival; // TT = completion - arrival
        p[i].waiting = p[i].turnaround - p[i].burst; // WT = TT - burst
        timeline[len] = time; // len update for Gantt chart
        timeline_pids[len] = p[i].pid;
        len++;
    }

    printf("\n--- FCFS ---\n");
    printTable(p, n);
    printGanttChart(timeline, timeline_pids, len);
    printAverages(p, n);
}
```

In FCFS, processes are sorted by order in which they arrive in the ready queue, and once a process begins running, it continues until completion without interruption. FCFS is quite efficient when jobs arrive in increasing order of length, but can run into complications if a larger job arrives before smaller jobs, leading to them being caught behind.

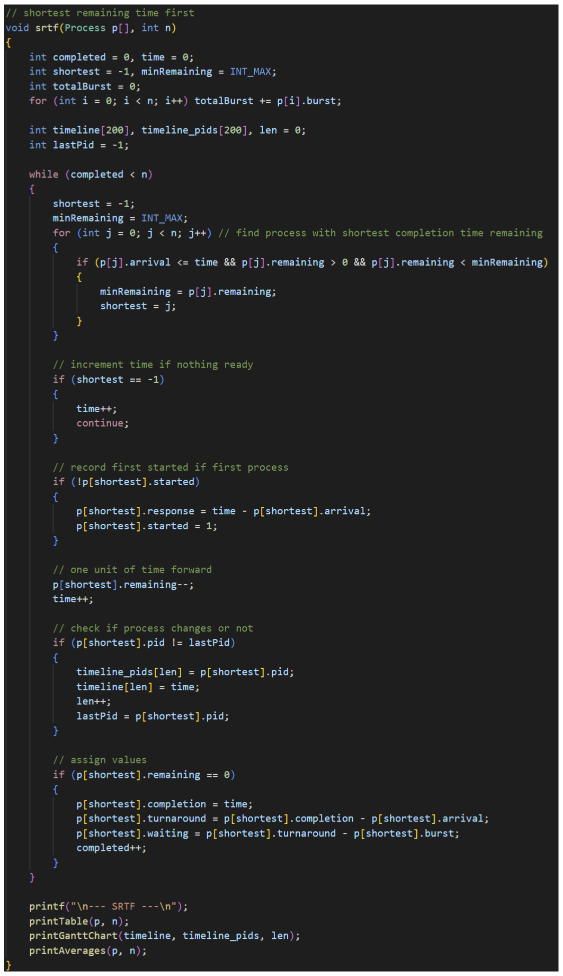
SJF



```c
// shortest job first
void sjf(Process p[], int n)
{
    int completed = 0, time = 0;
    int isDone[n];
    int timeline[100], timeline_pids[100], len = 0;
    for (int i = 0; i < n; i++) isDone[i] = 0;

    while (completed < n)
    {
        int idx = -1, minBurst = INT_MAX;
        for (int i = 0; i < n; i++)  // find shortest available job
        {
            if (!isDone[i] && p[i].arrival <= time && p[i].burst < minBurst)
            {
                minBurst = p[i].burst;
                idx = i;
            }
        }

        // increment time if nothing ready
        if (idx == -1) {
            time++;
            continue;
        }

        // assign values, same as in FCFS
        p[idx].response = time - p[idx].arrival;
        time += p[idx].burst;
        p[idx].completion = time;
        p[idx].turnaround = p[idx].completion - p[idx].arrival;
        p[idx].waiting = p[idx].turnaround - p[idx].burst;
        isDone[idx] = 1;
        completed++;
        timeline[len] = time;
        timeline_pids[len++] = p[idx].pid;
    }

    printf("\n--- SJF ---\n");
    printTable(p, n);
    printGanttChart(timeline, timeline_pids, len);
    printAverages(p, n);
}
```

In SJF, whenever a new process is to be started, available jobs are queued based on their burst time, with shortest jobs having the highest priority. The jobs then run without interruption to completion and a new job is chosen from the queue, updated to include any other new short jobs. This can be very efficient for some datasets, but sometimes longer jobs can be held indefinitely as newer short jobs keep arriving.

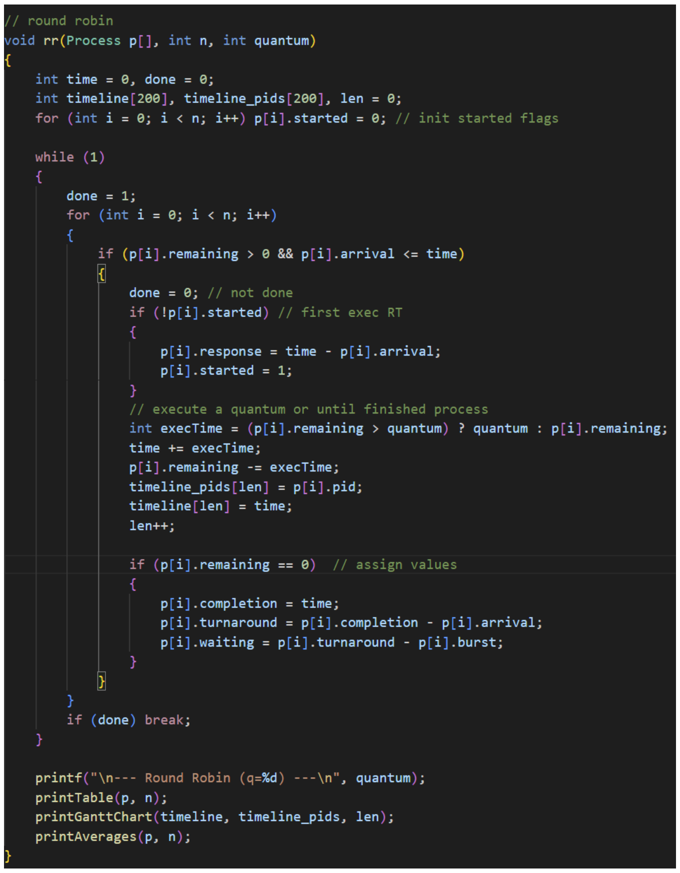
SRTF



```c
// shortest remaining time first
void srtf(Process p[], int n)
{
    int completed = 0, time = 0;
    int shortest = -1, minRemaining = INT_MAX;
    int totalBurst = 0;
    for (int i = 0; i < n; i++) totalBurst += p[i].burst;

    int timeline[200], timeline_pids[200], len = 0;
    int lastPid = -1;

    while (completed < n)
    {
        shortest = -1;
        minRemaining = INT_MAX;
        for (int j = 0; j < n; j++) // find process with shortest completion time remaining
        {
            if (p[j].arrival <= time && p[j].remaining > 0 && p[j].remaining < minRemaining)
            {
                minRemaining = p[j].remaining;
                shortest = j;
            }
        }

        // increment time if nothing ready
        if (shortest == -1)
        {
            time++;
            continue;
        }

        // record first started if first process
        if (!p[shortest].started)
        {
            p[shortest].response = time - p[shortest].arrival;
            p[shortest].started = 1;
        }

        // one unit of time forward
        p[shortest].remaining--;
        time++;

        // check if process changes or not
        if (p[shortest].pid != lastPid)
        {
            timeline_pids[len] = p[shortest].pid;
            timeline[len] = time;
            len++;
            lastPid = p[shortest].pid;
        }

        // assign values
        if (p[shortest].remaining == 0)
        {
            p[shortest].completion = time;
            p[shortest].turnaround = p[shortest].completion - p[shortest].arrival;
            p[shortest].waiting = p[shortest].turnaround - p[shortest].burst;
            completed++;
        }
    }

    printf("\n--- SRTF ---\n");
    printTable(p, n);
    printGanttChart(timeline, timeline_pids, len);
    printAverages(p, n);
}
```

SRTF is the preemptive version of SJF, meaning the algorithm continuously checks the ready queue, and if a new process arrives with a shorter remaining time than the currently running one, it has the CPU switch to the shorter job. This makes SRTF the algorithm with the lowest possible average turnaround time, but at the cost of lots of switching and more complexity. Like SJF, it may also starve longer tasks.

RR



```c
// round robin
void rr(Process p[], int n, int quantum)
{
    int time = 0, done = 0;
    int timeline[200], timeline_pids[200], len = 0;
    for (int i = 0; i < n; i++) p[i].started = 0; // init started flags

    while (1)
    {
        done = 1;
        for (int i = 0; i < n; i++)
        {
            if (p[i].remaining > 0 && p[i].arrival <= time)
            {
                done = 0; // not done
                if (!p[i].started) // first exec RT
                {
                    p[i].response = time - p[i].arrival;
                    p[i].started = 1;
                }
                // execute a quantum or until finished process
                int execTime = (p[i].remaining > quantum) ? quantum : p[i].remaining;
                time += execTime;
                p[i].remaining -= execTime;
                timeline_pids[len] = p[i].pid;
                timeline[len] = time;
                len++;

                if (p[i].remaining == 0)  // assign values
                {
                    p[i].completion = time;
                    p[i].turnaround = p[i].completion - p[i].arrival;
                    p[i].waiting = p[i].turnaround - p[i].burst;
                }
            }
        }
        if (done) break;
    }

    printf("\n--- Round Robin (q=%d) ---\n", quantum);
    printTable(p, n);
    printGanttChart(timeline, timeline_pids, len);
    printAverages(p, n);
}
```

In round robin, each process receives a fixed time slot and executes for that duration and is then placed at the end of the queue, or finishes. This ensures no longer processes monopolize the CPU, and guarantees shorter processes don't starve longer ones. However, if the time quantum is too small, excessive switching reduces efficiency, and can starve longer processes, and if it's too large, RR will behave closer to the simple but less efficient FCFS.

Conclusions

This project effectively demonstrates how scheduling decisions impact process performance metrics. By comparing algorithms on diverse datasets, you can visualize trade-offs between efficiency, fairness, and responsiveness, gaining a deeper understanding of CPU scheduling mechanisms in operating systems.