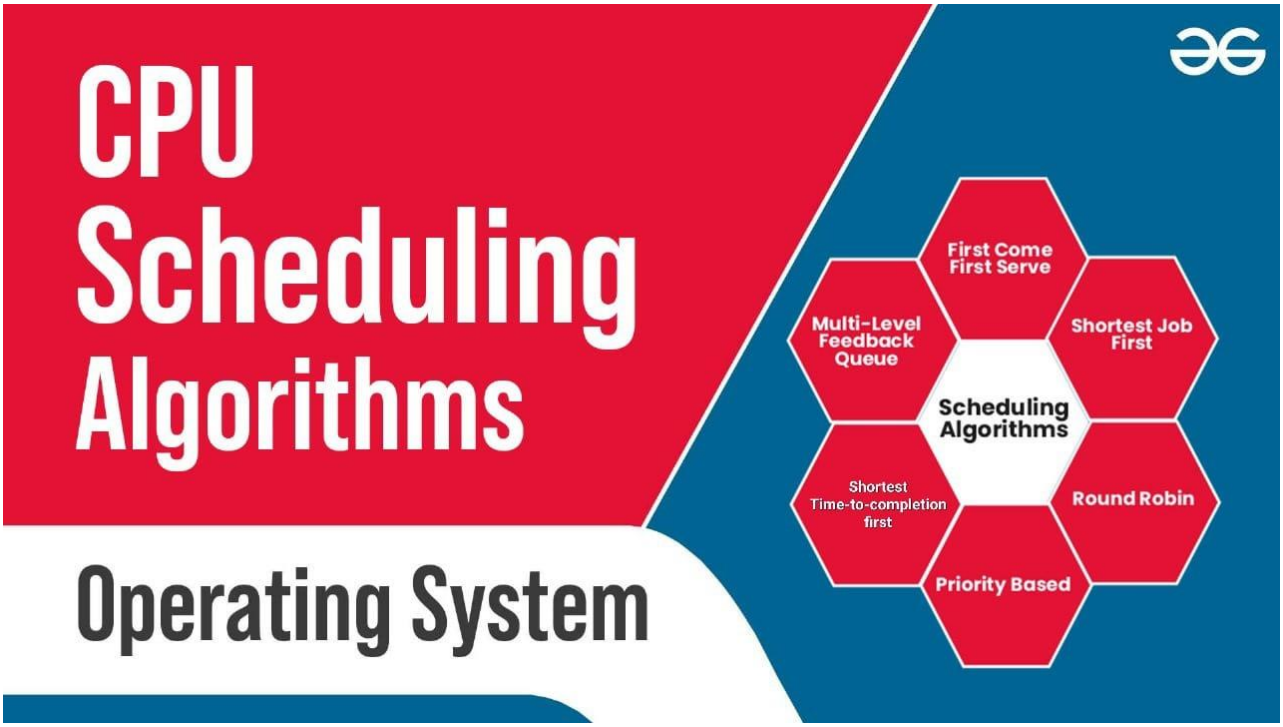

Submitted by:

"Ahmed Mahmoud Mohamad "

"Esraa Magdy Mosaad "

"Amna Ali Abd El-Wahab"

"Mohamad Khaled Abd El-Qader "

"Abdallah Hafez Saad "

3rd year

Computer Engineering

Faculty of Engineering at Shoubra

Benha University

Program description:

This process scheduling simulator program is a C-based implementation of different process scheduling algorithms that allows users to simulate the execution of processes in a system with multiple cores. The program reads a text file containing the input processes and their properties, as well as the number of cores available in the system. The program then prompts the user to select one of the available scheduling policies to use for simulating the execution of the processes. The available scheduling policies include:

1. First-come, first-served (FCFS)
2. Shortest Job First (SJF)
3. Round Robin (RR)
4. Priority scheduling
5. Multi-level Feedback Queue (MLFQ) scheduling
6. Stride scheduling
7. First time to completion first (FTCF)

For each scheduling policy, the program simulates the execution of the input processes on the available cores, and displays the average turnaround time and a Gantt chart showing the execution time of each process in each core. The program also provides options for the user to customize some policy parameters, such as time quantum for Round Robin policy and proportion of system time allocated to each process for Stride scheduling policy.

The program is designed to be modular, flexible, and user-friendly, with extensive error handling and informative user prompts. It enables users to experiment with different scheduling policies and parameters to evaluate their performance and optimize process scheduling in multi-core systems.

Design choices:

This process scheduling simulator program has several design choices that were made to enable accurate simulation and flexibility. Some of these design choices include:

1. Queue data structure: The program uses a queue data structure to store the input processes in their order of arrival.

2. Process struct: The program represents a process using a struct that contains its process ID, arrival time, burst time, priority, and I/O time.

3. Modular design: The program is modular, with each scheduling policy implemented as a separate function. This design choice makes it easy to add or modify policies without affecting the rest of the program.

4. Gantt chart: The program displays a Gantt chart showing the execution time of each process in each core. This visual representation of the execution order provides a detailed insight into the performance of each scheduling policy.

5. File input: The program reads the processes and the number of cores from external text files, allowing the user to input or modify these parameters without needing to change the code.

6. User input: The program prompts the user to select the desired scheduling policy and to input any required parameters, providing flexibility and allowing the user to experiment with different scheduling policies and parameters.

7. Error handling: The program includes error handling for file reading errors and input validation. This design choice ensures that the program is stable and returns informative error messages in case of invalid inputs or errors in reading files.

The explanation of our code & algorithms implemented:

- the process and scheduling queue structures

```
typedef struct
{
    char pid;
    int burst_time;
    int io_time;
    int io_duration;
    int priority;
    int arrive_time;
    int no_of_executions;
    int is_waiting;
    int start_waiting_time;
    int pass;
    int stride;
    int start;
    int end;
    int t;
} Process;
```

The process structure is defined to store information about a single process. It has various fields such as pid (process ID), burst_time (time required for the process to complete its execution), io_time (time spent by the process performing I/O operations), priority (priority of the process), and arrive_time (time at which the process arrives in the system), among others

```
typedef struct
{
    char pid;
    int start_time;
    int end_time;

} ProcessGantt;
```

The processGantt structure is defined to store information about a process's execution time, i.e., the start and end times of the process's execution.

```c
typedef struct
{
    Process *queue; // Array to hold the processes
    int front;
    int rear;
    int size;
} Queue;
```

The scheduling queue structure is defined to store a queue of processes that need to be executed. It has an array to hold the processes, and other fields such as front (index of the first element in the queue), rear (index of the last element in the queue), and size (number of elements in the queue).

- Create process and create queues functions

```c
Process createProcess(char pid, int burst_time, int io_time, int io_duration, int priority, int arrive_time)
{
    Process process;
    process.pid = pid;
    process.burst_time = burst_time;
    process.io_time = io_time;
    process.io_duration = io_duration;
    process.priority = priority;
    process.arrive_time = arrive_time;
    process.no_of_executions = 0;
    process.is_waiting = 0;
    process.start_waiting_time = -1;
    process.start = __INT_MAX__;
    process.end = 0;
    return process;
}
```

This function takes various attributes of a process such as (process ID), (time required for the process to complete its execution), (time spent by the process performing I/O operations), (duration of each I/O operation), (priority of the process), and (time at which the process arrives in the system) as its parameters.

The function creates a structure and initializes its fields with the values passed as arguments to the function. It also sets the no_of_executions, is_waiting, start_waiting_time , start and end fields of the process structure to default values.

The function returns the process structure created with the provided parameters. This function is helpful for creating a new process with the given attributes, which can then be added to the scheduling queue.

```c
Queue *createQueue(int size)
{
    Queue *queue = (Queue *)malloc(sizeof(Queue));
    queue->queue = (Process *)malloc(size * sizeof(Process));
    queue->front = -1;
    queue->rear = -1;
    queue->size = size;
    return queue;
}
```

The `createQueue` function creates a queue for storing processes for process scheduling. The function takes an integer `size` as its parameter, which specifies the maximum size of the queue. It first allocates memory on the heap for a `Queue` structure using the `malloc` function. This structure contains three fields - `queue`, `front`, and `rear` - which respectively represent the array of processes stored in the queue, the index of the front element, and the index of the rear element.

- Functions to check if a queue is empty or full respectively

```c
int isEmpty(Queue *queue)
{
    return queue->front == -1;
}

// Function to check if a queue is full
int isFull(Queue *queue)
{
    return (queue->rear + 1) % queue->size == queue->front;
}
```

➢ Reading processes data from files

```
Queue *inputQueueFromFile(const char *filename)
{
    FILE *file = fopen(filename, "r");
    if (file == NULL)
    {
        printf("Error opening file: %s\n", filename);
        return NULL;
    }

    int size;
    fscanf(file, "number of processes: %d\n", &size);
    Queue *queue = createQueue(size);

    char pid;
    int burst_time, io_time, priority, arrive_time, io_duration;
    for (int i = 0; i < size; ++i)
    {

        fscanf(file, "process: %c, start: %d, take: %d, I/O burst: %d, I/O duration: %d, Priority: %d\n"
        , &pid, &arrive_time, &burst_time, &io_time, &io_duration, &priority);
        Process process = createProcess(pid, burst_time, io_time, io_duration, priority, arrive_time);
        enqueue(queue, process);
    }

    fclose(file);

    return queue;
}
```

This function is used for reading input data from a file and setting up a queue of processes for process scheduling algorithms. The input data is expected to be in a specific format, as specified by the `fscanf` function calls in the function. Any deviation from this format may cause errors or unexpected behavior in the function. It first opens the file with the provided filename for reading. If the file does not exist or is not readable, the function returns `NULL`.

Next, the function reads the number of processes from the first line of the file and creates a new `Queue` structure with the specified size using the `createQueue` function.

The function then reads each line containing process information using the `fscanf` function, and constructs a new `Process` structure using the `createProcess` function. The new `Process` structure is then added to the queue using the `enqueue` function.

- Get ready process function

```
void get_ready(Queue *ready, Queue *queue, int time)
{
    while (queue->front != queue->rear)
    {
        if (queue->queue[queue->front].arrive_time == time)
            enqueue(ready, dequeue(queue));
        else
            return;
    }
    if (queue->front == queue->rear && queue->front != -1)
    {
        if (queue->queue[queue->front].arrive_time == time)
            enqueue(ready, dequeue(queue));
        return;
    }

}
```

The `get_ready` function takes a `Queue *ready` pointer, a `Queue *queue` pointer, and an `int time` as input. The purpose of this function is to move the processes that are ready to be executed from the `queue` to the `ready` queue.

- Run_another function:

```
void run_another(Queue *queue, int time, int *total_turnaround, int *last_execution, int core, char arr[][1000])
```

The `run_another` function takes a `Queue *queue` pointer, an `int time`, two `int` pointers `total_turnaround` and `last_execution`, an `int core`, and a 2D character array `arr` as input. The purpose of this function is to execute the process that is currently at the front of the `queue` on the specific `core`, and move it back to the `queue` if it is not finished.

- Move_processes function

```
void move_processes(Queue *destination, Queue *source)
{
    while (!isEmpty(source))
    {
        Process process = dequeue(source);
        enqueue(destination, process);
    }
}
```

The `move_processes` function takes two `Queue` pointers `destination` and `source` as input. The purpose of this function is to move all the elements from the `source` queue to the `destination` queue.

This function is useful for moving processes between various scheduling queues, such as between the ready queue and the blocked queue in an operating system

- All levels is empty function

```
int allLevelsEmpty(Queue *queues[], int no_of_levels)
```

The purpose of this function is to check if all the queues in the `queues` array are empty.

- Gantt chart function

```
void displayProcessesGantt(int n, int time, char arr[n][1000])
```

This function is used for visualizing the execution of processes on multiple cores in a multi-core scheduling algorithm.

- least remaining time function:

```
int leastremainingtime(Process *processes, int numProcesses, char *state, int current_time,
int current_cpu, int *cpu_used)
```

The purpose of this function is to find the process with the least remaining burst time that can run on the current CPU.

- Decrement burst time function:

```
void decrementtime(Process *processes, int numProcesses, char *state,
int *remaining, int *completed, int current_time)
```

The purpose of this function is to decrement the remaining burst time of each process by 1.

- Core reading function:

```
int cores_reading(const char *filename)···
```

The `cores_reading` function takes a string `filename` representing the path to a file as input and returns an integer value representing the number of cores available.

- Least arrive function:

```
int leastarrive(Process *processes, int numProcesses, char *state,
int current_time, int current_cpu, int *cpu_used)···
```

The purpose of this function is to find the process with the earliest arrival time that is ready to run on the given CPU core.

- Shortest job function:

```
int shortest_job(Process *processes, int *current_cpu,
int numProcesses, char *state, int current_time)···
```

The purpose of this function is to find the process with the shortest remaining burst time that is ready to run on an available CPU core.

- Highest priority function:

```
int highest_priority_job(Process *processes, int *current_cpu,
 int size, char *status, int current_time, int core)···
```

The purpose of this function is to find the process with the highest priority that is ready to run on the given CPU core.

- Shortest pass function:

```
int shortest_pass(Process *processes, int *current_cpu,
int numProcesses, char *state, int current_time)···
```

This function returns the index of the process with the shortest remaining burst time that is ready to run. If no such process is found, the function returns -1 it's used for stride policy.

Schedule algorithms implemented

1. Stride scheduling

```
void _stride(Queue *queue, int cores)···
```

The Stride scheduling algorithm assigns each process a "pass value" based on its priority and assigns a "stride value" to each process based on the number of cores available. The algorithm selects the process with the smallest pass value that is in the ready state to execute next.

2. The First-Come, First-Served (FCFS)

```
void fcfs(Queue *queue)···
```

In this algorithm, processes are scheduled in the order they arrive in the ready queue. Once a process starts executing, it continues until it finishes, and other processes have to wait for it to finish before starting their execution.

3. The Shortest Job First (SJF) scheduling

```
void sjf(Queue *queue)
```

The SJF Scheduling algorithm selects the process with the shortest burst time to execute first. The algorithm keeps track of the remaining time for each process and selects the one with the shortest remaining time to execute next. If there are multiple processes with the same shortest remaining time, the algorithm selects the one that arrived first.

4. The Shortest time to completion first scheduling

```
void stcf(Queue *queue)
```

This algorithm schedules processes based on their remaining burst time, with the process that has the shortest remaining burst time being executed first. It is a preemptive algorithm, which means that a process can be interrupted while executing, and the CPU can be reassigned to a process with a shorter remaining burst time. This algorithm aims to minimize the average waiting time and turnaround time, but can result in lower priority processes being starved of CPU time if they have longer burst times.

5. Non-preemptive priority schedule

```
void priority(Queue *queue)···
```

Priority Scheduling is a non-preemptive scheduling algorithm that allocates CPU time to processes based on a priority value assigned to each process. The process with the highest priority is executed first. If two processes have the same priority, then the process with the earliest arrival time is executed first. In this implementation, the processes are sorted based on their arrival time and priority value to create a priority queue. The algorithm executes the processes sequentially, waiting for a process to arrive if required, and then executes the highest priority process. It also simulates I/O operations if the process requires it. Processes are executed until all processes have completed their execution. After all processes have completed,

6. Round-robin (RR) scheduling

```
void rr(Queue *queue, int time_quantum)···
```

This algorithm first sorts the processes based on their arrival time, and then it moves the ready processes to the running processes by enqueuing them to the running queue. It then simulates the execution of the processes from the running queue for the time quantum. If a process's burst time exceeds the time quantum, the process is pre-empted and is enqueued back to the ready queue. If the process completes its execution within the time quantum, it is removed from the system. It also simulates I/O operations if the process requires it by freezing the process and enqueuing it back to the ready queue. If there are no processes ready to execute at a specific time, the algorithm skips that time and moves to the next available time.

7. Multi-level feedback queue scheduling

```
void mlfq(Queue *queue, int no_of_levels, int time_quantum, int boost_time)···
```

The multi-level feedback queue scheduling algorithm is designed to prioritize short jobs. The algorithm assigns processes to different levels of queues based on their priority and execution time, where the highest priority and shortest jobs are executed first. The algorithm also allows processes to move between queues based on their I/O requirements and execution history, giving shorter jobs more opportunities to execute and complete. Additionally, the algorithm boosts the priority of lower-level queues at regular intervals to prevent processes from waiting too long and ensure that short jobs are prioritized. Overall, the multi-level feedback queue scheduling algorithm is designed to increase the overall system throughput and minimize the average turnaround time of processes.

Instruction for using the simulator:

1. Extract the program files into a folder.

2. Open the DataSet.txt file and enter the processes that you want to schedule in the following format:

pid arrival_time burst_time io_time priority

The fields are separated by spaces. Each line represents a process.

3. Open the system.txt file and enter the number of available cores in the system.

4. Compile the program with a C compiler such as GCC.

5. Run the program from the command line or terminal.

6. When the program starts, it will prompt you to select a scheduling policy from a list of available policies.

7. Based on the selected policy, the program may prompt you to enter additional parameters such as time quantum, number of levels, and boost time.

8. Once the simulation is complete, the program will display the average turnaround time and a Gantt chart showing the execution time of the processes in each core.

9. You can repeat steps 2-8 with different inputs and policies to compare the performance of different scheduling algorithms.

Conclusions about performance of each scheduler:

The performance of each scheduler varies depending on the type of workload being executed. Here are some general conclusions about the performance of each scheduler with different kinds of workloads:

1. First-come, first-served (FCFS): performs well with long-running CPU-bound processes but can result in high turnaround time for short CPU-bound processes or processes with long I/O time. As this scheduling policy is non-preemptive, it may not be suitable for Real-time systems or systems requiring fast response time.

2. Shortest Job First (SJF): ideal for minimizing turnaround time for batch processing systems with CPU-bound processes. However, the optimal CPU burst time of each process needs to be known in advance, which is often unrealistic.

3. Round Robin (RR): good for interactive or time-sharing systems with processes of similar priority. With shorter time quantum values, RR performs well with short processes with few I/O operations.

4. Priority scheduling: Best for real-time systems or systems with processes requiring faster processing with higher priorities. However, it may experience Starvation of lower priority processes if not implemented well.

5. Multi-level Feedback Queue (MLFQ) scheduling: suitable for workloads with varying computation times that can be divided into multiple levels of priority. A recompilation step boosts time to exhausted processes, resulting in fairer allocation.

6. Stride scheduling: suitable for workloads with differing resource demands. This policy targets proportional allocation strategies, where processes with a higher proportional share of CPU time are allocated with more cycles than standard schemes.

In conclusion, each scheduling policy has its strengths and weaknesses, and the best policy depends on the system and the workload being handled. By selecting the appropriate scheduling policy, it is possible to improve system performance and optimize process scheduling in multi-core systems.

Performance:

Case 1:
number of processes: 4 , cores: 4
process: A, start: 0, take: 5, I/0 burst: 2, I/0 duration: 2, Priority: 2
process: B, start: 0, take: 10, I/0 burst: 0, I/O duration: 0, Priority: 1
process: C, start: 5, take: 9, I/0 burst: 2, I/O duration: 1, Priority: 0
process: D, start: 15, take: 6, I/0 burst: 0, I/0 duration: 0, Priority: 0

Case 2:
number of processes: 4 , cores: 1
process: A, start: 0, take: 5, I/0 burst: 2, I/0 duration: 2, Priority: 2
process: B, start: 0, take: 10, I/0 burst: 0, I/O duration: 0, Priority: 1
process: C, start: 5, take: 9, I/0 burst: 2, I/O duration: 1, Priority: 0
process: D, start: 15, take: 6, I/0 burst: 0, I/0 duration: 0, Priority: 0

Case 3:
number of processes: 5, cores: 4
process: A, start: 10, take: 5, I/0 burst: 2, I/O duration: 2, Priority: 2
process: B, start: 9, take: 10, I/0 burst: 0, I/O duration: 0, Priority: 1
process: C, start: 5, take: 9, I/0 burst: 2, I/O duration: 1, Priority: 0
process: D, start: 12, take: 6, I/0 burst: 0, I/O duration: 0, Priority: 0
process: E, start: 0, take: 3, I/0 burst: 1, I/O duration: 1, Priority: 2

Case 4:
number of processes: 3, cores: 2
process: A, start: 0, take: 5, I/0 burst: 2, I/0 duration: 2, Priority: 2
process: B, start: 0, take: 10, I/0 burst: 0, I/O duration: 0, Priority: 1
process: C, start: 0, take: 7, I/0 burst: 1, I/0 duration: 1, Priority: 1

1	Case 1	FCFS	SJF	STCF	RR (time quantum 4)	Priority	MLFQ (levels 3, quantum 4, boost 6)	Stride
2	Average Turnaround	9.5	9.5	9.5	10	9.5	10.5	9.5
3	Average Response	0	0	0	0.5	0	0.5	0
4	Best	SJF , PRIORITY, STRIDE, fcfs, stcf						
5								
6	Case 2							
7	Average Turnaround	16.25	16.25	14.75	18.5	16.5	20.25	22.25
8	Average Response	3.25	3.25	4.5	2.75	3.75	2.5	7
9	Best	FCFS and SJF						
10								
11	Case 3							
12	Average Turnaround	8.6	8.6	8.6	9	8.6	11.4	9.2
13	Average Response	0	0	0	0	0	1	0.4
14	Best	FCFS , SJF, STCF, Priority						
15								
16	Case 4							
17	Average Turnaround	12.33	12	13	12.33	12.33	15	12.33
18	Average Response	0.67	0.33	0.33	0.67	0.67	0.67	0.67
19	Best	SJF						

Thank you!