Simple Preemptive SFJ Scheduling Simulation

Operating System Concepts

Saman Mahdanian - 610399197

Introduction

The given code is an implementation of a scheduling algorithm called **Shortest Burst Time First (SBTF)**. The purpose of this algorithm is to schedule the execution of processes in a way that minimizes the average waiting time of processes. This is achieved by prioritizing the processes with the shortest burst time. Also, because of the limitations of the simulation procedure here we use a time quantom of 50ms.

The code also includes a logging feature that records the start and end times of each process as well as any preemption that occurs.

Implementation

The code uses C++17 thread library to implement the scheduling algorithm in a separate thread. The main thread is responsible for taking user input and inserting new processes into the scheduler's queue. The scheduler thread continuously runs and handles the execution of the processes.

The scheduler thread uses a `std::set` container to store the processes in the queue. The set is sorted by the burst time of the processes, with the process having the shortest burst time at the front. The scheduler thread takes the front process from the set, starts its execution and logs the start time. The process's burst time is decremented by a quantum time (50ms) and if it completes execution, the process is terminated and removed from the set. If the process still has burst time remaining, it is inserted back into the set.

The code also uses a `std::unordered_map` container to store the remaining burst time of each process. This is used to update the burst time of a process if it is preempted.

A std::mutex is used to synchronize access to the shared data structures between the main thread and the scheduler thread.

Implementation Notes:

- The program is written in C++ and makes use of several C++ standard library features such as the `iostream`, `set`, `chrono`, `thread`, `fstream`, `mutex`, and `unordered_map` libraries.
- The Process struct is used to represent a process. It contains the process ID (pid), the burst time (bt) and the arrival time (art) of the process. The `< operator` is overloaded for this struct to compare the burst time of two

processes. This is used by the set data structure to sort the processes in the ready queue.

- The `set<Process>` processes is used to store the processes in the ready queue. The set is sorted by the burst time of each process, with the process having the shortest burst time at the front.
- The `unordered_map<int, int>` remaining_time is used to store the remaining burst time of a process that was preempted.
- The `active_process_pid` variable is used to keep track of the currently active process.
- The `mtx` variable is used to implement a mutex lock for synchronizing access to the shared data structures.
- The `user_alive` variable is used to keep track of whether the user is still entering processes or not.
- The `user_alive_mtx` and scheduler_alive_mtx variables are used to implement mutex locks for synchronizing access to the `user_alive` variable.
- The `schedule()` function is the main function that is run by the scheduler thread. It runs in an infinite loop and repeatedly selects the process at the front of the set, starts executing it, and then preempts it if a process with a shorter burst time arrives. The function also logs all the process scheduling events, such as process start, process termination, and process preemption, in a file called scheduler.log.
- The `main()` function is responsible for creating the scheduler thread and providing a simple user interface for entering new processes. It runs in an infinite loop, where the user is prompted to enter the process ID and burst time of a new process. If the user enters 0 0, the loop exits, and the program ends.

Usage

To run the code, the user must enter the process ID and burst time (in seconds) of each process. The user can stop entering processes by entering ”0 0” as the process ID and burst time. The scheduler thread will continue to execute the existing processes and terminate when all processes have completed execution.

The scheduler thread will also log the start and end times of each process as well as any preemption that occurs in a file named ”scheduler.log”.

The code can be modified to change the quantum time or the scheduling algorithm as per the requirement.

Conclusion

The given code is an implementation of the Shortest Burst Time First (SBTF) scheduling algorithm using C++17 thread library. The code is designed to minimize the average waiting time of processes and includes logging feature to record the start and end times of each process and preemption. The code is well-commented and can be easily understood and modified as per the requirement.