Homework 1 Report

Course: BLG312E – Operating Systems
Homework: HW1 – Preemptive Priority-Based Scheduler

Name: Emre Safa Yalçın
Student Number: 150210090

1. Introduction

This report describes my implementation of a **preemptive priority-based process scheduler** for homework assignment HW1 in BLG312E. The scheduler reads process information from a `jobs.txt` file, spawns new child processes via `fork()` and `exec()`, and uses signals (`SIGSTOP`, `SIGCONT`) to preempt and resume processes as necessary.

The **key functionalities** are:

1. **Time slicing**: Each process runs for up to 3 seconds before being preempted.
2. **Priority scheduling**: Lower numerical priority means higher scheduling preference.
3. **Arrival times**: A job that has not yet arrived (based on its specified arrival time) is not considered for scheduling.
4. **Logging**: We maintain a `scheduler.log` that records major events (forking, executing, preempting, resuming, termination).

This document details the design decisions, scheduling flow, and testing approach. A separate file, **`answers.pdf`**, provides specific answers to the written questions in the homework.

---

2. Design Decisions

1. **Data Structure**:
    - I maintain an array of `Job` structs, each storing information like `name`, `arrival_time`, `priority`, `total_exec_time`, and `remaining_time`.
    - These are read from `jobs.txt` and processed in a main scheduling loop.
2. **Scheduling Algorithm**:
    - **Preemptive**: On each time slice expiration (3 seconds, or when a process completes early), we stop the currently running job (if unfinished) and select another job.
    - **Priority + Round Robin**: Among the processes that have arrived and are not finished, we pick the job with the **lowest** priority (numerical value). If there's a tie, we pick the job with the **earlier arrival time**. If no other job is ready, we continue the same job. Additionally, we enforce "no back-to-back" scheduling if another job is available, ensuring each partial job eventually gets CPU time.
3. **Signal Management**:

- We use `SIGSTOP` to pause a process when its time slice ends (or right after we fork it).
- We use `SIGCONT` to resume a paused process.

4. **Same-Priority Arrival Delay**:
    - If a job with the same priority arrives **later** than another job still in progress, the new job is delayed (not forked) until the older job finishes. This matches the homework requirement that earlier arrivals of the same priority must complete first.
5. **Logging**:
    - Every major event goes into `scheduler.log` in the format `[TIMESTAMP] [INFO] message`.
    - This includes forking, executing, "Time slice expired - Sending SIGSTOP," "Resuming jobX," and job completion lines.

---

3. Flow of Execution

1. **Initialization**:
    - The scheduler opens `jobs.txt`, parses the time slice (e.g. `TimeSlice 3`) and each job's parameters.
    - It sets up an array of `Job` structs, initializing `remaining_time` to the job's total CPU need.
2. **Main Loop**:
    - We track a `current_time`.
    - We **fork** (and immediately `SIGSTOP`) any newly arrived jobs that are not delayed by an older job of the same priority.
    - If there is no currently running job, we pick one based on priority and earliest arrival. If it's the job's **first** run, we simply `SIGCONT` it. If it's a **previously preempted** job, we log "Resuming jobX - SIGCONT."
    - The chosen job runs for up to 3 seconds, decrementing its `remaining_time`. If it finishes earlier, we log completion. Otherwise, at 3 seconds, we do `SIGSTOP` and pick another job for the next slice.
    - We repeat until all jobs have `finished = 1`.
3. **Termination**:
    - Once every job is completed, the scheduler ends.

---

4. Testing and Findings

- **Test Input**:

  ```
  nginx
  CopyEdit
  TimeSlice 3
  jobA 0 1 6
  jobB 2 2 9
  jobC 4 1 4
  ```

  This is the sample scenario from the homework PDF.

- **Expected Behavior**:

  1. jobA starts at time=0. Runs 3s, preempted.
  2. jobB starts at time=3. Runs 3s, preempted.
  3. jobA resumes at time=6, finishes at time=9.
  4. jobC is forked at time=9 because it shares priority with jobA but arrived later (time=4).
  5. jobC runs, alternates with jobB, etc.
  6. The final `scheduler.log` matches the sample format from the assignment.

- **Result**:
  The logs show a correct sequence of events. The scheduling ensures no single job hogs the CPU, higher priorities are respected, and arrivals are enforced. The final `scheduler.log` lines confirm all jobs complete.

---

5. Conclusion

We implemented a **preemptive priority-based scheduler** in C using `fork()`, `exec()`, and signals. The design meets the homework requirements:

- Correct usage of signals (`SIGSTOP`, `SIGCONT`),
- Priority-based selection with time slicing,
- Delay of same-priority, later arrivals,
- Detailed logging in `scheduler.log`.

In a real OS, additional features (like dynamic priority aging, deadlock detection, etc.) would be crucial. Nonetheless, for this homework, the described approach satisfies the given constraints and demonstrates a working scheduler.
