Emre Safa Yalçın - 150210090

1. Scheduling Fairness Analysis

Our scheduler uses **preemptive time slicing** and ensures that no job can continuously hold the CPU if another job is ready to run. Specifically:

1. **Preemptive Time Quantum**: Each job receives a maximum of 3 seconds of CPU time at once. After 3 seconds, the job is forced to yield (via `SIGSTOP`) so another waiting job can run. This prevents a single job from monopolizing the CPU.
2. **No Same-Job-Back-to-Back**: After a time slice expires for job X, if any other job is available, we run that other job next. This effectively cycles among the jobs, so no single job (even if it has higher priority) keeps running endlessly.
3. **Priority + First Arrival**: Among multiple ready jobs, we choose the one with the highest priority (lowest numerical value). If there is a tie in priority, we pick the earliest arrival. This ensures that higher priority jobs are generally served first, but once a job is preempted, we still give others a chance to execute if they are ready.

Fairness Implications:

- The **time slice** mechanism enforces a kind of “Round Robin” sharing among jobs at the same or lower priority, preventing starvation (because eventually each job receives a turn if it’s ready).
- By combining priority scheduling with Round Robin, we provide a balance: high-priority tasks do receive faster service, but they cannot indefinitely starve lower-priority tasks, since the time quantum and “no same-job-back-to-back” rule keep the CPU cycling.

Possible Improvements:

- **Dynamic Priorities**: If a lower-priority job waits too long, we could boost its priority to prevent indefinite postponement (aging).
- **Completely Fair Scheduling** (like Linux’s CFS) uses a weighted fair-share approach, factoring in each job’s share of the CPU over time.
- **Lottery Scheduling**: Each job gets tickets proportional to its priority; a random draw decides which job runs next—this can statistically ensure fairness while still respecting priority.

---

2. Edge Cases and Failure Scenarios

Below are some potential issues and how our scheduler handles (or could handle) them:

1. **Starvation**:
    - A lower-priority job might get overshadowed by multiple higher-priority jobs.

- **Our Implementation**: The mandatory time slice preemption and the rule “no same-job-back-to-back if another is available” help mitigate long waits. However, if new high-priority tasks continuously arrive, a lower-priority job might still wait longer. We do not implement aging or dynamic priority, so *true* starvation is still theoretically possible if high-priority jobs keep arriving.

2. **Process Not Responding (Hung Child)**:
    - A job might ignore signals or get stuck.
    - **Our Implementation**: We rely on `SIGSTOP`/`SIGCONT` for control and eventually `SIGTERM` if the job is finished or unresponsive. If a child process truly hangs (e.g., kernel-level uninterruptible wait), our code might stall. In practice, typical well-behaved user processes will stop/continue on these signals.
3. **Deadlock**:
    - True deadlock typically involves multiple processes waiting on resources. Our scheduler doesn’t handle resource locks explicitly, so if the processes themselves become deadlocked, the scheduler can only keep time-slicing them.
    - **Our Implementation**: We do not address resource locks directly. The scheduler itself does not cause deadlock, but it also does not detect or resolve deadlocks in child processes.
4. **Immediate Termination**:
    - A process might exit earlier than its stated execution time. We watch for child exits via `waitpid(..., WNOHANG)`.
    - **Our Implementation**: We detect if a process ended before using its entire quantum (`remaining_time` goes to 0 or the child is `waitpid`-ed). We then log “completed execution” and move on.
5. **Multiple Jobs Arriving at Same Time**:
    - If several processes share the same arrival time and priority, we break ties by whichever appears first in the input file.
    - **Our Implementation**: The tie is resolved in `pick_next_job()` by comparing arrival times, and if they are exactly equal, we pick whichever is encountered first in the `jobs` array.

Conclusion

Our scheduler ensures *some* level of fairness through **time slicing** and **no back-to-back re-selection** rules. It prevents one job from monopolizing the CPU if others are waiting. However, it doesn’t implement sophisticated fairness strategies like **dynamic priority aging** or **completely fair scheduling**, so starvation can still occur in edge cases. We address most straightforward failure scenarios (e.g., child finishing early, or ignoring signals) with checks and signals (`SIGSTOP`, `SIGCONT`, `SIGTERM`) but do not provide advanced deadlock detection or resource management.