Operating Systems Project 2

Thread-Based Process Simulation and Synchronization

Course: Operating Systems 4320–6320

Overview

In this project, you will simulate real-time process execution using threads and learn how operating systems handle synchronization. You will implement one classic synchronization problem using mutexes or semaphores.

Step-by-Step Instructions

1. Simulate Processes with Threads

- Reuse or modify your `processes.txt` file from Project 1.
- Each process becomes a thread.
- Simulate the CPU burst using `sleep()` or `Thread.sleep()`.

Example (Java):

```
class ProcessThread extends Thread {
    int pid, burstTime;
    public ProcessThread(int pid, int burstTime) {
        this.pid = pid;
        this.burstTime = burstTime;
    }
    public void run() {
        System.out.println("Process " + pid + " started.");
        try {
            Thread.sleep(burstTime * 1000);
        } catch (InterruptedException e) {}
        System.out.println("Process " + pid + " finished.");
    }
}
```

2. Add Synchronization with Locks or Semaphores

Choose **one** of the following problems and solve it using synchronization primitives:

2.1. Dining Philosophers

- Five threads represent philosophers.
- Each philosopher thinks, picks up two forks (locks), eats, and releases forks.
- Use strategies to avoid deadlocks (e.g., pick up lower-numbered fork first).

2.2. Readers-Writers

- Multiple readers can read simultaneously.
- Writers need exclusive access.
- Use mutexes and reader counters to manage access.

2.3. Producer-Consumer

- A producer thread adds items to a buffer.
- Consumer threads remove items.
- Use a bounded buffer, mutex, and semaphores to control access.

3. Show Thread Activity

- Print when each thread starts, waits for a lock, acquires it, and finishes.
- Example:

```
[Philosopher 2] Waiting for forks...
[Philosopher 2] Picked up fork 1 and 2
[Philosopher 2] Eating...
[Philosopher 2] Released forks
```

4. Submit Your Work

1. Source code with comments
2. Input file (`processes.txt`)
3. Output screenshot or terminal log
4. Short report (1–2 pages, PDF)

5. Grading Criteria

Task	Points
Thread creation from process input	25
Correct implementation of one synchronization problem	35
Output clarity (logs, execution order)	20
Report with explanation and screenshots	20

Tips

- Start by getting basic threads to work before adding synchronization.
- Test with multiple runs to catch random issues.
- Use print statements generously to debug.
- Java users: Try `ReentrantLock`, `Semaphore`, `Thread`.

Free Tools Notice

Everything you need to complete this project is completely free:

- **Programming Languages:** Java, C, and C++ are all free and widely supported.
- **Threading & Synchronization:** Java's `Thread`, `Semaphore`, and `ReentrantLock` classes are built-in. C/C++ users can use `pthread.h` or equivalent libraries.
- **Development Tools:** You may use free IDEs such as VS Code, Eclipse, IntelliJ (Community Edition), Code::Blocks, or terminal-based tools.
- **Compilation:** GCC (Linux/macOS), MinGW (Windows), or any other free compiler works well.
- **Documentation:** Overleaf is free and supports single-file LaTeX documents like this project.

You do not need to buy any software or tools. Everything can be done using free resources available online or already installed on your system.