Operating Systems Project 2:

Process Scheduling Simulation

Spring 2025

CSC 4320: Operating Systems

Team Members: Harry Cao, Sheila Corona, Son Nguyen

Disclaimer: One of us has an English degree, so this is meticulously explained & lengthy… enjoy!

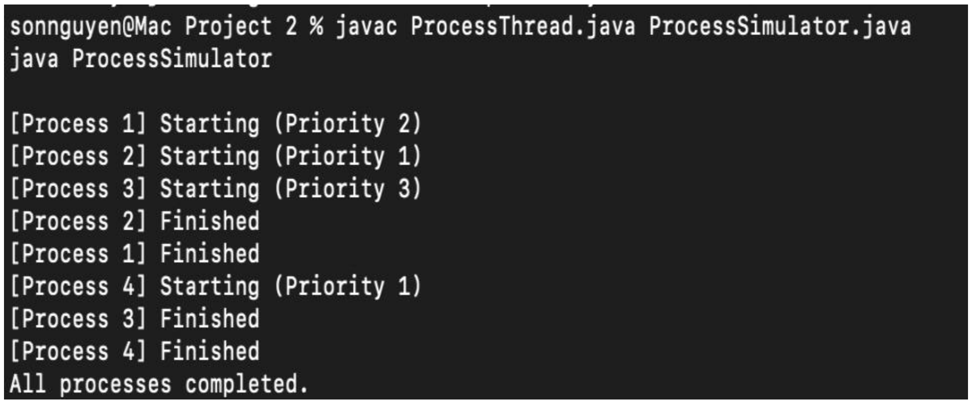
Introduction:

This project simulates real-time process execution using Java threads to explore synchronization in operating systems. Implementing the classic Producer-Consumer problem with semaphores and mutex locks demonstrates how shared resources are managed safely in a multi-threaded environment. The simulation also includes basic CPU scheduling behavior using process threads to represent arrival time, burst time, and priority. Overall, the project highlights how synchronization ensures proper coordination between concurrent tasks.

Simulating Processes with Threads:

To simulate real-time process execution, my group created a ProcessThread class that extends Thread. Each thread represents a single process, initialized with its process ID, arrival time, burst time, and priority, all of which are read from a processes.txt file. To simulate the delay before a process starts running (its arrival time), I used Thread.sleep() at the beginning of each thread's run() method. Then, to simulate the CPU burst, the thread sleeps again for the length of its burst time. This approach allows the simulation to loosely mimic how processes are scheduled and executed over time in an actual operating system.

The simulation doesn’t explicitly control thread priorities, instead, the priority value is read in and displayed, but actual thread execution is handled by the Java Virtual Machine’s scheduler, which may not honor those priorities. As a result, the order in which threads finish is based more on arrival and burst times than priority. This reflects real-world behavior, where many factors (like time slicing and system load) can influence when a process runs.



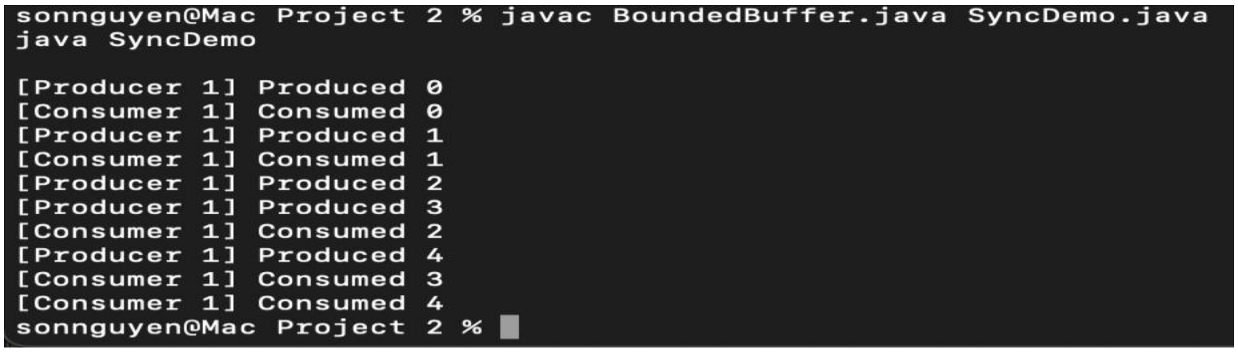
```
sonnguyen@Mac Project 2 % javac ProcessThread.java ProcessSimulator.java
java ProcessSimulator

[Process 1] Starting (Priority 2)
[Process 2] Starting (Priority 1)
[Process 3] Starting (Priority 3)
[Process 2] Finished
[Process 1] Finished
[Process 4] Starting (Priority 1)
[Process 3] Finished
[Process 4] Finished
All processes completed.
```

This output shows that multiple processes can begin executing around the same time, and that shorter burst times tend to finish first regardless of the assigned priority. It highlights how Java threads operate concurrently and how arrival and burst times affect execution order, even in a simplified CPU scheduling simulation like this one

Implementing Synchronization: Producer-Consumer

For the second part of the project, I implemented the classic **Producer-Consumer** synchronization problem using Java's concurrency tools. The main goal was to safely coordinate access to a shared resource which in this case was a bounded buffer between a producer thread and a consumer thread. Without synchronization, both threads could access the buffer at the same time, potentially causing race conditions or data corruption.

To prevent that, I built a bounded buffer of size 3 and used a combination of a ReentrantLock (used as a mutex) and two semaphores. The empty semaphore tracked how many open slots were available in the buffer, while the full semaphore tracked how many slots were filled. The mutex was used to protect the critical section, the part of the code where the buffer itself is read from or written to ensure only one thread could modify it at a time. The producer thread was responsible for generating items and placing them into the buffer. Before it could add an item, it had to acquire the empty semaphore (to ensure space was available) and lock the mutex. After inserting the item, it released the full semaphore to signal that something was available for consumption. Similarly, the consumer thread acquired full before trying to take an item and released empty after removing it, so that the producer would know a slot had opened.



```
sonnguyen@Mac Project 2 % javac BoundedBuffer.java SyncDemo.java
java SyncDemo

[Producer 1] Produced 0
[Consumer 1] Consumed 0
[Producer 1] Produced 1
[Consumer 1] Consumed 1
[Producer 1] Produced 2
[Producer 1] Produced 3
[Consumer 1] Consumed 2
[Producer 1] Produced 4
[Consumer 1] Consumed 3
[Consumer 1] Consumed 4
sonnguyen@Mac Project 2 %
```

This output shows that the producer can add multiple items if the buffer isn’t full, and the consumer waits appropriately until something is available to consume. Even though the threads run concurrently, the semaphores and lock ensure that the buffer’s state always remains consistent. There’s no overlap, missing values, or out-of-order access, which confirms that synchronization was successfully implemented.

Facing Challenges:

While working on this project, we faced a few challenges that helped deepen our understanding of thread synchronization. One of the first issues was a missing import statement like forgetting to include import java.util.concurrent.locks.ReentrantLock caused compilation errors until we realized the lock class wasn't being recognized by the compiler. Another key challenge was making sure semaphores were correctly acquired and released in the right order. A small mistake here could easily lead to deadlock or resource starvation, so we had to be intentional with how threads were synchronized. To ensure everything was functioning reliably, we ran the program multiple times and checked that the output consistently reflected safe, expected behavior across different execution orders.

Conclusion:

Through this project, our team gained valuable hands-on experience with key concepts in concurrent programming. We practiced thread creation and management in Java, simulated basic operating system process scheduling behavior, and implemented synchronization using semaphores and locks. These tasks helped us better understand how critical sections, race conditions, and thread-safe programming work in practice. Overall, this project deepened our appreciation for the challenges and importance of synchronization in real-world operating systems and concurrent applications.