Operating Systems Project 2:

Thread-Based Process Simulation and Synchronization

Name: Vu Dao

Panther ID: 002716984

Github: https://github.com/joey-congvu/CSC4320_Project2.git

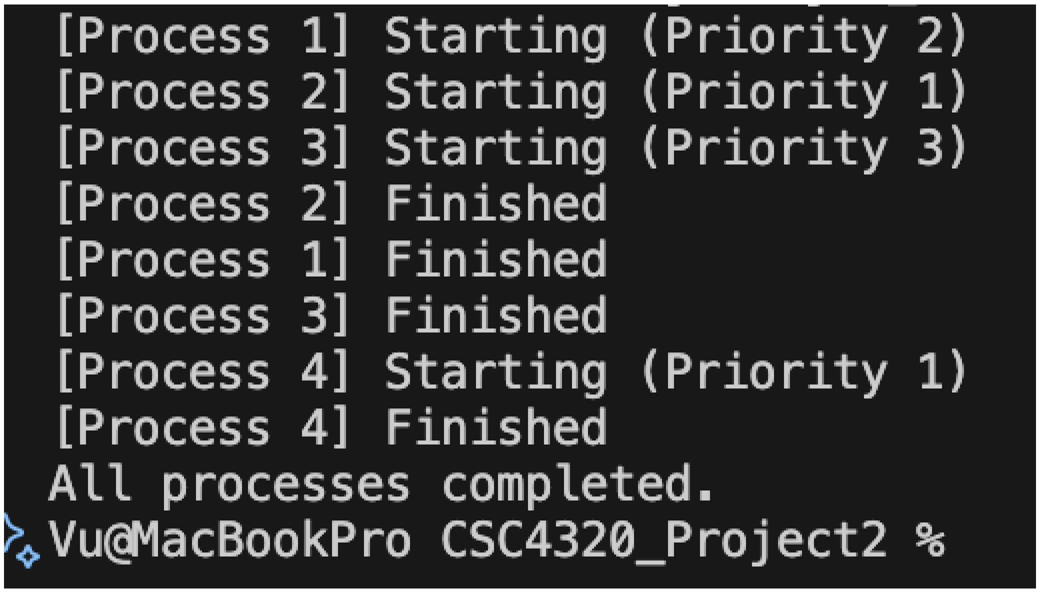
Introduction

This project simulates real-time process execution using Java threads so I can better understand synchronization in operating systems. I also implemented the classic Producer–Consumer problem using semaphores and a mutex lock to show how shared resources are safely handled in a multi-threaded environment. On top of that, I built a simple CPU-style process simulation where each process has its own arrival time, burst time, and priority. Altogether, the project highlights how synchronization keeps concurrent tasks from interfering with each other.

Simulating Processes with Threads

To model real processes, I created a ProcessThread class that extends Thread. Each thread represents a single process, and its data—PID, arrival time, burst time, and priority—comes from a processes.txt file (exactly the same processes from my first project). To simulate arrival time, the thread sleeps before it starts running. To simulate the CPU burst, it sleeps again for the burst duration. This gives a simple but clear idea of how processes arrive and run over time in an operating system.

The program reads the priority value but does not enforce priority scheduling. The actual execution order depends more on arrival time and burst time because the JVM controls thread scheduling internally. This is realistic, since in real systems many factors outside priority influence when a process actually runs.



```
[Process 1] Starting (Priority 2)
[Process 2] Starting (Priority 1)
[Process 3] Starting (Priority 3)
[Process 2] Finished
[Process 1] Finished
[Process 3] Finished
[Process 4] Starting (Priority 1)
[Process 4] Finished
All processes completed.
Vu@MacBookPro CSC4320_Project2 %
```

From the output, it's easy to see that multiple processes can run at the same time, and shorter burst times finish quickly even if their priority is lower. This shows how concurrency and timing affect the overall execution order, even in a basic simulation like this.

Implementing Synchronization: Producer–Consumer

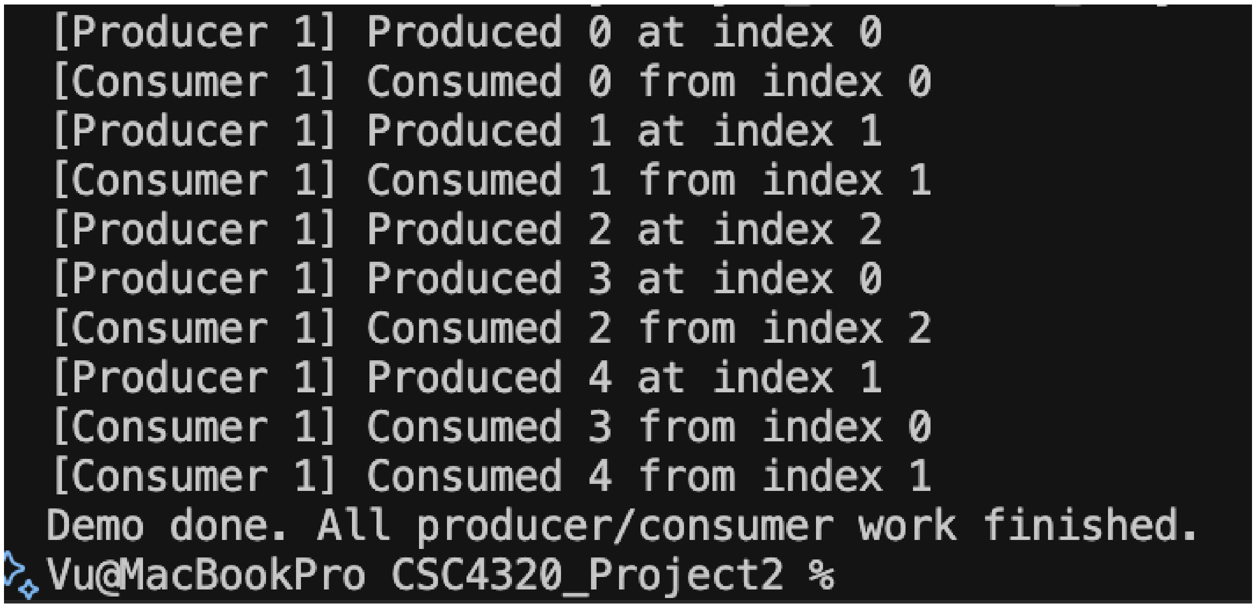
For the Producer–Consumer part, I implemented synchronization using Java semaphores and a ReentrantLock. The shared resource in this case was a bounded buffer of size 3. Without synchronization, both threads could access the buffer at the same time, which would lead to race conditions and corrupted data.

To avoid that, I used:

- an empty semaphore to track open slots,
- a full semaphore to track filled slots,
- and a mutex lock to protect the critical section where the buffer is read or written.

The producer must acquire empty before adding an item and lock the mutex before writing into the buffer. After inserting an item, it releases full to signal that something is available for the consumer.

The consumer does the opposite: it acquires full before removing an item and releases empty afterward so the producer knows a spot opened up.



```
[Producer 1] Produced 0 at index 0
[Consumer 1] Consumed 0 from index 0
[Producer 1] Produced 1 at index 1
[Consumer 1] Consumed 1 from index 1
[Producer 1] Produced 2 at index 2
[Producer 1] Produced 3 at index 0
[Consumer 1] Consumed 2 from index 2
[Producer 1] Produced 4 at index 1
[Consumer 1] Consumed 3 from index 0
[Consumer 1] Consumed 4 from index 1
Demo done. All producer/consumer work finished.
Vu@MacBookPro CSC4320_Project2 %
```

The output shows the expected behavior: the producer waits if the buffer is full, and the consumer waits if nothing is available. Even though the threads run concurrently, the buffer remains consistent with no overlapping writes or bad reads.

Facing Challenges

While working on this project, I ran into a few issues that helped reinforce how synchronization works. One early mistake was simply forgetting to import classes like ReentrantLock, which caused compilation errors until I realized what was missing. A bigger challenge was making sure semaphores were always acquired and released in the correct order. A small mistake there can easily lock up the whole program. I had to run the simulation multiple times to confirm that the producer and consumer always behaved correctly, even when thread timing changed between runs.

Conclusion

This project gave me solid hands-on experience with key ideas in concurrent programming. I created and managed threads in Java, simulated basic process scheduling with arrival and burst times, and used semaphores and locks to coordinate access to a shared resource. Working through these tasks helped me understand critical sections, race conditions, and thread-safe programming much more clearly. Overall, this project strengthened my understanding of why synchronization is essential in real operating systems and multi-threaded applications.