CIS 415 Operating Systems

Project 3 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Vikram Thirumaran

vikram

952046110

Report

Introduction

Project 3, titled Duck Park, is a multi-phase simulation that models the behavior of a theme park ride using advanced concepts in concurrency and inter-process communication. The project required the use of POSIX threads (`pthreads`), semaphores, and shared memory to create a realistic and concurrent system in which passenger threads enter the park, explore, acquire tickets, queue up, board cars, enjoy rides, and repeat the process. Each car has a limited number of seats and must follow strict loading, departure, and unloading protocols to ensure safety and fairness. The core goal of the project was to implement and coordinate multiple threads (passengers, cars, and a monitor) while preventing race conditions, deadlocks, or starvation scenarios.

The project was split into three distinct parts. In Part 1, we implemented a single car and a single passenger to build the foundation of thread coordination. Part 2 scaled this up to multiple concurrent passengers and cars, introducing more complex scheduling, car capacity enforcement, and ticketing. Part 3 further added inter-process communication using `mmap`, a monitor thread for live system reporting, and final simulation statistics. The design emphasized the use of semaphores for mutual exclusion, barriers for ride control, and timing constraints to simulate real-world delays and interactions. Throughout the project, we closely followed the rubric to ensure key features such as `FIFO` queueing, staggered arrivals, shared state monitoring, and proper statistics reporting were all robustly implemented.

Background

This project relied on several foundational operating systems concepts, including multithreading, synchronization primitives, and inter-process communication (IPC). At its core, we used POSIX threads (`pthread_create, pthread_join`) to simulate concurrent entities such as passengers, cars, and the system monitor. To safely coordinate access to shared resources like queues, ticket booths, and car states, we used semaphores (`sem_init, sem_wait, sem_post`) for mutual exclusion and thread signaling. These mechanisms allowed us to prevent race conditions while enabling concurrent operations like loading passengers or dequeuing from the ticket line.

One of the major challenges involved enforcing fair scheduling in a concurrent system. We implemented FIFO queues for both ticket and ride lines, ensuring that passengers were served in the order they arrived. To accomplish this, we maintained explicit queue structures with controlled access using semaphores. Additionally, for Part 3, we implemented a memory-mapped file (`mmap`) to create shared memory between the simulation and a separate monitor thread. This monitor read the live state of the simulation and printed updates every 5 seconds, simulating system-level diagnostics. We also made several key design decisions, such as using a round-robin departure order for cars and requiring full unloading of one car before another could begin, to model real-world ride operation constraints. The POSIX documentation, man pages, and concurrency pattern references were particularly helpful in refining synchronization logic and ensuring correctness under concurrent load.

Implementation

The implementation of *Duck Park* progressed through three stages, each building on the synchronization and concurrency mechanisms established in the previous part. In **Part 1**, I focused on a foundational setup with one passenger and one car, validating the logic of exploration, ticket acquisition, boarding, and riding without concurrent pressure. This served as a testbed for ensuring basic function and system timing before introducing

complexity. It also helped me experiment with initial semaphore usage and identify key synchronization points for later scaling.

In **Part 2**, I scaled the simulation to handle multiple concurrent passenger and car threads. I introduced a semaphore-guarded ticket booth so only one passenger could acquire a ticket at a time, and a ride queue implemented with additional semaphores and condition logic to control access to cars. Each car thread operated independently, loading passengers up to its capacity, starting the ride, and then unloading, all while coordinating through shared state variables and mutexes. I implemented a custom FIFO queue system for both ticket and ride lines to enforce fair order, ensuring early-arriving passengers always boarded first. Car loading was restricted to one car at a time, enforced via a shared `loading_lock` semaphore. This phase presented my first major concurrency challenges, particularly around avoiding race conditions during simultaneous car and passenger actions.

Part 3 significantly extended the complexity and realism of the simulation. I introduced a monitor thread and inter-process communication via `mmap`. Shared memory structures stored queue states, car status, and timing data, which the monitor accessed and printed every 5 seconds. The monitor used program-relative timestamps to track simulation behavior, supporting consistent and reproducible system logs. I also implemented logic for staggered passenger entry (every 2 seconds), car round-robin scheduling, and mandatory full unload before the next car could begin. To improve realism, passenger exploration time was randomized between 1–10 seconds using `rand()`, and cars departed either when full or after a 10-second timeout using time-based semaphores. Each passenger looped back into the park until they completed at least one ride, at which point they exited the simulation.

Performance Results and Discussion

The final simulation in Part 3 was tested with the following standard configuration: 20 passenger threads, 3 car threads (each with a capacity of 5), a ride duration of 8 seconds, and a car wait timeout of 10 seconds. Passengers entered the park every 2 seconds and explored for a randomized duration between 1 and 10 seconds before acquiring tickets. The system monitor reported status updates every 5 seconds, tracking ticket and ride queues, car occupancy, and elapsed simulation time.

The simulation successfully met all core performance criteria specified in the rubric:

- All 20 passengers completed at least one ride before the simulation ended.
- Cars did not depart early unless the timeout condition was met and at least one passenger was onboard.
- The ride queue followed FIFO order, with passengers boarding in the same sequence they queued.
- Car loading was restricted to one car at a time; no overlap or race conditions occurred.
- The system monitor displayed state snapshots (e.g. queue lengths, car fill levels) every 5 seconds, with correct timestamps based on simulation start.
- Final statistics were printed at the end of the simulation using program time (not system time).

```
Sample output:
===== FINAL STATS =====
Total simulation time: 127.82s
Total passengers served: 20
Total rides completed: 13
Average wait time in ticket queue: 4.81s
Average wait time in ride queue: 12.67s
Average car utilization: 84%
```

In addition, cars followed a round-robin departure order, improving fairness and simplifying scheduling. One area of performance tuning involved ensuring that cars didn't remain idle too long waiting for a full load. By introducing a 10-second timeout on the car loading semaphore, we avoided starvation while keeping overall throughput high.

The main performance bottlenecks were due to passenger exploration delays and car wait timeouts when passenger arrivals were sparse. However, these delays were intentional to reflect a realistic, time-based simulation. The car utilization metric (~84%) indicates that cars were mostly full on departure, satisfying the rubric's efficiency goal.

No deadlocks or livelocks were observed across multiple test runs. All concurrency issues were resolved through careful semaphore guarding and thread sequencing logic.

Conclusion

Completing the Duck Park simulation offered deep insight into the challenges and power of multithreaded programming. By building a system that mimicked the behavior of a real-world amusement park, I learned how difficult it can be to ensure fairness, efficiency, and correctness in a concurrent environment. One of the most valuable lessons came from implementing round-robin queuing for car departures designing a system where cars load and depart in a fixed sequence proved tricky at first, but once implemented properly, it led to more predictable system behavior and fairness across all threads.

Another key learning experience came from using time-based semaphores to manage car departure conditions. Initially, cars would wait indefinitely for passengers, causing deadlocks or inefficient underutilization. Adding a timed semaphore wait allowed each car to depart once full or after waiting 10 seconds, which drastically improved throughput while preserving synchronization guarantees. It also forced me to deeply understand and debug `sem_timedwait()` behavior, as well as to build in robust conditions to prevent premature departures or empty rides.

Overall, this project sharpened my skills in using POSIX threads, semaphores, and shared memory, and gave me confidence in debugging race conditions and deadlocks. Designing, testing, and refining this kind of simulation has prepared me to approach more complex systems programming tasks with discipline and foresight.