CS 415 Operating Systems
Fall 2025

Project #3 Report Collection

Author:

Rayna Patel

raynap

952017511

Report

Introduction

The goal of this project was to develop a Parallel Discrete-Event Simulation (PDES) system called "Duck Park." The project focuses on multi-threaded programming, complex synchronization, and Inter-Process Communication (IPC). The core objective was to simulate the interactions between passengers (who want to explore the amusement park and ride the roller coaster) and roller coaster cars (which load, run, and unload passengers). I implemented this using the C pthread library for concurrency, synchronization primitives (mutexes and condition variables) for thread safety, and a separate process for system monitoring.

Background

This project required an understanding of the Producer-Consumer problem and Readers-Writers concepts. Passengers act as producers (generating ride requests), and Cars act as consumers (processing passengers from the queue). For Part 1, I focused on basic thread creation and establishing a protocol between a single car and passenger. For Part 2, the complexity increased to N passengers and C cars. This required implementing a thread-safe Queue data structure and handling specific constraints: cars waiting for capacity or a timeout (W seconds), and cars unloading in strict FIFO order. For Part 3, I utilized fork() to create a child process that acts as a system monitor. This required IPC using Unix Pipes to transfer the state of the queues from the simulation (parent) to the monitor (child) for real-time display.

Implementation

Part 1 involved setting up the fundamental thread logic. I used pthread_create to make one passenger and one car. The main challenge was ensuring the output logs made logical sense (e.g., a car cannot depart before the passenger says they are boarding). I implemented a double protocol to help with this: the car signals the passenger to board, and the passenger signals back when they are seated. This guaranteed that the "Boarding" log always printed before the "Departed" log.

Part 2 introduced the Ride Queue using a linked list protected by a mutex (m_queue). To handle the "Complete Carload Rule" (all passengers must unboard before the car finishes), I implemented a 4-State protocol inside the Node structure:

1. *Boarded (State 1): Car signals passenger.*
2. *Allowed to Unboard (State 2): Car signals passenger after the ride.*
3. *Unboarded (State 3): Passenger confirms they have left.*
4. *Released (State 4): Car confirms passenger has left and allows the passenger thread to free memory. This prevented "Use-After-Free" race conditions where a passenger might destroy their condition variable while the car was still signaling it. For the FIFO Unloading constraint, I used a "Ticket System." As cars departed, they took a ticket number. Upon return, they waited on a global condition variable until their ticket number matched the next_unload_ticket counter.*

Part 3 added the monitoring system. I created a broadcast_state() function that formats the current contents of the Ticket and Ride queues into a string and writes it to a pipe (pipe_fd). The child process runs a loop reading *from this pipe and printing to the terminal. An important detail was avoiding deadlocks during broadcasting. Since broadcast_state needs to lock the queue mutexes to read data, calling it from inside a Car thread (which already holds the lock) causes a self-deadlock. I resolved this by unlocking the queue mutex, broadcasting, and immediately re-locking it.*

Performance Results and Discussion

I successfully implemented all three parts. The most difficult challenge was a race condition in Part 2 where the simulation would hang after about 10 seconds. This was caused by a lost signal problem, where a car would signal a passenger to unboard before the passenger had entered their wait state. I resolved this by relying on state integers inside while loops rather than relying solely on pthread_cond_signal.

Part 1 Output: The output below shows the correct handshake. The strict ordering of "Boarding" → "Departed" verifies the synchronization.

Part 2 Output: The output below demonstrates the multi-car logic. You can see cars departing partially full due to the timeout (W=1) and strictly adhering to the FIFO unloading order.

Part 3 Output: The screenshot below shows the Monitor in action. The [Monitor] blocks appear every 5 simulated seconds (as per my sampled implementation), accurately listing the IDs of passengers in both the Ticket Queue and Ride Queue.

Memory Safety

I utilized Valgrind to ensure memory safety.

For Part 2, I made sure that every malloc for a passenger node was paired with a free, only after the car explicitly released the passenger (State 4).

For Part 3, I used the flag --trace-children=yes to ensure Valgrind checked the Monitor process as well. The final logs showed zero memory leaks and zero errors

Conclusion

This project highlighted how complex coordinating parallel threads can be. While mutexes protect data, they do not dictate order; properly implemented Condition Variables and State Flags are required to orchestrate complex sequences like the roller coaster ride cycle. I learned that Deadlock Prevention isn't just about lock ordering, but also about ensuring signals are never lost. The protocols I designed were essential for stability. Additionally, implementing the Monitor via IPC reinforced how processes can cooperate to provide a full view of a system state.