CSE 331 – Assignment 12

Mahny Barazandehtar – 20210702004 – Section 4

Abstract

This assignment aims to compare two programming solutions—Multi-Threaded and Multi-Process approaches—for the “Bridge in Repair” problem. The evaluation focuses on execution time of each program for varying bridge capacities and number of cars. The programs are written in C language.

Introduction

The "Bridge in Repair" problem encapsulates a common synchronization challenge in computing, where multiple entities (cars) must access a limited resource (a bridge) without conflict. This problem is particularly significant in operating systems and parallel computing. The task requires that no two cars traveling in opposite directions cross the bridge simultaneously, thereby necessitating an efficient synchronization mechanism. We can achieve that by creating both Multi-Threaded and Multi-Process programs and using Semaphores to manage traffic flow over the bridge.

Semaphore is derived from the Ancient Greek words *sêma* meaning sign/mark/token and *phóros* meaning bearer/carrier, hence meaning Signal Carrier or “Apparatus for Signaling”.

In computing, semaphores serve as crucial synchronization tools used in operating systems to manage access to shared resources and prevent race conditions in concurrent programming.

They come in two types: **counting semaphores**, which can hold any non-negative integer value and are used for resources with multiple instances, and **binary semaphores** (mutexes), which can only be 0 or 1 and are used for single resources.

Semaphores support two primary atomic operations: **wait (P)** and **signal (V)**. The wait operation decreases the semaphore value and may block the calling thread if the value is zero, while the signal operation increases the semaphore value and may unblock a waiting thread.

Proper use of semaphores can significantly improve system performance and resource utilization by synchronizing concurrent threads or processes. However, they require careful handling to avoid complex issues like **deadlocks** and **busy waiting**, which can arise from improper semaphore management.

Implementation

a) Multi-Threaded Program

The program aims to simulate cars crossing a bridge with limited capacity using threads, semaphores, and synchronization primitives.

Each car is represented by a thread which calls the car() function which then calls the Arrive() and Depart() functions.

In Arrive(), the car checks if it can enter the bridge based on the bridge capacity and the direction of other cars. If not, it waits.

Crossing the bridge is simulated with a sleep function.

In Depart(), the car updates the shared variables to reflect its departure and signals that the bridge has more capacity.

The mutex semaphore ensures that only one thread can update the shared variables at a time, preventing race conditions.

The bridge semaphore controls the number of cars on the bridge, ensuring the bridge is not overloaded.

The execution time of the program was measure using the gettimeofday() function.

b) Multi-Process Program

The program aims to simulate cars crossing a bridge with limited capacity using processes, semaphores, and shared memory to manage synchronization and resource sharing.

Each car is represented by a separate child process using fork() and each child calls the Arrive() and Depart() functions.

In Arrive(), cars wait for bridge availability and update the shared state upon arrival.

Crossing the bridge is simulated with a sleep function.

Depart() updates the shared state and signals increased bridge capacity.

The mutex semaphore ensures that only one thread can update the shared variables at a time, preventing race conditions.

The bridge semaphore controls the number of cars on the bridge, ensuring the bridge is not overloaded.

The execution time of the program was measure using the gettimeofday() function.

Results

Table

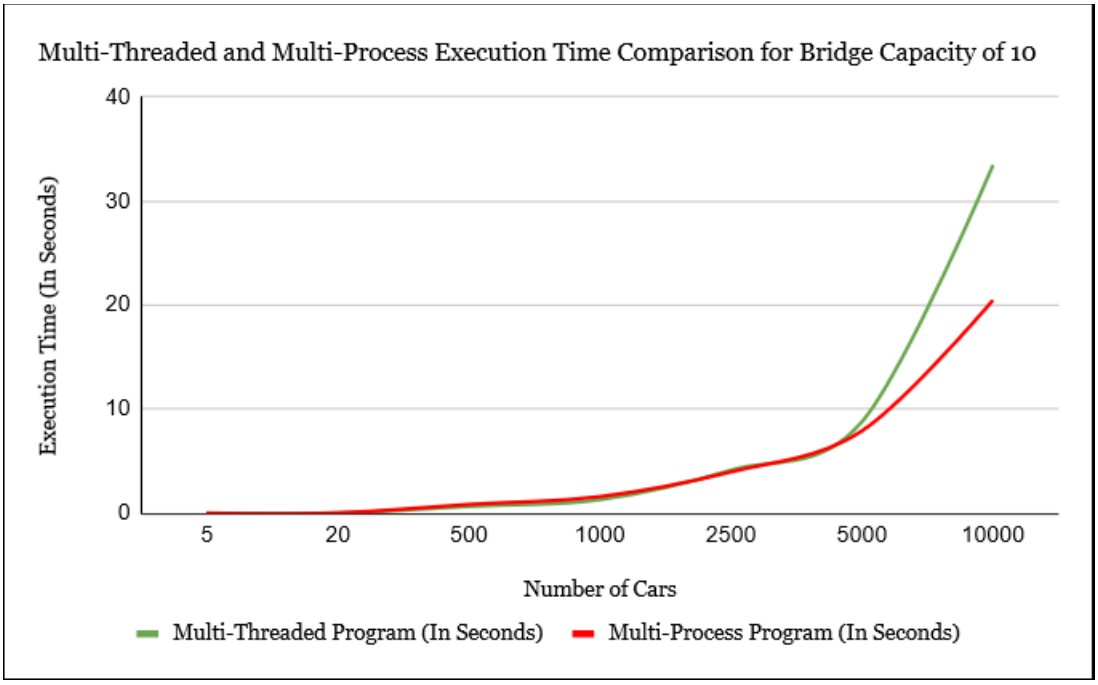
Bridge Capacity	Number of Cars	Multi-Threaded Program (In Seconds – 6 S.D.)	Multi-Process Program (In Seconds – 6 S.D.)
10	**5**	0.006682	0.012275
	20	0.016399	0.040458
	500	0.662305	0.853448
	1000	1.306598	1.600853
	2500	4.146115	3.975773
	5000	8.806087	7.921319
	10000	33.473671	20.490243
15	**5**	0.006906	0.01285
	20	0.010826	0.03158
	500	0.222117	0.74958
	1000	0.524985	1.466815
	2500	1.769522	3.704569
	5000	4.938963	8.605894
	10000	17.94602	19.503692
20	**5**	0.005209	0.015338
	20	0.010733	0.037487
	500	0.180100	0.748013
	1000	0.458669	1.469521
	2500	1.483388	3.84958
	5000	3.802641	8.635125
	10000	13.442869	19.762778

Note:

Due to system limitations, my laptop was unable to handle a sample size ranging from 2 to 300,000 cars and a bridge capacity between 2 and 20. Therefore, I adjusted the parameters to a more manageable range of 5 to 10,000 cars and a consistent capacity of 10 to 20.

Graphs

Multi-Threaded and Multi-Process Execution Time Comparison for Bridge Capacity of 10



Multi-Threaded and Multi-Process Execution Time Comparison for Bridge Capacity of 15

Multi-Threaded and Multi-Process Execution Time Comparison for Bridge Capacity of 20

Conclusion

Based on the collected data, we can see that the Multi-Threaded program consistently executes faster than the Multi-Process program across all scenarios of bridge capacity and car numbers. This suggests that the overhead associated with managing threads is lower than that with managing processes, which typically have higher costs for creation, management, and communication. Making the Multi-Threaded program the more efficient option between the two.

As the number of cars increases, the execution time for both Multi-Threaded and Multi-Process programs increases. However, the rate of increase in execution time is generally steeper for the Multi-Process program. This indicates that threads handle scaling in concurrent operations more efficiently than processes, likely due to better resource sharing and lower communication overhead within the same process memory space.

It was also observed that the Multi-Threaded program had a better CPU utilization because threads of the same process share memory and resources, leading to less context switching and better cache usage.

While the Multi-Process program showed a less efficient CPU usage due to more frequent context switches and the overhead associated with inter-process communication and memory management.