Assignment 2: Monitors and Semaphores

Nolan Tuttle & Mathew Hobson

College of Engineering and Technology, Grand Canyon University

CST-315: Operating Systems Lecture and Lab

Professor Citro

February 16, 2025

Description

The scenario we chose was the Reader-Writer problem. This problem is a common example of multiple threads/processes trying to access a shared resource all at one time. For our example, we used fork() to create child processes, creating multiple concurrent reader and writers. The Reader-Writer problem is when we have a shared memory area which we call the critical section. The critical section has the constraint that it cannot be modified by more than one thread/process and cannot be read while it is being written to. This problem is solved through synchronization mechanisms, namely, in this example, monitors and semaphores. It is imperative that synchronization mechanisms are used in this problem since serious process state issues can occur when concurrent access to a shared resource occurs. For example, if two processes access a shared resource concurrently but are also waiting for each other to finish accessing the resource, this can result in a deadlock case where they are both stuck in the blocked state waiting for one another. In this case, it's more about maintaining data integrity, if a writer writes to data at the same time as a reader reads it, the read might be inaccurate which can be very bad if that data being read affects other parts of the program. We have written two example programs, "semaphores.c" and "monitors.c", both using strictly monitors or semaphores respectively to simulate solving the Reader-Writer problem. Since we created our programs in the C programming language, we did our best to create a monitor construct, which does not exist as a construct in this language to be recognized by the compiler so the compiler can enforce mutual exclusion.

Pros and Cons

Feature	Monitors (Mutex & Conditions)	Semaphores
Thread Safety	Built-in mutex handling	Manual semaphore handling required
Blocking Mechanism	Uses wait() and notify()	Uses integer-based locks (wait/signal)
Ease of Use	Easier to read & maintain	Requires careful manual control
Concurrency Control	Readers can read together; Writers can lock access	Same behavior but uses simple counters
Encapsulation	Encapsulates all logic inside monitor functions	Uses explicit semaphore operations

Recommendation

Between using monitors or semaphores as the synchronization technique for the Reader-Writer problem, the best option is to use monitors. Monitors are usually easier and simpler to read, have a low risk of deadlocks, and have very good encapsulation, and not to mention they have the compiler enforce mutual exclusion, which is much safer than semaphores which have the programmer do it. Obviously, if using a programming language that does not support monitors, semaphores are the better choice. Semaphores are a very low-level data structure compared to monitors and give the user much more control over their behavior, leaving it up to the user to correctly enforce mutual exclusion. For learning applications, semaphores are a better choice because of the level of control, with simple changes to a program, intentional deadlocks and bad shared memory management can occur. However, in most cases and especially in the case of our Reader-Writer problem where concurrent access without synchronization can be disastrous, monitors are the best choice because of the safety they offer.

References

Stallings, W. (2019). *Operating Systems: Internals and Design Principles* (9th ed.). Pearson.

Tuttle, N., Hobson, M., (2025) *CST315* GitHub. https://github.com/nolantuttle/CST315.git