Understanding Semaphores in C with POSIX Threads

Ahmed Métwalli

December 25, 2024

Objective

This document provides an in-depth understanding of semaphores in C using POSIX Threads. By the end of this tutorial, students will:

- Understand the concept of semaphores and their use in thread synchronization.
- Learn how to handle critical sections to avoid race conditions.
- Explore three code snippets showcasing semaphore usage and related problems.

Introduction to Semaphores

A semaphore is a synchronization primitive used to control access to shared resources in a concurrent system such as multithreaded programs. It is often used to solve problems like race conditions, where multiple threads attempt to access and modify shared data simultaneously. A semaphore acts as a counter that tracks the availability of a resource.

Key Characteristics

- Binary Semaphore: Also known as a mutex, allows only one thread to access the critical section at a time.
- Counting Semaphore: Allows multiple threads to access a limited number of resources concurrently.
- Operations:
  - **sem_wait:** Decrements the semaphore value. If the value is zero, the thread is blocked until the semaphore becomes available.
  - **sem_post:** Increments the semaphore value, signaling that a resource is released.
- Thread Safety: Ensures that shared resources are accessed in a controlled and synchronized manner.

Problems Solved by Semaphores

- Prevents race conditions by ensuring mutual exclusion in critical sections.
- Avoids busy waiting, improving CPU efficiency.
- Synchronizes thread execution, allowing threads to wait for a specific condition to be satisfied.

Key Variable Explanations

- **tickets:** A shared resource representing the total number of tickets available for sale. Each thread decrements this value when it successfully sells a ticket.
- **turn:** A variable used in turn-based synchronization to determine which thread is allowed to enter the critical section. Threads alternate turns by updating this variable.
- **kill:** A control flag used in Snippet 1 to signal termination of threads. When set to 1, it forces threads to exit their loops.
- **sem:** A semaphore used in Snippet 3 to ensure mutual exclusion, allowing only one thread to enter the critical section at a time.
- **ids:** An array storing unique identifiers for threads in Snippet 3, primarily used for printing thread-specific messages.

Steps to Implement Code Snippet 1: Turn-Based Synchronization

Algorithm 1 Turn-Based Synchronization Algorithm

```
Initialize shared variables tickets, turn, and kill.
Create two threads thread1 and thread2.
while tickets > 0 do
    if turn == 1 (for thread1) or turn == 2 (for thread2) then
        Enter critical section.
        Decrement tickets if tickets ¿ 0.
        Print the remaining tickets.
        Pass the turn to the other thread.
    end if
end while
Wait for threads to finish execution.
Print the final number of tickets.
```

Expected Outcome

- Each thread alternates access to the critical section.
- **Drawback:** If a thread fails, the other is permanently blocked.

Steps to Implement Code Snippet 2: Race Condition Example

Algorithm 2 Race Condition Algorithm

```
Initialize shared variable tickets.
Create two threads thread1 and thread2.
while tickets > 0 do
    Each thread independently accesses and decrements tickets.
    Print the remaining tickets.
end while
Wait for threads to finish execution.
Print the final number of tickets.
```

Expected Outcome

- Threads compete for access to `tickets`.
- Unpredictable behavior due to race conditions.
- **Drawback:** Shared resource integrity is compromised.

Steps to Implement Code Snippet 3: Semaphore Synchronization

Algorithm 3 Semaphore Synchronization Algorithm

```
Initialize shared variable tickets and semaphore sem.
Set the semaphore sem to 1 (binary semaphore).
Create multiple threads.
while tickets > 0 do
    Call sem_wait(&sem) to enter the critical section.
    if tickets > 0 then
        Decrement tickets.
        Print the remaining tickets.
    end if
    Call sem_post(&sem) to exit the critical section.
end while
Wait for threads to finish execution.
Destroy the semaphore sem.
Print the final number of tickets.
```

Expected Outcome

- Threads safely access the critical section without interference.

- Proper synchronization ensures data integrity.
- **Advantage:** Avoids busy waiting and race conditions.

Comparison of Snippets

- **Snippet 1:** Demonstrates busy waiting and manual turn-switching but is prone to deadlocks and inefficiency.
- **Snippet 2:** Highlights race conditions and the need for synchronization mechanisms.
- **Snippet 3:** Implements a semaphore to achieve proper synchronization and thread safety.

Comparison Table

Aspect	**Snippet 1**	**Snippet 2**	**Snippet 3**
Synchronization Mechanism	Turn-based	None	Semaphore
Mutual Exclusion	Partial	None	Complete
Risk of Deadlock	High	N/A	Low
Efficiency	Low	Medium	High
Thread Safety	No	No	Yes

Discussion

Semaphores provide a robust mechanism for synchronizing threads and ensuring safe access to shared resources. By understanding the issues in Snippets 1 and 2, students can appreciate the value of semaphores in resolving synchronization problems.