Report for the Programming Assignment 3

Hamit Efe Çınar - 30925

Program Description

The Rideshare program is designed to simulate fans of two teams (A and B) finding spots in cars for a shared ride. Correct synchronization is looked out for to ensure that threads representing fans operate correctly without conflicts or deadlocks.

Synchronization Primitives Used

- Mutex Lock (**pthread_mutex_t**)
  The mutex lock is a critical component for protecting shared resources (**countA**, **countB**, **carID**). It is used in the **car_finder** function to ensure that only one thread at a time can modify these shared variables, providing mutual exclusion.
- Semaphores (**sem_t**)
  semA and **semB** are utilized to manage the access of fans from teams A and B, respectively. These semaphores are needed for controlling the flow of threads and ensuring that fans find spots in the cars according to the specified rules.
- Barriers (**pthread_barrier_t**)
  group_barrier and **captain_barrier** are used to synchronize threads into groups and to select a captain for each group. These barriers ensure that the threads proceed in a coordinated manner.

Implementation of Synchronization Primitives

Linux Zempahores-like pthread semaphores (got from the library 'semaphore.h') are used in this program to ensure synchronization. In addition to that, barriers and mutexes used in the program are from the pthread library. Synchronization primitives mentioned above are used in a suitable way that follows the instructions given in the lectures and the document. For instance, the mutex lock is acquired before any modification to shared counters and released appropriately to avoid deadlocks. Semaphores are initialized with zero counts (intra-process semaphores, semaphores that wait to be nonnegative) and are posted when a condition is met. The barriers are used to hold threads at specific points until all required threads reach that point.

Correctness Criteria

The implementation ensures:

- **Mutual Exclusion:** Mutex lock prevents simultaneous modification of shared resources.
- **Deadlock Avoidance:** The design avoids deadlock by ensuring that locks are acquired and released in a consistent order and that no thread holds a lock while waiting indefinitely.
- **Fairness and No Starvation:** Semaphore and barrier usage ensures that each fan eventually finds a spot in a car, and no fan is indefinitely blocked from progressing.

General Program Flow

Program Initialization

The program starts by setting up necessary synchronization mechanisms, including a mutex (**mutex**), two semaphores (**semA** and **semB**), and two barriers (**group_barrier** and **captain_barrier**). These are essential for managing access to shared resources and coordinating the threads representing the fans.

Creation of Threads

Once the initialization is complete, the program proceeds to create multiple threads, with each thread representing an individual fan. The number of threads (or fans) is determined based on the input arguments provided to the program, indicating how many fans are there from each team. The inputs' validity (each group size being an even number, and total number of supporters being a multiple of four) is checked and program is terminated if the validation fails. The created threads are crucial to the program's functionality, as they simulate the behavior of the fans seeking to find a spot in a car. Each thread is assigned to execute the **car_finder** function, which contains the main logic for the fans to find their respective places in the cars.

Thread Function(car_finder) Flow

The function begins by telling that it is looking for a car, and if possible, acquires the mutex lock. Once the lock is acquired, the fan's team counter (**countA** or **countB**) is incremented, depending on whether the fan is from team A or B.

Next, the function checks specific conditions to determine if a complete group can be formed. These conditions are based on the number of fans from each team waiting for a ride (2-2, 4-0 or 0-4 fans of A and B respectively). If the conditions are satisfied, the function posts (signals) the semaphores (**semA** or **semB**) for the appropriate team. This action allows waiting threads of the corresponding team to proceed. If the conditions are not met, the function releases the mutex lock to allow other threads to execute.

After this, the fan thread waits on its team's semaphore. It causes the thread to block until the semaphore is posted, indicating that there's an available spot in a car for the fan. Once the semaphore wait is over, the thread then waits at a barrier (**group_barrier**). This barrier ensures that the fans are grouped together before proceeding. All threads in a group (each consisting of 4 fans) must reach this barrier before any of them can continue.

After passing the **group_barrier**, the fan will tell that he/she has found a spot in the car. The next synchronization point is the **captain_barrier**. The threads again wait here, ensuring that the group is fully ready before proceeding. This barrier plays a crucial role in designating a captain for the group.

Finally, the group members increase the captain count as they continue from the second barrier, and the last one to increment the captain count among the group is selected as the captain, who say its ID, team and its cars’ ID. The captain increments the carID before he/she tells the carID that he/she is the captain. This is to distinguish different cars/groups. Once these actions are complete, the mutex lock is released, signaling the end of the critical section and allowing other threads to modify shared resources.

Joining Threads and Cleanup

After all threads have executed the **car_finder** function, the main thread waits for each of them to complete their execution. This is achieved through the use of the **pthread_join** function for each fan thread. Ensuring that all threads have completed their tasks is important for the program's consistency and to prevent any unwanted termination of the program while threads are still running.

Once all threads have completed their tasks and have been successfully joined, the program enters the cleanup phase. This phase is dedicated to the destruction of all the synchronization primitives that were initialized at the beginning. Destroying the mutex, semaphores, and barriers is a necessary to avoid any potential memory leaks.

Hamit Efe Çınar