CS307 PA3 REPORT

Batuhan Güzelyurt 31003

Synchronization Primitives Used

The synchronization primitives utilized in this implementation are:

- Semaphores (sem_t):
    - mutex: Acts as a mutual exclusion lock to protect shared variables.
    - tourEnd: Synchronizes the end of the tour between the tour guide and visitors.
    - arrival: Controls the arrival of new visitors when a tour is in progress.
- Mutex (pthread_mutex_t):
    - printMutex: Ensures atomicity of print statements to prevent interleaving outputs.

All synchronization primitives are standard and provided by the POSIX threads (pthreads) library. No custom implementations were necessary.

Implementation Details

Shared Variables

- groupSize: Number of visitors required to start a tour (excluding the guide).
- tourGuidePresent: Indicates if a tour guide is required (1) or not (0).
- totalVisitorsNeeded: Total number of individuals needed to start a tour (groupSize + tourGuidePresent).
- visitorsWaiting: Number of visitors currently waiting to start a tour.
- visitorsLeaving: Number of visitors who have left the tour.
- tourStarted: Boolean flag indicating whether a tour has started.
- guideThreadID: Thread ID of the tour guide (if applicable).

Synchronization Mechanisms

Mutual Exclusion with mutex Semaphore

The mutex semaphore ensures mutual exclusion when accessing shared variables. It is initialized to 1 (unlocked) and decremented (sem_wait) before entering critical sections, and incremented (sem_post) upon exiting.

Controlling Arrivals with arrival Semaphore

The arrival semaphore regulates the arrival of new visitors. Initially set to 1, it allows visitors to enter. When a tour starts, it is decremented to 0 to block new arrivals until the tour concludes, ensuring that no additional visitors interfere with the current tour.

Synchronizing Tour End with tourEnd Semaphore

The tourEnd semaphore facilitates synchronization at the end of a tour. If a tour guide is present, visitors wait on this semaphore after the tour until the guide signals that the tour has ended. This mechanism ensures that visitors do not leave prematurely.

Preventing Output Interleaving with printMutex

The printMutex mutex ensures that print statements from different threads do not interleave, providing coherent and readable output.

Flow of arrive and leave Methods

arrive Method Pseudocode:

```
function arrive()
  tid = current_thread_id()
  lock(printMutex)
  print("Thread ID: tid | Status: Arrived at the location.")
  unlock(printMutex)

  wait(arrival)  // Check if arrivals are allowed
  signal(arrival)

  wait(mutex)
  visitorsWaiting += 1
  currentVisitors = visitorsWaiting
```

```
  if not tourStarted and visitorsWaiting == totalVisitorsNeeded then
    if tourGuidePresent == 1 and guideThreadID == 0 then
      guideThreadID = tid

    lock(printMutex)
    print("Thread ID: tid | Status: There are enough visitors, the tour is starting.")
    unlock(printMutex)

    tourStarted = true
    wait(arrival)  // Block new arrivals
    signal(mutex)
  else
    signal(mutex)
    lock(printMutex)
    print("Thread ID: tid | Status: Only currentVisitors visitors inside, starting solo shots.")
    unlock(printMutex)
    // Proceed without waiting
  end if
end function
```

leave Method Pseudocode:

```
function leave()
  tid = current_thread_id()

  wait(mutex)
  if not tourStarted then
    visitorsWaiting -= 1
    signal(mutex)
```

```
    lock(printMutex)
    print("Thread ID: tid | Status: My camera ran out of memory while waiting, I am leaving.")
    unlock(printMutex)
  else
    signal(mutex)
    if tourGuidePresent == 1 and tid == guideThreadID then
      lock(printMutex)
      print("Thread ID: tid | Status: Tour guide speaking, the tour is over.")
      unlock(printMutex)
      for i from 1 to totalVisitorsNeeded - 1 do
        signal(tourEnd)
      end for
    else
      if tourGuidePresent == 1 then
        wait(tourEnd)
      end if
      lock(printMutex)
      print("Thread ID: tid | Status: I am a visitor and I am leaving.")
      unlock(printMutex)
    end if

    wait(mutex)
    visitorsLeaving += 1
    if visitorsLeaving == totalVisitorsNeeded then
      lock(printMutex)
      print("Thread ID: tid | Status: All visitors have left, the new visitors can come.")
      unlock(printMutex)
```

```
        visitorsWaiting = 0
        visitorsLeaving = 0
        tourStarted = false
        guideThreadID = 0
        signal(arrival)  // Allow new arrivals
      end if
      signal(mutex)
    end if
end function
```

Discussion of Synchronization Mechanisms

Choice and Implementation

Semaphores

Semaphores are chosen for their ability to control access to shared resources and to synchronize threads without busy-waiting. In this implementation:

- mutex: Serves as a binary semaphore (similar to a mutex lock) to protect critical sections where shared variables are accessed or modified.
- arrival: Controls the flow of new visitors. By decrementing this semaphore to zero, new arrivals are blocked when a tour starts.
- tourEnd: Allows the tour guide to signal the end of the tour to the visitors. Visitors wait on this semaphore if a guide is present.

Mutexes

- printMutex: Used to serialize access to output streams, preventing interleaving of print statements from different threads.

Adaptation for the Problem

- The arrival semaphore is adapted to act as a gatekeeper for new visitors, ensuring that no new visitors can enter once a tour has started.
- The tourEnd semaphore is used in a counting manner, where the tour guide signals it multiple times to release all waiting visitors.

- The mutex semaphore is used instead of a standard mutex to adhere to the requirement of using semaphores and barriers for synchronization.

Mutual Exclusion

By using the mutex semaphore to protect shared variables, race conditions are prevented. Only one thread can access or modify shared variables at a time, ensuring data consistency.

No Deadlocks

The order of semaphore operations and careful design of critical sections prevent deadlocks. Threads acquire and release semaphores in a consistent order, and no circular waiting occurs.

Liveness and Progress

- Visitors Proceeding Appropriately: Visitors who cannot join a tour proceed to take solo shots and eventually leave, ensuring they do not wait indefinitely.
- Tours Starting Correctly: Tours start only when the exact number of required visitors (and guide, if necessary) are present.
- Visitors Leaving After Tour: Visitors wait for the tour guide to announce the end of the tour (if a guide is present) before leaving, ensuring proper synchronization.

Absence of Busy-Waiting

Threads block on semaphores rather than spinning in loops, making efficient use of CPU resources and adhering to the requirement of avoiding busy-waiting.

Explanation of Thread Methods

arrive Method

The arrive method handles visitor arrival and determines whether a tour can start. Key steps include:

1. Announcement: The visitor prints a message indicating arrival.
2. Checking Arrivals: The visitor checks if arrivals are currently allowed by attempting to decrement the arrival semaphore.
3. Critical Section Entry: The visitor enters a critical section protected by the mutex semaphore.
4. Visitor Counting: The visitor increments visitorsWaiting.

5. Tour Start Decision:
    - If the required number of visitors is present, the tour starts:
        - The visitor assigns themselves as the guide if necessary.
        - A tour starting message is printed.
        - The tourStarted flag is set to true.
        - New arrivals are blocked by decrementing arrival to zero.
    - If not enough visitors are present, the visitor proceeds to take solo shots without waiting, printing an appropriate message.
6. Critical Section Exit: The mutex semaphore is incremented to allow other threads to enter.

leave Method

The leave method manages visitor and tour guide departure. Key steps include:

1. Critical Section Entry: The visitor acquires the mutex semaphore to check the tour state.
2. Departure Decision:
    - If the tour hasn't started, the visitor decrements visitorsWaiting and leaves after printing a message.
    - If the tour has started:
        - If the visitor is the tour guide, they announce the end of the tour and signal the tourEnd semaphore to release waiting visitors.
        - If the visitor is not the guide, they wait on the tourEnd semaphore if a guide is present.
        - After the tour ends, the visitor leaves, printing a message.
3. Visitor Counting: The visitor increments visitorsLeaving.
4. State Reset:
    - If all visitors have left, the last visitor resets shared variables and increments the arrival semaphore to allow new visitors.
5. Critical Section Exit: The mutex semaphore is incremented.

Synchronization Mechanisms Used

- Semaphores: Used extensively to control access and synchronize actions between threads.
- Critical Sections: Ensured by the mutex semaphore to prevent race conditions.
- Conditional Waiting: Visitors wait on the tourEnd semaphore only if a tour guide is present, preventing unnecessary blocking when no guide is involved.

Justification of Correctness

- Proper Sequencing: The use of semaphores ensures that events occur in the correct order (e.g., visitors do not leave before the tour guide announces the end).
- Data Integrity: Shared variables are modified only within protected critical sections, maintaining consistency.
- Avoidance of Starvation: All visitors eventually proceed (either by joining a tour or leaving after solo shots), ensuring no thread is indefinitely blocked.