CSE 511 Project 1: Synchronization with Serializers

GitHub Repository

Muni Bhavana Konidala(mzk6126), Naga Sri Hita Veleti(nkv5154)

October 2024

1 Design of your Synchronizer - Serializer Library - Layer A

The serializer is designed to control and synchronize multithreaded access using queues and crowds. It ensures orderly and safe execution of threads by managing their entry and exit in a controlled manner. The serializer designed manages an array of queues (Qs) and crowds (Cs), along with a mutex lock and Control Variables for synchronization. Each queue node of the queue has a condition, a pointer to the next queue node, and its own condition variable. Each crowd node tracks the number of threads in the crowd via a count variable. We have successfully created and destroyed the serializer, along with its associated queues and crowds as needed. The implementation provides functions for creating and destroying the serializer, queues, and crowds, as well as controlling how threads interact with these components. By using condition variables and mutex locks, the serializer manages the proper sequence of thread execution, allowing both serialized (via queues) and group (via crowds) access.

2 Functionality

Gaining possession of the serializer occurs in three scenarios: when entering the serializer, during dequeuing, and when leaving the crowd. Releasing possession of the serializer happens after enqueuing, after joining a crowd, and upon exiting the serializer.

3 Detailed Breakdown

3.1 Gain Possession:

Enter Serializer: A mutex lock is used to ensure that only one thread at a time can gain possession of the serializer.
Dequeue: Once a thread has entered the serializer, it enqueues and waits on its condition variable. If the condition evaluates to true, the thread dequeues; otherwise, it waits on the condition variable.
Leave Crowd: After the thread's task is completed, it regains possession of the serializer and decreases the crowd's count.

3.2 Possession Release:

After Enqueue: If the condition is false, a signal is sent to release possession, and the mutex lock is unlocked.
Join Crowd: When a thread joins a crowd, the crowd counter is incremented to enable parallelism. The possession of the serializer is then released, and a signal is sent to unlock the mutex.
Exit Serializer: After the process completes all its operations, it sends a signal to release possession of the serializer before exiting.
This structure ensures proper synchronization and parallelism while managing queue and crowd operations within the serializer.

4 Why used CV on Each Node of the Queue?

We first used a conditional variable (CV) on top of the serializer,but for that we had to broadcast to wake up the right node which turned out to be inefficient. Later we used a conditional variable on top of the queue but a challenge arose during the release of possession because it can be claimed by any of the three processes: *enter_serializer*, *dequeue*, or *leave_crowd*. Even if the condition for the waiting process on the CV after enqueuing is met and that process is at the head of the queue, possession might be taken by one of the other two processes before the head can do so. As a result, the head process could be pushed to the back of the queue maintained by the operating system for the CV. This creates a scenario where, when the next signal is sent, the process that gets woken up by the CV may not be the head of the queue anymore. This can lead to an infinite loop where the head never receives the signal to wake up. To prevent this, instead of relying on a single CV per queue, each node in the queue is assigned its own CV. This way, the signal is sent directly to the specific node that needs to wake up, ensuring that the correct process can claim possession without being displaced.

5 Problem Implemented Using Serializer

- **Monkey Crossing Problem**
  The serializer is used to implement the Monkey Crossing problem, which involves managing eastbound and westbound threads that must cross a rope without causing collisions. The following are the key conditions for each thread routine:

  - **Eastbound Thread (`east_cond`)** This condition ensures that an eastbound thread can proceed only if: The number of eastbound threads currently on the rope is less than the rope's capacity (`capacity`), there are no westbound threads currently crossing, and no westbound threads are waiting. This prevents collisions and ensures fairness by allowing eastbound threads only when the rope is free and no westbound monkeys are waiting.
  - **Westbound Thread (`west_cond`)** This condition allows a westbound thread to proceed if: The number of westbound threads on the rope is less than the rope's capacity, there are no eastbound threads on the rope, and no eastbound threads are waiting. This ensures that westbound threads can only cross when the rope is free from eastbound threads, and it prevents starvation by giving priority to waiting threads from the opposite direction if they exist.

  To ensure there's no starvation and first come first serve is served we just used one queue to add both east and west bound monkeys. These conditions are used to check the condition to dequeue the process so that it allows the thread to join its respective crowds once it is safe to do so.

- **Child Care Problem**
  The serializer is also used to implement the Child Care problem, which involves managing the arrival and departure of caregivers and children at a child-care center. The following are the key conditions for each thread routine:

  - **Caregiver Arrive (`caregiver_arrive_cond`)** This condition allows a caregiver to proceed without any restrictions, meaning that caregivers can arrive at any time.
  - **Child Depart (`child_depart_cond`)** This condition ensures that a child can only depart if there are children ready to depart (`CHILD_READY_DEPART > 0`), and there's atleast one Child who's in the process of departing(`CHILD_READY_DEPARTING > 0`), so that we ensure the serialization of Child arrived before depart. This guarantees that children only leave when it is safe to do so and when the proper conditions are met.
  - **Caregiver Depart (`caregiver_depart_cond`)** This condition allows a caregiver to depart if there are no children currently at the center (`CHILD_ARRIVED == 0`), no children are ready to depart (`CHILD_READY_DEPART == 0`), no children are in the process of departing (`CHILD_READY_DEPARTING == 0`), or there is more than one caregiver present (`CAREGIVER_ARRIVED > 1`) and There should be atleast one Caregiver arrived (`CAREGIVER_ARRIVED > 1`). This ensures that at least one caregiver remains at the center if children are present and also the serialization of Caregiver arrived before it can depart.
  - **Child Arrive (`child_arrive_cond`)** This condition allows a child to arrive only if: there is at least one caregiver already present (`CAREGIVER_ARRIVED > 0`). This ensures that children are only allowed to enter the center if there are caregivers available to supervise them.

  These conditions are used to control when a thread can be added to the crowd for arriving or departing once it is safe to do so. The conditions help maintain a safe environment in the child-care center, ensuring that the proper caregiver-to-child ratio is maintained at all times.

6 Distribution of Workload

The design and implementation for serializer is done by Muni Bhavana Konidala and Naga Sri Hita Veleti. Each problem implementation and testing has been taken by each of us.