OS LAB ASSIGNMENT-5

Thread Synchronization using Semaphores

Design Document

Group No: 17

Pranav Nyati (20CS30037)
Prerit Paliwal (20CS10046)
Vibhu (20CS10072)
Ashwani Kumar Kamal (20CS10011)

Basic Data Structures and Constructs:

1.) **Struct room:** an instance of it represents a hotel room with the following attributes:

- **room_no:** the room no is from 0 to N-1 (where N = total no of rooms)
- **count:** count is initialized to 0, and as a guest gets a room, the counter increases by one and can be increased to a max of 2 (before the rooms are cleaned, upon which the counter is again set to 0)
- **curr_occupied:** can be 0 or 1 based on whether the room is currently vacant or occupied by a guest.
- **guest_id:** the id of the guest thread (from 0 to Y-1) where Y = no of guests presently staying in the room

- **clean_time:** The time it will take to clean the room after two guests have occupied it (it is the sum of the actual initial sleep time of the two guests without considering the possibility of eviction)

2.)**Struct guest:** an instance of it represents a guest in the hotel with the following attributes
   - **guest_id:** represent the id (basically the id of the guest thread) of the guest from 0 to Y-1
   - **priority:** represents the priority of the guest in terms of getting a room (in case no room is vacant, and a lower priority guest has occupied a room, the higher priority guest can evict given the initial count is 1)

Apart from these structures for room and guest, we have considered four queues for managing the rooms as guests occupy them and leave, and then when they are cleaned and made available again.

3.) **Queue <room *> empty_q:** This queue holds the rooms just after they are cleaned (or initially when the program starts) and contains the rooms which have count = 0 (i.e., after being cleaned, no guest has occupied them till now).
   The queue is guarded by the binary semaphore q1_sem. When a guest thread wakes up from its initial random sleep , it first searches if a room is available in the empty_q (i.e., a room which has not been occupied by any guest till now). If such a room is available, the guest thread pops it and increases its count by 1 and stays in it (till completion of stay or eviction).

4.) **Priority_queue <room *, priority(int) > one_occ_q:** This queue is a priority queue (min Heap), and sorts rooms according to increasing order of the priority of the guests staying in them. This is the 2nd queue in the control flow for managing the rooms.
   If a room is popped out from empty_q, and is occupied by a guest, before the guest starts its sleep in that room , the guest thread pushes the room in this

queue. This queue stores all the rooms which are currently being occupied by their first occupying guest.

This queue is guarded by the binary semaphore q2_sem.

Now, when a guest is in this queue, two possibilities can happen:

(1) The guest completes its stay in the room: In this case, the guest is not evicted from the room before completion of its stay by any high priority guest, and thus for this room, its first occupant has completely stayed. In this case this room is searched through and removed from this **one_occ_q** and pushed into **one_free_q** (the 3rd queue) after (increasing the count to 2).

(2) The guest is evicted by a high priority guest (when the guest doesn't find any room free in either **empty_q** or **one_free_q**), in that case, the high priority guest will by itself remove this guest from **one_occ_q) ,** and itself occupy the room. In that case, the high priority guest thread will then push this room to **two_occ_q** (4th queue) , as the count for that room has become two.

5.) Queue <room *> one_free_q: This is the 3rd queue in the room management scheme, and holds the rooms which have been occupied by 1 guest, and that guest was able to complete its stay. So it holds the rooms which have a count of 1, and can have another guest to stay in them before cleaning of rooms occurs.

This queue is guarded by the binary semaphore q3_sem.

Any guest thread while searching for a room to stay in, first checks if there is a room (with count = 0) available in the **empty_q**, if yes, it occupies it, and if not, it then checks the **one_free_q** to see if there is a room available (with count = 1). Now, if even **one_free_q** is empty, so it means all rooms are currently occupied by some otbher guests. Then, it checks in the **one_occ_q**, if it can find a room which is currently occupied by a low-priority guest whom this guest can evict, if so, it evicts that guest and occupies it, else it goes for another round of random sleep before again searching for the rooms in the same sequence.

6.) Queue <room *> two_occ_q: This is the 4th queue in the room management scheme, which contains all the rooms which have achieved a count of 2. (Note that as soon as the 2nd guest occupies a room, the count of room is increased to 2, an the room is pushed to the two_occ_q before the 2nd guest begins its stay (sleep) in that room.
This queue is guarded by the binary semaphore q4_sem.
As soon as the last room which still has a count of 1, is occupied by the 2nd guest, the count for that room is increased to 2. The guest that just occupied it is spontaneously evicted, and all other guests that were in middle of their stay in the room are also evicted, so that the cleaners can be activated to clean the room, meanwhile, the guests keep waiting (blocking wait).
Once the cleaners clean all the rooms, all the rooms are made available at one go by putting them back into the empty_q, and then the waiting guests are signalled to wake them up (from blocking wait), upon which they again do a random sleep and the above cycle repeats again.

Semaphores and Mutexes used:

1.) Semaphores:

- **q1_sem, q2_sem, q3_sem, q4_sem (sem_t):** These are the 4 binary semaphores (initialized to 1) to provide mutual exclusion to the 4 queues.

- **write_sem (sem_t):** This is a binary semaphore (initialized to 1) to provide mutual exclusion to the different guest and cleaner threads when they write to the terminal (STDOUT).

- **room_sem (sem_t *):** This is an array of N (where N = no of rooms) signalling semaphores (initialized to 0), used by a higher priority guest thread to signal to a lower priority guest thread that it has been evicted by that particular high priority guest thread (Since the no of guests Y

> no of rooms N), it is better to use a signalling semaphore for a room rather than for each guest for this signalling purpose.
Mechanism by which such signalling is achieved: The lower priority guest does a **sem_timedwait** on the semaphore of the room in which it is staying with **a timeout equal to its stay (sleep) time** in the room. If the timeout occurs before a high priority guest posts to the low priority guest, it means that the low priority guest was able to complete its stay (sleep) in the room without being evicted. However, if a high priority guest evicts a low priority guest before the low priority guest has completed its stay, the high priority guest will do a sem_post on the semaphore corresponding to that room. In that case, the low priority guest thread waiting on it will come out of the wait before timeout occurs and thus will be evicted.
Note: Only those rooms will be signalled which have a guest staying in them and the guest is in the middle of its sleep (stay)

Therefore, **Total no of semaphores = 4 (for the queues) + 1 (for write to STDOUT) + N (for signalling of N rooms) = 5 + N**

2.) Mutex locks:

- **signal_mutex:** This mutex is used to broadcast to all the cleaner threads to start cleaning the rooms as soon as the each room has been occupied by 2 guests. For this, the cleaner threads wait on this **signal_mutex**, using **pthread_cond wait** on **signal_cond,** while the last guest thread signals them using **pthread_cond_broadcast.**

- **all_clean_mutex:** This mutex is used to broadcast to other waiting (blocking wait) cleaner threads, when the last cleaner thread has finished cleaning the last room to be cleaned. This is important that the remaining cleaners do a busy wait while the two_occ_q is empty(no new room to be cleaned) while some other cleaner thread is still cleaning the room. This will prevent the other threads from uselessly spinning in the while loop of the cleaner threads's code, as the queue will remain empty until the next cycle of cleaning. This

broadcasting is achieved using **pthread_cond_broadcast** on **all_clean_cond** by the last cleaning thread, while the other cleaning threads do a **pthread_cond_wait** on it.

- **signal_guests_mutex:** This mutex is used to broadcast to the waiting guests by the cleaner thread (that cleans the last room) that all rooms are available (in the empty_q) to be occupied again. The guests do a **pthread_cond_wait** on **signal_guests_cond** , and start occupying the rooms from empty_q, once the cleaner thread does a **pthread_cond_broadcast** on the same conditional variable.

- **temp_two_occ_q_mutex:** This is used to signal all the guest threads that are currently staying in a room as the 2nd occupant in that room, as soon as the last room which had a count of 1 gets a guest and thus all rooms get occupied twice. Thus, the other rooms, in which the 2nd guest was in the middle of its stay (sleep) are evicted by signalling the staying guests . For this, a **pthread_cond_broadcast** is done on **temp_two_occ_q_cond**, while the staying guests do a **pthread_cond_timedwait** on **temp_two_occ_q_cond** with a **timeout equal to their (stay) sleep period**. If the timeout expires before the broadcast is received, it means the 2nd guest was able to complete its stay and hence does not need to be evicted, else if the broadcast is received before the timeout expires, the guest gets evicted from room without completing the stay.

Therefore, **Total no of mutex locks = 4**
