OS LAB Assignment - 5

Design Report

Group Members

1. Pranav Mehrotra (20CS10085)
2. Saransh Sharma (20CS30065)
3. Anand Manojkumar Parikh (20CS10007)
4. Soni Aditya Bharatbhai (20CS10060)

Report

Class:

a. **Room:**

Room is the class of individual rooms in the system and is only used in the std::set custom data structure. It has the attributes of *room_id*, which will be unique to each room and is initialised in the main thread; apart from that, it has *guest_id* of the guest currently occupying the room, with *start_time*(when the guest occupied the room) and *stay_time*(nominal stay time of the guest, if not evicted by some other guest) of the guest. It also has the *tot_duration*(Total duration, the total time the room has been occupied since last cleaning) and the *occupancy* count(which can be 0,1,2). We have defined the necessary constructors and also overloaded the '<<' operator to help in printing the actions.

Data Structure:

a. **set<pair<int,Room>,cmp> rooms:**

This is the main data structure used in the whole program, it is used to maintain all the records about each room and to allot rooms when new guests arrive and finally the cleaner threads clean the rooms and reset this data structure.

We have defined a custom '*cmp*' struct, which has an overloaded '()' operator, which will be used by the set to find distinct entries and sort the entries according to our requirements.

Each entry is a pair of *(int, Room)*; where 'int' denotes the current priority of the room and the Room is an instance of the class defined above.

The custom comparator will first sort on the basis of the priority of the rooms, which can vary between 0 or guest priority or '*INF*' if occupancy is 2. If the rooms have the same priority, it will sort on the basis of '*occupancy*', and finally, if occupancy is also the same then it will just sort on '*room_id*'.

Usage of Semaphores:

A total of 4 semaphores have been used in this design, as described below-

1) **hotel_open:**
    - Binary semaphore, initial value = 1.
    - Describes whether the hotel is currently open or not.
    - Enforces synchronization and mutual exclusion between the cleaner and guest threads.
2) **hotel_close:**
    - Binary semaphore, initial value = 0.
    - Describes whether the hotel is currently closed or not.
    - Enforces synchronization and mutual exclusion between the cleaner and guest threads.
3) **guest_book:**
    - Binary semaphore, initial value = 1.
    - Describes whether the guest records are available for change or not.
    - Enforces synchronization and mutual exclusion amongst the guest threads.
4) **cleaner_book**:
    - Binary semaphore, initial value = 1.
    - Describes whether the cleaner records are available for change or not.
    - Enforces synchronization and mutual exclusion amongst the cleaner threads.

The general idea is this:

Guest Threads-

- All guests must wait in line to request a room, much the same way guests wait in the lobby before checking in for the first time in a hotel and getting a room number.
- If this is the first guest (after a cleaning phase), then the guest must wait till the hotel is cleaned. The remaining guests still wait in line behind the first guest.
- Once the guest gets a chance to request for a room, one of the following things may happen:
    - An empty room is found, in which case, occupy it immediately. Sleep.
    - No empty room is found and the least important occupant amongst all the rooms has priority at least as much as this guest, in which case, no room is allocated to the guest. Sleep (outside the hotel, not inside) before requesting a room again.

    - No empty room is found and the least important occupant amongst all the rooms has lesser priority than this guest, in which case, a signal is sent to the guest thread sleeping in that room, and that thread is woken up.
- If this is the 2*N the guest, then do not go to sleep, since we know they will be evicted instantly anyways, so post the hotel_close semaphore to let cleaners know that all occupancies have hit 2*N.
- In each of the above steps, (room taken / refused / 2*N th guest), the guest thread posts the guest_book semaphore **BEFORE** going to the sleep state, so that other guests can access the guest_book while this thread sleeps.
- After the sleep is completed (either normally or abruptly evicted by another higher priority guest), the guest must again wait on the guest_book semaphore to update the status while leaving the hotel, much the same way guests need to wait on the front desk before checking out. A check is applied to test whether the guest left normally or was evicted. In case the guest was evicted, the guest does not make any changes to the book, assuming that the person who evicted them had already made the changes.
- Finally, sleep before requesting a room again.

Cleaner Threads-

- All cleaners must wait in line before entering the hotel (wait on semaphore cleaner_book), much the same way employees of any organization must mark their attendance / entry time in a record.
- If it is the first cleaner thread (after an occupancy phase), then the cleaner must wait for the hotel to close. Other cleaner threads are waiting on cleaner_book.
- Once a cleaner has gotten hold of the record, they choose a random room out of the list of rooms to clean and remove that room from the list. Also, conditionally evict any guest that might be staying in the room. Finally, post the cleaner_book semaphore so other cleaners can proceed.
- Clean the room for time proportional to amount of time room was used (via sleep)
- After cleaning, the cleaners must again wait on the cleaner_book semaphore to update their status to "finished", much the same way employees must mark their time of leaving.
- If this is the Nth cleaner (meaning all N rooms are cleaned), then signal the hotel_open semaphore so guests can now enter.
- Then again wait for the cleaner_book before cleaning a new room.

Parallelism guarantees:

Parallelism in the whole procedure is achieved as follows:

- The only sequential code is within the critical sections. The critical sections only deal with the update of previously defined data structures. Each critical section takes time proportional to log(N) (N = no. of rooms), for finding/insertion/deletion from a std::set.
- Actual work on the data structures like staying in and cleaning a room are respectively done by the guest and cleaner threads concurrently. This duration is in the order of tens of seconds, which comprises the major time duration of execution.

Possibilities of race conditions and avoidance techniques:

Signals in a multithreaded environment lead to hidden race conditions, which can disrupt the expected flow of code.

Despite this, the unconventional choice to use signals in this multithreaded environment is because we need to wake up sleeping threads.
Without using signals, we would have to use a semaphore (or pthread mutex) for doing a timed_wait (or cond_timedwait) on it and then possibly post (or broadcast condition) from the other thread trying to wake up this thread.

We would end up needing at least 1 semaphore for each room! It would slow down the code because of unnecessary waiting and posting semaphores (or locking and unlocking mutex locks) and occupy more memory for each semaphore object. It also leads to more races (not elaborated here) which, although can be handled, make the code messy.

So, instead of this, to wake up a thread, we simply send a SIGUSR1 using pthread_kill() to the sleeping thread. SIGUSR1 is used since it is a user-configurable signal (with a custom signal handler implemented, which basically does nothing and simply returns) since over-writing the system's signals is not recommendable.

The sleeping thread uses sigtimedwait() where the timeout is equal to sleep duration in that room. This basically makes the thread wait until any of the signals specified in the mask argument is received or timeout occurs (it may also exit abnormally if any other signals are sent, but since we only send SIGUSR1, we do not need to worry about this).

However, this seemingly simple setting creates 2 race conditions, which are mentioned with solutions-

Race 1:

SIGUSR1 is sent by thread B (from within the critical section) to thread A, which is outside the critical section, but before it calls sigtimedwait().

Soln 1:

Use pthread_sigmask() via the wrapper signal_blocker() to mask SIGUSR1 in thread A BEFORE exiting the critical section. Since thread B can only enter the critical section (and consequently signal thread A) AFTER thread A has left, the signal has definitely been blocked by thread A before thread B signals it. Now any incoming SIGUSR1 signals are queued in thread A before it calls sigtimedwait(), so the above race is removed.

Race 2:

Thread B decides to send SIGUSR1 to thread A. But before it can do so, thread A's sigtimedwait() times out. Then thread B sends the signal to thread A (since even though thread A has finished sleeping, it has not updated its status in the shared data structures since the critical section is currently being run by thread B) while it is waiting on the semaphore guest_book. This signal interrupts sem_wait() and it comes out abruptly.

Soln 2:

This is simply an extension to soln 1. If sigtimedwait() has timed out, the signal mask of the current thread is returned to its original state. So, SIGUSR1 is currently blocked (and queued if received). Now, we only unblock the signal AFTER entering the critical section, so the sem_wait() is not interrupted by the signal anymore. Also, if there is any remaining signal (at most 1, as guaranteed by the data structures), it is immediately consumed by executing the signal handler.