HW6: Concurrent ProHgramming

In this homework you will build a **concurrent** Airline Reservation System (ARS). The ARS provides essential functionalities:

* Book: reserve a seat on a flight and issue a ticket
* Cancel: cancel a ticket
* Change: cancel a ticket and book another flight **atomically**

The ARS needs to work correctly in a concurrent environment where multiple threads can perform any of the above operations without external synchronization. Apparently, the ARS needs to take care of everything internally, such as acquiring locks.

A summary of your programming tasks:

* Complete the implementation of ars.c to make it work in a single-threaded environment (pass "./test" and "./main" without arguments)
* Use a mutex lock to enforce thread safety and allow multiple threads to access the ARS simultaneously (pass "./main <n>", where n > 1)
* Improve the scalability with fine-grained locks to allow concurrent processing on different flights (make "./main <n>" run much faster)
* Enable wait-if-no-seat (book\_flight\_can\_wait). User can wait until a seat is made available on the target flight (pass "./cond <n>", where n > 5)

**The skeleton code**

Read the functions in ars.c. In this system, a ticket is uniquely identified by a combination of **<user-id, flight-id, ticket-number>**. To obtain a ticket, a user (more accurately, an agent thread on behalf of the user) calls book\_flight() with a user-id and a flight number. Once booked, the ARS will save the combination of the three numbers in an array associated to the flight data structure (flights[i]->tickets[j]). To cancel or change a ticket, user must provide the correct three-number combination. Accordingly, tickets are "non-transferable" because user-id must be the same.

ARS doesn't maintain seat numbers. It only needs to make sure the flights are never over-booked.

The skeleton code contains most of the required data structures and functions. In your submission you can only add/change code in ars.c but not any other files. Similar to homeworks 4 and 5, the autograder will only copy and use your **ars.c** and ignore the other files in your submission. Feel free to change any files for debugging but remember to test with the original ones for your final solution.

test.c, main.c, and wait.c are programs that test the correctness and efficiency of the ARS. Your will need to make all of them work as expected.

**Detailed tasks**

1. Your first task is to implement the change\_flight function to make the system fully functional with one thread. To verify it works, compile the "**test**" program and run it. It will show "no error" if it works as expected. However, it is not thread-safe and if you compile and run "main" with an argument >= 2, such as "make main; ./main 2" it will almost certainly fail due to race conditions (Murphy's Law).
2. A global mutex lock (biglock) is provided in ars.c but not used. Your second task is to employ this lock to provide thread-safety to the ARS. You need to perform lock/unlock in book/cancel/change functions. You **don't** need to use them in ars\_init() and dump\_tickets(). Once this is done, "./main <n>" should work correctly with multiple threads (try ./main 2 and ./main 3).
3. Have you observed the slowdown as the number of threads increases? Now, let's look at the code in main.c. Each worker thread performs 10 million operations by random. Since ars uses one big lock, all of the incoming requests must execute sequentially. Can we make it more efficient by exploiting more concurrency? Of course. Since there is no shared data between different flights, we can give each flight its own lock which allows operations on different flights to run concurrently without blocking each other. Your third task is to replace the biglock with some per-flight mutex locks. Be careful with **deadlocks**! Once this is done, "./main 10" shall finish within 5 seconds.
4. The book\_flight function will return -1 if the flight is fully booked, which can be quite annoying. To better serve the customers, you need to roll out a new service called "book\_flight\_can\_wait", which always returns a valid ticket to the caller! If a flight is fully booked, the function must wait until some one have cancelled a ticket. You need to use **condition variables** to achieve this. You will need to replace the code in book\_flight\_can\_wait() and add necessary structures/code to a few places in ars.c. Once this is done, "./wait <n>" should work correctly with any n > 5. For example, "./wait 8" should finish within 5 seconds. See the lists below for available pthread library functions.

pthread functions that you can use:

* pthread\_mutex\_init, pthread\_mutex\_lock, pthread\_mutex\_unlock
* pthread\_cond\_init, pthread\_cond\_wait, pthread\_cond\_signal

**Don't use the following functions:**

* pthread\_cond\_timedwait
* pthread\_cond\_broadcast

**Debugging tips (in gdb):**

While segmentation faults or failed assertions often print out a message to give you some clue, deadlocks can make you wait indefinitely without seeing anything. Sometimes you will have to use a debugger to hunt the bugs, which is often painful. Here are a few tips that might be useful if you're using gdb from command line.

* If program hangs there due to deadlocks, use "CTRL-c" to interrupt the program and inspect the execution context.
* "tui enable": a smart way to show source code.
* "info threads" or "i th": see all threads.
* "thread 2" or "t 2": select a thread in the list.
* "bt": see the stack backtrace of the current thread.
* "frame 1" or "fr 1": switch to a function frame in the stack trace. main() should correspond to the largest frame number.
* "info locals" or "i lo": inspect local variables.