Lift Simulation Program - Writeup

# Achieving Mutual Exclusion

Mutual exclusion in both implementations was used to coordinate the interaction of the lift request handler (Lift-R) and lift processes/threads with two primary resources: the request buffer and the log file. This was required to ensure output to the log file was not overwritten/interleaved, and to prevent race conditions when interacting with the buffer.

## Using Threads

When using threads, mutual exclusion is achieved using the POSIX threads library (Pthreads). The lift request handler and lifts are all individual threads which have access to some of the same resources (using pointers to locations on the heap which all the threads share), which include the request buffer – a struct containing an array of Request structs as well as other related values (including a mutex and conditions) - and the log file, as a direct file pointer previously opened.

For mutual exclusion with the file pointer, a simple mutex lock (pthread\_mutex\_t) was used. When Lift-R or any of the lifts want to log an action, they first call pthread\_mutex\_lock(), importing the mutex lock corresponding to the buffer (which all threads have access to as a shared pointer to the heap). This function will check to see if the mutex is currently ‘locked’, indicating another thread is currently writing to the log file which has called pthread\_mutex\_lock() previously. If it is locked, the function will cause the thread to wait until it becomes unlocked (and it is not locked again by another thread first). Once unlocked, the function will itself lock the mutex and allow it to continue. The thread will then perform the logging operation using calls to fprintf, knowing no other thread in the program can currently be editing the file as other threads attempting to do so would be blocked on their own pthread\_mutex\_lock() calls. Once writing is complete, the process calls pthread\_mutex\_unlock() on the same mutex, unlocking it and allowing other threads to perform logging.

Operations on the buffer prevent simultaneous reads/writes in the same way, using a different shared mutex between all threads, though the add/remove operations have an additional issue: What should happen if, once a lock on the mutex has been gained, the buffer is empty for a retrieval or full for an addition? This is solved using Pthreads’ condition variables (condition\_t). Like mutexes, these have been made to be shared between all threads. Any thread can at any time ‘signal’ one of these variables, indicating some event. If a thread retrieving from the buffer finds the buffer to be empty, it will begin to wait on the ‘added to buffer’ condition (calling pthread\_cond\_wait() on the pthread\_cond\_t addedCond & with the buffer mutex). This will unlock the imported mutex (here the buffer mutex) so other threads can again manipulate the buffer. If a thread adds a request to the buffer, it will then signal the ‘added to buffer’ condition (calling pthread\_cond\_signal() on addedCond), indicating an item has been added. This will then cause the waiting thread to attempt to regain a lock on the buffer (effectively a call to pthread\_mutex\_lock) and, once received, continue running. Since multiple threads can be waiting on the same condition and more than one may be informed when a condition is signalled, the logical case they were originally waiting due to (the buffer being empty) is checked again before continuing with the retrieval, and the condition wait is repeated if it is again the case, ensuring mutual exclusion. The same operations are also performed when adding to the buffer, but with the case being a full buffer and the condition being ‘removed from buffer’.

Additionally, to avoid cases of unlimited waiting, a timed condition wait was implemented instead of a normal one during buffer retrieval. This causes the wait to ‘time out’ after a given amount of time and abort the attempt at retrieval, done in case the retrieval attempt was initiated at the same time that the last element was removed from the buffer and after Lift-R has stopped adding elements to the buffer. Conditions are also used when marking the buffer as completed (empty with no more requests incoming), by checking the number of elements in the buffer each time the ‘removed’ condition is signalled until the buffer is empty, at which point it is marked as completed.

## Using Processes (POSIX implementation)

Semaphores (sem\_t) were used to ensure mutual exclusion during when using processes, and the ‘mmap’ function was used to create shared memory between processes.

For the initial lock on the log-file and on the buffer, binary semaphores were used – semaphores made to keep a value of either 0 or 1. These function very similarly to mutexes when using threads –before the log file/buffer is accessed an attempt to wait on the semaphore associated with the resource is made (sem\_wait() importing the semaphore). This function attempts to reduce the value of the semaphore by 1 – if it is 0, the process will wait until the value is increased, at which point it is then reduced, and the process continues. Since the semaphores used here can only have a value of 0 or 1, this is effectively the same as locking a mutex. Once operation with the resource is done, the associated semaphore is signalled (via sem\_post()), increasing its value by 1 and effectively acting as an ‘unlock’.

The buffer concurrency issue solved using conditions when using threads was also solved here using semaphores. Two additional semaphores exist associated with the buffer – ‘full’ (initial value 0) and ‘empty’ (initial value equal to the buffer size). Whenever a request is added to the buffer ‘full’ is signalled and whenever one is removed ‘empty’ is signalled. Before add/remove operations even attempt to get a lock on the buffer, they will wait on empty/full respectively. This way, retrieve operations will wait for there to be at least one element in the buffer and remove operations will wait for at least one to be free. Signalling a semaphore also means that only one of the processes waiting on the semaphore will continue, eliminating the need for an additional check that the buffer is not empty/full.

# Testing

I am confident that my program works perfectly as I have tested it extensively. As can be seen in the ‘Testing’ folder, I have tested both lift sim A’s and B’s functionality in a variety of different cases, changing the buffer size, number of requests in the input file and the time taken to complete a request. This included a number of special edge cases that I believed could potentially cause issues (e.g. a buffer size of 1 (less than the number of lifts), a buffer size larger than the total number of requests). I found my program to work perfectly for all tested scenarios, which I verified by looking at the created output file and ensuring values were valid and by running the program in ‘DEBUG’ mode, ensuring all the additional information printed out to the terminal also indicated things were working correctly. I also tested both simulations using valgrind and found no memory leaks, though some ‘still reachable’ blocks are present, in part A due to pthread library functions and in part B as these blocks are later freed in the parent process, though the child processes still see them as ‘unfreed’ when they complete.

One bug that was found during testing was that the implementation using processes effectively ignored the ‘buffer size’ argument when running – using a buffer of size 20 (and operating correctly with it) regardless of what was input. The actual buffer was being initialised correctly – the issue was that the ‘empty’ semaphore was being initialised to 20, meaning Lift-R would continue adding until it was reduced to 0 (always 20 adds). I hypothesise this caused it to simply overwrite the memory past the buffer, which must have been within legal boundaries during testing, allowing the ‘buffer’ of size 20 to be read without segmentation faults during execution.