# OPERATING SYSTEMS ASSIGNMENT REPORT (SEM. 1, 2020)

The work of Rados, Kristian (19764285)

# DISCUSSION

## HOW MUTUAL EXCLUSION IS ACHIEVED

In the **lift_sim_A simulation** mutual exclusion between threads is achieved using a shared mutex lock via the pthread_mutex_lock() and pthread_mutex_unlock() functions in combination with a conditional variable via the pthread_cond_wait() or pthread_cond_timedwait() and pthread_cond_signal() functions. The first thread to begin its operations will lock the mutex on the buffer (named bufferLock) and check for a condition. In the case of liftR() this condition is that the buffer is full. If this condition is true, then the thread will unlock the bufferLock and wait on the condition variable to be signalled; otherwise the thread will continue to perform operations on a request. In the case of lift(), this checked condition is that the buffer is empty and that the pthread_cond_timedwait() hasn't timed out.

Once the conditional variable is signalled a thread that is waiting on it will block the conditional variable again and get a lock on the mutex bufferLock. The thread will then perform its request operations (the critical section) before sleeping for the time specified by the command line parameter in the case of lift(). Once done with this, the thread then signals the conditional variable and unlocks the mutex, allowing another thread to perform its critical section. Timed waits are put in place to prevent the system from entering deadlock in certain situations.

In the **lift_sim_B simulation** mutual exclusion between processes is achieved by using semaphores via the sem_wait() or sem_timedwait() and sem_post() functions. There are three semaphore variables used to ensure mutual exclusion: 'empty', 'full' and 'mutex'. The 'mutex' semaphore functions much like the mutex in lift_sim_A in that it can only have a value of 1 or 0, hence only ever allowing a single process to access the shared resources at a time. 'Empty' is initialised to the size of the buffer and hence its value represents the number of empty spots in the buffer, while 'full' is initialised to 0 and conversely represents the number of full spots in the buffer.

For the liftR() process to work there must be an empty spot in the buffer, hence it waits on the 'empty' semaphore to have a value > 0. The lift() processes conversely must have at least one full spot in the buffer and so wait on the 'full' semaphore to have a value > 0. They both must then wait on the 'mutex' before entering the critical section. Once done with their critical section they signal mutex via sem_post(mutex), allowing another process to enter its critical section, and they signal the semaphore that represents the change the process has made: in the case of liftR() this means signalling and thus incrementing 'full', as it has filled a spot in the buffer. Timed waits are put in place to prevent the system from entering deadlock in certain situations.

## ACCESS TO SHARED RESOURCES

Both lift_sim_A and lift_sim_B make use of two structs named Shared and Info. The Shared struct contains all of the shared data and variables that must be shared by all of the created threads/processes (including the buffer) and a single instance of this struct is created, with

pointers to the struct being shared throughout the program. The Info struct acts as a wrapper struct for the Shared struct that also contains extra variables that are needed by each individual lift thread/process. This allows all of the information needed by the lifts to be stored in a single struct that can be passed to the thread or process when it's created. Each lift gets its own Info struct instance, but they all contain a pointer to the same Shared struct.

For the threads in lift_sim_A they all have access to the memory that needs to be shared as it is created in the main thread before the other threads are created and the pointers are passed to each thread. In lift_sim_B the Shared and Info structs are mapped to shared memory that is accessible to all child processes before the process is forked. The first fork is given the Shared struct in liftR() and the other 3 forked processes are given their respective Info structs as passed into lift().

# PROBLEM CASES

When lift_sim_A is run with a buffer size of 1 and t == 0, the simulation stutters forward and runs far slower than it should. This occurs because the simulation is repeatedly entering some form of deadlock, causing the simulation to freeze momentarily before it is continued by the lift waits timing out, with the process then repeating. The result is that even though t == 0, the simulation will take a significant period of time to finish when it should be finishing almost immediately. The effect can be particularly seen when the program is compiled with VERBOSE=1.

When lift_sim_B is run with a buffer size of 1 and t > 0 the simulation will occasionally be cut short. Not all of the requests from the input file will be processed and the simulation will end, with two requests being consumed at the end despite the buffer having a size of 1.

# SAMPLE INPUTS/OUTPUTS

The A simulation run with a buffer of size 5 and a processing time of 0 seconds per request.

```
> ./lift_sim_A 5 0
Running simulation...

Finished simulation.
Output has been logged to 'sim_out.txt'.
```

The B simulation run with an invalid sim_input.txt file (too few requests)

```
> ./lift_sim_B 3 1
Error: Incorrect format in sim_input line, must be between 50 and 100 requests (lines)
```

The A simulation run with an invalid command line parameter: [m] is too low

```
> ./lift_sim_A 0 1
Error: Buffer size has been specified as 0, but it must be > 0
```

```
> ./lift_sim_B 6 2
Running simulation...
Lift-R: #1
Lift-R: #2
Lift-R: #3
Lift-R: #4
Lift-R: #5
Lift-R: #6
Lift-1: 1 -> 7 -> 6
Lift-R: #7
Lift-2: 1 -> 7 -> 8
Lift-R: #8
```

Here the B simulation is run with the VERBOSE conditional compilation option, a buffer of size 6 and a processing time of 2 seconds per request. Each line is a request either being added or removed from the buffer. We can see that Lift-R fills the buffer before the lifts start consuming them. Only a small portion of the terminal output is shown here.

```
1   ---------------------------------------------
2     New Lift Request From Floor 18 to Floor 8
3     Request No: 1
4   ---------------------------------------------
5
6   ---------------------------------------------
7     New Lift Request From Floor 19 to Floor 12
8     Request No: 2
9   ---------------------------------------------
10
11  Lift-2 Operation
12  Previous position: Floor 1
13  Request: Floor 19 to Floor 12
14  Detail operations:
15      Go from Floor 1 to Floor 19
16      Go from Floor 19 to Floor 12
17      #movement for this request: 25
18      #request: 1
19      Total #movement: 25
20  Current position: Floor 12
21
22  Lift-3 Operation
23  Previous position: Floor 1
24  Request: Floor 18 to Floor 8
25  Detail operations:
26      Go from Floor 1 to Floor 18
27      Go from Floor 18 to Floor 8
28      #movement for this request: 27
29      #request: 1
30      Total #movement: 27
31  Current position: Floor 8
32
33  ---------------------------------------------
34    New Lift Request From Floor 3 to Floor 5
35    Request No: 3
36  ---------------------------------------------
```

Here is a small sample from the beginning of the sim_output.txt file after running ./lift_sim_A 2 0. We can see that the buffer is filled before Lift-2 and Lift-3 execute those two requests.