# COMP304: Operating Systems
# Project 2: Spacecraft Control with Pthreads

Betül DEMİRTAŞ - 68858
Doğa Ege İNHANLI - 69033

P.s: All the details of our project can be found in the repository at: https://github.com/bdemirtas18/beOS
P.s: You can compile our code by this command: gcc -g project_2.c -lpthread
P.s: Example run command: ./a.out -p 0.2 -t 120 -s 10 -n 30

## PART 1:

In the first part, we were asked to implement spacecraft control mechanisms without emergency and starvation conditions.

In order for the simulation to be in real time, we used the time(NULL) function which gives the current time. To keep the track of the waiting jobs, we used separate queues for launching, landing and assembling. Also, we create the main threads for the Control Tower and the Launching Job that is waiting at time zero. Then, we created a while loop so that with the randomized probability we can create a new Job thread in every SIMULATION_T until it reaches the total real simulation time. The Job threads basically constructs a new job with its necessary information such as id, type, request time etc. and calls Enqueue function to push the job into the corresponding waiting queue. To prevent the race condition among the same type of jobs, which are waiting to be pushed into the corresponding queue, we used pthread_mutex_lock and pthread_mutex_unlock functions.

In the meantime, the already created Control Tower thread creates two new threads for Pad A and Pad B. In both of these threads, the real simulation time is checked continuously. In Pad A, we can only operate Landing and Launching Jobs. Since the Landing Job is prior to the Launching Job, if there are any waiting Jobs on the landing queue, the pad accepts it first. It pops the selected job from the corresponding queue by calling Dequeue function, then we use the given pthread_sleep() function in order to simulate the time the Jobs take. The PopJob function also assigns ending time, turnaround time, pad and status information to the popped Job and saves these assigned information to the 'events.log' file. During this popping stage, we used mutex locks to prevent the race condition among the jobs aiming to leave from the same queue. Similar to Pad A, the landing is prior to the assembly operation in pad B. In order to make sure that there is only one job being done at each pad at a time, we created mutexes for both of the pads. Additionally, in order to prevent the situation where both of the pads try to pop the same Landing Job from the queue and operate on it, we created a mutex for the landing operation.

## PART 2:

In this part, first we tried to prevent starvation for ground jobs. For this purpose, we added a counter-based control mechanism. In the earlier part we designed the pads so that the landing jobs were prior. We reimplemented these parts to give priority above landing when 3 or more launches (for Pad A) or 3+ assemblies (for Pad B) are waiting to be scheduled. However, this may create starvation for the orbit jobs. To be able to prevent it, we designed a new mechanism which controls the consecutive job type counts. It only allows at most two Launch and one Assembly Jobs to be operated consecutively and the pads have to operate a Landing Job afterwards if there are any in the waiting queue.

## PART 3:

For the Emergency Jobs, we created a new queue and new threads for each of two pieces. In the while loop we created for the probabilistic creation of new Jobs, we added another condition which creates and pushes the new Emergency Jobs every 40t seconds (in real time). In the threads of the Pad A and Pad A, we gave priority to the emergency queue by adding it as the first if condition. If there is a Job in the emergency queue, any Pad which is available pops it from the queue and operates on it. Since both of the Pads can operate on an Emergency Job, we added a mutex to prevent the race condition among the Pads.

## KEEPING LOGS:

To be able to keep track of required informations for the log we added four new attributes to the Job struct:

```
time_t req_time
time_t end_time
time_t ta_time
int pad
```

We assign the id, type and req_time when the Job struct is constructed and assign the end_time, ta_time and pad when the Job is popped from the queue. Before we free the popped Job, we save this information to the ‘event.log’ file.

To print the queues continuously, we created a new thread, which constructs the printing structure on the pdf. To start printing after “n” seconds, which is given by the user as a command line argument, we checked the current real time continously.

REFERENCES:

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*. Hoboken, N.J: Wiley, 2019.

[2] *C Library Function - time()*. [Online]. Available: https://www.tutorialspoint.com/c_standard_library/c_function_time.htm. [Accessed: 26-Apr-2022].

[3] L. L. N. L. Blaise Barney, “POSIX Threads programming,” *LLNL HPC Tutorials*. [Online]. Available: https://hpc-tutorials.llnl.gov/posix/. [Accessed: 26-Apr-2022].

[4] P. Prinz and T. Crawford, “C in a Nutshell,” *O'Reilly Online Learning*. [Online]. Available: https://www.oreilly.com/library/view/c-in-a/0596006977/re86.html. [Accessed: 26-Apr-2022].

[5] “Difference between deadlock and starvation in os,” *GeeksforGeeks*, 30-Sep-2019. [Online]. Available: https://www.geeksforgeeks.org/difference-between-deadlock-and-starvation-in-os/. [Accessed: 26-Apr-2022].