**COMP 304 PROJECT 2**

**Our Github Repository**: https://github.com/edemirbas17/COMP304\_Project\_2

**We completed implementations for all parts. All parts are working successfully.**

**COMPILATION and EXECUTION**

For compilation, type **gcc project\_2.c -lpthread** to the terminal. In order to run the code type **./a.out -p 0.2 -t 120 -s 10 -n 30** to the terminal. At the end of run, our program will create a file called **events.log** and print outputs to the terminal.

**Important note:** While we print the queues to the terminal, we separated landing, launching, assembly queues and pad queues (A and B). We prefer this GUI since in this way, we can observe these queues separately and see transition from job queues to the pad queues.

**PART 1**

For the first part, we started our simulation and initialized our variables, pthread\_t(s) and mutexes in main. We also constructed our queues for landing, launching, assembly, padA, padB and emergency inside of main similar to pseudocode given in the PS. In addition to this, we recorded **start\_time** of the simulation inside of the main with **time(NULL)** command. We identified **end\_time** with respect to summation of **start\_time** and **simulationTime**. All these variables (mutexes, start\_time, end\_time) are declared globally in our implementation.

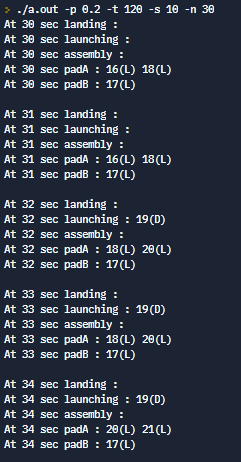
Then, we implemented LandingJob, LaunchJob and AssemblyJob methods in a similar way with pseudocode given in the PS. Firstly, we recorded **currentTime** before entering the while loop with **time(NULL)** command. Then, we started while loop if condition holds **currentTime < end\_time**. In the while loop, after using **pthread\_sleep** for t seconds, we generated random possibility with **rand()** command. If this created random possibility smaller than predetermined values in the description, we generated our jobs. For counting number of jobs and recording their ID(s), we created a global **counter**. Since this counter is a shared variable, we have locked counter mutex before incrementing this counter. Then, we created our job and record its ID with respect to counter value. Also, we recorded arrival time of the job by recording current time in this region and subtracting start time from the recorded current time. We have added **int arrival\_time** variable to the **Job struct** in the **queue.c**. Then, we enqueued this created job into corresponding queue. Before adding job to the queue, we also utilized from mutexes of queues.

In our **Pad\_A** and **Pad\_B** implementations we followed similar way described in the implementation of LandingJob, LaunchJob, etc. First, we recorded currentTime and check whether currentTime is smaller than end\_time or not in the while loop condition. If the pad queue is empty, then pad thread sleeps for t seconds. Before checking emptiness of our pad queues, we utilized from corresponding mutexes. If our pad is not empty, we dequeue one job from our queue and used pthread\_sleep with respect to sleep time of dequeued job. We recorded sleep time of jobs in the job struct as **type variable**. After sleep, we recorded current time again and calculated **termination\_time** of job. **We added termination\_time variable into Job struct**.

In the **ControlTower** implementation, we first recorded currentTime and checked whether it is smaller than end\_time or not before entering the while loop. In the while loop, we created another while loop which is checking whether **landing\_queue** is empty or not (since landing cannot wait in the part 1). If landing\_queue is not empty, then we locked **padA\_mutex and** **padB\_mutex** and compared the total time of jobs inside of these queues. **For this comparison, we added getTotalTime method into queue.c**. This method iterates over the queues and sums the **job.types** inside of the queues. By using this comparison, we determined which queue should take landing job (the one with smaller total time). Then, we dequeue landing job from landing queue and enqueue to the corresponding pad queue. After this inner while loop, we added assembly jobs and launch jobs into corresponding pads. For these jobs, we have also used mutexes for enqueue, dequeue operations.

The following screenshot is output of our results from part1 implementations. As can be seen from results, the first job with ID = 1 is launching job (departing). Then, the jobs are given to pads with respect to total time of jobs in the pads. These results belong to our final implementation, that’s why part 2, part 3 and part 4 (keeping logs) can be observed from these screenshots as well.

tablo içeren bir resim

Açıklama otomatik olarak oluşturuldu 

**PART 2**

In this part the requirement was solving the starvation problem with given two conditions:

• (a) No more pieces are waiting to land

• (b) 3 or more launches or 3+ assemblies are waiting to be scheduled on the ground.

To solve this issue, we first break the priority of the landing queue in control tower when the assembly or launching queue size get 3 and they were not ready to operate in pads. Then we give the priority of using pads to assembly and launching if their queue size reach 3, otherwise priority stay in the landings. With this way we protected to “no more pieces are waiting to land (which already implemented in part 1)”, and “3 or more launches or 3+ assemblies are waiting to be scheduled on the ground” conditions. Note that in this part the jobs not assigned to the front of the pad queue as implemented in the emergency it just solves the starvation problem and assigning to the pads to do not wait all the landing jobs finish.

The sample output of the queues at different seconds:

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

As you see there is not any starvation in landing assembly or launching but this can’t be proof there were not any starvation, so we increase the probability P to 0.6 (while executing the code via command line argument) which created many landing assembly or launching job as seen in below:

Text

Description automatically generated

Regardless of probability (P) value and simulation time we can see that with our solution to the starvation problem there is no starvation (No landing job, no assembly or launching with size>=3) and following two conditions satisfied.

• (a) No more pieces are waiting to land SATISFIED ✓

• (b) 3 or more launches or 3+ assemblies are waiting to be scheduled on the ground. SATISFIED✓

**For the answer to the given question in project description:**

When a situation which launching and assembly job created heavily more than the landing job, there can be starvation for the pieces waiting in the orbit. From an example probability situation which is can be seen above assembly already fulfilled the Pad B when there was heavily more launching job created than the landing this can be also situation for the Pad A. To solve this issue aging can be implemented. When the waiting time of the landing pieces in orbit increases, they can be prioritized with the aging.

**PART 3**

In this part our requirements were initializing the emergency situation in simulation and solving this situation. We first create 2 emergency condition in every **40t** seconds in “**EmergencyJob(void \*arg**)” function. Then we implement a “**Enqueue\_Front(Queue \*pQueue, Job j)**” function in **queue.c** file to enqueue a job in front of a queue order. The reason of this implementation, queues are working with first in first out but in an emergency situation the emergency job (last job) should be first out. Then we change the control tower and give priority to emergency, and we make sure that when an emergency job created all the other pad assigning’s braked, and emergency job directly added to the in front of the pad queues.

For the pad assignment we check the time required to finish emergency jobs and we choose the best options. The following conditions are satisfied for the pad chosen.

* If the sleep time of the pad B not finished and Pad A can do two emergency job until Pad B sleep finishes, then Pad A does 2 emergencies.
* If the sleep time of the pad A not finished and Pad B can do two emergency job until Pad A sleep finishes, then Pad A does 2 emergencies.
* If best required time to finish 2 emergency job was separating the job Pad A and Pad B, then the jobs get separates the Pads.

With this implementation turnaround of the emergency jobs are observed as expected.

The output of the log file to show Part 3 result:

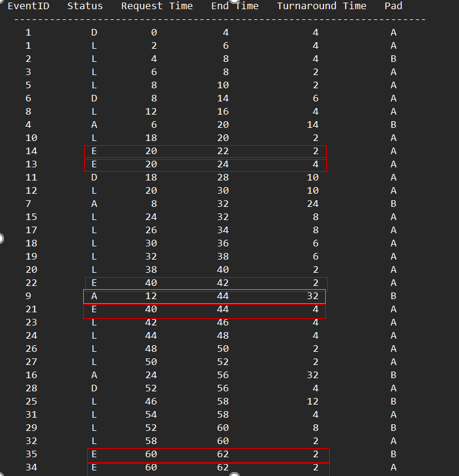
metin, ekran, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

From the log file it can be directly seen that the emergency job directly assigned to the pads, and they operated as soon as possible so their turn around time is very small. But since emergency period so small it shown in only every 40t(80) second of the simulation. Because of that reason three scenario which I explained in above may not be seen so we increase the emergency frequency to every 10t (20) second.

**Note that: This is done only for reporting purposes, our code emergency frequency is as described in the project description.**

Result is below:



For example, when we increase the emergency frequency at 40 second since the Pad B busy (indicated blue line) the program calculated when Pad A assigned by the two-emergency turnaround time of the emergencies are get much smaller, so the Pad A does two of the emergencies. At 60 second since both pad A and Pad B not in long job when emergency separated them the turnaround time get very smaller, so the jobs separated. So, we can conclude that our implementation gives smallest turnaround time results.

**PART 4 (Keeping Logs)**

In order to keep logs, we implemented a method called **recordLogs**. This method takes job and pad as input arguments and prints log statements in to events.log file. For modifying this file, we utilized from **file\_mutex** in order to prevent data corruption.

In order to print waiting queues starting from time n, we added a method called **printQueues** to the **project\_2.c**. Inside of this method, we called **printQueue method** for each queue. This method was added into queue.c which is iterating over elements of queue. We initialized another thread called **printing\_thread** for printing queues concurrently. Users need to specify n in the command window before executing the output file like “**./a.out -n 30”**.

**Important note:** While we print the queues to the terminal, we separated landing, launching, assembly queues and pad queues (A and B). We prefer this GUI since in this way, we can observe these queues separately and see transition from job queues to the pad queues.

The following screenshot is demonstrating how -n argument should be specified before running the code.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

The **following screenshot** is demonstrating our example log file which is called **events.log**.

tablo içeren bir resim

Açıklama otomatik olarak oluşturuldu

