CSE344 - Homework#3

Muhammet Talha Memişoğlu

210104004009

Introduction

The objective of this assignment is to simulate a satellite-communication scheduling system using multithreading and synchronization techniques in C. In this system, satellite threads request engineer service with different priority levels, and engineer threads serve these requests accordingly. The project makes extensive use of pthread, semaphore, and mutex APIs under the POSIX threading model and was tested on Debian 11.

Code Explanation

Initialization

- The program begins by defining the necessary global variables for priority levels, number of satellites and engineers, and connection timeout windows.
- Two mutexes (**engineerMutex, servedMutex**) and one semaphore (**newRequest**) are initialized for synchronization.

Satellite Struct

Each satellite is represented by a structure that contains:

- Satellite ID
- Priority
- Thread ID
- A binary semaphore (**request_handled**) used for engineer response
- A served flag indicating whether satellite is still handling by engineer when timeout is reached.

Main Function

- Initializes threads and semaphores
- Spawns a number of engineer threads equal to NUM_ENGINEERS
- Spawns satellite threads with random priorities equal to NUM_SATELLITE
- Waits for all satellite threads to terminate using **pthread_join**
- Waits for a while to ensure all engineers finish their work before exiting

- Cancels and joins engineer threads to simulate graceful exit
- Calls **cleanup()** to release resources

Thread Logic

Satellite Thread

Each satellite:

- Initializes its semaphore
- Locks the engineer mutex to add itself to a shared queue based on priority
- Signals availability using **sem_post(&newRequest)**
- Waits (with timeout) using **sem_timedwait()** for an engineer to serve it

```
// Wait for an engineer to handle the request, or timeout
int result = sem_timedwait(&sat->request_handled, &timeout);
```

- If timed out and **sat->served** is ture, it means engineer that is assigned to that satellite hasn't completed its job so we won't initiate our timeout mechanism.
- If timed out and **sat->served** is false, it means any engineer is not assigned to that satellite. Therefore, it removes itself from the queue and logs a timeout message
- Destroys its semaphore before exit

Engineer Thread

Each engineer:

- Waits on **newRequest** semaphore for satellite requests
- Pops the highest-priority satellite from the queue using **pop_highest_priority()**
- Sets the served flag and simulates a 4–6 second processing time randomly.
- Signals the satellite's **request_handled** semaphore after serving
- Increments the available engineers counter and logs service completion

---

Thread and Semaphore Flow

Satellite Thread Flow:

1. Request service
2. Wait for a limited connection window
3. If served not in time but assigned to a engineer → success, else if served in time → success , else → timeout

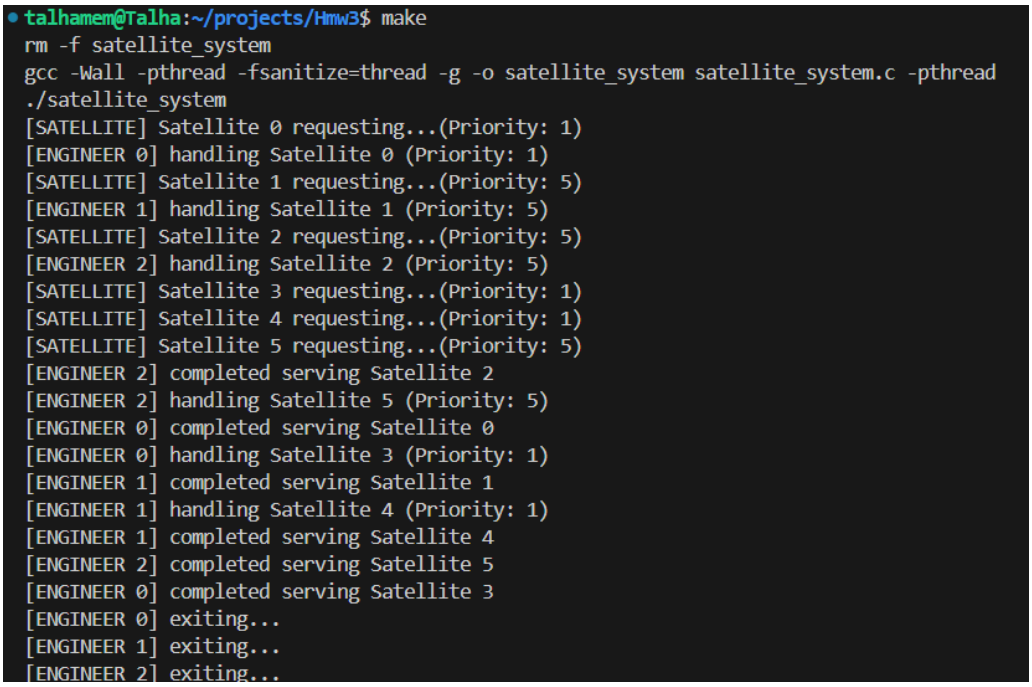
Engineer Thread Flow:

1. Wait for request signal
2. Pick highest priority request
3. Serve and update shared state
4. Repeat until canceled in the end of the program

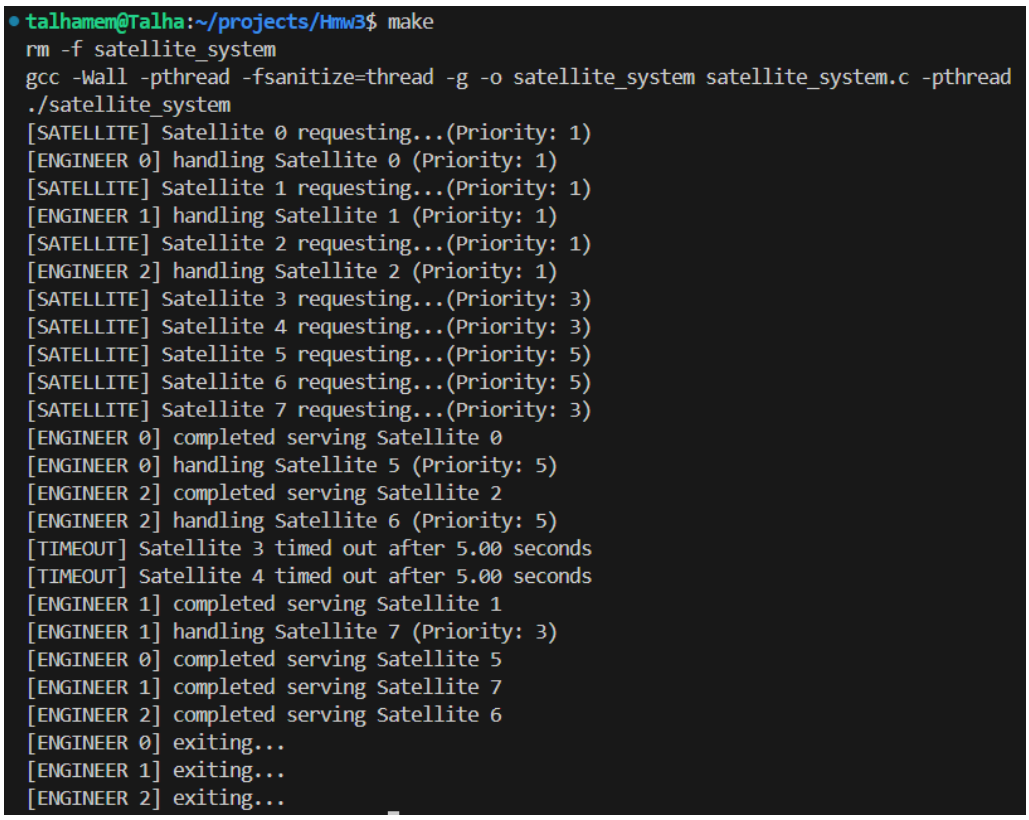
Inter-thread Synchronization:

- **engineerMutex** protects access to the shared request queue and global engineer availability
- **servedMutex** ensures correct access to the served flag
- **sem_timedwait** on per-satellite semaphores prevents indefinite blocking and continue if timeout occurs.
- **newRequest** semaphore acts as a signaling mechanism between satellite and engineer threads. When a satellite posts a request, it calls sem_post(&newRequest). Engineer threads wait on this semaphore (sem_wait(&newRequest)), ensuring they are only activated when a new satellite request is available.

Screenshots



Testing with 5 Satellites and 3 Engineers



Testing with 8 Satellites and 3 Engineers

Conclusion

In this project, one of the main challenges was preventing the timeout mechanism from being triggered while an engineer was still actively working on a satellite request. To address this, I introduced an additional semaphore and a served flag that is set to true as soon as an engineer begins processing a request. This ensures that the satellite thread recognizes it is being served, even if the task is not yet completed. Furthermore, to ensure that all engineer threads have sufficient time to complete their work before the program terminates, I added an additional 9-second delay at the end of the main function. Ultimately, I successfully completed all aspects of the assignment, fulfilling the all requirements.