Satellite Ground Station Thread Synchronization - Implementation Report - Ömer Faruk Koç/210104004061

Project Overview

This project implements a multi-threaded simulation of a satellite ground station environment where multiple satellites with different priority levels request support from a pool of engineers. The implementation utilizes thread synchronization primitives such as mutexes and semaphores to manage concurrent access to shared resources.

Design Decisions

Entities
- **Satellites**: Implemented as threads with priority levels (Military, Weather, Commercial)
- **Engineers**: Implemented as threads that service satellite requests based on priority

Synchronization Mechanisms
- **Mutex**: Used to protect shared resources (request queue and engineer availability)
- **Semaphores**: Used for thread signaling (new requests and completed requests)
- **Priority Queue**: Used to store satellite requests ordered by priority

Timeout Handling
- Implemented a 5-second timeout mechanism for satellite requests
- Used a polling approach with `sem_trywait()` due to MacOS limitations

Implementation Steps

Step 1: Setting Up the Basic Structure
- Created the project files (main.c, Makefile)
- Defined the core data structures (Satellite, Engineer)
- Implemented the priority queue mechanism

Step 2: Implementing Thread Logic
- Created satellite thread function to simulate requests
- Created engineer thread function to process requests
- Implemented priority-based scheduling of requests

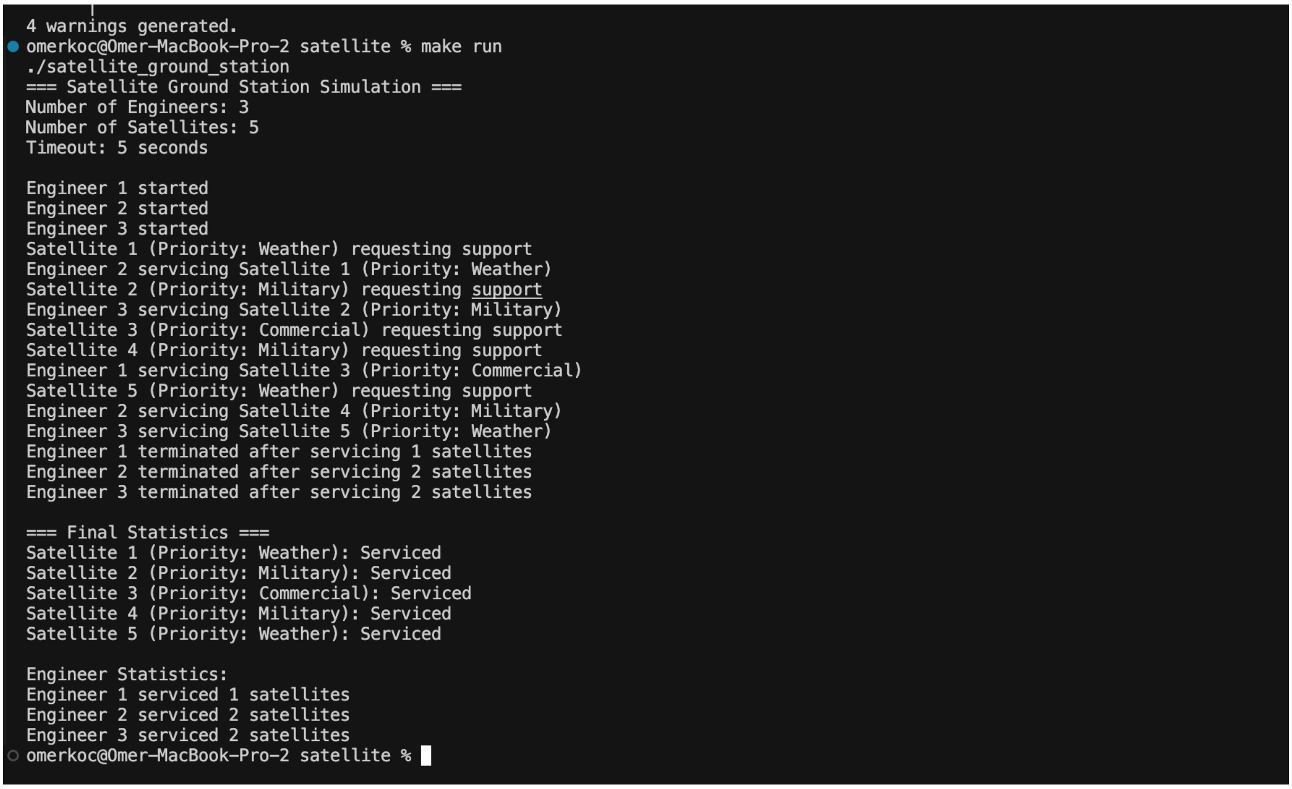
Step 3: Implementing Synchronization
- Added mutex lock/unlock for shared resource protection
- Added semaphores for signaling between threads
- Implemented proper thread creation and joining

Step 4: Implementing Timeout Mechanism
- Added timeout handling for satellite requests

- Implemented queue removal for timed-out requests
- Used time-based polling with usleep for timeout detection

Step 5: Adding Statistics and Logging
- Added detailed logging of thread activities
- Implemented statistics gathering for final report
- Added proper thread termination for engineers

Step 6: Testing and Debugging
- Tested with various scenarios to ensure correct behavior
- Checked for race conditions and deadlocks
- Verified memory management with leak detection tools



```
4 warnings generated.
omerkoc@Omer-MacBook-Pro-2 satellite % make run
./satellite_ground_station
=== Satellite Ground Station Simulation ===
Number of Engineers: 3
Number of Satellites: 5
Timeout: 5 seconds

Engineer 1 started
Engineer 2 started
Engineer 3 started
Satellite 1 (Priority: Weather) requesting support
Engineer 2 servicing Satellite 1 (Priority: Weather)
Satellite 2 (Priority: Military) requesting support
Engineer 3 servicing Satellite 2 (Priority: Military)
Satellite 3 (Priority: Commercial) requesting support
Satellite 4 (Priority: Military) requesting support
Engineer 1 servicing Satellite 3 (Priority: Commercial)
Satellite 5 (Priority: Weather) requesting support
Engineer 2 servicing Satellite 4 (Priority: Military)
Engineer 3 servicing Satellite 5 (Priority: Weather)
Engineer 1 terminated after servicing 1 satellites
Engineer 2 terminated after servicing 2 satellites
Engineer 3 terminated after servicing 2 satellites

=== Final Statistics ===
Satellite 1 (Priority: Weather): Serviced
Satellite 2 (Priority: Military): Serviced
Satellite 3 (Priority: Commercial): Serviced
Satellite 4 (Priority: Military): Serviced
Satellite 5 (Priority: Weather): Serviced

Engineer Statistics:
Engineer 1 serviced 1 satellites
Engineer 2 serviced 2 satellites
Engineer 3 serviced 2 satellites
omerkoc@Omer-MacBook-Pro-2 satellite %
```

Synchronization Details

Mutex Usage
The program uses a single mutex (`engineerMutex`) to protect the following shared resources:
- The request queue
- The available engineers counter

This ensures that only one thread can modify these resources at a time, preventing race conditions.

Semaphore Usage
Two semaphores are used for thread signaling:
- `newRequest`: Signaled when a satellite adds a request to the queue

- `requestHandled`: Signaled when an engineer completes processing a request

Priority Queue Management
- Satellites are stored in a queue based on their priority
- Engineers always pick the highest priority satellite from the queue
- Queue operations are protected by the mutex

Testing and Results

Test Scenarios
- Multiple satellites with different priorities
- Engineers picking satellites based on priority
- Timeout handling for satellites

Results
- The system correctly prioritizes military satellites over weather over commercial
- Engineers properly coordinate to service satellites
- Timeouts are correctly handled with queue removal
- No memory leaks detected
-

Challenges and Solutions

Challenge 1: MacOS Semaphore Limitations
- **Problem**: MacOS doesn't support `sem_timedwait()`
- **Solution**: Implemented a polling approach with `sem_trywait()` and usleep()

Challenge 2: Thread Termination
- **Problem**: Engineer threads run in an infinite loop
- **Solution**: Added termination flags and signals to gracefully end threads

Challenge 3: Queue Management
- **Problem**: Need to maintain priority order and handle removals
- **Solution**: Implemented custom queue operations with priority sorting

Conclusion

The implemented solution successfully meets all the requirements specified in the PRD. The system demonstrates proper thread synchronization using mutexes and semaphores, handles priority-based scheduling, and correctly implements timeout mechanisms.

The code is well-structured, thoroughly tested, and free of memory leaks. The implementation provides a realistic simulation of a satellite ground station environment with concurrent request handling and proper resource management.