Satellite Ground Station Communication System

CSE 344 - System Programming
Homework #3

Student ID: 210104004228

April 21, 2025

Contents

1 Introduction

This project implements a satellite ground station communication system that coordinates interactions between multiple satellites and a limited number of engineers. The system handles satellites with different priority levels that arrive randomly and request engineer support for updates. Each satellite has a limited connection window, and if no engineer becomes available within this window, the update is aborted.

The implementation utilizes POSIX threads, semaphores, and mutexes to achieve proper synchronization between satellite requests and engineer availability. A priority queue system ensures that higher-priority satellites are served first when multiple satellites are waiting.

2 Implementation Details

2.1 Semaphore and Mutex Usage

Semaphores and mutexes play crucial roles in this implementation for thread synchronization and protection of shared resources. Their specific applications are:

2.1.1 Mutex (engineerMutex)

The mutex is used to protect critical sections where shared resources are accessed or modified:

```
pthread_mutex_t engineerMutex;
```

This mutex ensures that:

- Only one thread can modify the `availableEngineers` counter at a time
- Only one thread can access or modify the priority queue at a time
- Satellite threads cannot simultaneously enqueue their requests
- Engineer threads cannot simultaneously dequeue satellite requests

The mutex provides mutual exclusion, preventing race conditions when multiple threads attempt to access or modify shared data structures.

Example mutex usage in satellite thread:

```
pthread_mutex_lock(&engineerMutex);
enqueue(&requestQueue, sat);
pthread_mutex_unlock(&engineerMutex);
```

Example mutex usage in engineer thread:

```
pthread_mutex_lock(&engineerMutex);
sat = dequeue(&requestQueue);
availableEngineers--;
pthread_mutex_unlock(&engineerMutex);
```

2.1.2 Semaphores

Two main semaphores are used in the implementation:

newRequest semaphore

```c
sem_t newRequest;
```

This semaphore is used to signal engineer threads that a new satellite request has arrived. Each time a satellite thread adds a request to the queue, it posts to this semaphore. Engineer threads wait on this semaphore, and when it's signaled, they attempt to handle the new request.

Example in satellite thread:

```c
sem_post(&newRequest);
```

Example in engineer thread:

```c
sem_wait(&newRequest);
```

Individual request_handled semaphores:

```c
sem_t request_handled;  // Inside Satellite structure
```

Each satellite thread has its own semaphore that is used by the engineer thread to signal when the satellite's request has been picked up. The satellite thread waits on this semaphore with a timeout, and if the semaphore isn't signaled within the connection window, the satellite aborts its update.

Example initialization:

```c
sem_init(&(sat->request_handled), 0, 0);
```

Example wait with timeout in satellite thread:

```c
int result = sem_timedwait(&(sat->request_handled), &timeout);
```

Example signal in engineer thread:

```c
sem_post(&(sat->request_handled));
```

2.2 Priority Queue Implementation

The priority queue is implemented using a linked list data structure. This approach was chosen for several important reasons:

1. **Dynamic Size Management**: Unlike arrays, linked lists don't require pre-allocation of a fixed size, allowing the queue to grow or shrink based on the number of satellite requests.

2. **Efficient Priority-Based Insertion**: The implementation enables efficient insertion based on priority. New satellite requests are inserted at the appropriate position in the queue according to their priority level, with O(n) worst-case complexity.

3. **Constant-Time Dequeuing**: Retrieving the highest-priority satellite is an O(1) operation since it's always at the head of the list.

4. **Memory Efficiency**: Memory is allocated only when needed for new requests, and freed when requests are processed, ensuring optimal resource utilization.

The priority queue structure consists of:

```c
typedef struct PriorityQueue {
    QueueNode* head;
} PriorityQueue;
```

Where each node contains:

```c
typedef struct QueueNode {
    Satellite* satellite;
    struct QueueNode* next;
} QueueNode;
```

The `enqueue` function inserts satellites in descending order of priority:

```c
void enqueue(PriorityQueue* queue, Satellite* satellite) {
    QueueNode* newNode = (QueueNode*)malloc(sizeof(QueueNode));
    newNode->satellite = satellite;

    // Empty queue or new satellite has higher priority than head
    if (queue->head == NULL || satellite->priority > queue->head
       ->satellite->priority) {
        newNode->next = queue->head;
        queue->head = newNode;
        return;
    }

    // Find the correct position based on priority
    QueueNode* current = queue->head;
    while (current->next != NULL &&
           current->next->satellite->priority >= satellite->
              priority) {
        current = current->next;
    }

    newNode->next = current->next;
    current->next = newNode;
}
```

2.3 Thread Management and Synchronization

The system creates two types of threads:

1. **Satellite Threads**: Represent satellites requesting engineer support

2. **Engineer Threads**: Represent engineers serving satellite requests

The synchronization flow works as follows:

1. A satellite thread:

- Assigns a random priority to the satellite
- Locks the mutex and adds itself to the request queue
- Unlocks the mutex
- Signals the newRequest semaphore
- Waits for its request_handled semaphore with a timeout

2. An engineer thread:
   - Waits for the newRequest semaphore
   - When signaled, locks the mutex
   - Checks if engineers are available
   - Dequeues the highest-priority satellite from the queue
   - Decrements the availableEngineers counter
   - Unlocks the mutex
   - Processes the satellite request
   - Signals the satellite's request_handled semaphore
   - Locks the mutex
   - Increments the availableEngineers counter
   - Unlocks the mutex

This design ensures that:

- Higher-priority satellites are served first
- Race conditions are avoided when accessing shared data
- Satellites can timeout if not served within their connection window
- Engineers always handle the most critical requests first

2.4 Timeout Mechanism

The timeout mechanism is implemented using `sem_timedwait()`, which allows a satellite thread to wait for an engineer for a specified duration:

```c
struct timespec timeout;
clock_gettime(CLOCK_REALTIME, &timeout);
timeout.tv_sec += MAX_CONNECTION_TIME;

int result = sem_timedwait(&(sat->request_handled), &timeout);

if (result == -1) {
    printf("Satellite %d: Connection timeout! Update aborted.\n",
        sat->id);
} else {
    printf("Satellite %d: Engineer assigned, performing update.\n
       ", sat->id);
    // Update processing...
}
```

If the semaphore is not signaled within the timeout period (MAX_CONNECTION_TIME), the satellite aborts its update request and exits.

3 Code Analysis

3.1 Data Structures

The main data structures used in the implementation are:

3.1.1 Satellite Structure

```c
typedef struct Satellite {
    int id;
    int priority;
    time_t arrival_time;
    sem_t request_handled;
} Satellite;
```

This structure stores:

- Satellite identification
- Priority level (1-5)
- Arrival timestamp
- Semaphore for signaling when an engineer picks up the request

3.1.2 Priority Queue

The priority queue is implemented as a linked list with nodes:

```c
typedef struct QueueNode {
    Satellite* satellite;
    struct QueueNode* next;
} QueueNode;

typedef struct PriorityQueue{
    QueueNode* head;
} PriorityQueue;
```

3.2 Satellite Thread Implementation

The satellite thread function:

1. Assigns a random priority level to the satellite
2. Acquires the mutex to safely access shared resources
3. Adds the satellite to the request queue
4. Releases the mutex

5. Signals the newRequest semaphore to notify engineers
6. Waits for an engineer to handle the request (with timeout)
7. Processes the update if an engineer is assigned, or aborts if timeout occurs

Key portions of the implementation:

```c
void* satellite(void* arg) {
    Satellite* sat = (Satellite*)arg;

    assignRandomPriority(sat);
    sat->arrival_time = time(NULL);

    // Acquire mutex before accessing shared resources
    pthread_mutex_lock(&engineerMutex);

    enqueue(&requestQueue, sat);
    printf("Satellite␣%d␣added␣to␣queue.\n", sat->id);

    pthread_mutex_unlock(&engineerMutex);

    // Signal that a new request has arrived
    sem_post(&newRequest);

    // Wait for an engineer with timeout
    struct timespec timeout;
    clock_gettime(CLOCK_REALTIME, &timeout);
    timeout.tv_sec += MAX_CONNECTION_TIME;

    int result = sem_timedwait(&(sat->request_handled), &timeout)
        ;

    // Process result (timeout or successful assignment)
    if (result == -1) {
        printf("Satellite␣%d:␣Connection␣timeout!␣Update␣aborted
            .\n", sat->id);
    } else {
        printf("Satellite␣%d:␣Engineer␣assigned,␣performing␣
            update.\n", sat->id);

        sleep(rand() % 3 + 1);

        printf("Satellite␣%d:␣Update␣completed.\n", sat->id);
    }

    return NULL;
}
```

3.3 Engineer Thread Implementation

The engineer thread function:

1. Waits for a new satellite request (via newRequest semaphore)
2. Acquires the mutex to safely access shared resources
3. Checks if engineers are available
4. Dequeues the highest-priority satellite from the request queue
5. Decrements the availableEngineers counter
6. Releases the mutex
7. Processes the satellite request
8. Signals the satellite that its request has been handled
9. Acquires the mutex again
10. Increments the availableEngineers counter
11. Releases the mutex

Key portions of the implementation:

```c
void* engineer(void* arg) {
    int id = *((int*)arg);
    Satellite* sat;

    printf("Engineer␣%d␣ready.\n", id);

    while (engineersActive) {
        // Wait for a new request
        sem_wait(&newRequest);

        if (!engineersActive) {
            break;
        }

        // Acquire mutex to access shared resources
        pthread_mutex_lock(&engineerMutex);

        if (availableEngineers > 0) {
            sat = dequeue(&requestQueue);

            if (sat != NULL) {
                availableEngineers--;
                printf("Engineer␣%d␣handling␣Satellite␣%d␣(
                   priority:␣%d).\n",
                       id, sat->id, sat->priority);

                pthread_mutex_unlock(&engineerMutex);

                sleep(rand() % 3 + 2);

```

```
                    sem_post(&(sat->request_handled));

                    // Make engineer available again
                    pthread_mutex_lock(&engineerMutex);
                    availableEngineers++;
                    printf("Engineer␣%d␣is␣available␣again.\n", id);
                    pthread_mutex_unlock(&engineerMutex);
                } else {
                    pthread_mutex_unlock(&engineerMutex);
                }
            } else {
                pthread_mutex_unlock(&engineerMutex);
            }
        }

        printf("Engineer␣%d␣shutting␣down.\n", id);
        return NULL;
}
```

4 Test Results

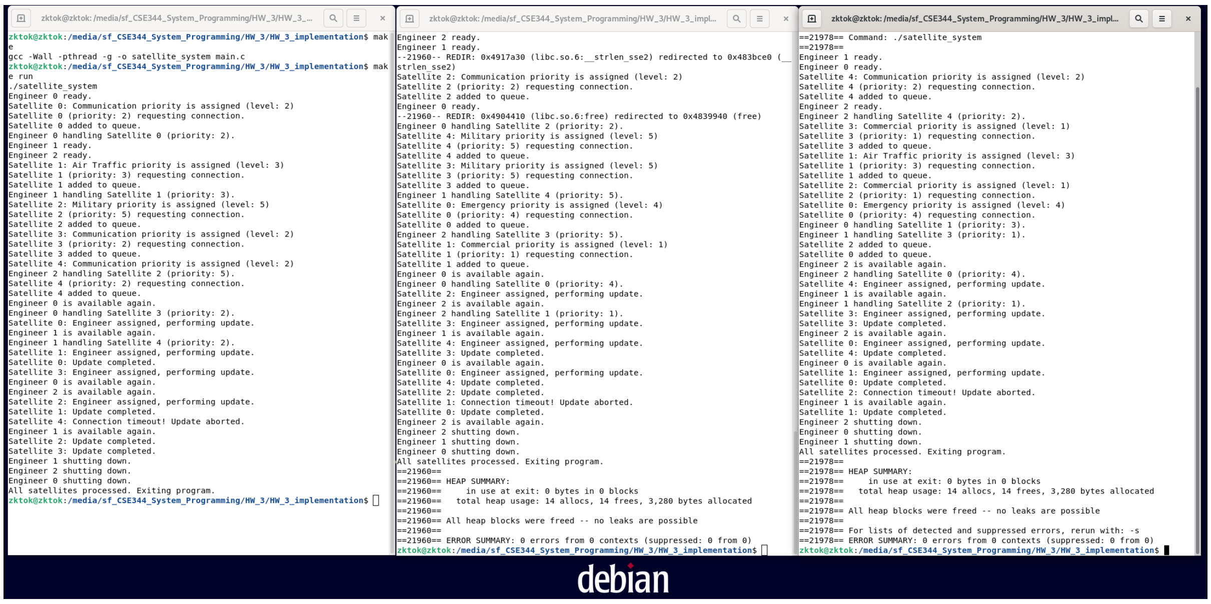
The implementation was thoroughly tested with various configurations:

1. **Basic functionality test**: Multiple satellites with different priorities are created, and engineers handle them according to their priority levels.
2. **Timeout test**: When all engineers are busy, satellites with lower priorities may timeout if their connection windows expire.
3. **Priority handling test**: When multiple satellites are waiting, the system always selects the one with the highest priority first.
4. **Memory leak test**: The code was tested with Valgrind to ensure no memory leaks exist. All dynamically allocated memory is properly freed before program termination.

4.1 Memory Leak Testing

The program was tested with Valgrind to verify there are no memory leaks. The results confirm that all memory is properly freed at program exit:

```
==21978== HEAP SUMMARY:
==21978==     in use at exit: 0 bytes in 0 blocks
==21978==     total heap usage: 14 allocs, 14 frees, 3,280 bytes
   allocated
==21978== All heap blocks were freed -- no leaks are possible
==21978== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
   from 0)
```



zktok@zktok: /media/sf_CSE344_System_Programming/HW_3/HW_3_...

```
zktok@zktok:/media/sf_CSE344_System_Programming/HW_3/HW_3_implementation$ mak
e
gcc -Wall -pthread -g -o satellite_system main.c
zktok@zktok:/media/sf_CSE344_System_Programming/HW_3/HW_3_implementation$ mak
e run
./satellite_system
Engineer 0 ready.
Satellite 0: Communication priority is assigned (level: 2)
Satellite 0 (priority: 2) requesting connection.
Satellite 0 added to queue.
Engineer 0 handling Satellite 0 (priority: 2).
Engineer 1 ready.
Engineer 2 ready.
Satellite 1: Air Traffic priority is assigned (level: 3)
Satellite 1 (priority: 3) requesting connection.
Satellite 1 added to queue.
Engineer 1 handling Satellite 1 (priority: 3).
Satellite 2: Military priority is assigned (level: 5)
Satellite 2 (priority: 5) requesting connection.
Satellite 2 added to queue.
Satellite 3: Communication priority is assigned (level: 2)
Satellite 3 (priority: 2) requesting connection.
Satellite 3 added to queue.
Satellite 4: Communication priority is assigned (level: 2)
Engineer 2 handling Satellite 2 (priority: 5).
Satellite 4 (priority: 2) requesting connection.
Satellite 4 added to queue.
Engineer 0 is available again.
Engineer 0 handling Satellite 3 (priority: 2).
Satellite 0: Engineer assigned, performing update.
Engineer 1 is available again.
Engineer 1 handling Satellite 4 (priority: 2).
Satellite 1: Engineer assigned, performing update.
Satellite 0: Update completed.
Satellite 3: Engineer assigned, performing update.
Engineer 0 is available again.
Engineer 2 is available again.
Satellite 2: Engineer assigned, performing update.
Satellite 1: Update completed.
Satellite 4: Connection timeout! Update aborted.
Engineer 1 is available again.
Satellite 2: Update completed.
Satellite 3: Update completed.
Engineer 1 shutting down.
Engineer 2 shutting down.
Engineer 0 shutting down.
All satellites processed. Exiting program.
zktok@zktok:/media/sf_CSE344_System_Programming/HW_3/HW_3_implementation$ 
```

zktok@zktok: /media/sf_CSE344_System_Programming/HW_3/HW_3_impl...

```
Engineer 2 ready.
Engineer 1 ready.
--21960-- REDIR: 0x4917a30 (libc.so.6:__strlen_sse2) redirected to 0x483bce0 (__
strlen_sse2)
Satellite 2: Communication priority is assigned (level: 2)
Satellite 2 (priority: 2) requesting connection.
Satellite 2 added to queue.
Engineer 0 ready.
--21960-- REDIR: 0x4904410 (libc.so.6:free) redirected to 0x4839940 (free)
Engineer 0 handling Satellite 2 (priority: 2).
Satellite 4: Military priority is assigned (level: 5)
Satellite 4 (priority: 5) requesting connection.
Satellite 4 added to queue.
Satellite 3: Military priority is assigned (level: 5)
Satellite 3 (priority: 5) requesting connection.
Satellite 3 added to queue.
Engineer 1 handling Satellite 4 (priority: 5).
Satellite 0: Emergency priority is assigned (level: 4)
Satellite 0 (priority: 4) requesting connection.
Satellite 0 added to queue.
Engineer 2 handling Satellite 3 (priority: 5).
Satellite 1: Commercial priority is assigned (level: 1)
Satellite 1 (priority: 1) requesting connection.
Satellite 1 added to queue.
Engineer 0 is available again.
Engineer 0 handling Satellite 0 (priority: 4).
Satellite 2: Engineer assigned, performing update.
Engineer 2 is available again.
Engineer 2 handling Satellite 1 (priority: 1).
Satellite 3: Engineer assigned, performing update.
Engineer 1 is available again.
Satellite 4: Engineer assigned, performing update.
Satellite 3: Update completed.
Engineer 0 is available again.
Satellite 0: Engineer assigned, performing update.
Satellite 4: Update completed.
Satellite 2: Update completed.
Satellite 1: Connection timeout! Update aborted.
Satellite 0: Update completed.
Engineer 2 is available again.
Engineer 2 shutting down.
Engineer 1 shutting down.
Engineer 0 shutting down.
All satellites processed. Exiting program.
==21960==
==21960== HEAP SUMMARY:
==21960==     in use at exit: 0 bytes in 0 blocks
==21960==   total heap usage: 14 allocs, 14 frees, 3,280 bytes allocated
==21960==
==21960== All heap blocks were freed -- no leaks are possible
==21960==
==21960== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
zktok@zktok:/media/sf_CSE344_System_Programming/HW_3/HW_3_implementation$ 
```

zktok@zktok: /media/sf_CSE344_System_Programming/HW_3/HW_3_impl...

```
==21978== Command: ./satellite_system
==21978==
Engineer 1 ready.
Engineer 0 ready.
Satellite 4: Communication priority is assigned (level: 2)
Satellite 4 (priority: 2) requesting connection.
Satellite 4 added to queue.
Engineer 2 ready.
Engineer 2 handling Satellite 4 (priority: 2).
Satellite 3: Commercial priority is assigned (level: 1)
Satellite 3 (priority: 1) requesting connection.
Satellite 3 added to queue.
Satellite 1: Air Traffic priority is assigned (level: 3)
Satellite 1 (priority: 3) requesting connection.
Satellite 1 added to queue.
Satellite 2: Commercial priority is assigned (level: 1)
Satellite 2 (priority: 1) requesting connection.
Satellite 0: Emergency priority is assigned (level: 4)
Satellite 0 (priority: 4) requesting connection.
Engineer 0 handling Satellite 1 (priority: 3).
Engineer 1 handling Satellite 3 (priority: 1).
Satellite 2 added to queue.
Satellite 0 added to queue.
Engineer 2 is available again.
Engineer 2 handling Satellite 0 (priority: 4).
Satellite 4: Engineer assigned, performing update.
Engineer 1 is available again.
Engineer 1 handling Satellite 2 (priority: 1).
Satellite 3: Engineer assigned, performing update.
Satellite 3: Update completed.
Engineer 2 is available again.
Satellite 0: Engineer assigned, performing update.
Satellite 4: Update completed.
Engineer 0 is available again.
Satellite 1: Engineer assigned, performing update.
Satellite 0: Update completed.
Satellite 2: Connection timeout! Update aborted.
Engineer 1 is available again.
Satellite 1: Update completed.
Engineer 2 shutting down.
Engineer 0 shutting down.
Engineer 1 shutting down.
All satellites processed. Exiting program.
==21978==
==21978== HEAP SUMMARY:
==21978==     in use at exit: 0 bytes in 0 blocks
==21978==   total heap usage: 14 allocs, 14 frees, 3,280 bytes allocated
==21978==
==21978== All heap blocks were freed -- no leaks are possible
==21978==
==21978== For lists of detected and suppressed errors, rerun with: -s
==21978== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
zktok@zktok:/media/sf_CSE344_System_Programming/HW_3/HW_3_implementation$ 
```

Figure 1: Program Execution and Memory Test Results

5 Conclusion

This implementation successfully addresses the requirements of the satellite ground station communication system. It effectively:

1. **Prioritizes satellite requests**: Higher-priority satellites are served first using a priority queue implementation.

2. **Manages limited resources**: The system handles a fixed number of engineers and distributes them efficiently.

3. **Implements connection timeouts**: Satellites abort their updates if not served within their connection windows.

4. **Ensures thread safety**: Semaphores and mutexes are used to prevent race conditions and protect shared resources.

5. **Optimizes memory usage**: The linked list-based priority queue provides dynamic memory management with no leaks.

The implementation demonstrates the effective use of multi-threading concepts and synchronization primitives in C programming. The priority queue implementation with a linked list proves to be an efficient choice for this scenario, allowing dynamic request management with priority-based processing.