GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING
DEPARTMENT

CSE344 Systems Programming
Spring 2025
Homework 3 Report

Osmancan Bozali
220104004011

1. Introduction:

This report analyzes and documents the implementation of a satellite ground station coordination system that manages communications between Earth and multiple satellites. The system simulates a priority-based queuing system where satellites with varying priority levels request engineer support for updates. There are three engineers available to assist satellites, and each satellite has a limited connection window to establish communication. If no engineer becomes available within the allowed connection window, the satellite aborts the update and leaves.

The system is implemented in C using multithreading, synchronization mechanisms (semaphores and mutexes), and a priority queue to manage satellite requests based on their priorities. The implementation showcases the use of concurrent programming techniques to handle resource allocation in a time-constrained environment.

1.1 Objective

The main objective of this system is to:

- Coordinate communications between satellites and engineers
- Serve satellites based on their priority levels
- Handle timeout scenarios when engineers are not available
- Properly manage shared resources using synchronization mechanisms
- Implement a priority queue for efficient request handling

1.2 System Components

The system consists of the following key components:

- Satellite threads that represent satellites requesting connections
- Engineer threads that represent engineers handling requests
- A priority queue to manage satellite requests
- Synchronization mechanisms (semaphores and mutexes)
- Timeout handling for connection windows

2. Code Explanation:

2.1 Data Structures

The implementation uses two primary data structures:

SatelliteRequest Structure

```c
typedef struct {
    int id; // Satellite ID
    int priority; // Priority of the request
    sem_t requestHandled; // Semaphore for request handling
    bool isHandled; // Flag to check if request is handled
    bool hasTimedOut; // Flag to check if request has timed out
    time_t timeout; // Timeout duration in seconds
} SatelliteRequest;
```

This structure represents a satellite request and contains:

- The satellite's ID and priority level
- A semaphore for signaling when the request is handled
- Flags for tracking request status
- A timeout value for the connection window

RequestQueue Structure

```
typedef struct {
    SatelliteRequest* requests[NUM_SATELLITES]; // Array of
requests
    int size; // Current size of the queue
} RequestQueue;
```

This structure implements a priority queue as a binary heap:

- An array of satellite request pointers
- A size field to track the current number of requests in the queue

2.2 Global Variables

The system uses several global variables for coordination:

```
int availableEngineers = NUM_ENGINEERS;
RequestQueue requestQueue = {.size = 0};
pthread_mutex_t engineerMutex = PTHREAD_MUTEX_INITIALIZER;
sem_t newRequest;
```

These globals maintain the system state and provide synchronization points:

- `availableEngineers` tracks the number of free engineers
- `requestQueue` holds the satellite requests ordered by priority
- `engineerMutex` protects access to shared resources
- `newRequest` semaphore signals when new requests arrive

2.3 Priority Queue Implementation

The priority queue is implemented as a binary max-heap with the following operations:

Enqueue Function

```
void enqueue(SatelliteRequest* request) {
    // Add request to the end of the queue
    int i = requestQueue.size++;
    requestQueue.requests[i] = request;
    // Rearrange the heap by heapifying up
    while (i > 0) {
        int parent = (i - 1) / 2;
        if (requestQueue.requests[parent]->priority >=
requestQueue.requests[i]->priority) break;
        // Swap with parent if higher priority
        SatelliteRequest* temp = requestQueue.requests[parent];
        requestQueue.requests[parent] = requestQueue.requests[i];
        requestQueue.requests[i] = temp;
        i = parent;
    }
}
```

This function:

- Adds a new request to the end of the heap
- Performs "heapify up" to maintain the max-heap property
- Ensures higher priority requests bubble up to the top

Dequeue Function

```
SatelliteRequest* dequeue() {
    if (requestQueue.size == 0) return NULL; // No requests to
dequeue
    // Get the highest priority request (root of the heap)
    SatelliteRequest* highest = requestQueue.requests[0];
    requestQueue.requests[0] = requestQueue.requests[--
requestQueue.size]; // Replace root with last element
    // Rearrange the heap by heapifying down
    int i = 0;
    while (true) { // Check if we need to swap with children
```

```
        int leftChild = 2 * i + 1;
        int rightChild = 2 * i + 2;
        int highest_idx = i;
        if (leftChild < requestQueue.size &&
requestQueue.requests[leftChild]->priority >
requestQueue.requests[highest_idx]->priority) highest_idx =
leftChild;
        if (rightChild < requestQueue.size &&
requestQueue.requests[rightChild]->priority >
requestQueue.requests[highest_idx]->priority) highest_idx =
rightChild;
        if (highest_idx == i) break; // No swap needed
        // Swap with the highest priority child
        SatelliteRequest* temp = requestQueue.requests[i];
        requestQueue.requests[i] =
requestQueue.requests[highest_idx];
        requestQueue.requests[highest_idx] = temp;
        i = highest_idx;
    }
    return highest;
}
```

This function:

- Returns the highest priority request (the root of the heap)
- Replaces the root with the last element in the heap
- Performs "heapify down" to maintain the max-heap property
- Ensures the new root is the highest priority request

RemoveRequest Function

```
void removeRequest(int satelliteId) {
    for (int i = 0; i < requestQueue.size; i++) { // Find the
request with the given satellite ID
        if (requestQueue.requests[i]->id == satelliteId) {
            requestQueue.requests[i] = requestQueue.requests[--
requestQueue.size]; // Replace with last element
            // Rearrange the heap by heapifying up or down
            int parent = (i - 1) / 2;
            if (i > 0 && requestQueue.requests[i]->priority >
requestQueue.requests[parent]->priority) { // Heapify up
                while (i > 0) {
                    parent = (i - 1) / 2;
                    if (requestQueue.requests[parent]->priority
>= requestQueue.requests[i]->priority) break;

                    SatelliteRequest* temp =
requestQueue.requests[parent];
                    requestQueue.requests[parent] =
requestQueue.requests[i];
                    requestQueue.requests[i] = temp;
                    i = parent;
                }
            } else { // Heapify down
                while (true) {
                    int leftChild = 2 * i + 1;
                    int rightChild = 2 * i + 2;
                    int highest = i;
                    if (leftChild < requestQueue.size &&
requestQueue.requests[leftChild]->priority >
requestQueue.requests[highest]->priority) highest = leftChild;
                    if (rightChild < requestQueue.size &&
requestQueue.requests[rightChild]->priority >
requestQueue.requests[highest]->priority) highest = rightChild;
                    if (highest == i) break;
                    SatelliteRequest* temp =
requestQueue.requests[i];
                    requestQueue.requests[i] =
requestQueue.requests[highest];
                    requestQueue.requests[highest] = temp;
                    i = highest;
                }
            }
            break;
        }
    }
```

```
}
```

This function:

- Finds and removes a specific request from the queue (used when a request times out)
- Maintains the heap property by heapifying up or down as necessary

2.4 Thread Functions

The system uses two main thread functions:

Satellite Thread Function

```
void* satellite(void* arg) {
    int id = *((int*)arg);
    free(arg); // Free the allocated memory for satellite ID

    // Create request with random priority and timeout
    int priority = rand() % 5 + 1;
    SatelliteRequest* request = malloc(sizeof(SatelliteRequest));
    if (request == NULL) {
        fprintf(stderr, "Memory allocation failed for satellite
request\n");
        return NULL;
    }
    // Initialize request
    request->id = id;
    request->priority = priority;
    request->isHandled = false;
    request->hasTimedOut = false;
    request->timeout = rand() % MAX_TIMEOUT + 1; // Random
timeout between 1 and MAX_TIMEOUT
    sem_init(&request->requestHandled, 0, 0); // Initialize
semaphore

    sleep(rand() % 2);  // Random delay

    // Add request to queue with mutex protection
    pthread_mutex_lock(&engineerMutex);
    printf("[SATELLITE] Satellite %d requesting (priority %d)\n", 
id, priority);
    enqueue(request);
    sem_post(&newRequest); // Signal engineer that a new request 
is available
    pthread_mutex_unlock(&engineerMutex);

    // Wait with timeout
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += request->timeout;

    if (sem_timedwait(&request->requestHandled, &ts) != 0 && 
!request->isHandled) {
        // Timeout occurred
        pthread_mutex_lock(&engineerMutex);
        request->hasTimedOut = true;
        removeRequest(id);
        printf("[TIMEOUT] Satellite %d timeout %ld seconds.\n", 
id, request->timeout);
        pthread_mutex_unlock(&engineerMutex);

        sem_destroy(&request->requestHandled);
        free(request);
        return NULL;
    }
    return NULL; // Request handled successfully
}
```

This function:

- Creates a satellite request with random priority and timeout
- Adds the request to the priority queue
- Signals that a new request is available
- Waits for the request to be handled with a timeout
- Handles timeout scenarios by removing the request from the queue

Engineer Thread Function

```
void* engineer(void* arg) {
```

```c
    int id = *((int*)arg);
    free(arg); // Free the allocated memory for engineer ID

    while (1) {
        sem_wait(&newRequest); // Wait for new request
        pthread_mutex_lock(&engineerMutex);

        if (requestQueue.size > 0) { // Check if there are
requests in the queue
            SatelliteRequest* request = dequeue(); // Dequeue the
highest priority request

            // Check for exit signal (priority -1)
            if (request->priority == -1) {
                pthread_mutex_unlock(&engineerMutex);
                printf("[ENGINEER %d] Exiting...\n", id);
                sem_destroy(&request->requestHandled);
                free(request);
                return NULL;
            }

            // Check if timed out
            if (request->hasTimedOut) {
                pthread_mutex_unlock(&engineerMutex);
                continue;
            }

            // Handle request
            request->isHandled = true;
            sem_post(&request->requestHandled);
            availableEngineers--;
            printf("[ENGINEER %d] Handling Satellite %d (Priority
%d)\n", id, request->id, request->priority);
            pthread_mutex_unlock(&engineerMutex);

            // Process time
            sleep(3 + (rand() % 2)); // Simulate processing time
(3-5 seconds)
            printf("[ENGINEER %d] Finished Satellite %d\n", id,
request->id);
            // Cleanup resources
            sem_destroy(&request->requestHandled);
            free(request);

            pthread_mutex_lock(&engineerMutex);
            availableEngineers++;
            pthread_mutex_unlock(&engineerMutex);
        } else {
            pthread_mutex_unlock(&engineerMutex);
        }
    }
}
```

This function:

- Waits for new requests to become available
- Dequeues the highest priority request
- Checks for exit signals and timeouts
- Handles the request by signaling the satellite
- Simulates processing time
- Updates the available engineers count

2.5 Main Function

The main function initializes the system and creates threads:

```
int main() {
    pthread_t satellite_threads[NUM_SATELLITES];
    pthread_t engineer_threads[NUM_ENGINEERS];

    srand(time(NULL));
    sem_init(&newRequest, 0, 0);

    // Start engineer threads
    for (int i = 0; i < NUM_ENGINEERS; i++) {
        int* id = malloc(sizeof(int));
        if (id == NULL) {
            fprintf(stderr, "Memory allocation failed for
engineer ID\n");
            return 1;
        }
        *id = i;
        pthread_create(&engineer_threads[i], NULL, engineer, id);
    }

    // Start satellite threads
    for (int i = 0; i < NUM_SATELLITES; i++) {
        int* id = malloc(sizeof(int));
        if (id == NULL) {
            fprintf(stderr, "Memory allocation failed for
satellite ID\n");
            return 1;
        }
        *id = i;
        pthread_create(&satellite_threads[i], NULL, satellite,
id);
        usleep(500000);
    }

    // Wait for satellites to finish
    for (int i = 0; i < NUM_SATELLITES; i++) {
        pthread_join(satellite_threads[i], NULL);
    }

    // Signal engineers to exit
    for (int i = 0; i < NUM_ENGINEERS; i++) {
        SatelliteRequest* exitRequest =
malloc(sizeof(SatelliteRequest));
        if (exitRequest == NULL) {
            fprintf(stderr, "Memory allocation failed for exit
request\n");
            return 1;
        }
        exitRequest->priority = -1;
        exitRequest->isHandled = false;
        exitRequest->hasTimedOut = false;
        sem_init(&exitRequest->requestHandled, 0, 0);

        pthread_mutex_lock(&engineerMutex);
        enqueue(exitRequest);
        pthread_mutex_unlock(&engineerMutex);

        sem_post(&newRequest);
    }

    // Wait for engineers to exit
    for (int i = 0; i < NUM_ENGINEERS; i++) {
        pthread_join(engineer_threads[i], NULL);
    }

    // Cleanup resources
    sem_destroy(&newRequest);
    pthread_mutex_destroy(&engineerMutex);

    return 0;
}
```

This function:

- Initializes random number generation and semaphores
- Creates threads for engineers and satellites
- Waits for satellites to finish their requests
- Sends exit signals to engineers
- Waits for engineers to exit
- Cleans up resources

3. Testing and Results:

3.1 Basic Functionality Test

This test verifies the core functionality of the system with the default configuration (5 satellites and 3 engineers).

Test Configuration:

- Use the default parameters: `NUM_SATELLITES=5`, `NUM_ENGINEERS=3`
- No code modifications required

Test Execution:

$ make clean

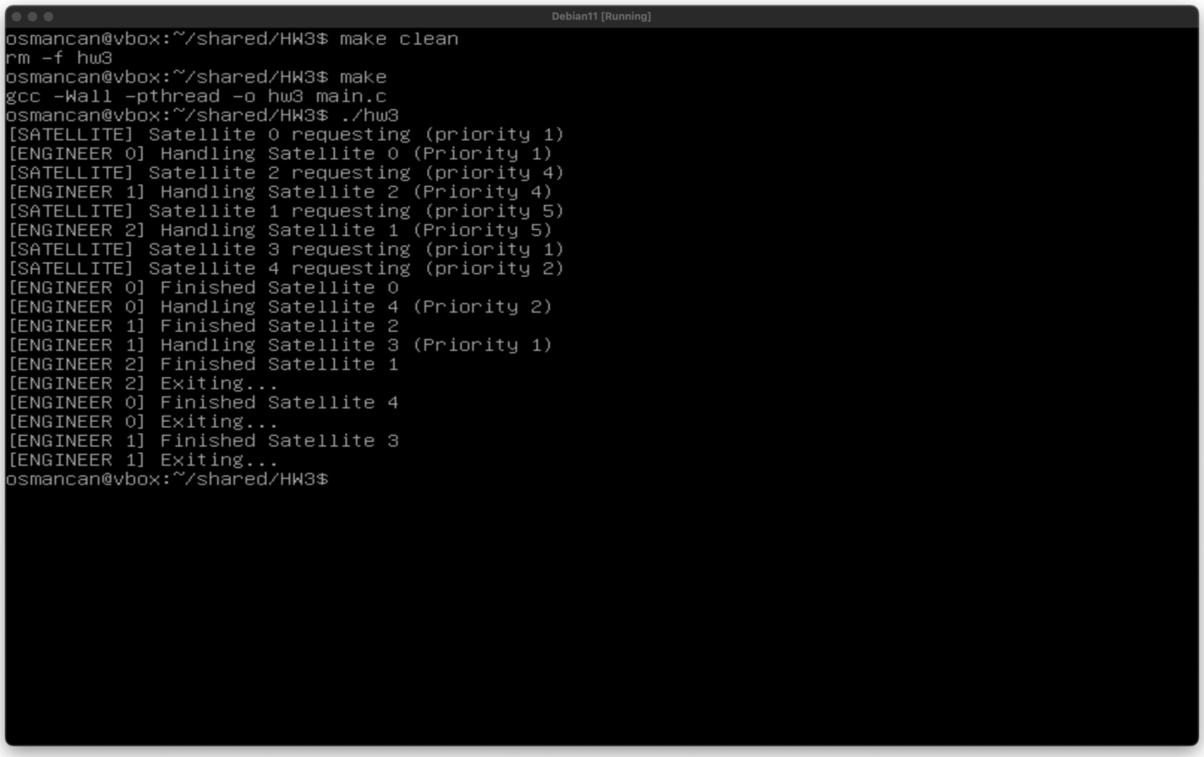
$ make

$ ./hw3

Expected Results:

- Satellites should request connections with random priorities
- Engineers should handle requests in priority order
- Some satellites may time out if all engineers are busy
- All satellites should eventually be served or time out
- All engineers should exit cleanly

Actual Results: Matching with the expected result



```
osmancan@vbox:~/shared/HW3$ make clean
rm -f hw3
osmancan@vbox:~/shared/HW3$ make
gcc -Wall -pthread -o hw3 main.c
osmancan@vbox:~/shared/HW3$ ./hw3
[SATELLITE] Satellite 0 requesting (priority 1)
[ENGINEER 0] Handling Satellite 0 (Priority 1)
[SATELLITE] Satellite 2 requesting (priority 4)
[ENGINEER 1] Handling Satellite 2 (Priority 4)
[SATELLITE] Satellite 1 requesting (priority 5)
[ENGINEER 2] Handling Satellite 1 (Priority 5)
[SATELLITE] Satellite 3 requesting (priority 1)
[SATELLITE] Satellite 4 requesting (priority 2)
[ENGINEER 0] Finished Satellite 0
[ENGINEER 0] Handling Satellite 4 (Priority 2)
[ENGINEER 1] Finished Satellite 2
[ENGINEER 1] Handling Satellite 3 (Priority 1)
[ENGINEER 2] Finished Satellite 1
[ENGINEER 2] Exiting...
[ENGINEER 0] Finished Satellite 4
[ENGINEER 0] Exiting...
[ENGINEER 1] Finished Satellite 3
[ENGINEER 1] Exiting...
osmancan@vbox:~/shared/HW3$
```

3.2 Priority Queue Test

This test specifically verifies that the priority queue correctly prioritizes satellite requests based on their priority values.

Test Configuration: Modify the satellite function to use fixed priorities instead of random ones:

```
// Modify in satellite() function
// Replace: int priority = rand() % 5 + 1;
// With:
int priority;
if (id == 0) priority = 1;      // Lowest priority
else if (id == 1) priority = 2;
else if (id == 2) priority = 3;
else if (id == 3) priority = 4;
else priority = 5;              // Highest priority
// Also remove the random delay in the satellite thread
// Comment out the line: sleep(rand() % 2);  // Random delay
// Make timeout 4-5 seconds for all satellites
// Replace: request->timeout = rand() % MAX_TIMEOUT + 1;
// With:
request->timeout = rand() % MAX_TIMEOUT + 1;
// Add sleep(3) to engineer thread before while loop
```

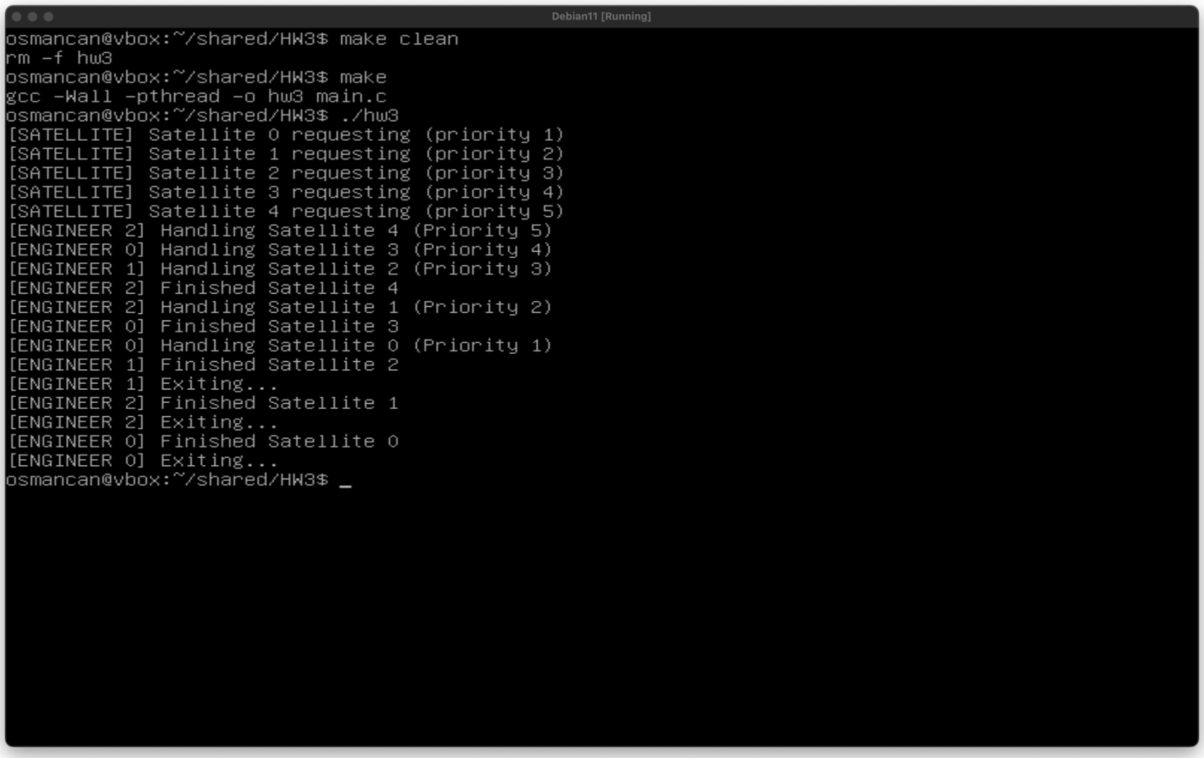
Test Execution:

```
$ make clean

$ make

$ ./hw3
```

Expected Results:

- Satellite 4 (priority 5) should be served first
- Satellite 3 (priority 4) should be served second
- Satellite 2 (priority 3) should be served third
- Satellite 1 (priority 2) should be served fourth
- Satellite 0 (priority 1) should be served last or time out

Actual Results: Matching with the expected result



```
osmancan@vbox:~/shared/HW3$ make clean
rm -f hw3
osmancan@vbox:~/shared/HW3$ make
gcc -Wall -pthread -o hw3 main.c
osmancan@vbox:~/shared/HW3$ ./hw3
[SATELLITE] Satellite 0 requesting (priority 1)
[SATELLITE] Satellite 1 requesting (priority 2)
[SATELLITE] Satellite 2 requesting (priority 3)
[SATELLITE] Satellite 3 requesting (priority 4)
[SATELLITE] Satellite 4 requesting (priority 5)
[ENGINEER 2] Handling Satellite 4 (Priority 5)
[ENGINEER 0] Handling Satellite 3 (Priority 4)
[ENGINEER 1] Handling Satellite 2 (Priority 3)
[ENGINEER 2] Finished Satellite 4
[ENGINEER 2] Handling Satellite 1 (Priority 2)
[ENGINEER 0] Finished Satellite 3
[ENGINEER 0] Handling Satellite 0 (Priority 1)
[ENGINEER 1] Finished Satellite 2
[ENGINEER 1] Exiting...
[ENGINEER 2] Finished Satellite 1
[ENGINEER 2] Exiting...
[ENGINEER 0] Finished Satellite 0
[ENGINEER 0] Exiting...
osmancan@vbox:~/shared/HW3$ _
```

3.3 Timeout Handling Test

This test specifically verifies that the timeout mechanism works correctly for satellites that aren't served in time.

Test Configuration: Modify the timeout and engineer processing time:

```
// Reduce MAX_TIMEOUT to ensure some satellites time out
// In #define section:
#define MAX_TIMEOUT 2  // Reduced from 5 to 2

// Increase engineer processing time
// In engineer() function:
// Replace: sleep(3 + (rand() % 2));
// With:
sleep(5);  // Fixed longer processing time
```

Test Execution:

$ make clean

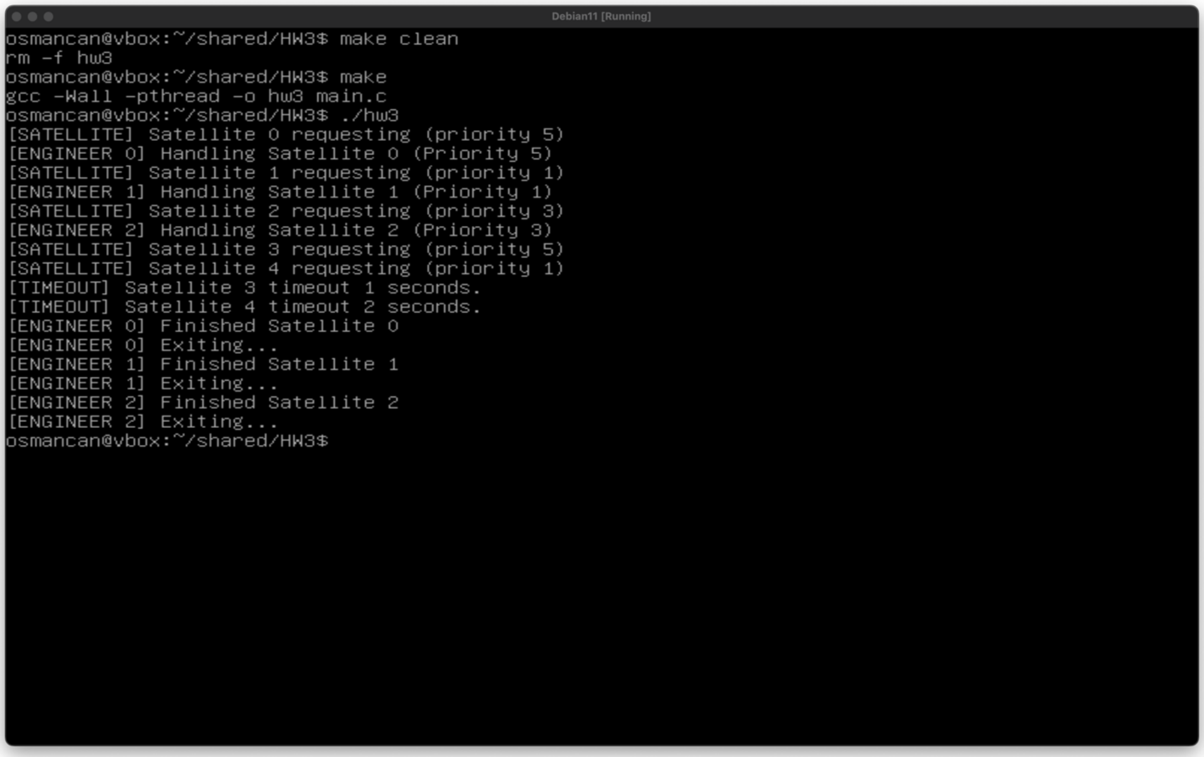
$ make

$ ./hw3

Expected Results:

- With longer processing times and shorter timeouts, more satellites should time out
- The system should handle these timeouts gracefully, removing timed-out requests from the queue
- Engineers should continue processing other requests after timeouts occur

Actual Results: Matching with the expected result



```
osmancan@vbox:~/shared/HW3$ make clean
rm -f hw3
osmancan@vbox:~/shared/HW3$ make
gcc -Wall -pthread -o hw3 main.c
osmancan@vbox:~/shared/HW3$ ./hw3
[SATELLITE] Satellite 0 requesting (priority 5)
[ENGINEER 0] Handling Satellite 0 (Priority 5)
[SATELLITE] Satellite 1 requesting (priority 1)
[ENGINEER 1] Handling Satellite 1 (Priority 1)
[SATELLITE] Satellite 2 requesting (priority 3)
[ENGINEER 2] Handling Satellite 2 (Priority 3)
[SATELLITE] Satellite 3 requesting (priority 5)
[SATELLITE] Satellite 4 requesting (priority 1)
[TIMEOUT] Satellite 3 timeout 1 seconds.
[TIMEOUT] Satellite 4 timeout 2 seconds.
[ENGINEER 0] Finished Satellite 0
[ENGINEER 0] Exiting...
[ENGINEER 1] Finished Satellite 1
[ENGINEER 1] Exiting...
[ENGINEER 2] Finished Satellite 2
[ENGINEER 2] Exiting...
osmancan@vbox:~/shared/HW3$
```

3.4 Stress Test

This test verifies the system's behavior under high load with many more satellites than engineers.

Test Configuration: Modify the number of satellites:

```
// In #define section:
#define NUM_SATELLITES 30  // Increased from 5 to 30
```

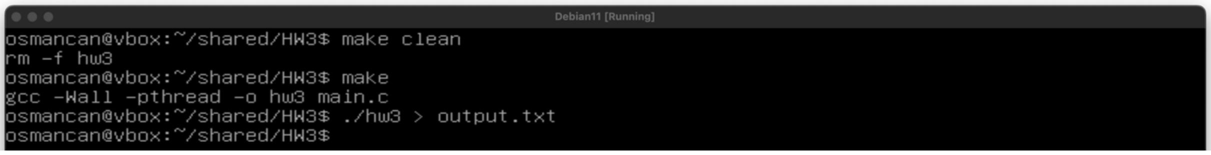
Test Execution:

```
$ make clean

$ make

$ ./hw3  > output.txt
```

Expected Results:

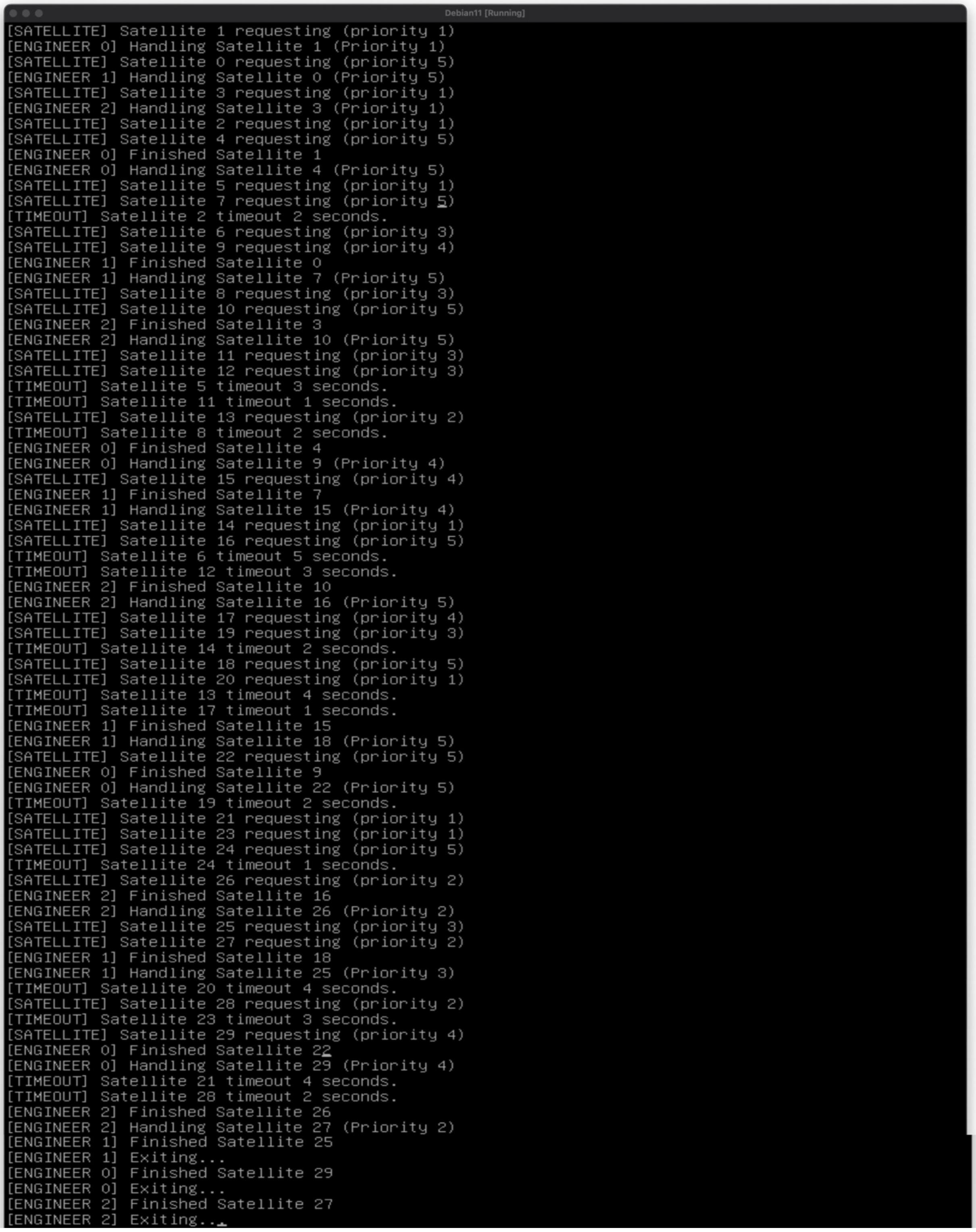
- The system should handle a large number of concurrent requests
- Higher priority satellites should still be served first
- Many satellites will likely time out due to limited engineers
- The system should remain stable and not crash

Actual Results: Matching with the expected result



```
osmancan@vbox:~/shared/HW3$ make clean
rm -f hw3
osmancan@vbox:~/shared/HW3$ make
gcc -Wall -pthread -o hw3 main.c
osmancan@vbox:~/shared/HW3$ ./hw3 > output.txt
osmancan@vbox:~/shared/HW3$
```

output.txt content: (output was too large for terminal window, so it is written to a text file in order to show it fully)


Debian11 [Running]

```
[SATELLITE] Satellite 1 requesting (priority 1)
[ENGINEER 0] Handling Satellite 1 (Priority 1)
[SATELLITE] Satellite 0 requesting (priority 5)
[ENGINEER 1] Handling Satellite 0 (Priority 5)
[SATELLITE] Satellite 3 requesting (priority 1)
[ENGINEER 2] Handling Satellite 3 (Priority 1)
[SATELLITE] Satellite 2 requesting (priority 1)
[SATELLITE] Satellite 4 requesting (priority 5)
[ENGINEER 0] Finished Satellite 1
[ENGINEER 0] Handling Satellite 4 (Priority 5)
[SATELLITE] Satellite 5 requesting (priority 1)
[SATELLITE] Satellite 7 requesting (priority 5)
[TIMEOUT] Satellite 2 timeout 2 seconds.
[SATELLITE] Satellite 6 requesting (priority 3)
[SATELLITE] Satellite 9 requesting (priority 4)
[ENGINEER 1] Finished Satellite 0
[ENGINEER 1] Handling Satellite 7 (Priority 5)
[SATELLITE] Satellite 8 requesting (priority 3)
[SATELLITE] Satellite 10 requesting (priority 5)
[ENGINEER 2] Finished Satellite 3
[ENGINEER 2] Handling Satellite 10 (Priority 5)
[SATELLITE] Satellite 11 requesting (priority 3)
[SATELLITE] Satellite 12 requesting (priority 3)
[TIMEOUT] Satellite 5 timeout 3 seconds.
[TIMEOUT] Satellite 11 timeout 1 seconds.
[SATELLITE] Satellite 13 requesting (priority 2)
[TIMEOUT] Satellite 8 timeout 2 seconds.
[ENGINEER 0] Finished Satellite 4
[ENGINEER 0] Handling Satellite 9 (Priority 4)
[SATELLITE] Satellite 15 requesting (priority 4)
[ENGINEER 1] Finished Satellite 7
[ENGINEER 1] Handling Satellite 15 (Priority 4)
[SATELLITE] Satellite 14 requesting (priority 1)
[SATELLITE] Satellite 16 requesting (priority 5)
[TIMEOUT] Satellite 6 timeout 5 seconds.
[TIMEOUT] Satellite 12 timeout 3 seconds.
[ENGINEER 2] Finished Satellite 10
[ENGINEER 2] Handling Satellite 16 (Priority 5)
[SATELLITE] Satellite 17 requesting (priority 4)
[SATELLITE] Satellite 19 requesting (priority 3)
[TIMEOUT] Satellite 14 timeout 2 seconds.
[SATELLITE] Satellite 18 requesting (priority 5)
[SATELLITE] Satellite 20 requesting (priority 1)
[TIMEOUT] Satellite 13 timeout 4 seconds.
[TIMEOUT] Satellite 17 timeout 1 seconds.
[ENGINEER 1] Finished Satellite 15
[ENGINEER 1] Handling Satellite 18 (Priority 5)
[SATELLITE] Satellite 22 requesting (priority 5)
[ENGINEER 0] Finished Satellite 9
[ENGINEER 0] Handling Satellite 22 (Priority 5)
[TIMEOUT] Satellite 19 timeout 2 seconds.
[SATELLITE] Satellite 21 requesting (priority 1)
[SATELLITE] Satellite 23 requesting (priority 1)
[SATELLITE] Satellite 24 requesting (priority 5)
[TIMEOUT] Satellite 24 timeout 1 seconds.
[SATELLITE] Satellite 26 requesting (priority 2)
[ENGINEER 2] Finished Satellite 16
[ENGINEER 2] Handling Satellite 26 (Priority 2)
[SATELLITE] Satellite 25 requesting (priority 3)
[SATELLITE] Satellite 27 requesting (priority 2)
[ENGINEER 1] Finished Satellite 18
[ENGINEER 1] Handling Satellite 25 (Priority 3)
[TIMEOUT] Satellite 20 timeout 4 seconds.
[SATELLITE] Satellite 28 requesting (priority 2)
[TIMEOUT] Satellite 23 timeout 3 seconds.
[SATELLITE] Satellite 29 requesting (priority 4)
[ENGINEER 0] Finished Satellite 22
[ENGINEER 0] Handling Satellite 29 (Priority 4)
[TIMEOUT] Satellite 21 timeout 4 seconds.
[TIMEOUT] Satellite 28 timeout 2 seconds.
[ENGINEER 2] Finished Satellite 26
[ENGINEER 2] Handling Satellite 27 (Priority 2)
[ENGINEER 1] Finished Satellite 25
[ENGINEER 1] Exiting...
[ENGINEER 0] Finished Satellite 29
[ENGINEER 0] Exiting...
[ENGINEER 2] Finished Satellite 27
[ENGINEER 2] Exiting...
```

3.5 Edge Case: Equal Priorities Test

This test verifies how the system handles satellites with equal priorities.

Test Configuration: Modify the satellite function to assign the same priority to all satellites:

```c
// In satellite() function:
// Replace: int priority = rand() % 5 + 1;
// With:
int priority = 3;  // Fixed equal priority for all satellites
```

Test Execution:

$ make clean

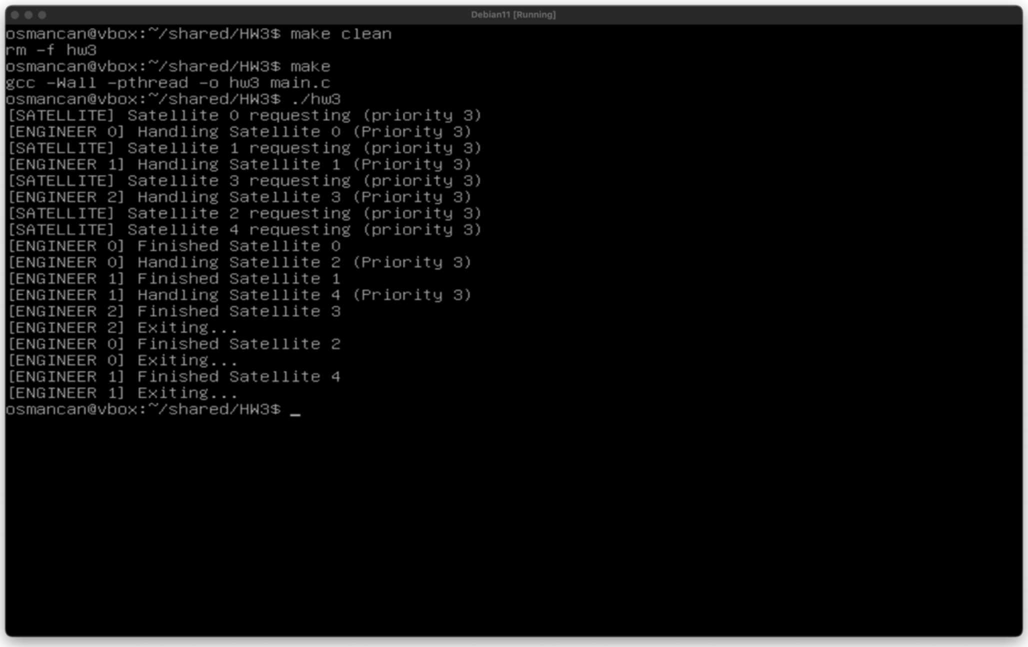
$ make

$ ./hw3

Expected Results:

- With equal priorities, satellites should be served in the order they arrive
- The first satellites to request should be the first to be served
- The system should handle equal priorities correctly

Actual Results: Matching with the expected result



```
osmancan@vbox:~/shared/HW3$ make clean
rm -f hw3
osmancan@vbox:~/shared/HW3$ make
gcc -Wall -pthread -o hw3 main.c
osmancan@vbox:~/shared/HW3$ ./hw3
[SATELLITE] Satellite 0 requesting (priority 3)
[ENGINEER 0] Handling Satellite 0 (Priority 3)
[SATELLITE] Satellite 1 requesting (priority 3)
[ENGINEER 1] Handling Satellite 1 (Priority 3)
[SATELLITE] Satellite 3 requesting (priority 3)
[ENGINEER 2] Handling Satellite 3 (Priority 3)
[SATELLITE] Satellite 2 requesting (priority 3)
[SATELLITE] Satellite 4 requesting (priority 3)
[ENGINEER 0] Finished Satellite 0
[ENGINEER 0] Handling Satellite 2 (Priority 3)
[ENGINEER 1] Finished Satellite 1
[ENGINEER 1] Handling Satellite 4 (Priority 3)
[ENGINEER 2] Finished Satellite 3
[ENGINEER 2] Exiting...
[ENGINEER 0] Finished Satellite 2
[ENGINEER 0] Exiting...
[ENGINEER 1] Finished Satellite 4
[ENGINEER 1] Exiting...
osmancan@vbox:~/shared/HW3$ _
```

3.6 Memory Leak Test

This test verifies that the system doesn't leak memory during operation.

Test Configuration: No code modifications required

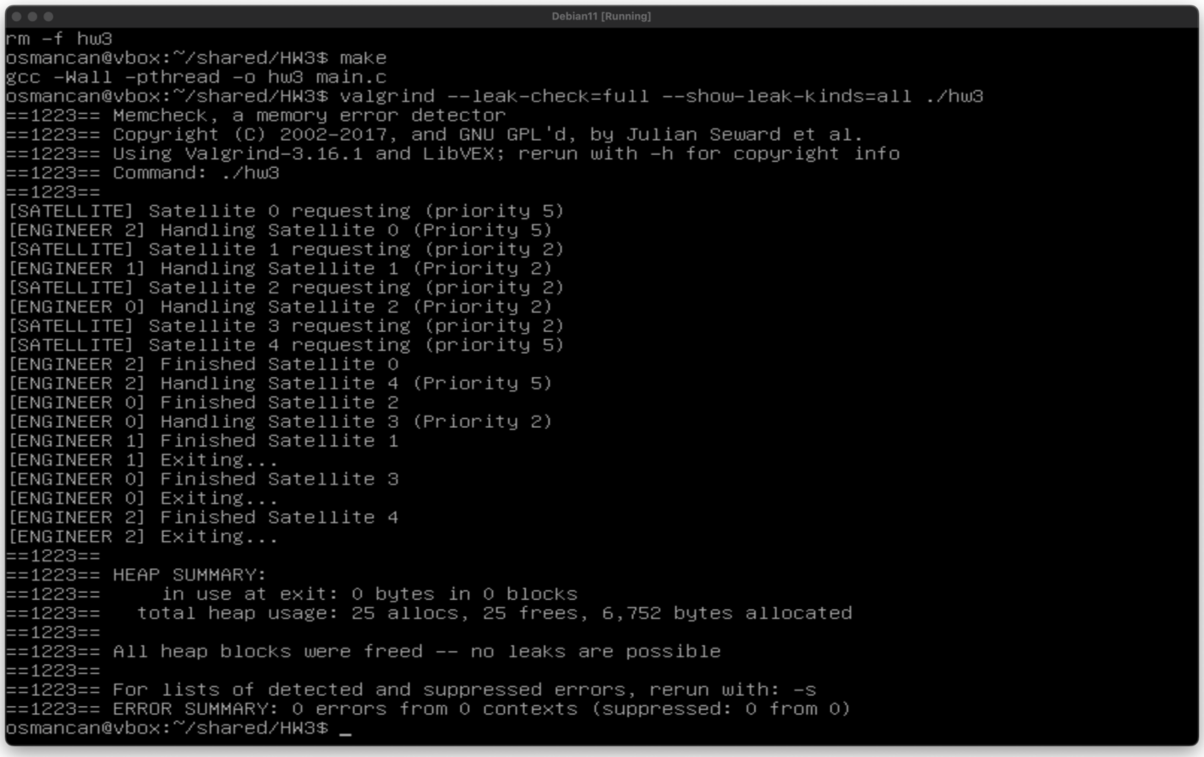
Test Execution:

```
$ make clean

$ make

$ valgrind --leak-check=full --show-leak-kinds=all ./hw3
```

Expected Results:

- Valgrind should report no memory leaks
- All allocated memory should be properly freed
- All semaphores and mutexes should be properly destroyed

Actual Results: Matching with the expected result



```
rm -f hw3
osmancan@vbox:~/shared/HW3$ make
gcc -Wall -pthread -o hw3 main.c
osmancan@vbox:~/shared/HW3$ valgrind --leak-check=full --show-leak-kinds=all ./hw3
==1223== Memcheck, a memory error detector
==1223== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1223== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==1223== Command: ./hw3
==1223==
[SATELLITE] Satellite 0 requesting (priority 5)
[ENGINEER 2] Handling Satellite 0 (Priority 5)
[SATELLITE] Satellite 1 requesting (priority 2)
[ENGINEER 1] Handling Satellite 1 (Priority 2)
[SATELLITE] Satellite 2 requesting (priority 2)
[ENGINEER 0] Handling Satellite 2 (Priority 2)
[SATELLITE] Satellite 3 requesting (priority 2)
[SATELLITE] Satellite 4 requesting (priority 5)
[ENGINEER 2] Finished Satellite 0
[ENGINEER 2] Handling Satellite 4 (Priority 5)
[ENGINEER 0] Finished Satellite 2
[ENGINEER 0] Handling Satellite 3 (Priority 2)
[ENGINEER 1] Finished Satellite 1
[ENGINEER 1] Exiting...
[ENGINEER 0] Finished Satellite 3
[ENGINEER 0] Exiting...
[ENGINEER 2] Finished Satellite 4
[ENGINEER 2] Exiting...
==1223==
==1223== HEAP SUMMARY:
==1223==     in use at exit: 0 bytes in 0 blocks
==1223==   total heap usage: 25 allocs, 25 frees, 6,752 bytes allocated
==1223==
==1223== All heap blocks were freed -- no leaks are possible
==1223==
==1223== For lists of detected and suppressed errors, rerun with: -s
==1223== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
osmancan@vbox:~/shared/HW3$ _
```

Conclusion:

The satellite ground station synchronization system successfully implements a priority-based concurrent processing system for satellite requests. The system effectively uses:

- **Multithreading**: Separate threads for satellites and engineers allow concurrent operations
- **Synchronization Mechanisms**: Mutexes and semaphores coordinate access to shared resources
- **Priority Queue**: Ensures higher priority satellites are served first
- **Timeout Handling**: Gracefully handles cases where engineers aren't available within the connection window

The implementation demonstrates several important concepts in concurrent programming:

1. **Resource Management**: The system carefully tracks available engineers and allocates them to satellite requests.
2. **Priority-Based Scheduling**: Higher priority satellites are given preference, which is crucial in real-world scenarios where some communications are more critical than others.
3. **Bounded Waiting**: Satellites have a maximum wait time before abandoning their request
4. **Proper Cleanup**: All resources (memory, semaphores, mutexes) are properly cleaned up.