**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

|  |
| --- |
| 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) {  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

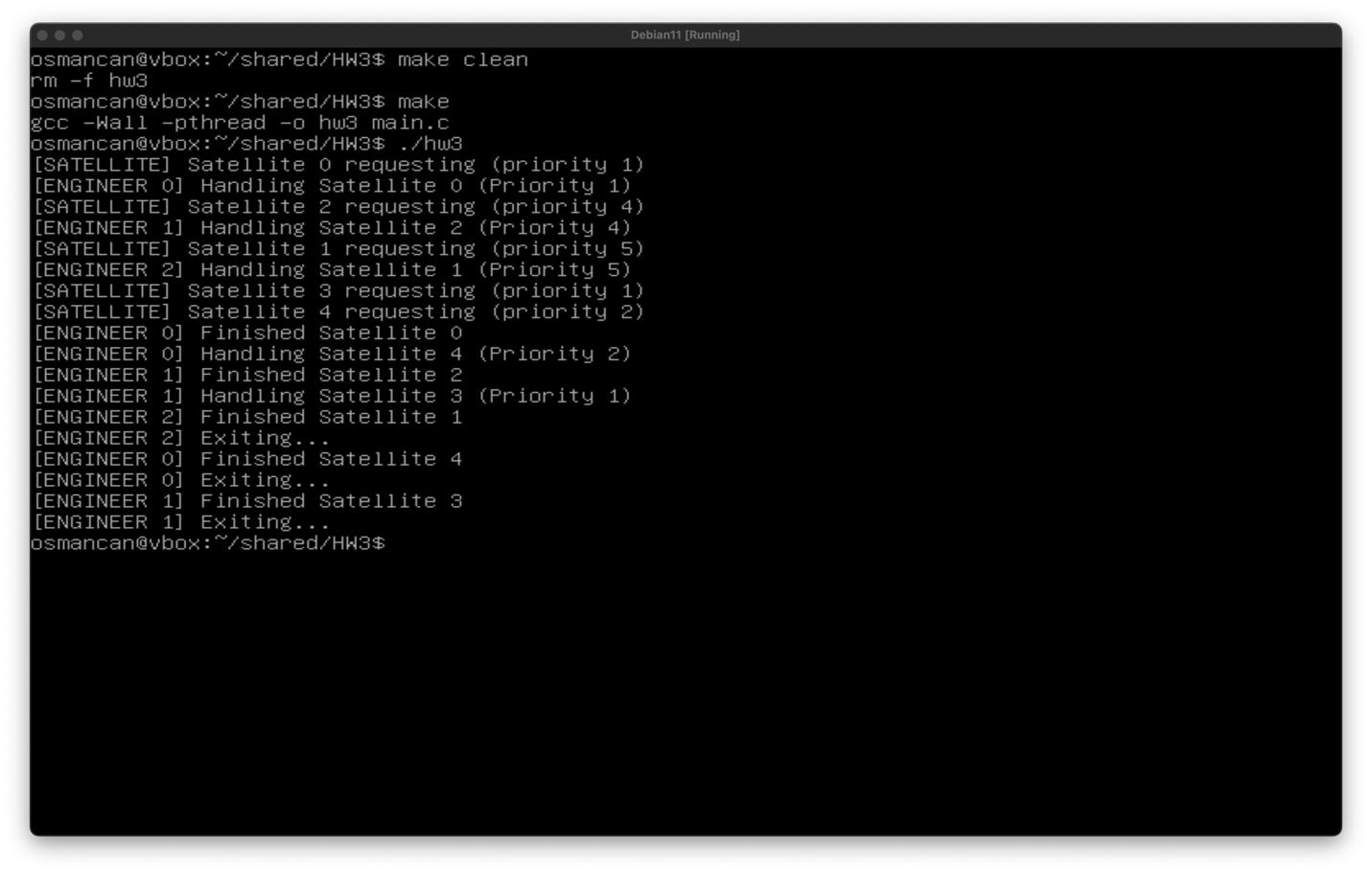
$ make

$ ./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



#### **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

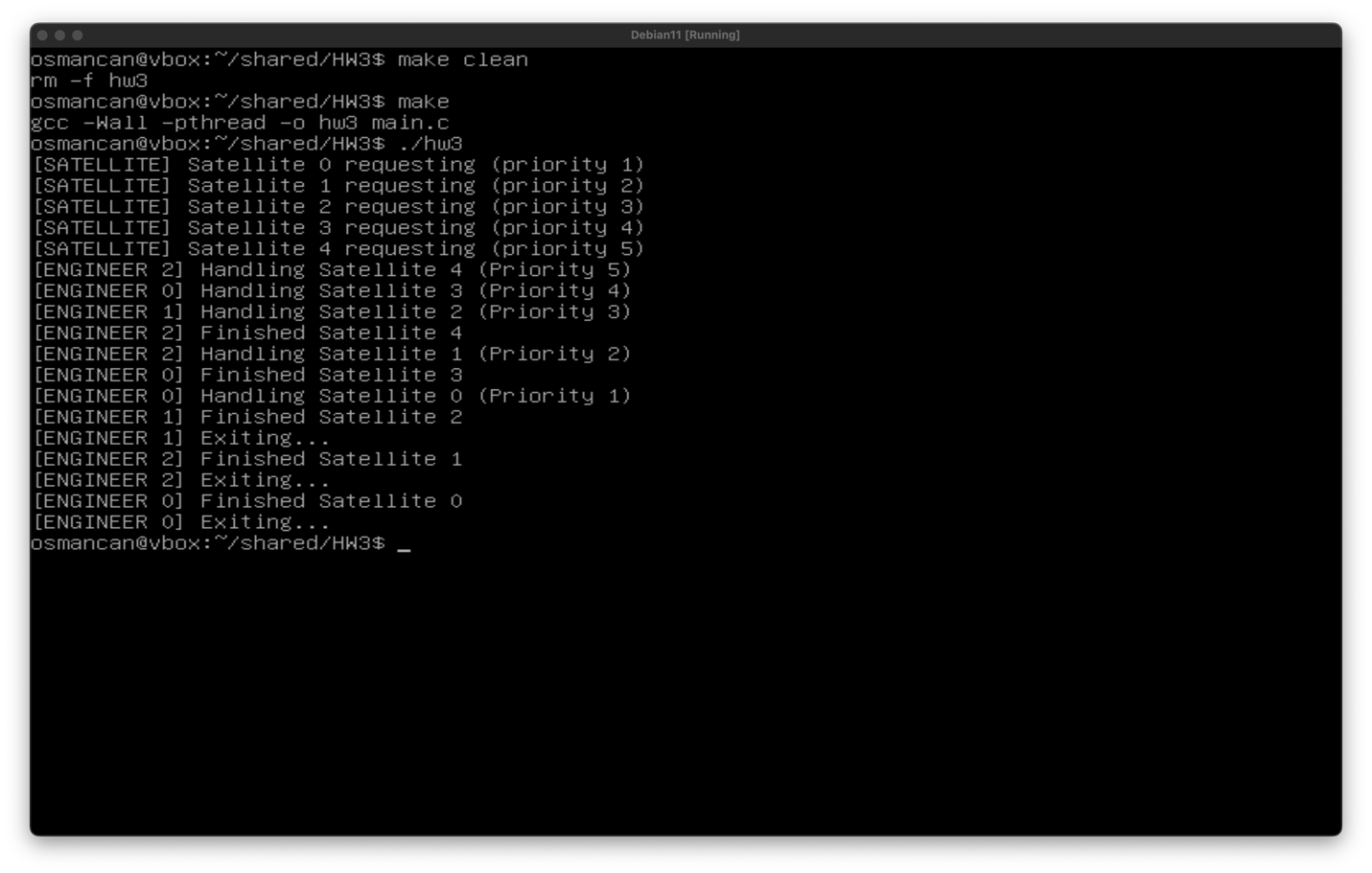
$ make

$ ./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



#### **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

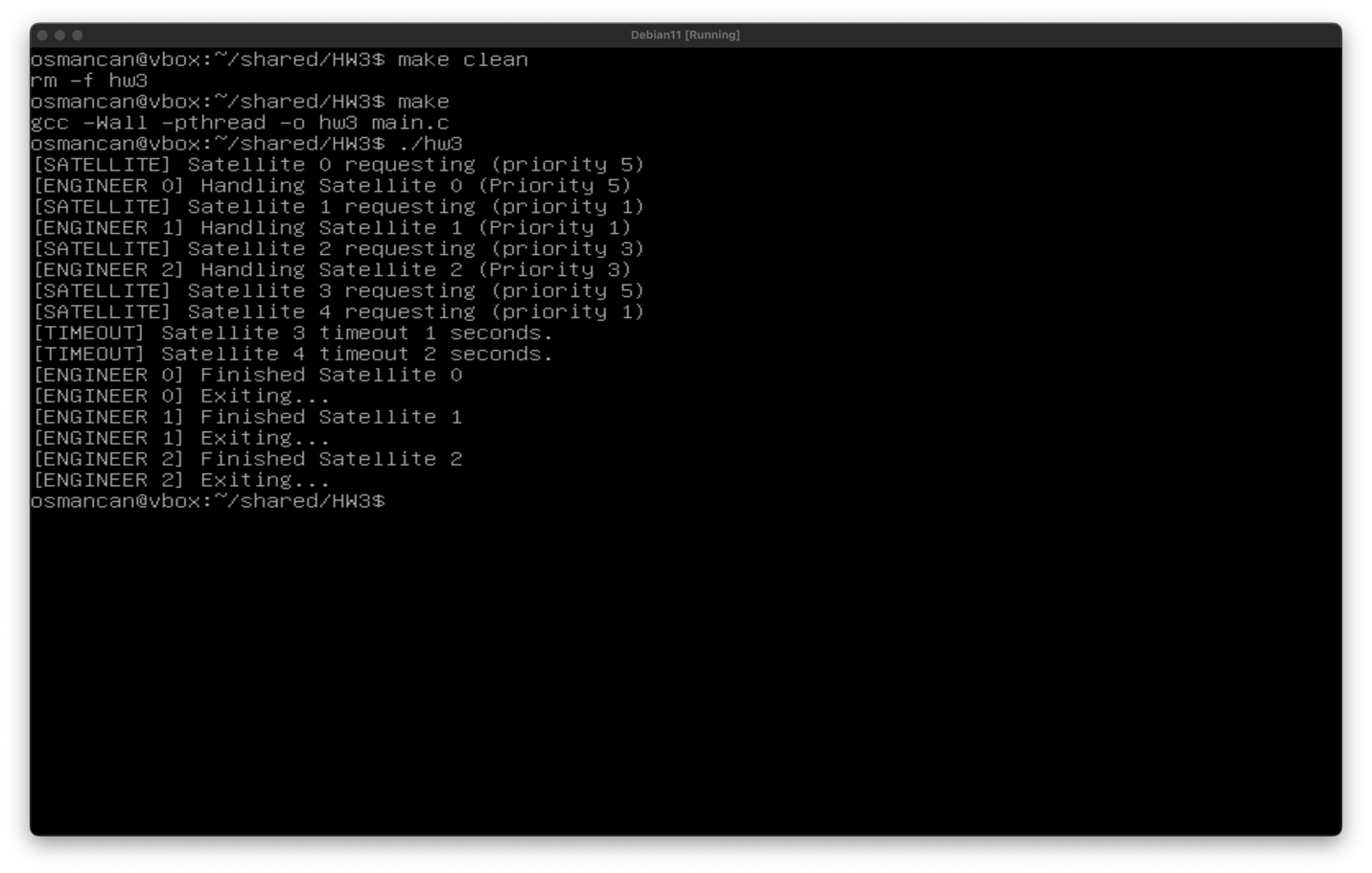
$ make

$ ./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



#### **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

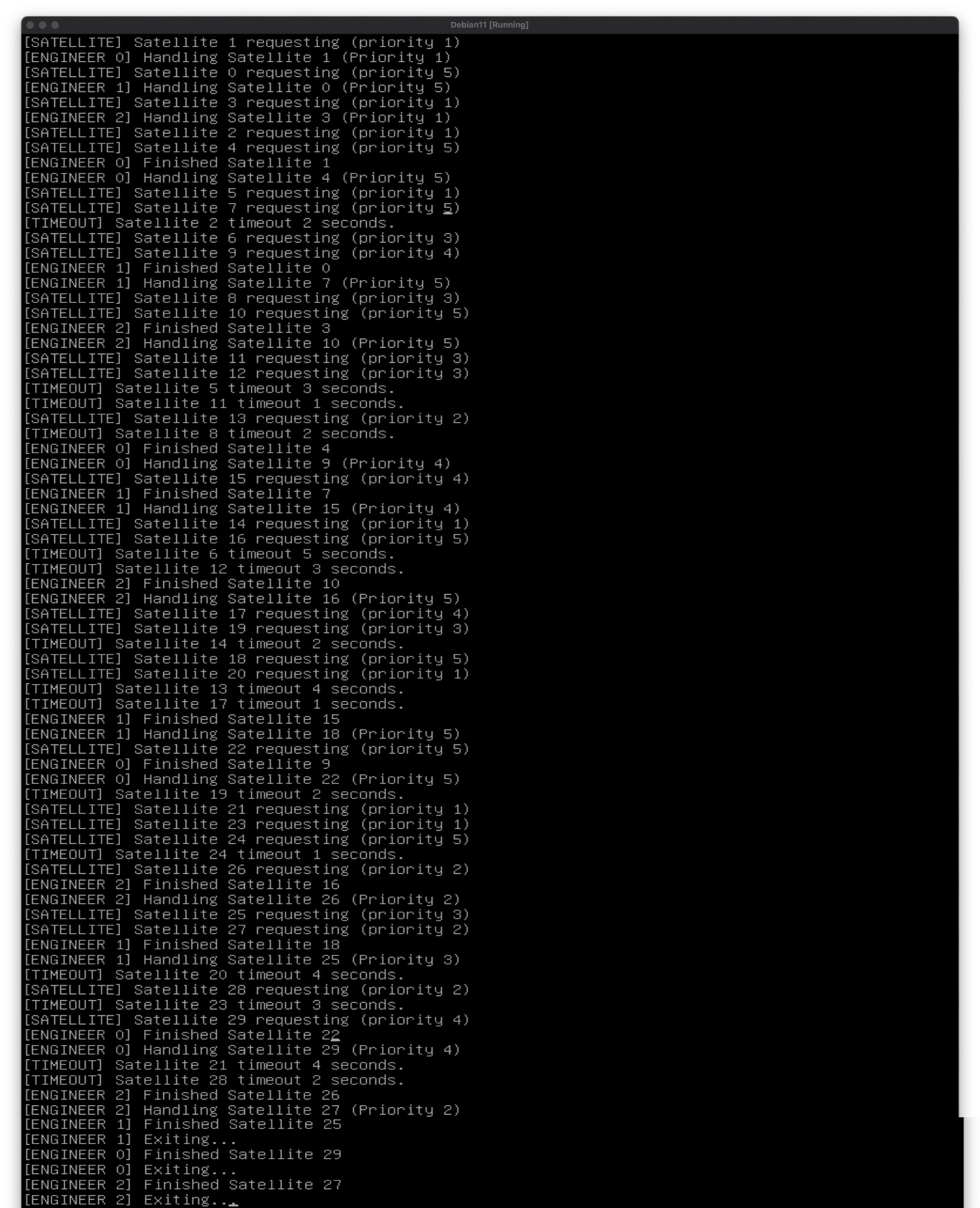
**Expected Results:**

* 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



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



#### **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:

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

**Test Execution:**

$ make clean

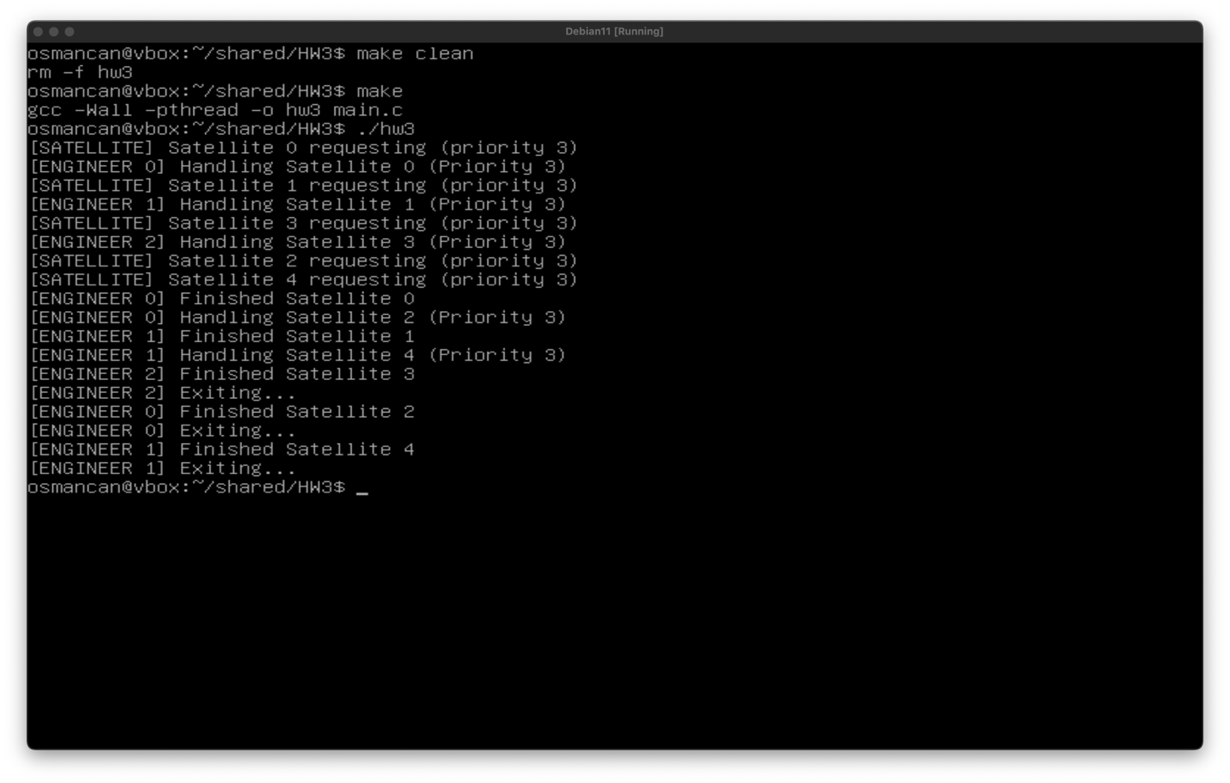
$ make

$ ./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



#### **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

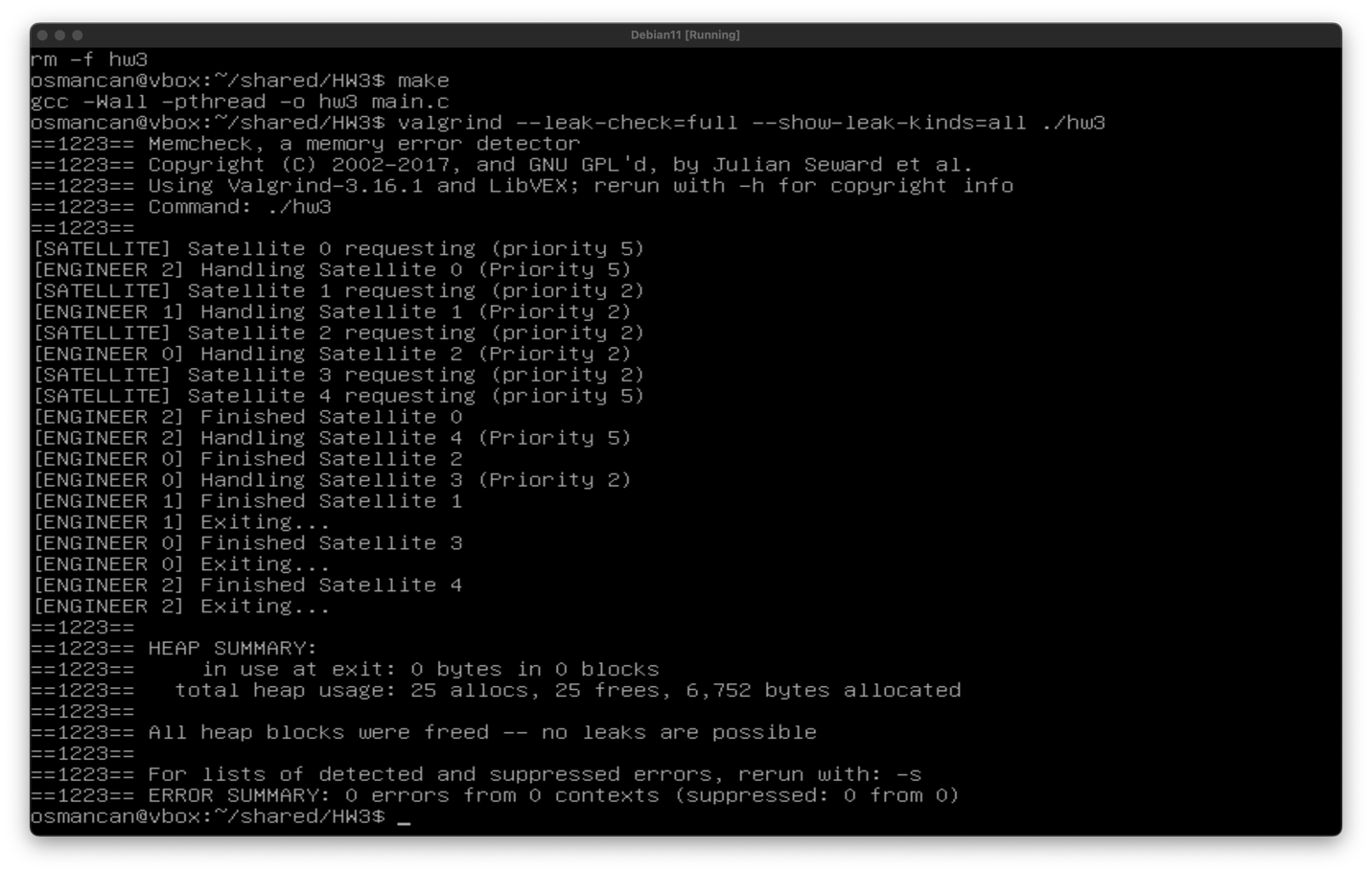
$ make

$ 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



## **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.