CSE344 -- HOMEWORK #3

23/04/2025

Cemal BOLAT

STUDENT NO: 210104004010

Introduction:

This project presents the implementation and analysis of a multi-threaded simulation of a satellite ground station. In this system, satellites with varying priority levels request service from a limited number of engineers within strict connection windows. The main objective is to simulate a real-time scheduling scenario where higher-priority satellites are served before lower-priority ones, using thread synchronization techniques. To achieve this, the system is designed using two types of threads: satellite() and engineer(), which represent the clients (satellites) and servers (engineers), respectively. Synchronization is handled through semaphores and mutexes to ensure safe access to shared resources, including a priority queue for satellite requests and a counter for available engineers. The simulation emphasizes critical real-time constraints and demonstrates how concurrency mechanisms such as semaphores and mutexes can be used to manage access, prevent race conditions, and prioritize service requests. The entire system is tested under various scenarios to verify correctness, responsiveness, and adherence to the described requirements.

TESTING (all scenerios created by me):

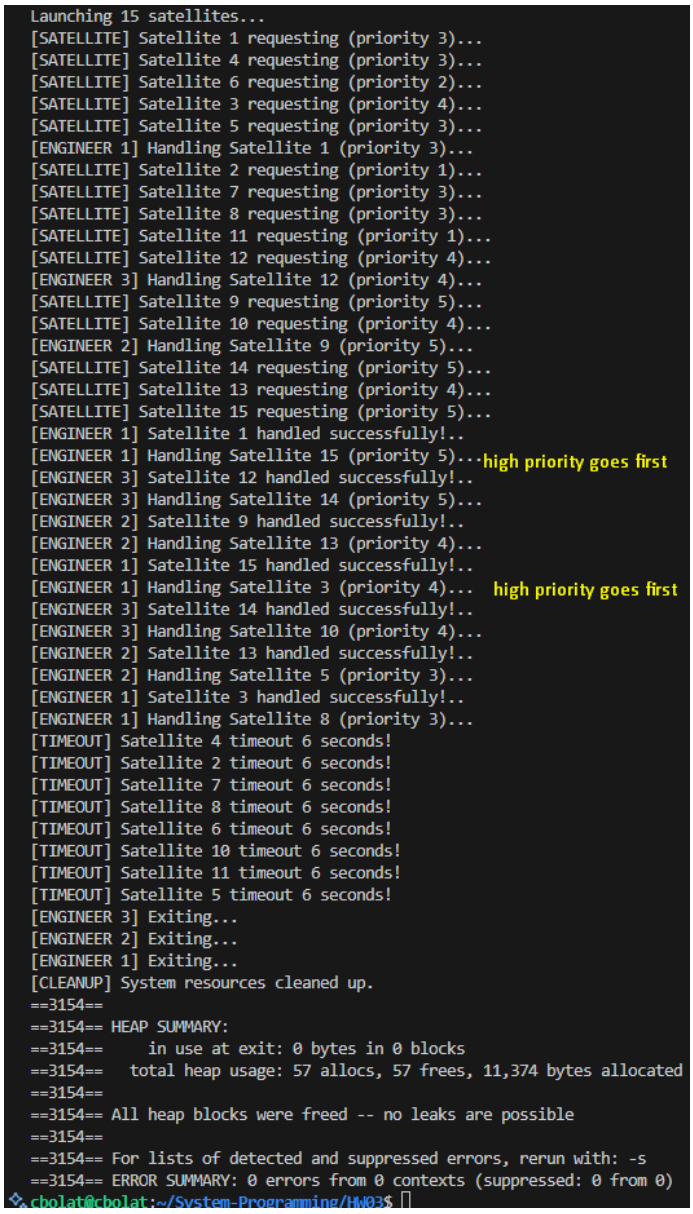
TESTING WITHOUT PROBLEM

```
cbolat@cbolat:~/System-Programming/HW03$ valgrind --tool=memcheck ./SatelliteSystem
==4008== Memcheck, a memory error detector
==4008== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4008== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==4008== Command: ./SatelliteSystem
==4008==
Launching 5 satellites...
[SATELLITE] Satellite 1 requesting (priority 4)...
[ENGINEER 1] Handling Satellite 1 (priority 4)...
[SATELLITE] Satellite 2 requesting (priority 5)...
[SATELLITE] Satellite 3 requesting (priority 1)...
[ENGINEER 2] Handling Satellite 2 (priority 5)...
[SATELLITE] Satellite 4 requesting (priority 4)...
[ENGINEER 3] Handling Satellite 4 (priority 4)...
[SATELLITE] Satellite 5 requesting (priority 1)...
[ENGINEER 1] Finished Satellite 1
[ENGINEER 1] Handling Satellite 5 (priority 1)...
[ENGINEER 3] Finished Satellite 4
[ENGINEER 3] Handling Satellite 3 (priority 1)...
[ENGINEER 2] Finished Satellite 2
[ENGINEER 1] Finished Satellite 5
[ENGINEER 3] Finished Satellite 3
[ENGINEER 2] Exiting...
[ENGINEER 3] Exiting...
[ENGINEER 1] Exiting...
[CLEANUP] System resources cleaned up.
==4008==
==4008== HEAP SUMMARY:
==4008==     in use at exit: 0 bytes in 0 blocks
==4008==   total heap usage: 27 allocs, 27 frees, 7,774 bytes allocated
==4008==
==4008== All heap blocks were freed -- no leaks are possible
==4008==
==4008== For lists of detected and suppressed errors, rerun with: -s
==4008== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@cbolat:~/System-Programming/HW03$
```

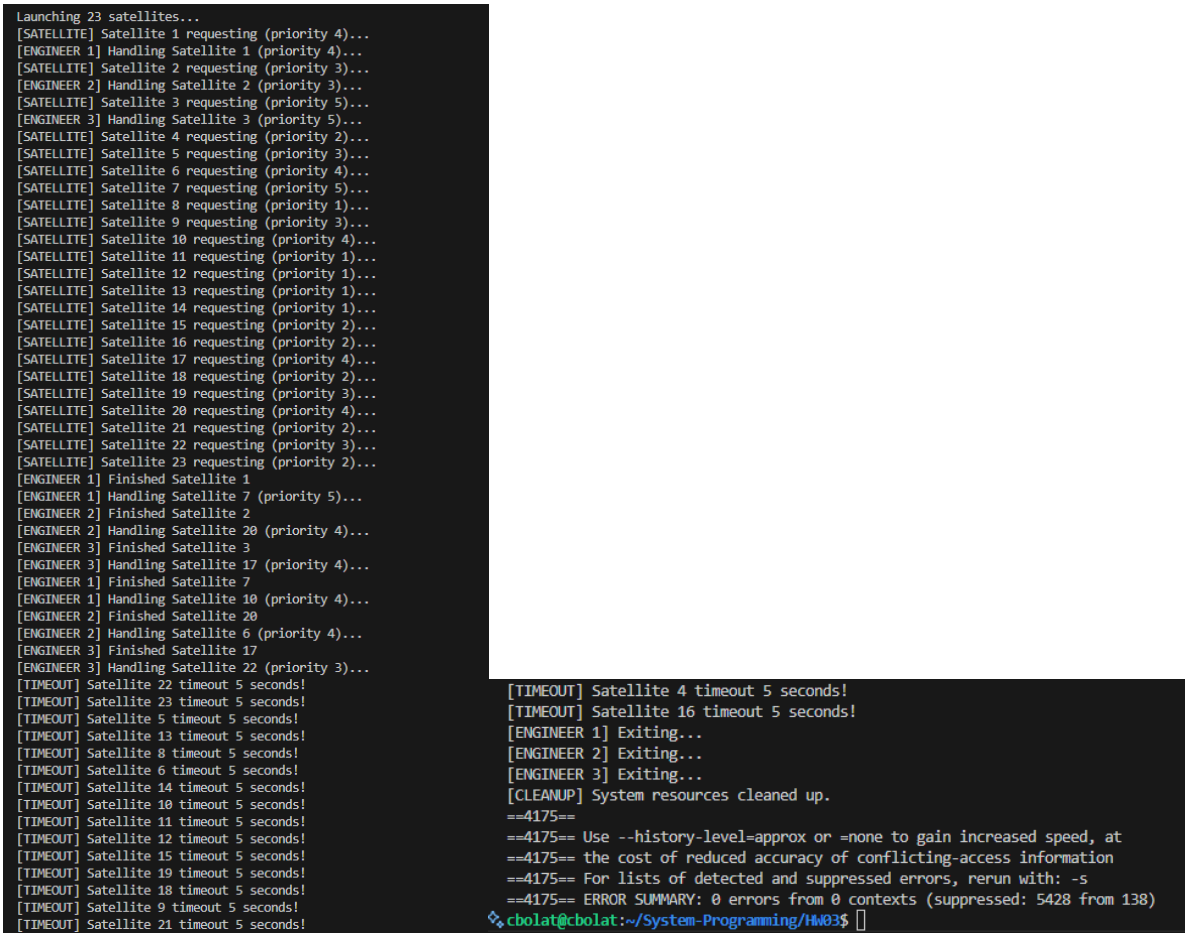
```
cbolat@cbolat:~/System-Programming/HW03$ valgrind --tool=helgrind ./SatelliteSystem
==4069== Helgrind, a thread error detector
==4069== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==4069== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==4069== Command: ./SatelliteSystem
==4069==
Launching 3 satellites...
[SATELLITE] Satellite 1 requesting (priority 4)...
[ENGINEER 1] Handling Satellite 1 (priority 4)...
[SATELLITE] Satellite 2 requesting (priority 5)...
[ENGINEER 2] Handling Satellite 2 (priority 5)...
[SATELLITE] Satellite 3 requesting (priority 2)...
[ENGINEER 3] Handling Satellite 3 (priority 2)...
[ENGINEER 1] Finished Satellite 1
[ENGINEER 2] Finished Satellite 2
[ENGINEER 3] Finished Satellite 3
[ENGINEER 3] Exiting...
[ENGINEER 2] Exiting...
[ENGINEER 1] Exiting...
[CLEANUP] System resources cleaned up.
==4069==
==4069== Use --history-level=approx or =none to gain increased speed, at
==4069== the cost of reduced accuracy of conflicting-access information
==4069== For lists of detected and suppressed errors, rerun with: -s
==4069== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 464 from 103)
cbolat@cbolat:~/System-Programming/HW03$
```

Valgrind output is shows there is no leaks & data races on no purposes

TESTING MORE SATELLITES WITH LOW TIMEOUT TIME TO SEE TIMEOUTS



Data race check for next page...



Valgrind output is shows there is no leaks & data races on no purposes

TESTING MORE SATELLITES WITH HIGHTIMEOUT TIME TO SEE ENGINEERS CAN HANDLE A LOT SATELLITES

```
cbolat@cbolat:~/System-Programming/HW03$ valgrind --tool=helgrind ./SatelliteSystem
==3773== Helgrind, a thread error detector
==3773== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==3773== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==3773== Command: ./SatelliteSystem
==3773==
Launching 10 satellites...
[SATELLITE] Satellite 1 requesting (priority 2)...
[ENGINEER 1] Handling Satellite 1 (priority 2)...
[SATELLITE] Satellite 2 requesting (priority 5)...
[ENGINEER 2] Handling Satellite 2 (priority 5)...
[SATELLITE] Satellite 3 requesting (priority 4)...
[ENGINEER 3] Handling Satellite 3 (priority 4)...
[SATELLITE] Satellite 4 requesting (priority 3)...
[SATELLITE] Satellite 5 requesting (priority 3)...
[SATELLITE] Satellite 6 requesting (priority 5)...
[SATELLITE] Satellite 7 requesting (priority 4)...
[SATELLITE] Satellite 8 requesting (priority 1)...
[SATELLITE] Satellite 9 requesting (priority 1)...
[SATELLITE] Satellite 10 requesting (priority 3)...
[ENGINEER 1] Finished Satellite 1
[ENGINEER 1] Handling Satellite 6 (priority 5)...
[ENGINEER 2] Finished Satellite 2
[ENGINEER 2] Handling Satellite 7 (priority 4)...
[ENGINEER 3] Finished Satellite 3
[ENGINEER 3] Handling Satellite 10 (priority 3)...
[ENGINEER 1] Finished Satellite 6
[ENGINEER 1] Handling Satellite 5 (priority 3)...
[ENGINEER 2] Finished Satellite 7
[ENGINEER 2] Handling Satellite 4 (priority 3)...
[ENGINEER 3] Finished Satellite 10
[ENGINEER 3] Handling Satellite 9 (priority 1)...
[ENGINEER 1] Finished Satellite 5
[ENGINEER 1] Handling Satellite 8 (priority 1)...
[ENGINEER 2] Finished Satellite 4
[ENGINEER 3] Finished Satellite 9
[ENGINEER 1] Finished Satellite 8
[ENGINEER 3] Exiting...
[ENGINEER 2] Exiting...
[ENGINEER 1] Exiting...
[CLEANUP] System resources cleaned up.
==3773==
==3773== Use --history-level=approx or =none to gain increased speed, at
==3773== the cost of reduced accuracy of conflicting-access information
==3773== For lists of detected and suppressed errors, rerun with: -s
==3773== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 1833 from 113)
cbolat@cbolat:~/System-Programming/HW03$
```

Clearly see that priority goes high to low after first 3 requests.

Valgrind output is shows there is no leaks & data races on no purposes

Project Development Steps:

The development of the satellite ground station simulation project was carried out through a structured, step-by-step approach to ensure thread-safe operations and correct priority-based scheduling. Below is a detailed breakdown of the main development stages:

Data Initialization
The `t_data` structure is initialized to hold shared resources such as the satellite list, engineer threads, priority queue head, and synchronization primitives. The number of satellites is randomly generated using the current system time to introduce variability in each simulation run.

Memory Allocation and Safety Checks
Dynamic memory is allocated for the satellites array based on the generated satellite count. Safety checks are applied to ensure successful memory allocation. In case of any allocation failure, the program gracefully exits after cleanup.

Synchronization Setup
A mutex (`engineerMutex`) is initialized to protect shared data, particularly the `availableEngineers` counter and the `requestQueue`. A semaphore named `newRequest` is also initialized to synchronize communication between satellites and engineers, signaling when a new request is available.

Engineer Thread Creation
Three engineer threads are created using `pthread_create()`, each represented by the `engineer()` function. These threads continuously wait for satellite requests and serve the one with the highest priority from the queue.

Satellite Thread Creation
A number of satellite threads are launched, each with a unique ID. The `satellite()` function simulates each satellite attempting to connect and request service within a time-constrained window.

Thread Synchronization and Cleanup
After all satellite threads are joined, a `stop` flag is set to signal the engineers to exit. Each engineer thread is then released via `sem_post()`. Finally, all engineer threads are joined, and resources are properly freed using the `cleanup()` function to prevent memory leaks.

MAIN FUNCTION:

```c
int main() {
    t_data data;
    data.requestQueueHead = NULL;
    data.stop = 0;
    data.availableEngineers = MAX_ENGINEERS;

    // Initialize random seed using current time
    time_t t = time(NULL);
    if (t == ((time_t)-1)) {
        perror("time");
        return EXIT_FAILURE;
    }


    srand(t);

    // Randomly determine the number of satellites to create
    data.satelliteCount = (rand() % MAX_SATELLITES) + 1;
    data.satellites = (Satellites*)malloc(sizeof(Satellites) * data.satelliteCount);
    if (data.satellites == NULL) {
        perror("Failed to allocate memory for satellite threads");
        cleanup(&data);
        return EXIT_FAILURE;
    }

    // Initialize synchronization primitives
    // Mutex for protecting shared engineer availability data
    if (pthread_mutex_init(&data.engineerMutex, NULL) != 0) {
        perror("pthread_mutex_init");
        cleanup(&data);
        exit(EXIT_FAILURE);
    }

    // Semaphore for signaling new requests to engineer threads
    if (sem_init(&data.newRequest, 0, 0) != 0) {
        perror("sem_init");
        cleanup(&data);
        exit(EXIT_FAILURE);
    }

    printf("Launching %d satellites...\n", data.satelliteCount);
```

```c

    // Create engineer threads
    for (int i = 0; i < MAX_ENGINEERS; i++) {
        data.engineers[i].id = i + 1;
        if (pthread_create(&data.engineers[i].threadId, NULL, engineer, &data) != 0) {
            perror("Failed to create engineer thread");
            cleanup(&data);
            exit(EXIT_FAILURE);
        }
    }

    // Create satellite threads
    for (int i = 0; i < data.satelliteCount; i++) {
        data.satellites[i].id = i + 1;
        if (pthread_create(&data.satellites[i].threadId, NULL, satellite, &data) != 0) {
            perror("Failed to create satellite thread");
            cleanup(&data);
            exit(EXIT_FAILURE);
        }
    }

    // Wait for all satellite threads to finish
    for (int i = 0; i < data.satelliteCount; i++) {
        if (pthread_join(data.satellites[i].threadId, NULL) != 0) {
            perror("Failed to join satellite thread");
        }
    }

    data.stop = 1; // Signal engineers to stop processing

    // Notify engineers to wake up and check for the stop signal
    for (int i = 0; i < MAX_ENGINEERS; i++) {
        if (sem_post(&data.newRequest) != 0) {
            perror("sem_post: newRequest");
        }
    }

    // Wait for all engineer threads to finish
    for (int i = 0; i < MAX_ENGINEERS; i++) {
        pthread_join(data.engineers[i].threadId, NULL);
    }

    cleanup(&data); // Clean up resources

    return 0;
}
```

Satellite Thread Implementation

Each satellite is represented by a thread executing the satellite() function. This function models the behavior of a satellite attempting to connect with the ground station and request support from an engineer. The process includes preparing a request, entering the priority queue, and either receiving service within a timeout window or leaving the system.

Step-by-Step Execution:

1. Request Allocation and Identification
   A new SatelliteRequest structure is dynamically allocated. The satellite identifies itself by comparing its thread ID to those stored in data->satellites. This ID is essential for logging and queue operations.
2. Request Initialization
   Each request is assigned:
   a. A unique id
   b. A random priority between 1 and 5 (where higher numbers likely indicate more critical satellites)
   c. The arrivalTime
   d. Flags valid = 1 and being_processed = 0 for later queue management
3. Semaphore Initialization
   A semaphore requestHandled is initialized. It will later be signaled by an engineer thread when this satellite's request is picked up.
4. Request Submission
   The request is pushed into the shared priority queue using insertRequest(), under mutex protection (engineerMutex). This ensures thread-safe queue manipulation.
5. Signal Engineer Threads
   The newRequest semaphore is posted to alert idle engineers that a new request is available.
6. Timeout Handling
   A timeout is configured using clock_gettime() and sem_timedwait() to enforce the maximum wait time (TIMEOUT_SECONDS) for the satellite. If the satellite is not served within this time:
   a. A timeout message is printed
   b. The valid flag is set to 0
   c. If the request is still pending (not picked by an engineer), it is removed from the queue and resources are freed
7. Cleanup on Success
   If the request is handled in time, the satellite cleans up its semaphore and frees its request object under mutex protection to prevent race conditions during simultaneous access.

SATELLITE FUNCTION:

```c
void* satellite(void* arg) {
    t_data* data = (t_data*)arg;

    // Allocate memory for satellite request
    SatelliteRequest* request = malloc(sizeof(SatelliteRequest));
    if (request == NULL) {
        perror("malloc: SatelliteRequest");
        pthread_exit(NULL);
    }

    // Get the satellite ID from the thread ID
    int satelliteId = -1;
    pthread_t threadId = pthread_self();
    for (int i = 0; i < data->satelliteCount; i++) {
        if (data->satellites[i].threadId == threadId) {
            satelliteId = data->satellites[i].id;
            break;
        }
    }

    // Initialize the request
    request->id = satelliteId;
    request->priority = rand() % 5 + 1;
    request->arrivalTime = time(NULL);
    request->valid = 1;
    request->being_processed = 0;

    // Initialize the semaphore for request handling
    if (sem_init(&request->requestHandled, 0, 0) != 0) {
        perror("sem_init: requestHandled");
        free(request);
        pthread_exit(NULL);
    }

    printf("[SATELLITE] Satellite %d requesting (priority %d)...\n", request->id, request->priority);

    pthread_mutex_lock(&data->engineerMutex); // Lock the mutex to protect shared data
    insertRequest(data, request); // Insert the request into the queue
    pthread_mutex_unlock(&data->engineerMutex);

    // Signal to the engineer that a new request is available
    if (sem_post(&data->newRequest) != 0) {
        perror("sem_post: newRequest");
        pthread_exit(NULL);
    }
```

CONTINUES...

```c
    struct timespec ts;
    if (clock_gettime(CLOCK_REALTIME, &ts) == -1) { //
        perror("clock_gettime");
        pthread_exit(NULL);
    }
    ts.tv_sec += TIMEOUT_SECONDS;

    // Wait for the engineer to handle the request or timeout
    int waitResult = sem_timedwait(&request->requestHandled, &ts);
    if (waitResult == -1) {
        if (errno == ETIMEDOUT) { // Check if the wait timed out
            printf("[TIMEOUT] Satellite %d timeout %d seconds!\n", request->id, TIMEOUT_SECONDS);

            pthread_mutex_lock(&data->engineerMutex); // Lock the mutex to protect shared data

            request->valid = 0;

            if (!request->being_processed) { // If the request is not being processed, remove it from the queue
                removeRequestFromQueue(data, request->id);

                sem_destroy(&request->requestHandled);

                free(request);
            }

            pthread_mutex_unlock(&data->engineerMutex);
        }
        else {
            perror("sem_timedwait");
            sem_destroy(&request->requestHandled); // Destroy the semaphore if wait failed
            free(request);
        }

        pthread_exit(NULL);
        return NULL;
    }

    // If the request was handled successfully, wait for the engineer to finish processing
    pthread_mutex_lock(&data->engineerMutex);
    sem_destroy(&request->requestHandled);
    free(request);
    pthread_mutex_unlock(&data->engineerMutex);
    pthread_exit(NULL);
    return NULL;
}
```

Engineer Thread Implementation

The engineer() function models the behavior of an engineer at the ground station who waits for incoming satellite requests and serves them based on their priority. This function runs in a loop, responding to new requests signaled by the newRequest semaphore.

Step-by-Step Execution:

1. **Engineer Identification**
   Each engineer thread first determines its own ID by matching its thread ID against those stored in the data->engineers array. This allows for informative logging and better thread traceability.
2. **Main Processing Loop**
   The engineer thread enters an infinite loop where it waits for a signal that a new satellite request is available:
   a. It calls sem_wait(&data->newRequest), blocking until a satellite posts a new request.
   b. If the data->stop flag is set, the loop is exited, and the thread prepares to terminate.
3. **Request Handling (Priority Queue)**
   Under mutex protection (engineerMutex), the engineer retrieves the highest priority request using popHighestPriorityRequest(). If no request is found (which can rarely happen due to race conditions), a warning is logged and the loop continues.
4. **Request Processing**
   Once a valid request is obtained:
   a. The being_processed flag is set to indicate the engineer has taken ownership of the request.
   b. The availableEngineers counter is decremented to reflect that this engineer is now busy.
   c. The engineer simulates work using sleep(PROCESSING_TIME).
5. **Post-Processing and Acknowledgement**
   After processing:
   a. The engineer checks if the request is still valid (it may have timed out or been removed).
   b. If valid, the requestHandled semaphore is posted, signaling the corresponding satellite that its request was successfully completed.
   c. If invalid (e.g., satellite gave up waiting), the request's resources are cleaned up directly.
   d. The availableEngineers counter is incremented to show availability.

```c
void* engineer(void* arg) {
    t_data* data = (t_data*)arg;

    int engineerId = -1;
    pthread_t threadId = pthread_self();
    for (int i = 0; i < MAX_ENGINEERS; i++) {
        if (data->engineers[i].threadId == threadId) {
            engineerId = data->engineers[i].id;
            break;
        }
    }

    while (1) {
        if (sem_wait(&data->newRequest) != 0) {
            perror("sem_wait: newRequest");
            pthread_exit(NULL);
        }

        if (data->stop) {
            break;
        }

        pthread_mutex_lock(&data->engineerMutex);
        SatelliteRequest* request = popHighestPriorityRequest(data);

        if (request == NULL) {
            pthread_mutex_unlock(&data->engineerMutex);
            fprintf(stderr, "[ENGINEER %d] Warning: No request found in queue.\n", engineerId);
            continue;
        }

        request->being_processed = 1;
        int request_id = request->id;
        int request_priority = request->priority;
        int request_valid = request->valid;

        data->availableEngineers--;
        printf("[ENGINEER %d] Handling Satellite %d (priority %d)...\n",
               engineerId, request_id, request_priority);
        pthread_mutex_unlock(&data->engineerMutex);


        sleep(PROCESSING_TIME);

        pthread_mutex_lock(&data->engineerMutex);

        if (request_valid && request->valid) {
            printf("[ENGINEER %d] Finished Satellite %d\n",
                   engineerId, request_id);

            if (sem_post(&request->requestHandled) != 0) {
                perror("sem_post: requestHandled");
            }
        } else {
            sem_destroy(&request->requestHandled);
            free(request);
            data->availableEngineers++;

            pthread_mutex_unlock(&data->engineerMutex);
            break;
        }

        request->being_processed = 0;

        data->availableEngineers++;

        pthread_mutex_unlock(&data->engineerMutex);
    }

    printf("[ENGINEER %d] Exiting...\n", engineerId);
    pthread_exit(NULL);
    return NULL;
}
```

CLEANUP FUNCTION:

```
5   void cleanup(t_data* data) {
6       if (data == NULL) {
7           fprintf(stderr, "cleanup: Null data pointer!\n");
8           return;
9       }
10
11      pthread_mutex_lock(&data->engineerMutex);
12      Node* current = data->requestQueueHead;
13      while (current != NULL) {
14          Node* temp = current;
15          if (current->request != NULL) {
16              sem_destroy(&current->request->requestHandled);
17              free(current->request);
18          }
19          current = current->next;
20          free(temp);
21      }
22      data->requestQueueHead = NULL;
23      pthread_mutex_unlock(&data->engineerMutex);
24
25      if (data->satellites != NULL) {
26          free(data->satellites);
27          data->satellites = NULL;
28      }
29
30      if (pthread_mutex_destroy(&data->engineerMutex) != 0) {
31          perror("pthread_mutex_destroy");
32      }
33
34      if (sem_destroy(&data->newRequest) != 0) {
35          perror("sem_destroy");
36      }
37
38
39      printf("[CLEANUP] System resources cleaned up.\n");
40  }
```