CSE 344

Homework #3

200104004074

Bedirhan Ömer Aksoy

1) Introduction

In my implementation, main function creates engineer threads first, then creates satellite threads with 1 second delay between them and after the jobs done, joins the satellite threads, then wake engineers as there are new requests to trigger the break condition from the while loop and stop them waiting for new request, then joins all engineer threads after ensuring no threads waiting for any semaphore. Then the program destroys the requestsHandled semaphore array and newRequest semaphore. Finally, program free's all requestQueue nodes one by one.

Engineer function runs in an infinite while loop and waits for a new request. When a request arrives, first it checks if any satellites are active and if the request queue is null. If they are, it breaks the while loop and exits since there is no any satellite requests to process anymore. If not, in a mutex lock to prevent other engineers to process the same satellite, it pops a request from the requestQueue, prints the "[ENGINEER %d] handling…" message, sleeps a random time up to 7 seconds (I made it 7 to test the priority queue since 5 seconds is too short to make a queue for 5 satellites with 1 second waiting between their creation). Then checks for the requestQueue whether it's empty or not, if it's not empty, posts a newRequest to update available engineers to try again.

Satellite function sets it's priority between 1 to 5 (1 is most prior and 5 is least prior) randomly. Then prints the request text on the console, then inserts request with inserRequest(req) function and posts newRequest semaphore to wake available engineers to handle the request. Then function waits for the requestHandled semaphore with the time struct to check the timeout condition. After request handled or timeout occurred, function decreases activeSatellites variable in a mutex lock and returns null.

2) Code Explanation

2.0) global variables:

I have defined NUM_SATELLITES, NUM_ENGINEERS, MAX_TIMEOUT variables to make these macros dynamic. Request struct defined for making requests as an object which has satellite_id and priority features. Node struct has defined for handling request queue as a request linked list in a queue based on their priority.

I have initialized the availableEngineers and activeSatellites variables as the NUM_ENGINEERS and NUMB_SATELLITES macros at the beginning since all engineers are available and all satellites are active at the beginning. requestQueue head node initialized as NULL. Initialized engineerMutex to be sure one engineer handles one satellite in a mutex lock. newRequest and requestHandled[] semaphores are initialized for waking threads and mutexes.

```
8   #define NUM_SATELLITES 5
9   #define NUM_ENGINEERS 3
10  #define MAX_TIMEOUT 10
11
12  typedef struct {
13      int satellite_id;
14      int priority;
15  } Request;
16
17  typedef struct Node {
18      Request request;
19      struct Node* next;
20  } Node;
21
22  int availableEngineers = NUM_ENGINEERS;
23  Node* requestQueue = NULL;
24  pthread_mutex_t engineerMutex;
25  sem_t newRequest;
26  sem_t requestHandled[NUM_SATELLITES];
27  int activeSatellites = NUM_SATELLITES;
```

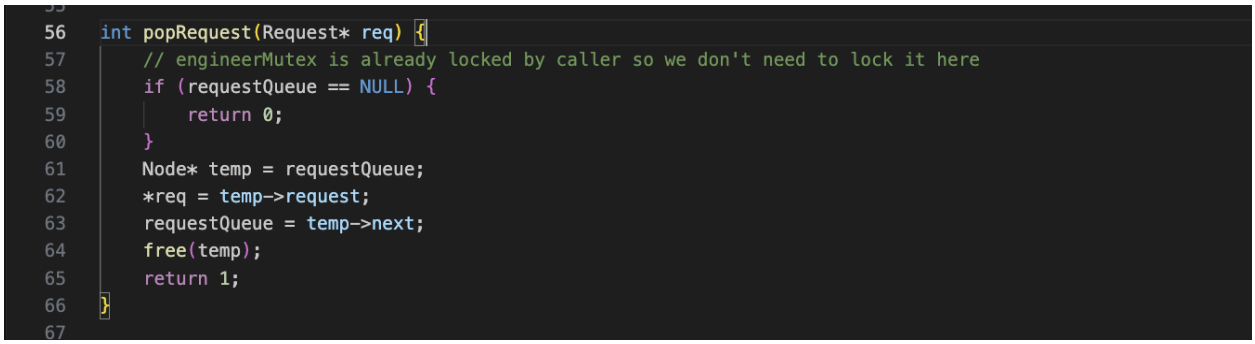
2.1) void insertRequest(Request req): This function inserts new request to requestQueue inside of an engineerMutex lock, since this function does not called inside the engineer() function, we need to apply engineerMutex lock to ensure the security of the requesQueue that only one satellite request written to the requestQueue linked list at a time.

Function first allocates memory for the new node, then locks the engineerMutex and enters critical region. If the queue is empty or new request has a lower number priority (higher priority), new node becomes the head node. If not, it means the requestQueue is not empty or the new node has higher number (lower priority) and it traverses the whole queue until it finds a node with greater or equal priority node to insert before it in the queue. After it's done, unlocks the engineerMutex.

```
29  void insertRequest(Request req) {
30      Node* newNode = (Node*)malloc(sizeof(Node));
31      if (!newNode) {
32          fprintf(stderr, "Memory allocation failed\n");
33          exit(1);
34      }
35      newNode->request = req;
36      newNode->next = NULL;
37
38      pthread_mutex_lock(&engineerMutex);
39      // insert at head if queue is empty OR new req has lower number (higher priority)
40      if (requestQueue == NULL || req.priority < requestQueue->request.priority) {
41          newNode->next = requestQueue;
42          requestQueue = newNode;
43      }
44      else {
45          Node* current = requestQueue;
46          // traverse until finding node with greater or equal priority
47          while (current->next != NULL && current->next->request.priority <= req.priority) {
48              current = current->next;
49          }
50          newNode->next = current->next;
51          current->next = newNode;
52      }
53      pthread_mutex_unlock(&engineerMutex);
54  }
```

2.2) int popRequest(Request* req): This function has called inside of the engineer() function so engineerMutex is already locked. That's why we don't need to lock it again, the program is in the critical section already and only one engineer processes the requestQueue.

It first checks whether the requestQueue node (head node) is NULL or not. If it is, returns 0. If not, creates a temp Request node, then assigns the req* pointer to the head node (highest priority node of the queue), then assigns the next node as the new head node and free's the temp node, then returns 1 to indicate that the pop request was handled successfully.



```
56  int popRequest(Request* req) {
57      // engineerMutex is already locked by caller so we don't need to lock it here
58      if (requestQueue == NULL) {
59          return 0;
60      }
61      Node* temp = requestQueue;
62      *req = temp->request;
63      requestQueue = temp->next;
64      free(temp);
65      return 1;
66  }
67
```

2.3) void* satellite(void* arg): Satellite function sets it's id from the argument it takes, then sets it's priority randomly between 1 to 5 and prints the satellite request message to console. Then satellite inserts a request with insertRequest function and posts newRequest semaphore to wake engineers up to handle the request.

Then function gets the time and calculates the timeout time and waits it's request to handled in the given time in a sem_timedwait() function with using the requestHandled[id] semaphore to track it's request's status.

If it's not handled in the given time, it prints the timeout message (and removes it's request from the requestQueue by calling removeRequest() function. After satellite time is out or request is handled, it locks the engineerMutex to decrease activeSatellites variable, after decreasing, unlocks the engineerMutex and returns NULL.

```
68  void* satellite(void* arg) {
69      int id = *(int*)arg;
70      srand(time(NULL) ^ id);
71      int priority = (rand() % 5) + 1;
72      Request req = {id, priority};
73
74      printf("[SATELLITE %d] requesting (priority %d)\n", id, priority);
75
76      insertRequest(req);
77      sem_post(&newRequest);
78
79      struct timespec ts;
80      clock_gettime(CLOCK_REALTIME, &ts);
81      ts.tv_sec += MAX_TIMEOUT;
82      if (sem_timedwait(&requestHandled[id], &ts) == -1) {
83          printf("[TIMEOUT] Satellite %d timeout %d second.\n", id, MAX_TIMEOUT);
84      }
85
86      pthread_mutex_lock(&engineerMutex);
87      activeSatellites--;
88      pthread_mutex_unlock(&engineerMutex);
89
90      return NULL;
91  }
```

2.4) void* engineer(void* arg): This function runs in an infinite while loop and engineers waits for newRequest semaphore in a sem_wait. If a request wakes an engineer and locks the engineerMutex to check if there are any active satellites and if the requestQueue is empty or not, then unlocks the engineerMutex lock. If there are active satellites, it locks the engineerMutex again and checks for available engineers again and pops the most prior request with popRequest function.

If engineer gets request, first it posts requestHandled[req.satellite_id] semaphore to wake the waiting satellite to prevent timeout. Then it prints the engineer handling satellite message and sleeps a random time between 1 to 5 seconds. After the sleep, engineer prints that engineer has finished the satellite work on the console. Then engineer locks the engineerMutex to update available engineers since it's job done and can receive new task, checks the requestQueue node whether it's NULL or not.

If it's not null, engineer posts newRequest to wake other engineers and unlocks the engineerMutex to end the critical region. If engineer could not get newRequest, it posts newRequest semaphore again to allow other engineers to try. If there are no active satellites and requestQueue is NULL, engineer breaks the infinite loop and prints exiting message and returns null.

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

    while (1) {
        sem_wait(&newRequest);

        // check if we should exit
        pthread_mutex_lock(&engineerMutex);
        if (activeSatellites == 0 && requestQueue == NULL) {
            pthread_mutex_unlock(&engineerMutex);
            break;
        }
        pthread_mutex_unlock(&engineerMutex);

        Request req;
        int gotRequest = 0;

        // try to process a request
        pthread_mutex_lock(&engineerMutex);
        if (availableEngineers > 0 && popRequest(&req)) {
            availableEngineers--;
            gotRequest = 1;
        }
        pthread_mutex_unlock(&engineerMutex);

        if (gotRequest) {
            // handle the request
            printf("[ENGINEER %d] Handling Satellite %d (Priority %d)\n", id, req.satellite_id, req.priority);
            sleep((rand() % 5) + 1);
            printf("[ENGINEER %d] Finished Satellite %d\n", id, req.satellite_id);
            sem_post(&requestHandled[req.satellite_id]);

            // update available engineers
            pthread_mutex_lock(&engineerMutex);
            availableEngineers++;
            if (requestQueue != NULL) {
                sem_post(&newRequest);
            }
            pthread_mutex_unlock(&engineerMutex);
        }
        else {
            // repost newRequest to allow other engineers to try
            sem_post(&newRequest);
        }
    }

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

2.5) int main(): Function defines satellite_threads and engineer_threads, satellite_ids and engineer_ids integer arrays to hold satellite and engineer ids.After initializing engineerMutex as NULL and initializing newRequest semaphore, function initializes requestHandled[] semaphore array.

Then engineer threads are created with &engineer_threads[i] thread and &engineer_ids[i] integer in a for loop from zero to NUM_ENGINEERS. Then satellite threads are created with &satellite_threads[i] and &satellite_ids[i] integer in a for loop from zero to NUM_SATELLITES. Between each satellite creation, function waits 0.1 seconds.

After creating engineer and satellite threads, function waits for satellite threads to terminate and join threads in a for loop one by one. After joining satellite threads, function posts newRequest semaphore to trigger the break condition in engineer threads to print exit message and terminate inside of the engineerMutex lock.

After terminating engineer threads, function joins engineer threads in a loop. Then function destroys the engineerMutex and newRequest semaphore. Then destroys requestHandled[] semaphore array in a for loop. Then function free's the requestQueue elements one by one to ensure all allocated blocks are free'd and main function returns 0 to terminate the program.

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

    pthread_mutex_init(&engineerMutex, NULL);
    sem_init(&newRequest, 0, 0);
    for (int i = 0; i < NUM_SATELLITES; i++) {
        sem_init(&requestHandled[i], 0, 0);
    }

    for (int i = 0; i < NUM_ENGINEERS; i++) {
        engineer_ids[i] = i;
        pthread_create(&engineer_threads[i], NULL, engineer, &engineer_ids[i]);
    }

    for (int i = 0; i < NUM_SATELLITES; i++) {
        satellite_ids[i] = i;
        pthread_create(&satellite_threads[i], NULL, satellite, &satellite_ids[i]);
        usleep(100000);
    }

    for (int i = 0; i < NUM_SATELLITES; i++) {
        pthread_join(satellite_threads[i], NULL);
    }

    // wake engineers to check exit condition
    pthread_mutex_lock(&engineerMutex);
    if (activeSatellites == 0 && requestQueue == NULL) {
        for (int i = 0; i < NUM_ENGINEERS; i++) {
            sem_post(&newRequest);
        }
    }
    pthread_mutex_unlock(&engineerMutex);

    // join engineer threads
    for (int i = 0; i < NUM_ENGINEERS; i++) {
        pthread_join(engineer_threads[i], NULL);
    }

    pthread_mutex_destroy(&engineerMutex);
    sem_destroy(&newRequest);
    for (int i = 0; i < NUM_SATELLITES; i++) {
        sem_destroy(&requestHandled[i]);
    }

    while (requestQueue != NULL) {
        Node* temp = requestQueue;
        requestQueue = requestQueue->next;
        free(temp);
    }

    return 0;
}
```

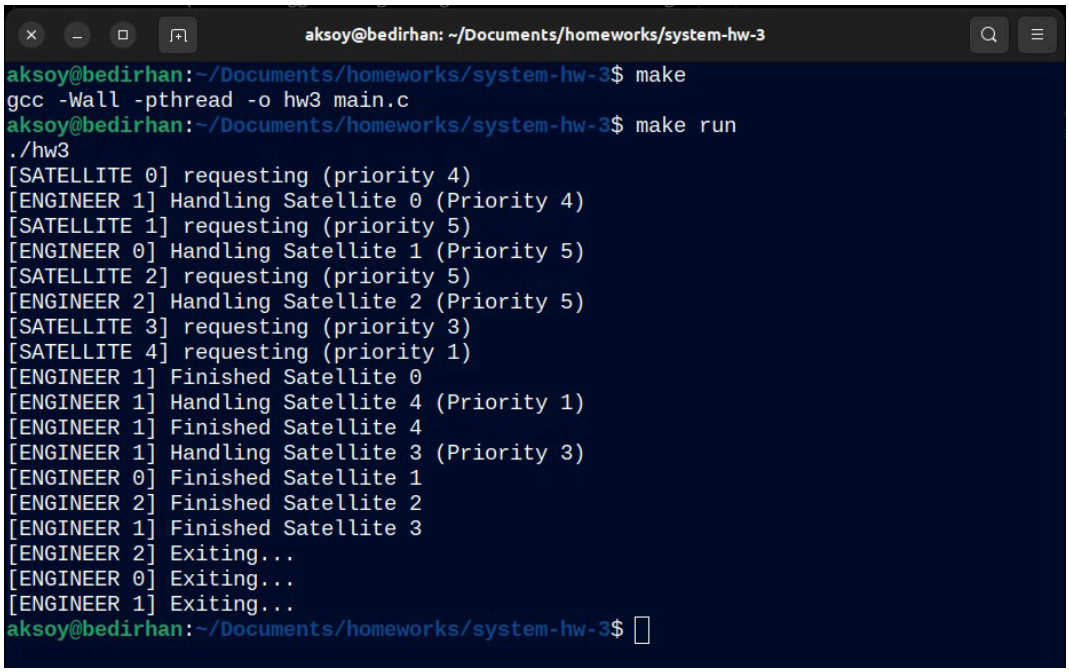
2.6) void removeRequest(int satellite_id): I have realized that when a satellite goes timeout, I'm not removing that satellite from the requestQueue, so that the engineers still handles the timeouted satellites. So I have implemented a removeRequest function to remove timeouted satellites between a mutex lock to prevent the handling of the timeouted satellites.

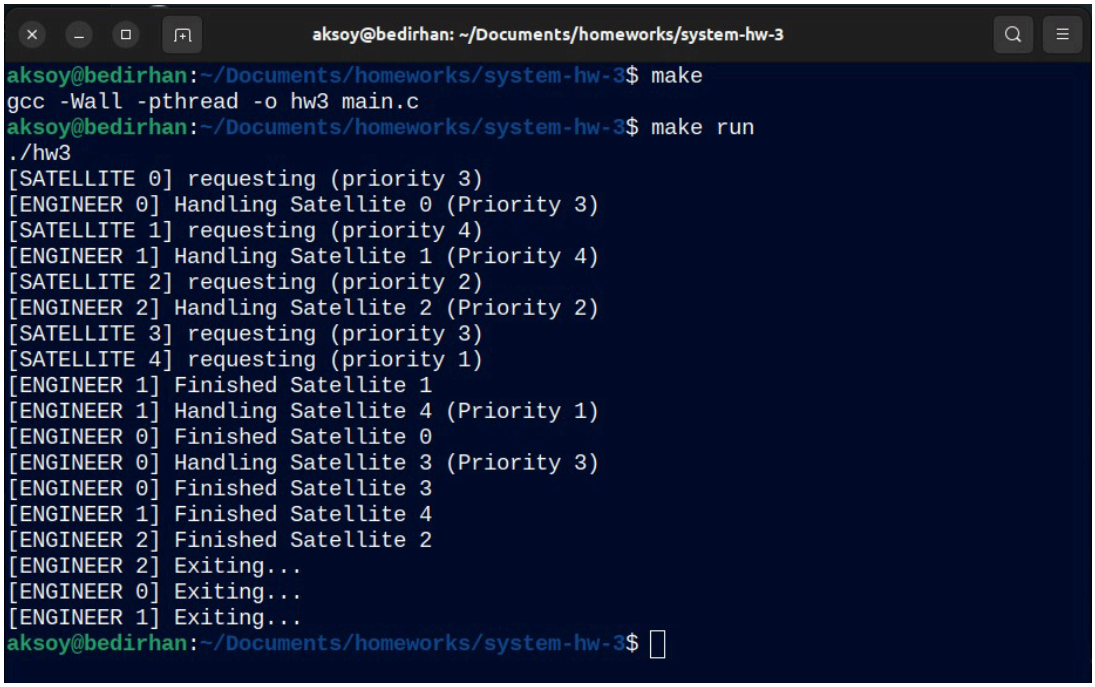
This function takes the satellite_id as an argument and searches for the given satellite_id to remove that request from the requestQueue linked list since that satellite has reached to it's timeout time. It removes the node from the list and free's that node and returns.

```
67  // function to remove a request from the queue by satellite_id
68  void removeRequest(int satellite_id) {
69      pthread_mutex_lock(&engineerMutex);
70      Node* current = requestQueue;
71      Node* prev = NULL;
72
73      // traverse the queue to find the request
74      while (current != NULL) {
75          if (current->request.satellite_id == satellite_id) {
76              // found the node to remove
77              if (prev == NULL) {
78                  // removing the head
79                  requestQueue = current->next;
80              }
81              else {
82                  // removing a non-head node
83                  prev->next = current->next;
84              }
85              free(current);
86              pthread_mutex_unlock(&engineerMutex);
87              return;
88          }
89          prev = current;
90          current = current->next;
91      }
92      // request not found
93      pthread_mutex_unlock(&engineerMutex);
94  }
95
```

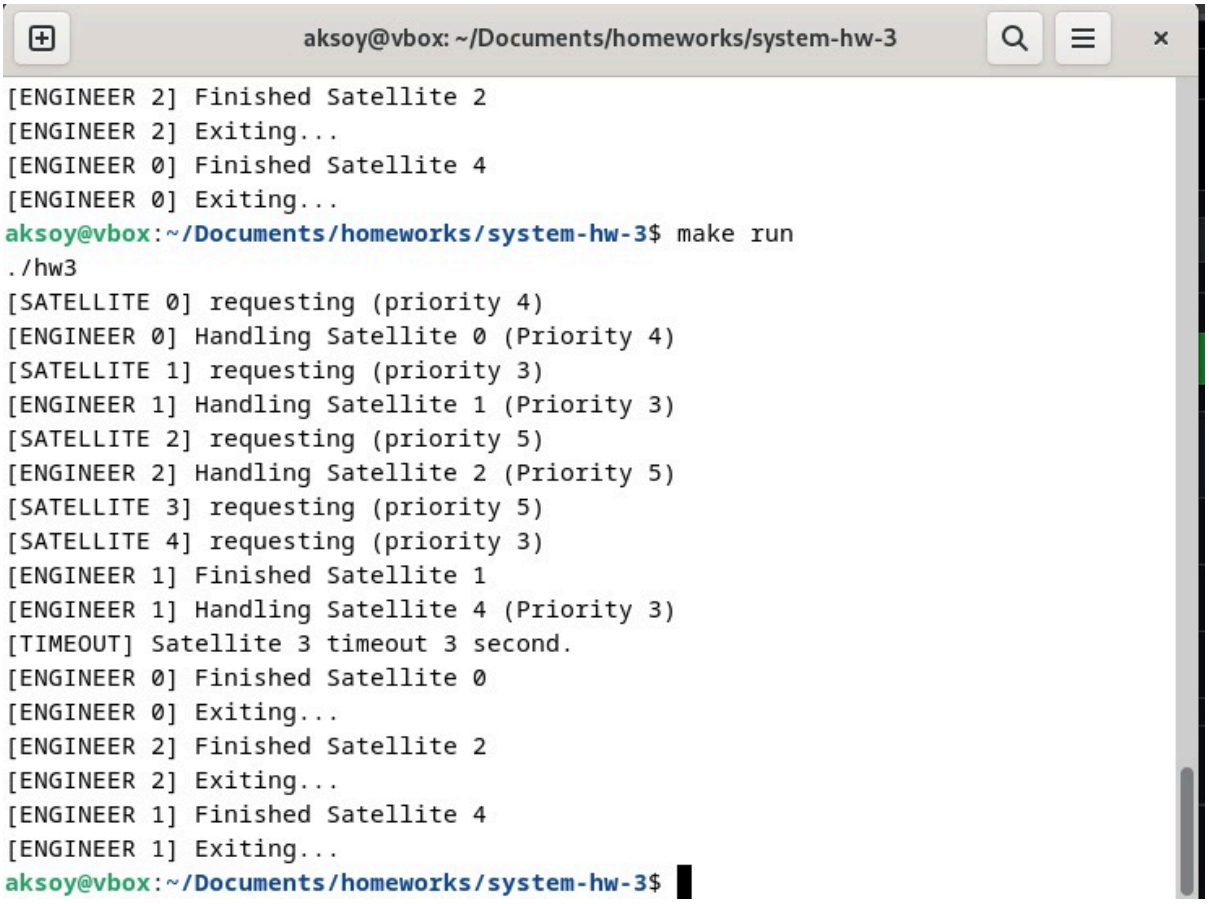
3) Screenshots

Testing program in my Ubuntu and Debian 12 machines:





Testing the program's timeout algorithm with changing timeout (3 second) and satellite working times (random time interval between 1 to 7 second) on Debian 12 virtual box:



4) Conclusion

I have encountered challenges when implementing the priority queue in the requestQueue linked list. I have misunderstood the priority order so I have to update it as the number decreases, the priority increases (1 is the most prior to 5 is the least prior). I have tried to lock the engineerMutex inside of the popRequest while the engineerMutex has already locked in the caller function, so it caused to a deadlocked and program waits indefinitely. It takes too much time to figure out and fix it.

The creation order of the satellites and engineer causes deadlocks too since engineers waits for newRequest and this semaphore is posted inside of the satellite function. First I have tried to create satellites first and then create engineer threads which works for one satellite and other engineers was stays idle. I have solved it with posting engineerMutex at the end of the inside of the engineer function itself to update available engineers and creating engineers first before satellites to prevent deadlocks.

The timeout algorithm was a bit confusing since the satellite request should be removed from the request queue if it reached its timeout time. I have implemented another function called remove request at the end of my working on this project to ensure no timeouted satellite was handled by engineers.