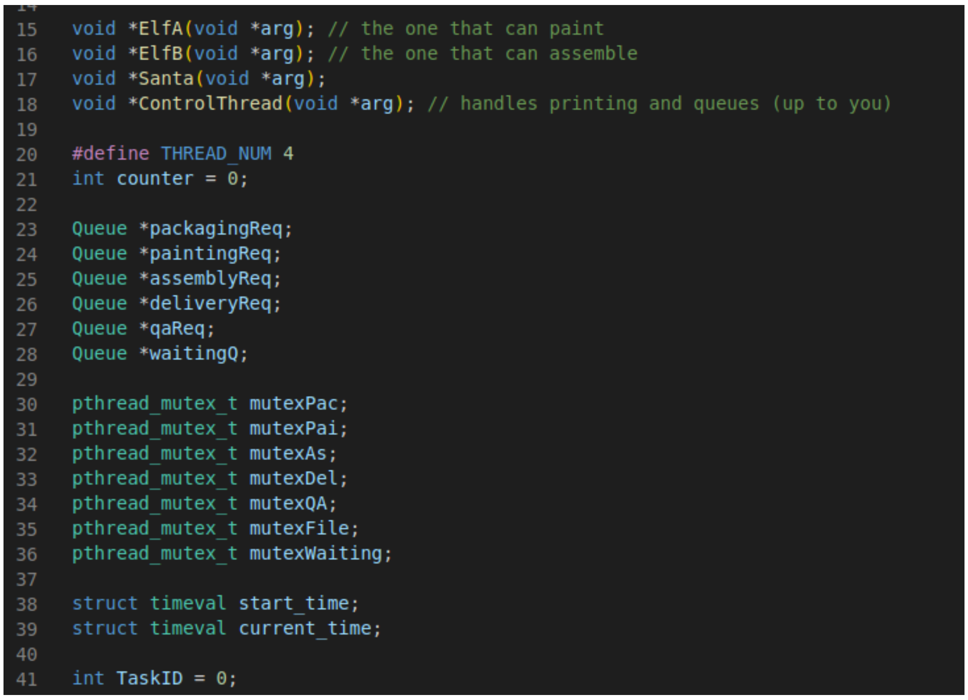
COMP 304 PROJECT

PART1:

Explanation of global variables:

ElfA ElfB Santa are threads which will do the explained job from pdf. We used control thread for grouping the tasks and enqueue them into their relative queues. PackagingReq is for packaging tasks which will be done by elves, paintigReq is for painting tasks which will be done by elf A, assemblyReq is for assembling tasks which will be done by elf B, deliveryReq and qaReq are for delivery and qa tasks which will be done by Santa and waitinQ is an intermediate queue which will hold tasks for type 4 and type 5 gifts (when done one task, the other task will wait for the other thread in the waiting queue). Counter was initially for time managment but then used as an ID manager. Start_time and current_time are for time management of threads. TaskID is for controlling the current task number.

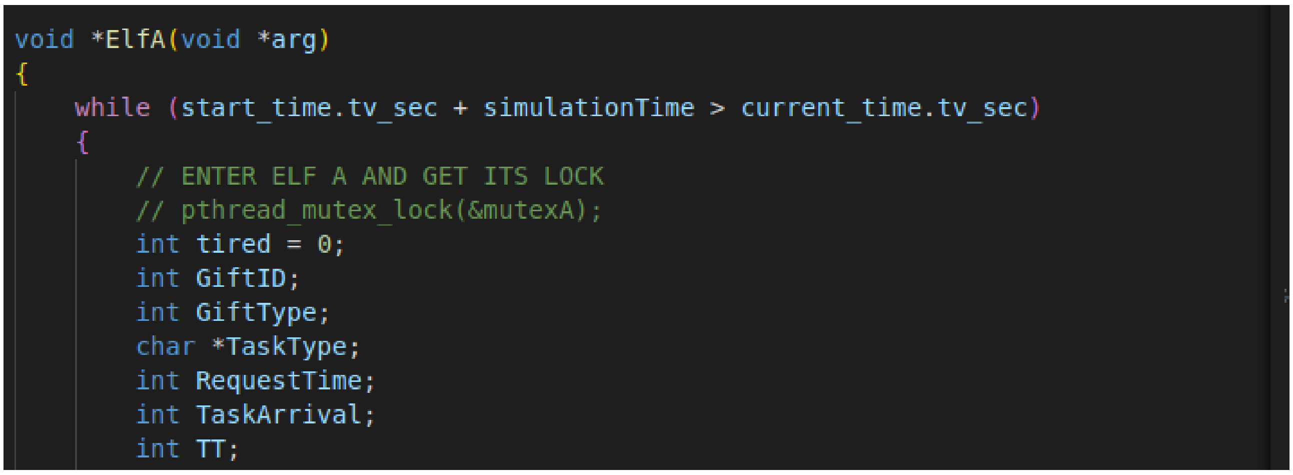


```c
void *ElfA(void *arg); // the one that can paint
void *ElfB(void *arg); // the one that can assemble
void *Santa(void *arg);
void *ControlThread(void *arg); // handles printing and queues (up to you)

#define THREAD_NUM 4
int counter = 0;

Queue *packagingReq;
Queue *paintingReq;
Queue *assemblyReq;
Queue *deliveryReq;
Queue *qaReq;
Queue *waitingQ;

pthread_mutex_t mutexPac;
pthread_mutex_t mutexPai;
pthread_mutex_t mutexAs;
pthread_mutex_t mutexDel;
pthread_mutex_t mutexQA;
pthread_mutex_t mutexFile;
pthread_mutex_t mutexWaiting;

struct timeval start_time;
struct timeval current_time;

int TaskID = 0;
```

In the main function we constructed queues, initialized mutexes, initialized events.log file, created threads with assigned functions, joined threads, destroyed mutexes and queues.

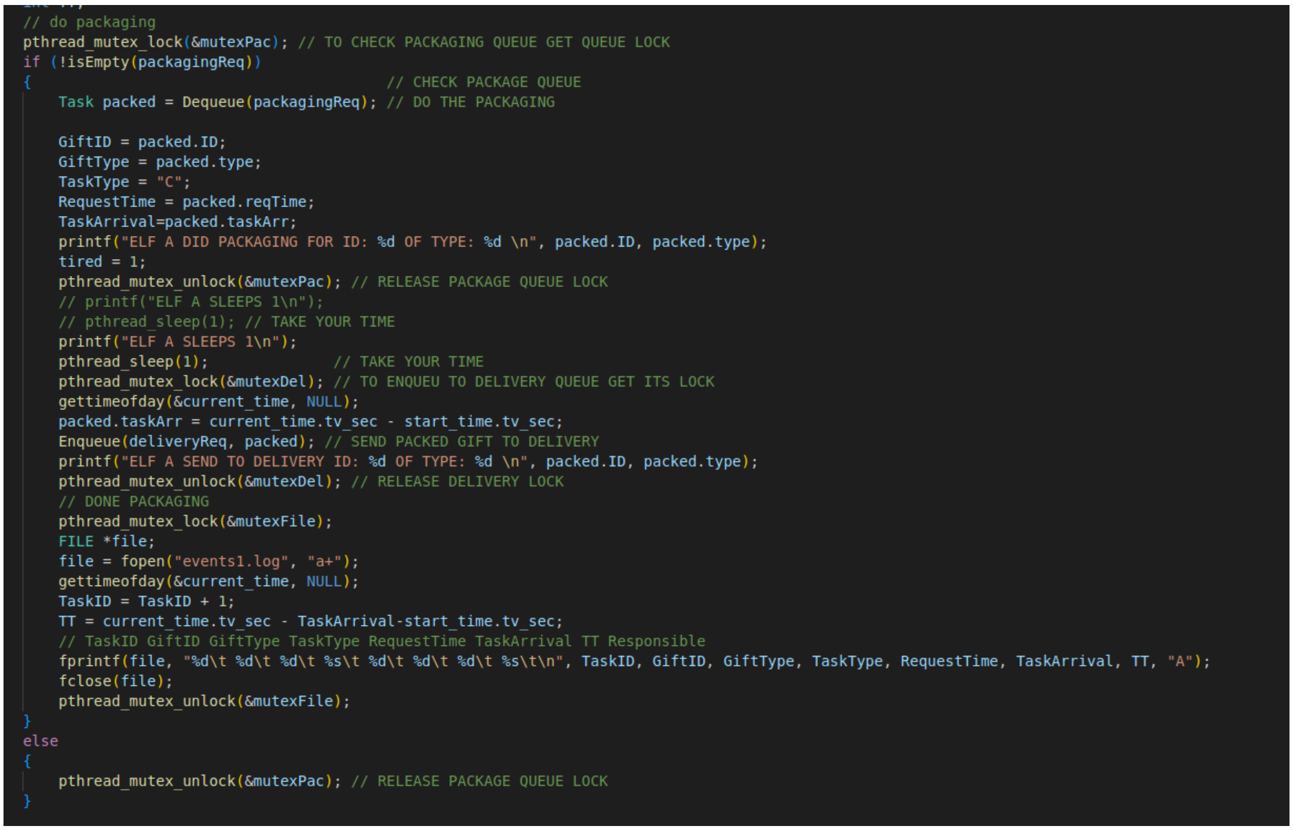
ElfA:

Starts a while loop which will until simulation time is achieved and then continue running for 30 more seconds to finish the jobs it may have. Every while loop starts with initializing file data variables.



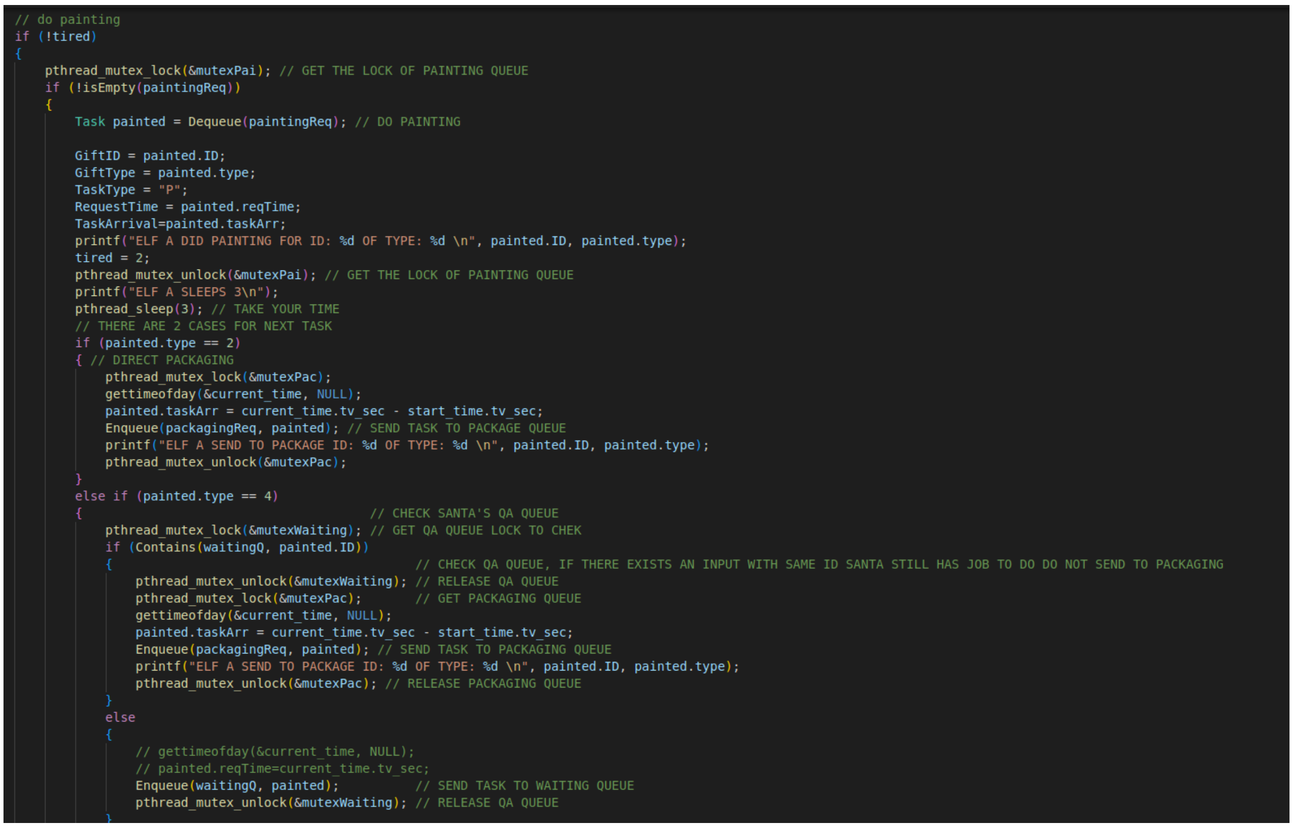
```
void *ElfA(void *arg)
{
    while (start_time.tv_sec + simulationTime > current_time.tv_sec)
    {
        // ENTER ELF A AND GET ITS LOCK
        // pthread_mutex_lock(&mutexA);
        int tired = 0;
        int GiftID;
        int GiftType;
        char *TaskType;
        int RequestTime;
        int TaskArrival;
        int TT;
```

Tired is for prioritizing tasks of the thread and the rest are named same as the pdf.

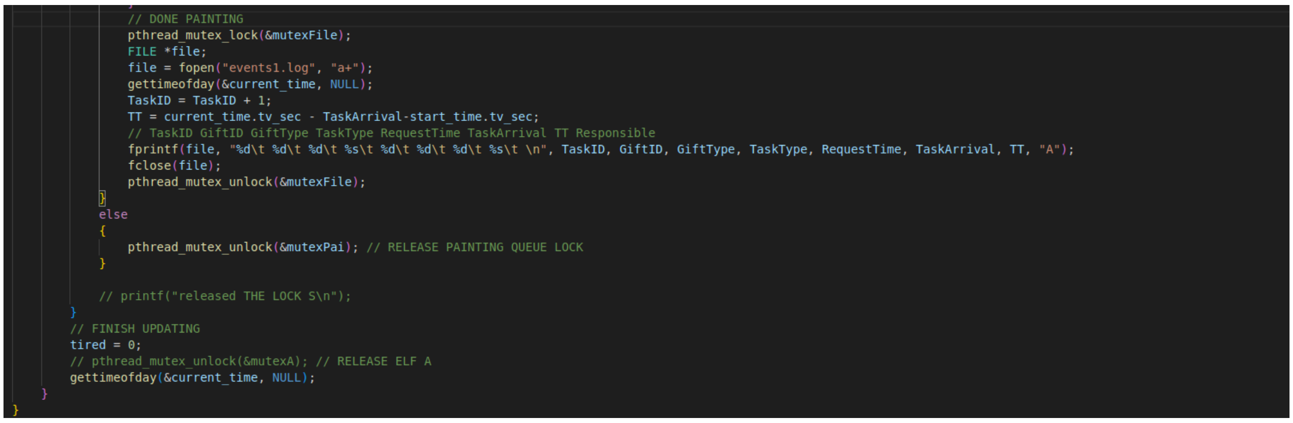


```
// do packaging
pthread_mutex_lock(&mutexPac); // TO CHECK PACKAGING QUEUE GET QUEUE LOCK
if (!isEmpty(packagingReq))
{                                         // CHECK PACKAGE QUEUE
    Task packed = Dequeue(packagingReq); // DO THE PACKAGING

    GiftID = packed.ID;
    GiftType = packed.type;
    TaskType = "C";
    RequestTime = packed.reqTime;
    TaskArrival=packed.taskArr;
    printf("ELF A DID PACKAGING FOR ID: %d OF TYPE: %d \n", packed.ID, packed.type);
    tired = 1;
    pthread_mutex_unlock(&mutexPac); // RELEASE PACKAGE QUEUE LOCK
    // printf("ELF A SLEEPS 1\n");
    // pthread_sleep(1); // TAKE YOUR TIME
    printf("ELF A SLEEPS 1\n");
    pthread_sleep(1);              // TAKE YOUR TIME
    pthread_mutex_lock(&mutexDel); // TO ENQUEU TO DELIVERY QUEUE GET ITS LOCK
    gettimeofday(&current_time, NULL);
    packed.taskArr = current_time.tv_sec - start_time.tv_sec;
    Enqueue(deliveryReq, packed); // SEND PACKED GIFT TO DELIVERY
    printf("ELF A SEND TO DELIVERY ID: %d OF TYPE: %d \n", packed.ID, packed.type);
    pthread_mutex_unlock(&mutexDel); // RELEASE DELIVERY LOCK
    // DONE PACKAGING
    pthread_mutex_lock(&mutexFile);
    FILE *file;
    file = fopen("events1.log", "a+");
    gettimeofday(&current_time, NULL);
    TaskID = TaskID + 1;
    TT = current_time.tv_sec - TaskArrival-start_time.tv_sec;
    // TaskID GiftID GiftType TaskType RequestTime TaskArrival TT Responsible
    fprintf(file, "%d\t %d\t %d\t %s\t %d\t %d\t %d\t %s\t\n", TaskID, GiftID, GiftType, TaskType, RequestTime, TaskArrival, TT, "A");
    fclose(file);
    pthread_mutex_unlock(&mutexFile);
}
else
{
    pthread_mutex_unlock(&mutexPac); // RELEASE PACKAGE QUEUE LOCK
}
```

Elf A can do packaging or painting. It was given that packaging is prioritized. To check and possibly manipulate the packaging queue we get the packaging queue mutex. This mutexPac is used only when the packagingReq is needed. We check for an existence of a Task node in the queue, if it exists we continue to work in the if block, if there are no tasks waiting in packaging queue we release the mutex lock and check for painting queue, exactly the same way. In the case of an existence of a task in package queue, we first dequeue it and get the necessary information for log file writing and set some task variables. We update the tired variable to prevent the entrance of elfA into painting task right after finishing the packaging task. We do unlocking for the packaging queue, since our job with the queue is done, then we sleep the elf A thread for 1 second. The following task for every possible gift type is delivery, so we lock the delivery queue with respecting mutex,enqueue the dequeued and finished packaging task after updating its task arrival time into delivery queue. After unlocking the delivery queue, we get the file mutex to write data into it, open file, update task Id and current time within the mutex since these are global variables, close the file and unlock file mutex.

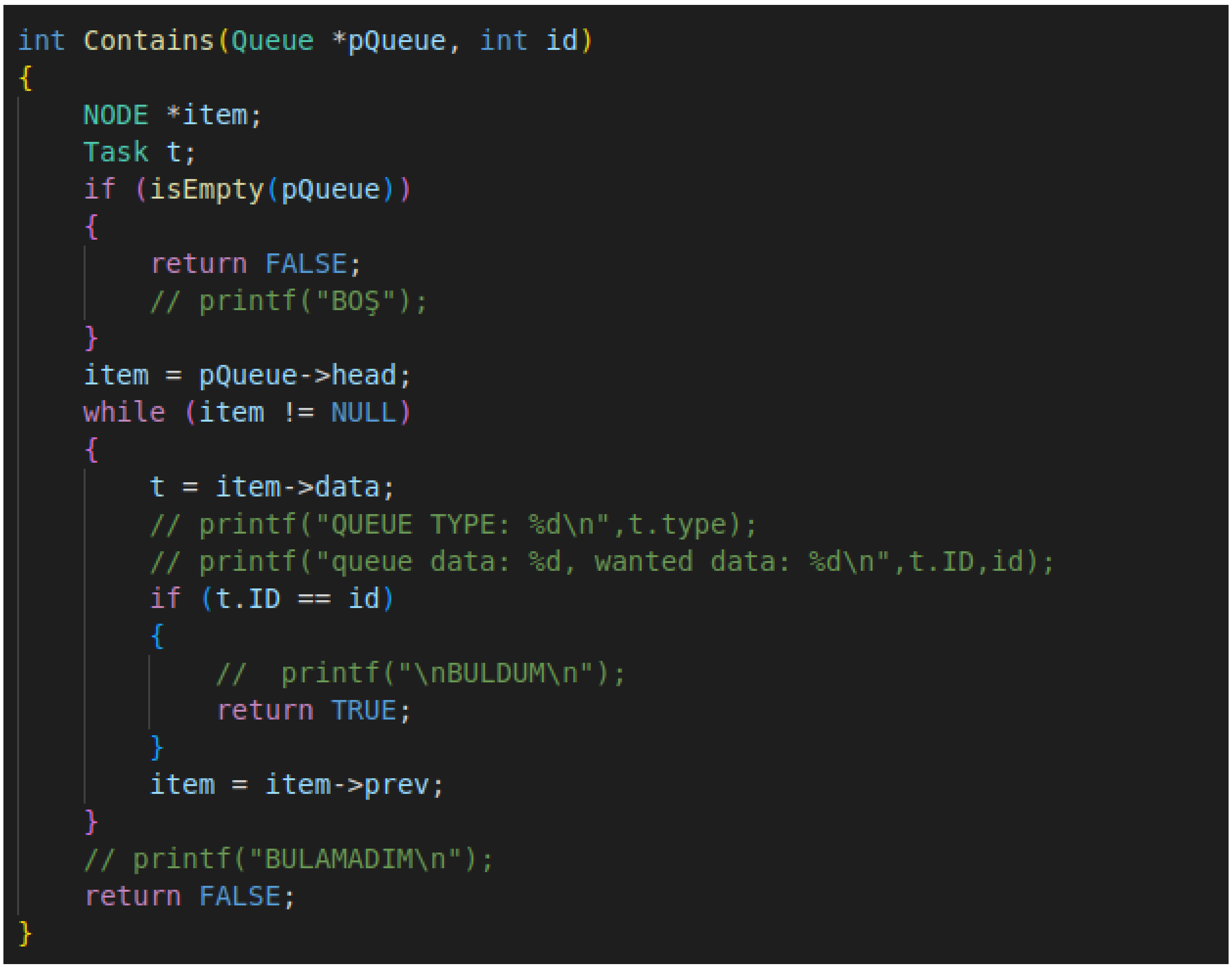


```
// do painting
if (!tired)
{
    pthread_mutex_lock(&mutexPai); // GET THE LOCK OF PAINTING QUEUE
    if (!isEmpty(paintingReq))
    {
        Task painted = Dequeue(paintingReq); // DO PAINTING

        GiftID = painted.ID;
        GiftType = painted.type;
        TaskType = "P";
        RequestTime = painted.reqTime;
        TaskArrival=painted.taskArr;
        printf("ELF A DID PAINTING FOR ID: %d OF TYPE: %d \n", painted.ID, painted.type);
        tired = 2;
        pthread_mutex_unlock(&mutexPai); // GET THE LOCK OF PAINTING QUEUE
        printf("ELF A SLEEPS 3\n");
        pthread_sleep(3); // TAKE YOUR TIME
        // THERE ARE 2 CASES FOR NEXT TASK
        if (painted.type == 2)
        { // DIRECT PACKAGING
            pthread_mutex_lock(&mutexPac);
            gettimeofday(&current_time, NULL);
            painted.taskArr = current_time.tv_sec - start_time.tv_sec;
            Enqueue(packagingReq, painted); // SEND TASK TO PACKAGE QUEUE
            printf("ELF A SEND TO PACKAGE ID: %d OF TYPE: %d \n", painted.ID, painted.type);
            pthread_mutex_unlock(&mutexPac);
        }
        else if (painted.type == 4)
        {                                       // CHECK SANTA'S QA QUEUE
            pthread_mutex_lock(&mutexWaiting); // GET QA QUEUE LOCK TO CHEK
            if (Contains(waitingQ, painted.ID))
            {                                       // CHECK QA QUEUE, IF THERE EXISTS AN INPUT WITH SAME ID SANTA STILL HAS JOB TO DO DO NOT SEND TO PACKAGING
                pthread_mutex_unlock(&mutexWaiting); // RELEASE QA QUEUE
                pthread_mutex_lock(&mutexPac);       // GET PACKAGING QUEUE
                gettimeofday(&current_time, NULL);
                painted.taskArr = current_time.tv_sec - start_time.tv_sec;
                Enqueue(packagingReq, painted); // SEND TASK TO PACKAGING QUEUE
                printf("ELF A SEND TO PACKAGE ID: %d OF TYPE: %d \n", painted.ID, painted.type);
                pthread_mutex_unlock(&mutexPac); // RELEASE PACKAGING QUEUE
            }
            else
            {
                // gettimeofday(&current_time, NULL);
                // painted.reqTime=current_time.tv_sec;
                Enqueue(waitingQ, painted);          // SEND TASK TO WAITING QUEUE
                pthread_mutex_unlock(&mutexWaiting); // RELEASE QA QUEUE
            }
```



```
                // DONE PAINTING
                pthread_mutex_lock(&mutexFile);
                FILE *file;
                file = fopen("events1.log", "a+");
                gettimeofday(&current_time, NULL);
                TaskID = TaskID + 1;
                TT = current_time.tv_sec - TaskArrival-start_time.tv_sec;
                // TaskID GiftID GiftType TaskType RequestTime TaskArrival TT Responsible
                fprintf(file, "%d\t %d\t %d\t %s\t %d\t %d\t %d\t %s\t \n", TaskID, GiftID, GiftType, TaskType, RequestTime, TaskArrival, TT, "A");
                fclose(file);
                pthread_mutex_unlock(&mutexFile);
            }
            else
            {
                pthread_mutex_unlock(&mutexPai); // RELEASE PAINTING QUEUE LOCK
            }

            // printf("released THE LOCK S\n");
        }
        // FINISH UPDATING
        tired = 0;
        // pthread_mutex_unlock(&mutexA); // RELEASE ELF A
        gettimeofday(&current_time, NULL);
    }
}
```

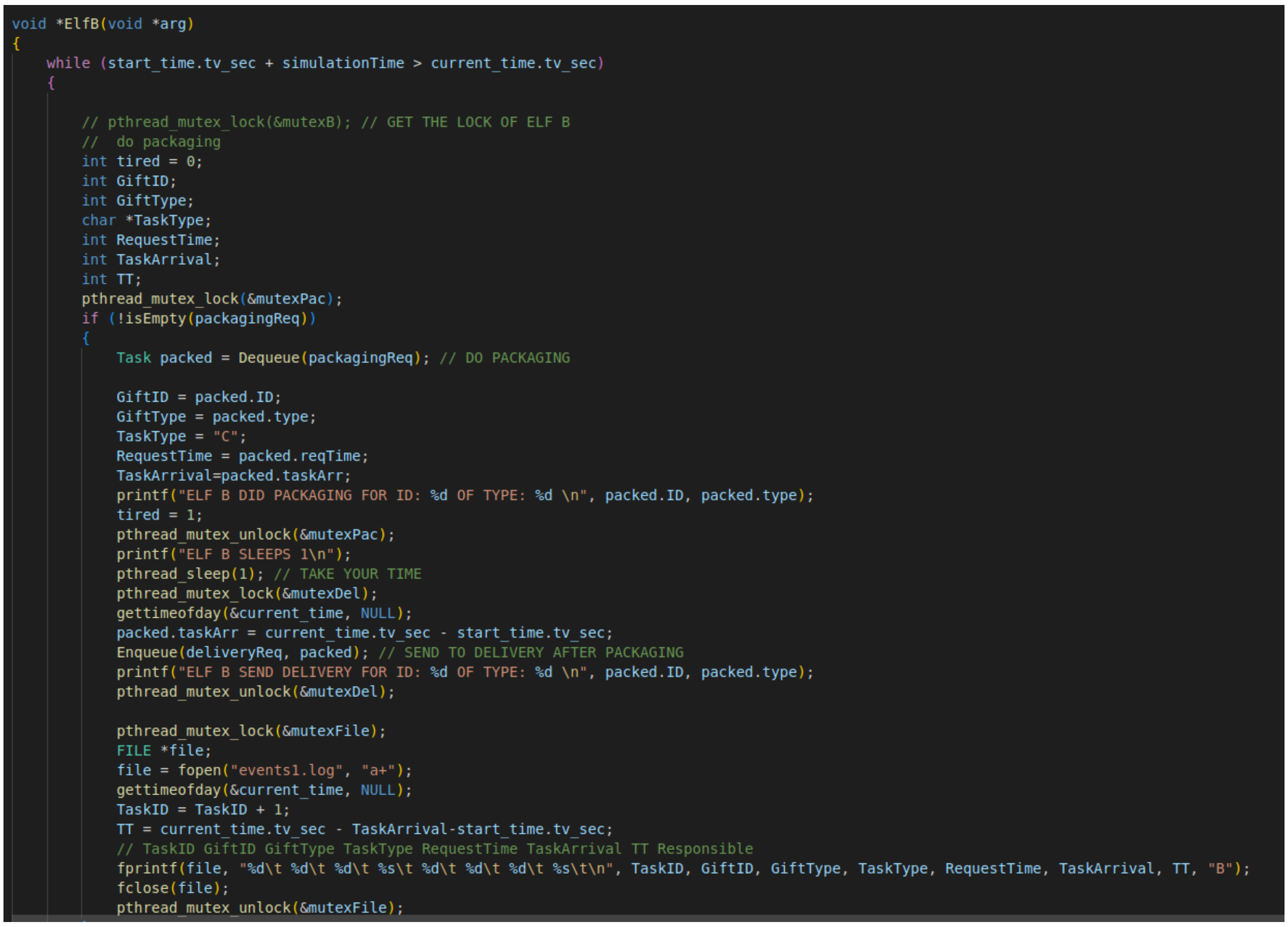
Elf A can do painting if it didn't do packaging in the current time. We check it with tired data. If the elf is not tired from a previous job, we enter into painting job, since we will do this job if there exists a task in painting queue, we first get the lock for painting queue to check. If there exists a task waiting in painting queue we continue with the if block else we release the lock, update current time and get into another while cycle within the thread. In the case of a waiting paint request, we dequeue the painting queue, update the data to write into the file, release the lock of painting queue after getting the task, do the painting job for 3 seconds with sleep, then we check the type of the gift. There are 2 cases: if gift type is 2, the next task is simply packaging, so we enqueue the dequeued task the same way we did for the packed tasked in the first part explained above into the packaging queue with the control of mutex locks; however, if gift type is 4 santa and elf A need to work together. For this concurrent working, we have an intermediate queue, any task that is done by elf A or santa is added to waiting queue to indicate that task waits for the other one to be sent to packaging. We wrote a simple contains function in queue class.



```
int Contains(Queue *pQueue, int id)
{
    NODE *item;
    Task t;
    if (isEmpty(pQueue))
    {
        return FALSE;
        // printf("BOŞ");
    }
    item = pQueue->head;
    while (item != NULL)
    {
        t = item->data;
        // printf("QUEUE TYPE: %d\n",t.type);
        // printf("queue data: %d, wanted data: %d\n",t.ID,id);
        if (t.ID == id)
        {
            //  printf("\nBULDUM\n");
            return TRUE;
        }
        item = item->prev;
    }
    // printf("BULAMADIM\n");
    return FALSE;
}
```

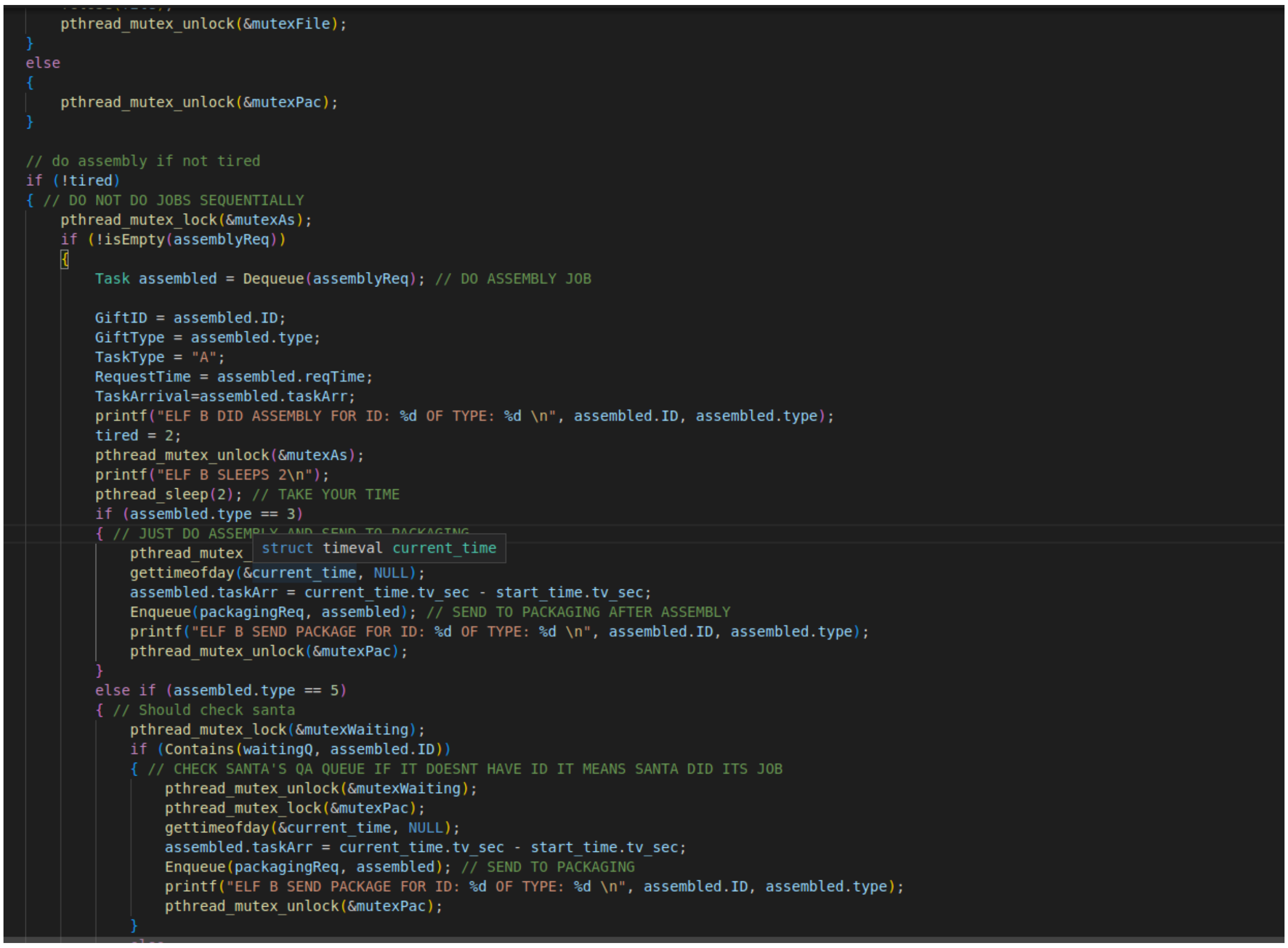
The task currently finished being in the waiting queue means that its qa is already done by Santa. In this case the dequeued task is enqueued into packaging queue. While both checking the contains functions and enqueue we do locking and unlocking just like we did before. If the waiting queue does not contain the currently finished task, it means the task needs to wait for santa to do the qa. To let santa know that the task is waiting in the queue we enqueue the current task into waiting queue without updating its task arrival time because it was waiting already for the Santa. We also do the file writing at the end of all this enqueues.

ElfB:



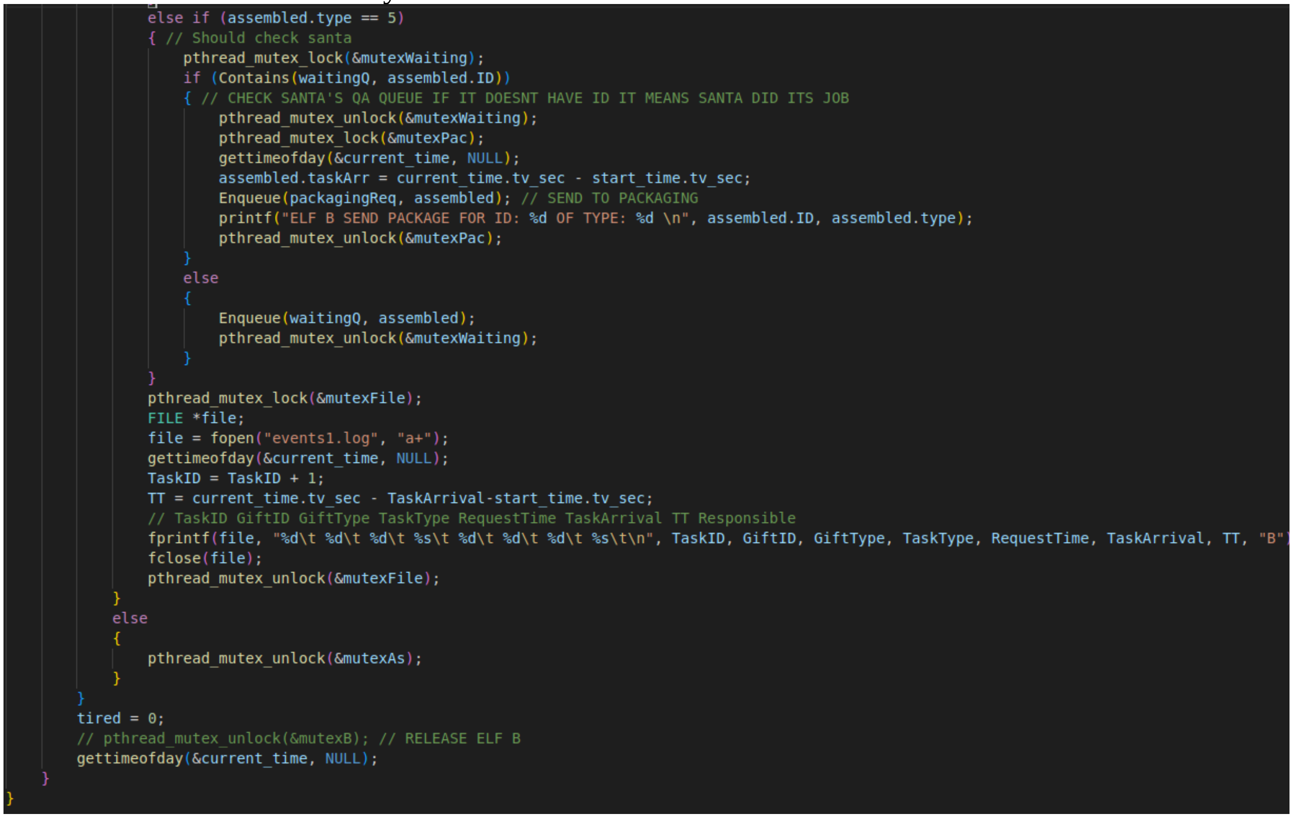
```c
void *ElfB(void *arg)
{
    while (start_time.tv_sec + simulationTime > current_time.tv_sec)
    {

        // pthread_mutex_lock(&mutexB); // GET THE LOCK OF ELF B
        //  do packaging
        int tired = 0;
        int GiftID;
        int GiftType;
        char *TaskType;
        int RequestTime;
        int TaskArrival;
        int TT;
        pthread_mutex_lock(&mutexPac);
        if (!isEmpty(packagingReq))
        {
            Task packed = Dequeue(packagingReq); // DO PACKAGING

            GiftID = packed.ID;
            GiftType = packed.type;
            TaskType = "C";
            RequestTime = packed.reqTime;
            TaskArrival=packed.taskArr;
            printf("ELF B DID PACKAGING FOR ID: %d OF TYPE: %d \n", packed.ID, packed.type);
            tired = 1;
            pthread_mutex_unlock(&mutexPac);
            printf("ELF B SLEEPS 1\n");
            pthread_sleep(1); // TAKE YOUR TIME
            pthread_mutex_lock(&mutexDel);
            gettimeofday(&current_time, NULL);
            packed.taskArr = current_time.tv_sec - start_time.tv_sec;
            Enqueue(deliveryReq, packed); // SEND TO DELIVERY AFTER PACKAGING
            printf("ELF B SEND DELIVERY FOR ID: %d OF TYPE: %d \n", packed.ID, packed.type);
            pthread_mutex_unlock(&mutexDel);

            pthread_mutex_lock(&mutexFile);
            FILE *file;
            file = fopen("events1.log", "a+");
            gettimeofday(&current_time, NULL);
            TaskID = TaskID + 1;
            TT = current_time.tv_sec - TaskArrival-start_time.tv_sec;
            // TaskID GiftID GiftType TaskType RequestTime TaskArrival TT Responsible
            fprintf(file, "%d\t %d\t %d\t %s\t %d\t %d\t %d\t %s\t\n", TaskID, GiftID, GiftType, TaskType, RequestTime, TaskArrival, TT, "B");
            fclose(file);
            pthread_mutex_unlock(&mutexFile);
```

Elf B is implemented with the exact same logic, since its descriptions match elf A at the logic level, the difference is the queues and sleep times.



```c
        pthread_mutex_unlock(&mutexFile);
    }
    else
    {
        pthread_mutex_unlock(&mutexPac);
    }

    // do assembly if not tired
    if (!tired)
    { // DO NOT DO JOBS SEQUENTIALLY
        pthread_mutex_lock(&mutexAs);
        if (!isEmpty(assemblyReq))
        {
            Task assembled = Dequeue(assemblyReq); // DO ASSEMBLY JOB

            GiftID = assembled.ID;
            GiftType = assembled.type;
            TaskType = "A";
            RequestTime = assembled.reqTime;
            TaskArrival=assembled.taskArr;
            printf("ELF B DID ASSEMBLY FOR ID: %d OF TYPE: %d \n", assembled.ID, assembled.type);
            tired = 2;
            pthread_mutex_unlock(&mutexAs);
            printf("ELF B SLEEPS 2\n");
            pthread_sleep(2); // TAKE YOUR TIME
            if (assembled.type == 3)
            { // JUST DO ASSEMBLY AND SEND TO PACKAGING
                pthread_mutex_ struct timeval current_time
                gettimeofday(&current_time, NULL);
                assembled.taskArr = current_time.tv_sec - start_time.tv_sec;
                Enqueue(packagingReq, assembled); // SEND TO PACKAGING AFTER ASSEMBLY
                printf("ELF B SEND PACKAGE FOR ID: %d OF TYPE: %d \n", assembled.ID, assembled.type);
                pthread_mutex_unlock(&mutexPac);
            }
            else if (assembled.type == 5)
            { // Should check santa
                pthread_mutex_lock(&mutexWaiting);
                if (Contains(waitingQ, assembled.ID))
                { // CHECK SANTA'S QA QUEUE IF IT DOESNT HAVE ID IT MEANS SANTA DID ITS JOB
                    pthread_mutex_unlock(&mutexWaiting);
                    pthread_mutex_lock(&mutexPac);
                    gettimeofday(&current_time, NULL);
                    assembled.taskArr = current_time.tv_sec - start_time.tv_sec;
                    Enqueue(packagingReq, assembled); // SEND TO PACKAGING
                    printf("ELF B SEND PACKAGE FOR ID: %d OF TYPE: %d \n", assembled.ID, assembled.type);
                    pthread_mutex_unlock(&mutexPac);
                }
```

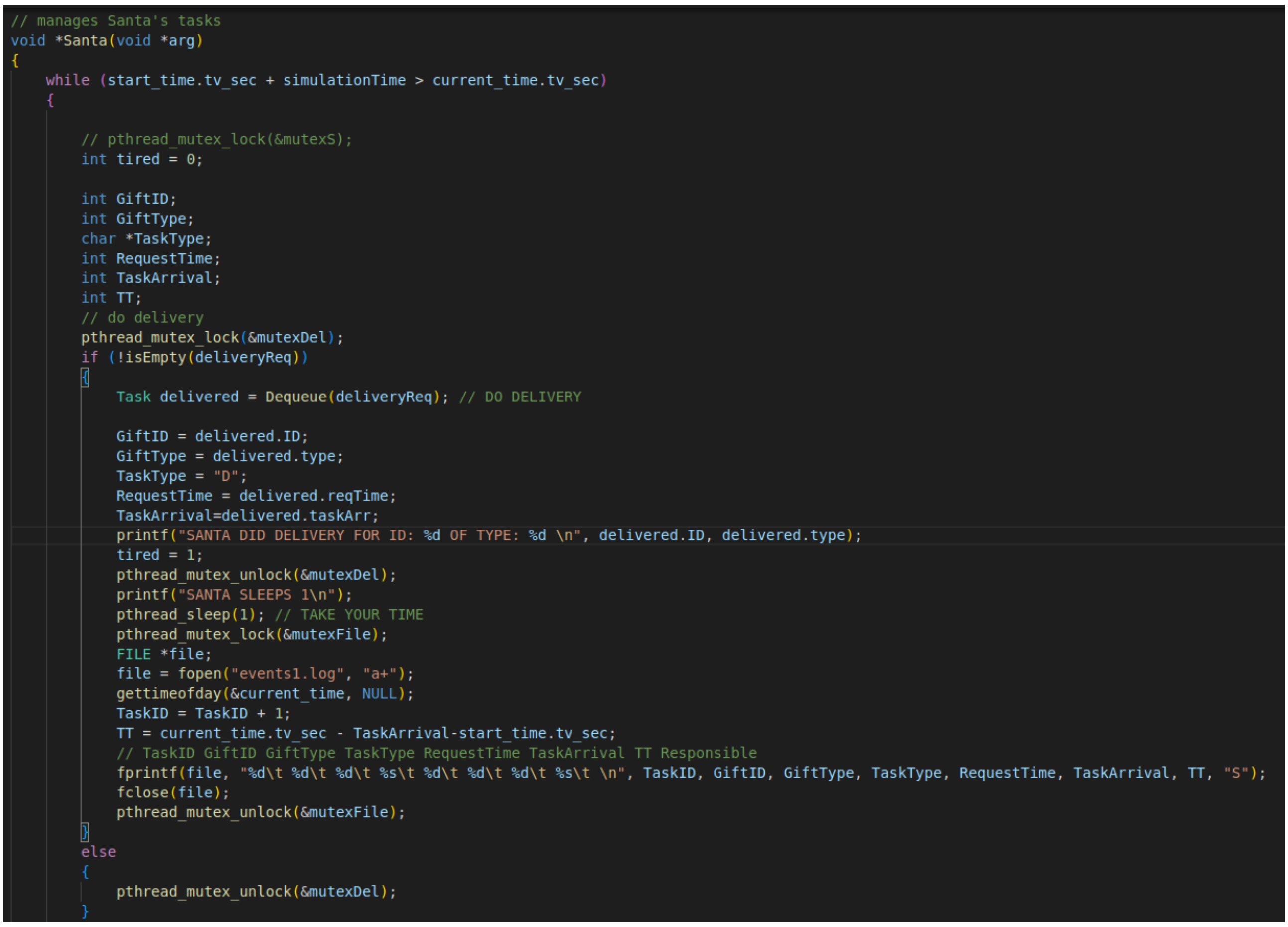
Here can be seen the similarity of elf A and elf B.



```c
                else if (assembled.type == 5)
                { // Should check santa
                    pthread_mutex_lock(&mutexWaiting);
                    if (Contains(waitingQ, assembled.ID))
                    { // CHECK SANTA'S QA QUEUE IF IT DOESNT HAVE ID IT MEANS SANTA DID ITS JOB
                        pthread_mutex_unlock(&mutexWaiting);
                        pthread_mutex_lock(&mutexPac);
                        gettimeofday(&current_time, NULL);
                        assembled.taskArr = current_time.tv_sec - start_time.tv_sec;
                        Enqueue(packagingReq, assembled); // SEND TO PACKAGING
                        printf("ELF B SEND PACKAGE FOR ID: %d OF TYPE: %d \n", assembled.ID, assembled.type);
                        pthread_mutex_unlock(&mutexPac);
                    }
                    else
                    {
                        Enqueue(waitingQ, assembled);
                        pthread_mutex_unlock(&mutexWaiting);
                    }
                }
                pthread_mutex_lock(&mutexFile);
                FILE *file;
                file = fopen("events1.log", "a+");
                gettimeofday(&current_time, NULL);
                TaskID = TaskID + 1;
                TT = current_time.tv_sec - TaskArrival-start_time.tv_sec;
                // TaskID GiftID GiftType TaskType RequestTime TaskArrival TT Responsible
                fprintf(file, "%d\t %d\t %d\t %s\t %d\t %d\t %d\t %s\t\n", TaskID, GiftID, GiftType, TaskType, RequestTime, TaskArrival, TT, "B"
                fclose(file);
                pthread_mutex_unlock(&mutexFile);
            }
            else
            {
                pthread_mutex_unlock(&mutexAs);
            }
        }
        tired = 0;
        // pthread_mutex_unlock(&mutexB); // RELEASE ELF B
        gettimeofday(&current_time, NULL);
    }
}
```

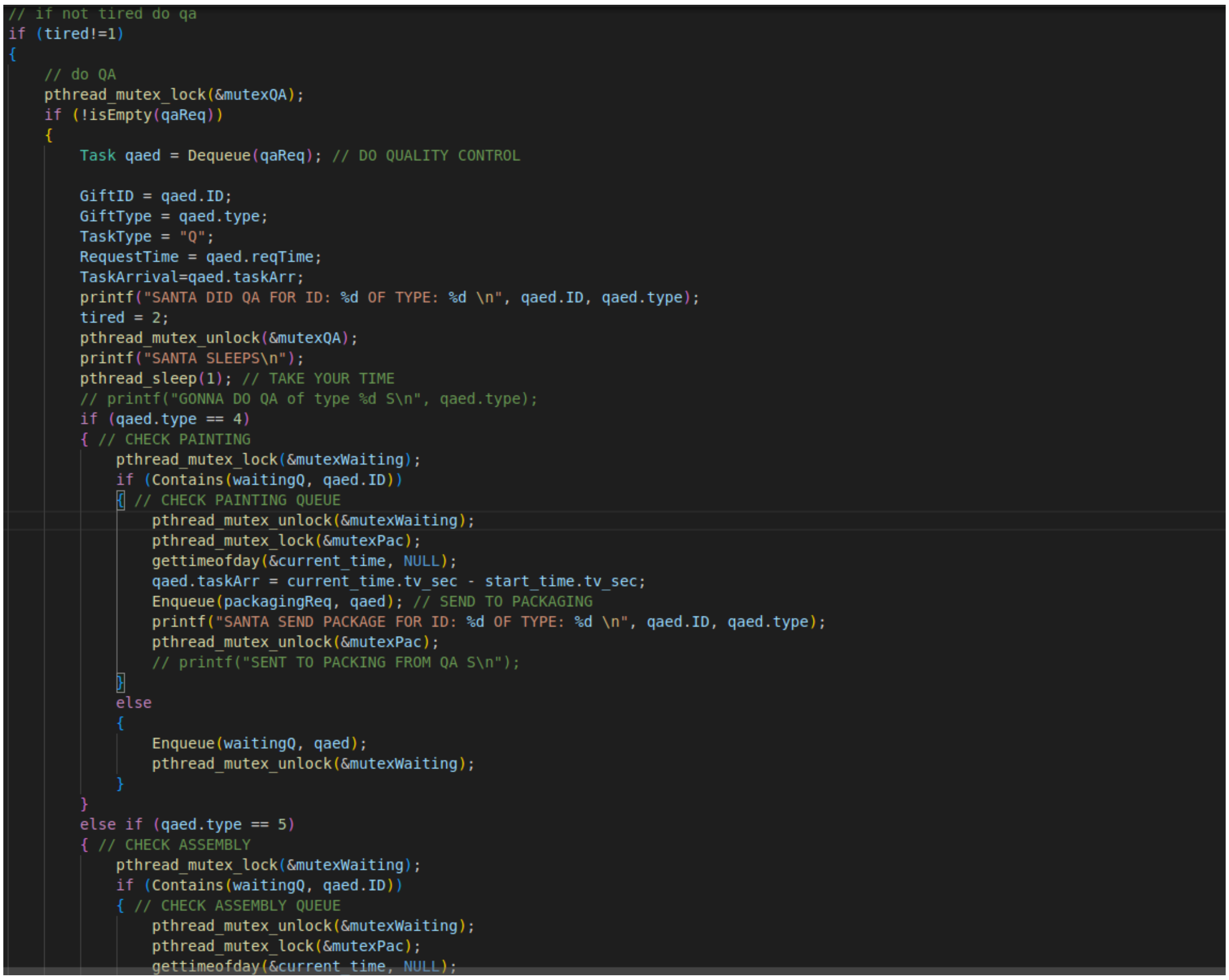
Santa is also implemented very similarly to elves.

Santa:



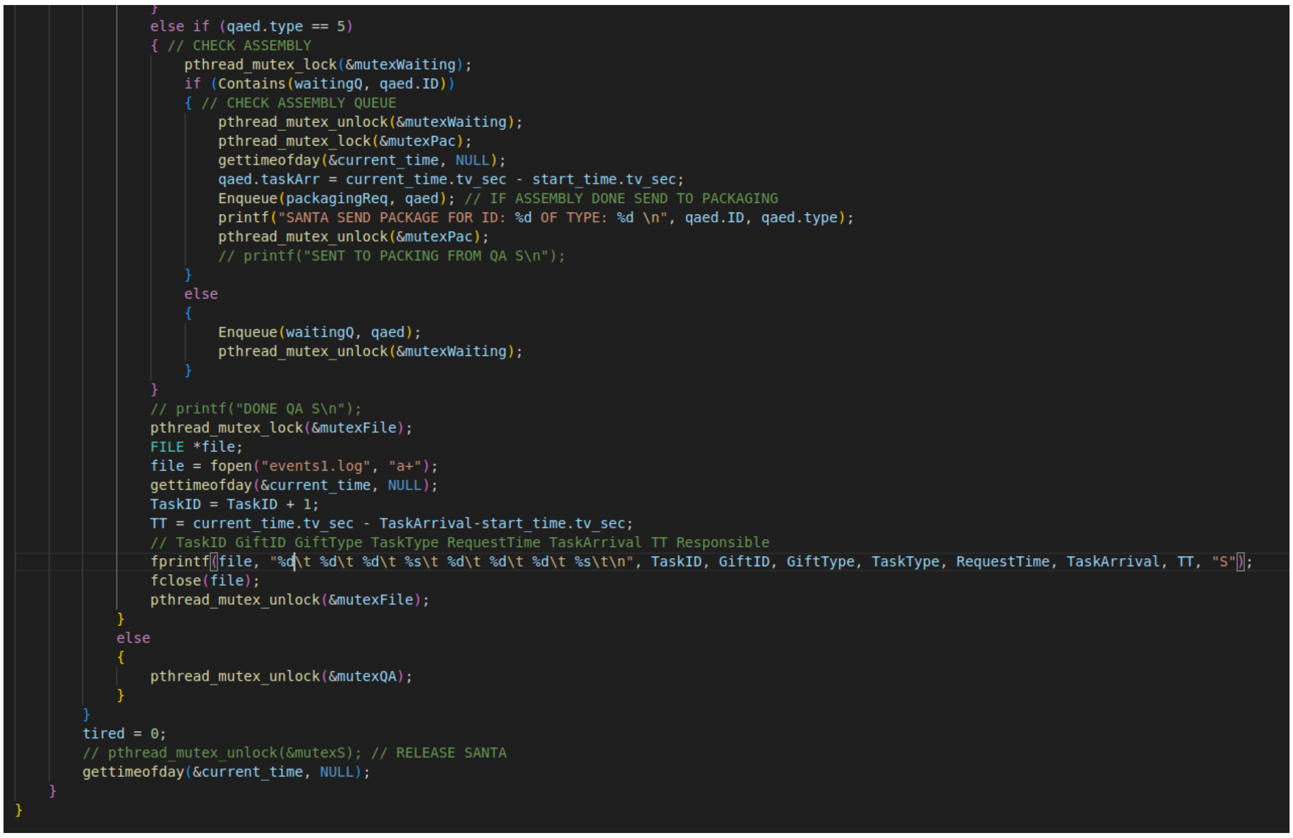
```c
// manages Santa's tasks
void *Santa(void *arg)
{
    while (start_time.tv_sec + simulationTime > current_time.tv_sec)
    {

        // pthread_mutex_lock(&mutexS);
        int tired = 0;

        int GiftID;
        int GiftType;
        char *TaskType;
        int RequestTime;
        int TaskArrival;
        int TT;
        // do delivery
        pthread_mutex_lock(&mutexDel);
        if (!isEmpty(deliveryReq))
        {
            Task delivered = Dequeue(deliveryReq); // DO DELIVERY

            GiftID = delivered.ID;
            GiftType = delivered.type;
            TaskType = "D";
            RequestTime = delivered.reqTime;
            TaskArrival=delivered.taskArr;
            printf("SANTA DID DELIVERY FOR ID: %d OF TYPE: %d \n", delivered.ID, delivered.type);
            tired = 1;
            pthread_mutex_unlock(&mutexDel);
            printf("SANTA SLEEPS 1\n");
            pthread_sleep(1); // TAKE YOUR TIME
            pthread_mutex_lock(&mutexFile);
            FILE *file;
            file = fopen("events1.log", "a+");
            gettimeofday(&current_time, NULL);
            TaskID = TaskID + 1;
            TT = current_time.tv_sec - TaskArrival-start_time.tv_sec;
            // TaskID GiftID GiftType TaskType RequestTime TaskArrival TT Responsible
            fprintf(file, "%d\t %d\t %d\t %s\t %d\t %d\t %d\t %s\t \n", TaskID, GiftID, GiftType, TaskType, RequestTime, TaskArrival, TT, "S");
            fclose(file);
            pthread_mutex_unlock(&mutexFile);
        }
        else
        {
            pthread_mutex_unlock(&mutexDel);
        }
```

The difference is that the prioritized task of Santa is delivery which wont send the dequeued task to anyone. So only one queue management is enough for these tasks.



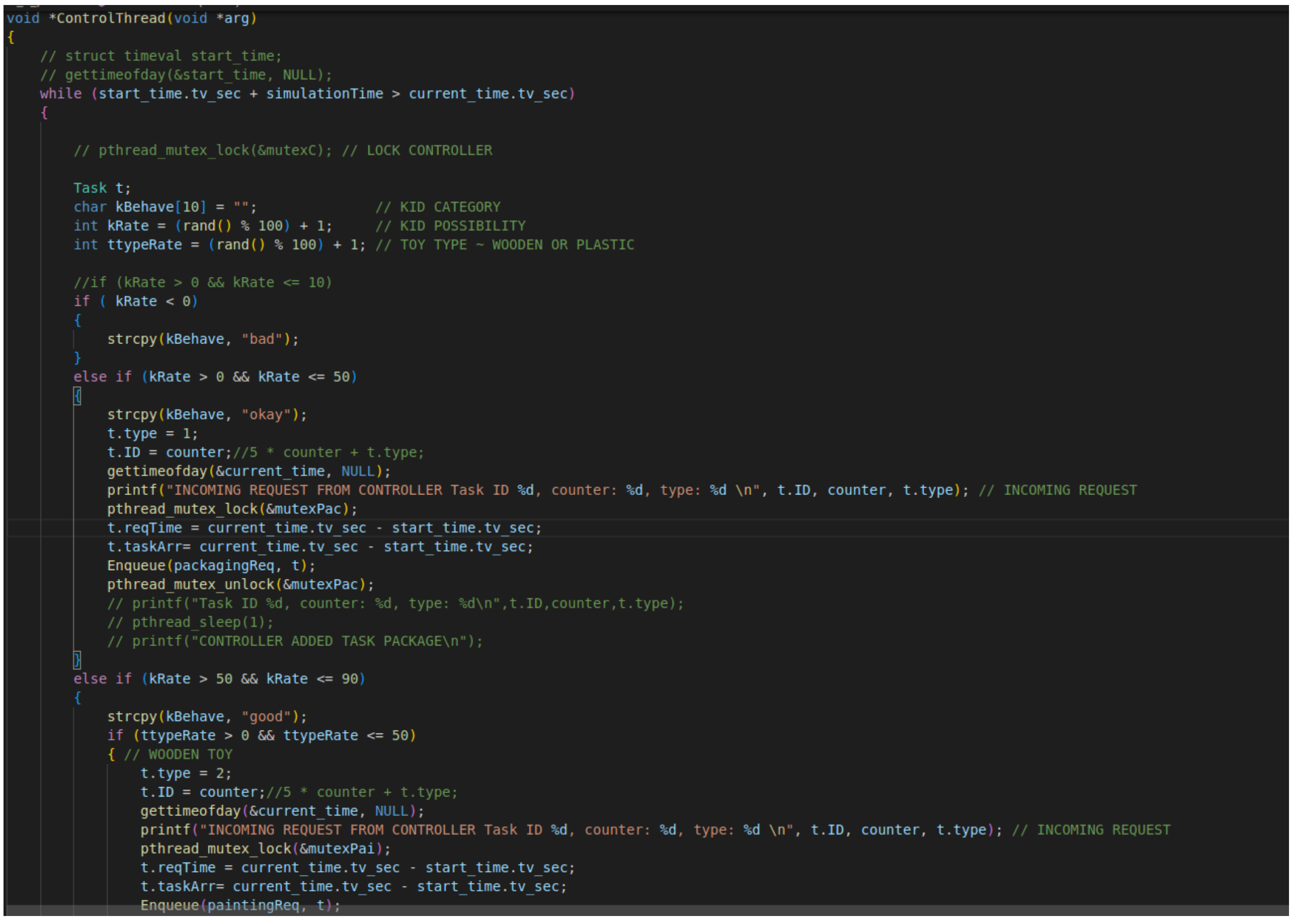
```
// if not tired do qa
if (tired!=1)
{
    // do QA
    pthread_mutex_lock(&mutexQA);
    if (!isEmpty(qaReq))
    {
        Task qaed = Dequeue(qaReq); // DO QUALITY CONTROL

        GiftID = qaed.ID;
        GiftType = qaed.type;
        TaskType = "Q";
        RequestTime = qaed.reqTime;
        TaskArrival=qaed.taskArr;
        printf("SANTA DID QA FOR ID: %d OF TYPE: %d \n", qaed.ID, qaed.type);
        tired = 2;
        pthread_mutex_unlock(&mutexQA);
        printf("SANTA SLEEPS\n");
        pthread_sleep(1); // TAKE YOUR TIME
        // printf("GONNA DO QA of type %d S\n", qaed.type);
        if (qaed.type == 4)
        { // CHECK PAINTING
            pthread_mutex_lock(&mutexWaiting);
            if (Contains(waitingQ, qaed.ID))
            { // CHECK PAINTING QUEUE
                pthread_mutex_unlock(&mutexWaiting);
                pthread_mutex_lock(&mutexPac);
                gettimeofday(&current_time, NULL);
                qaed.taskArr = current_time.tv_sec - start_time.tv_sec;
                Enqueue(packagingReq, qaed); // SEND TO PACKAGING
                printf("SANTA SEND PACKAGE FOR ID: %d OF TYPE: %d \n", qaed.ID, qaed.type);
                pthread_mutex_unlock(&mutexPac);
                // printf("SENT TO PACKING FROM QA S\n");
            }
            else
            {
                Enqueue(waitingQ, qaed);
                pthread_mutex_unlock(&mutexWaiting);
            }
        }
        else if (qaed.type == 5)
        { // CHECK ASSEMBLY
            pthread_mutex_lock(&mutexWaiting);
            if (Contains(waitingQ, qaed.ID))
            { // CHECK ASSEMBLY QUEUE
                pthread_mutex_unlock(&mutexWaiting);
                pthread_mutex_lock(&mutexPac);
                gettimeofday(&current_time, NULL);
```

The qa task is same with painting or packaging, the only difference is that in both types of gifts, santa needs to check the waiting queue.



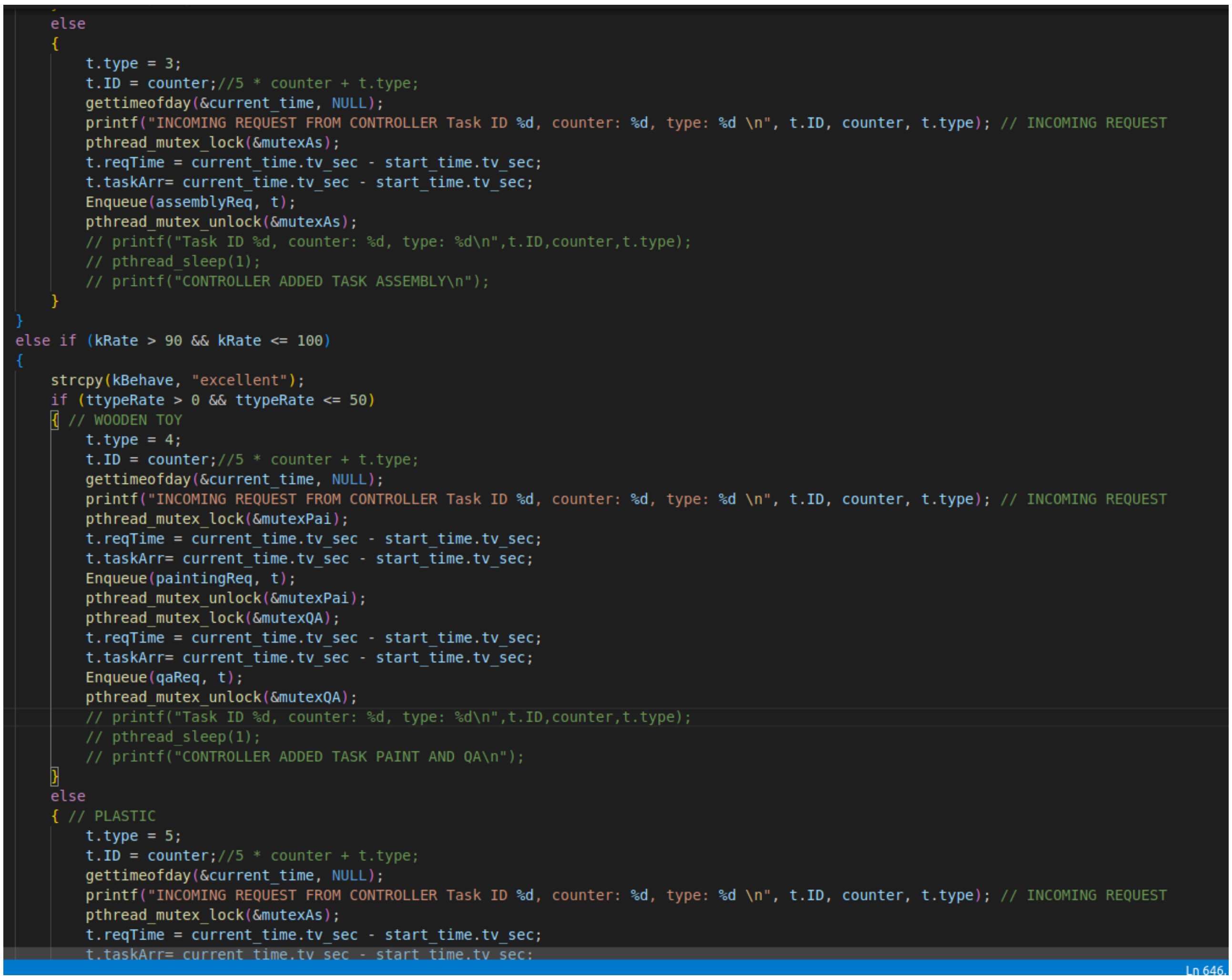
```c
                }
                else if (qaed.type == 5)
                { // CHECK ASSEMBLY
                    pthread_mutex_lock(&mutexWaiting);
                    if (Contains(waitingQ, qaed.ID))
                    { // CHECK ASSEMBLY QUEUE
                        pthread_mutex_unlock(&mutexWaiting);
                        pthread_mutex_lock(&mutexPac);
                        gettimeofday(&current_time, NULL);
                        qaed.taskArr = current_time.tv_sec - start_time.tv_sec;
                        Enqueue(packagingReq, qaed); // IF ASSEMBLY DONE SEND TO PACKAGING
                        printf("SANTA SEND PACKAGE FOR ID: %d OF TYPE: %d \n", qaed.ID, qaed.type);
                        pthread_mutex_unlock(&mutexPac);
                        // printf("SENT TO PACKING FROM QA S\n");
                    }
                    else
                    {
                        Enqueue(waitingQ, qaed);
                        pthread_mutex_unlock(&mutexWaiting);
                    }
                }
                // printf("DONE QA S\n");
                pthread_mutex_lock(&mutexFile);
                FILE *file;
                file = fopen("events1.log", "a+");
                gettimeofday(&current_time, NULL);
                TaskID = TaskID + 1;
                TT = current_time.tv_sec - TaskArrival-start_time.tv_sec;
                // TaskID GiftID GiftType TaskType RequestTime TaskArrival TT Responsible
                fprintf(file, "%d\t %d\t %d\t %s\t %d\t %d\t %d\t %s\t\n", TaskID, GiftID, GiftType, TaskType, RequestTime, TaskArrival, TT, "S");
                fclose(file);
                pthread_mutex_unlock(&mutexFile);
            }
            else
            {
                pthread_mutex_unlock(&mutexQA);
            }
        }
        tired = 0;
        // pthread_mutex_unlock(&mutexS); // RELEASE SANTA
        gettimeofday(&current_time, NULL);
    }
}
```

We do rewriting of the same code for it to be clear when reading the code.
The controller thread in our algorithm doesn’t manage the file management but it manages the gifts first entrance to their respected queues.
Controller:



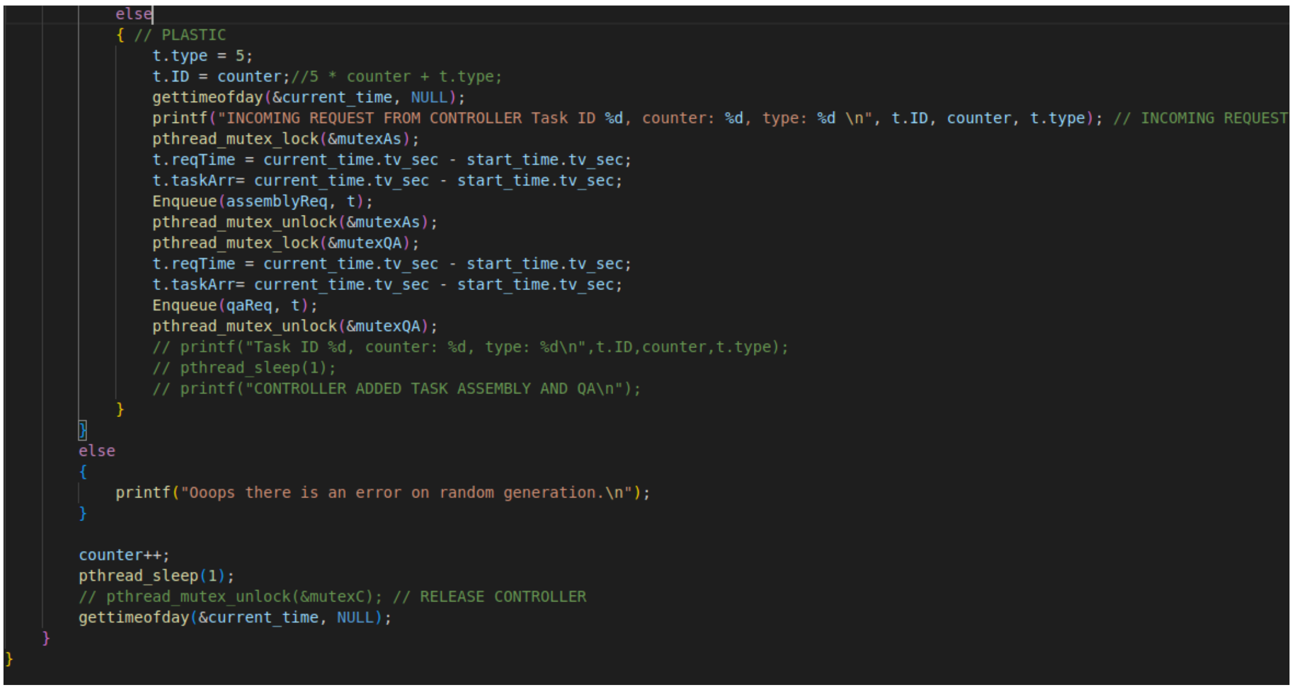
```c
void *ControlThread(void *arg)
{
    // struct timeval start_time;
    // gettimeofday(&start_time, NULL);
    while (start_time.tv_sec + simulationTime > current_time.tv_sec)
    {

        // pthread_mutex_lock(&mutexC); // LOCK CONTROLLER

        Task t;
        char kBehave[10] = "";              // KID CATEGORY
        int kRate = (rand() % 100) + 1;     // KID POSSIBILITY
        int ttypeRate = (rand() % 100) + 1; // TOY TYPE ~ WOODEN OR PLASTIC

        //if (kRate > 0 && kRate <= 10)
        if ( kRate < 0)
        {
            strcpy(kBehave, "bad");
        }
        else if (kRate > 0 && kRate <= 50)
        {
            strcpy(kBehave, "okay");
            t.type = 1;
            t.ID = counter;//5 * counter + t.type;
            gettimeofday(&current_time, NULL);
            printf("INCOMING REQUEST FROM CONTROLLER Task ID %d, counter: %d, type: %d \n", t.ID, counter, t.type); // INCOMING REQUEST
            pthread_mutex_lock(&mutexPac);
            t.reqTime = current_time.tv_sec - start_time.tv_sec;
            t.taskArr= current_time.tv_sec - start_time.tv_sec;
            Enqueue(packagingReq, t);
            pthread_mutex_unlock(&mutexPac);
            // printf("Task ID %d, counter: %d, type: %d\n",t.ID,counter,t.type);
            // pthread_sleep(1);
            // printf("CONTROLLER ADDED TASK PACKAGE\n");
        }
        else if (kRate > 50 && kRate <= 90)
        {
            strcpy(kBehave, "good");
            if (ttypeRate > 0 && ttypeRate <= 50)
            { // WOODEN TOY
                t.type = 2;
                t.ID = counter;//5 * counter + t.type;
                gettimeofday(&current_time, NULL);
                printf("INCOMING REQUEST FROM CONTROLLER Task ID %d, counter: %d, type: %d \n", t.ID, counter, t.type); // INCOMING REQUEST
                pthread_mutex_lock(&mutexPai);
                t.reqTime = current_time.tv_sec - start_time.tv_sec;
                t.taskArr= current_time.tv_sec - start_time.tv_sec;
                Enqueue(paintingReq, t);
```

We enqueued the tasks with given information. Randomness is generated in the beginning of the while cycle.



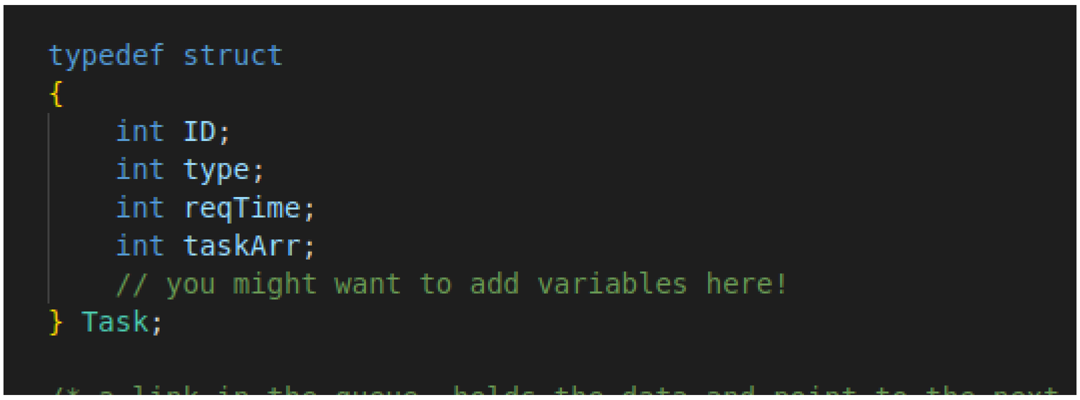
```
    else
    {
        t.type = 3;
        t.ID = counter;//5 * counter + t.type;
        gettimeofday(&current_time, NULL);
        printf("INCOMING REQUEST FROM CONTROLLER Task ID %d, counter: %d, type: %d \n", t.ID, counter, t.type); // INCOMING REQUEST
        pthread_mutex_lock(&mutexAs);
        t.reqTime = current_time.tv_sec - start_time.tv_sec;
        t.taskArr= current_time.tv_sec - start_time.tv_sec;
        Enqueue(assemblyReq, t);
        pthread_mutex_unlock(&mutexAs);
        // printf("Task ID %d, counter: %d, type: %d\n",t.ID,counter,t.type);
        // pthread_sleep(1);
        // printf("CONTROLLER ADDED TASK ASSEMBLY\n");
    }
}
else if (kRate > 90 && kRate <= 100)
{
    strcpy(kBehave, "excellent");
    if (ttypeRate > 0 && ttypeRate <= 50)
    { // WOODEN TOY
        t.type = 4;
        t.ID = counter;//5 * counter + t.type;
        gettimeofday(&current_time, NULL);
        printf("INCOMING REQUEST FROM CONTROLLER Task ID %d, counter: %d, type: %d \n", t.ID, counter, t.type); // INCOMING REQUEST
        pthread_mutex_lock(&mutexPai);
        t.reqTime = current_time.tv_sec - start_time.tv_sec;
        t.taskArr= current_time.tv_sec - start_time.tv_sec;
        Enqueue(paintingReq, t);
        pthread_mutex_unlock(&mutexPai);
        pthread_mutex_lock(&mutexQA);
        t.reqTime = current_time.tv_sec - start_time.tv_sec;
        t.taskArr= current_time.tv_sec - start_time.tv_sec;
        Enqueue(qaReq, t);
        pthread_mutex_unlock(&mutexQA);
        // printf("Task ID %d, counter: %d, type: %d\n",t.ID,counter,t.type);
        // pthread_sleep(1);
        // printf("CONTROLLER ADDED TASK PAINT AND QA\n");
    }
    else
    { // PLASTIC
        t.type = 5;
        t.ID = counter;//5 * counter + t.type;
        gettimeofday(&current_time, NULL);
        printf("INCOMING REQUEST FROM CONTROLLER Task ID %d, counter: %d, type: %d \n", t.ID, counter, t.type); // INCOMING REQUEST
        pthread_mutex_lock(&mutexAs);
        t.reqTime = current_time.tv_sec - start_time.tv_sec;
        t.taskArr= current_time.tv_sec - start_time.tv_sec;
```

We defined the kid types and toy types for easy debugging and easy reading of the code.



```
        else
        { // PLASTIC
            t.type = 5;
            t.ID = counter;//5 * counter + t.type;
            gettimeofday(&current_time, NULL);
            printf("INCOMING REQUEST FROM CONTROLLER Task ID %d, counter: %d, type: %d \n", t.ID, counter, t.type); // INCOMING REQUEST
            pthread_mutex_lock(&mutexAs);
            t.reqTime = current_time.tv_sec - start_time.tv_sec;
            t.taskArr= current_time.tv_sec - start_time.tv_sec;
            Enqueue(assemblyReq, t);
            pthread_mutex_unlock(&mutexAs);
            pthread_mutex_lock(&mutexQA);
            t.reqTime = current_time.tv_sec - start_time.tv_sec;
            t.taskArr= current_time.tv_sec - start_time.tv_sec;
            Enqueue(qaReq, t);
            pthread_mutex_unlock(&mutexQA);
            // printf("Task ID %d, counter: %d, type: %d\n",t.ID,counter,t.type);
            // pthread_sleep(1);
            // printf("CONTROLLER ADDED TASK ASSEMBLY AND QA\n");
        }
    }
    else
    {
        printf("Ooops there is an error on random generation.\n");
    }

    counter++;
    pthread_sleep(1);
    // pthread_mutex_unlock(&mutexC); // RELEASE CONTROLLER
    gettimeofday(&current_time, NULL);
  }
}
```

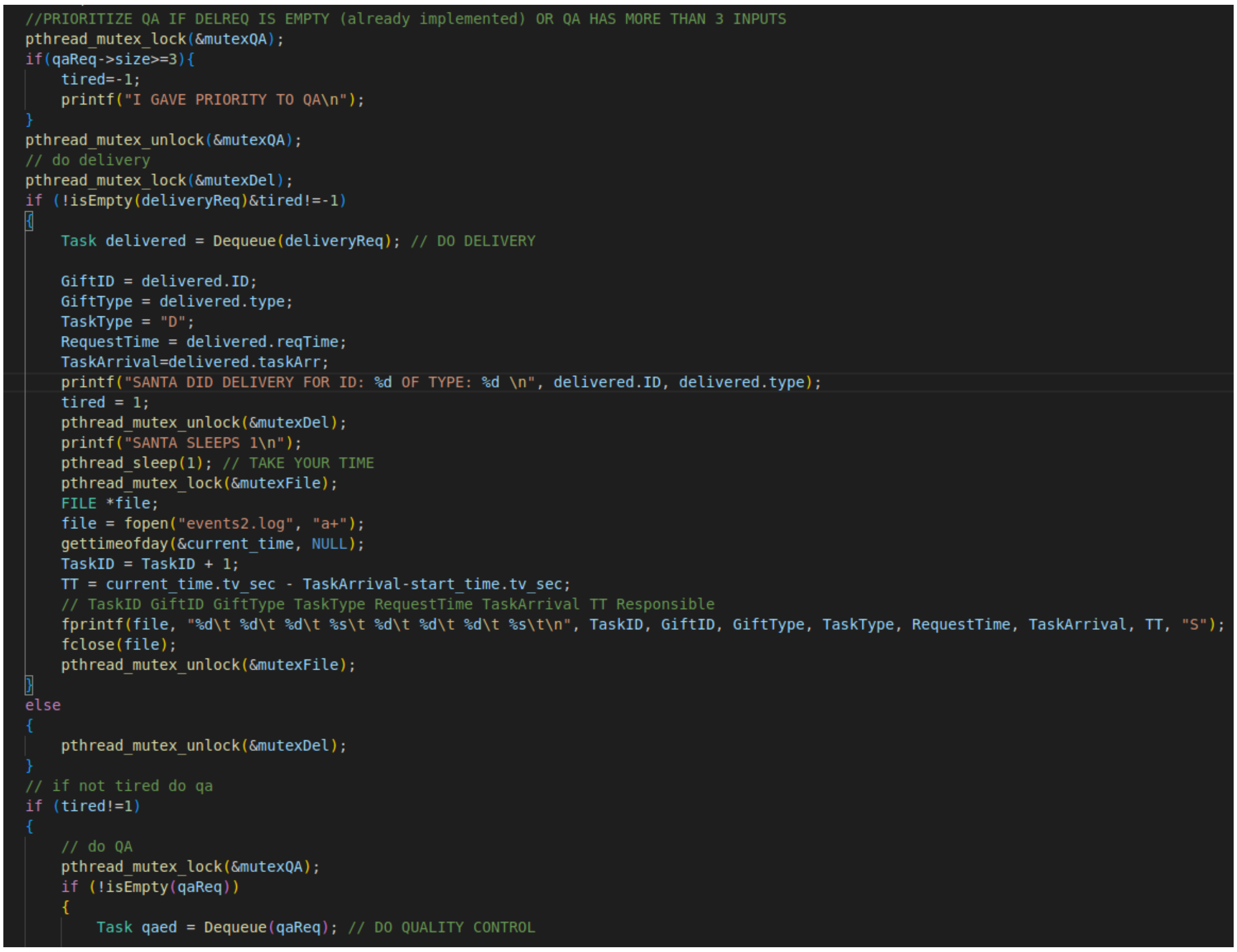
All enqueues are done by safety locks and every task is enqueued by initializing their id, type,request type and task arrival time.



```c
typedef struct
{
    int ID;
    int type;
    int reqTime;
    int taskArr;
    // you might want to add variables here!
} Task;

/* a link in the queue, holds the data and point to the next
```

To easily initialize and update arrival and request times we defined new integers in data structure in queue.

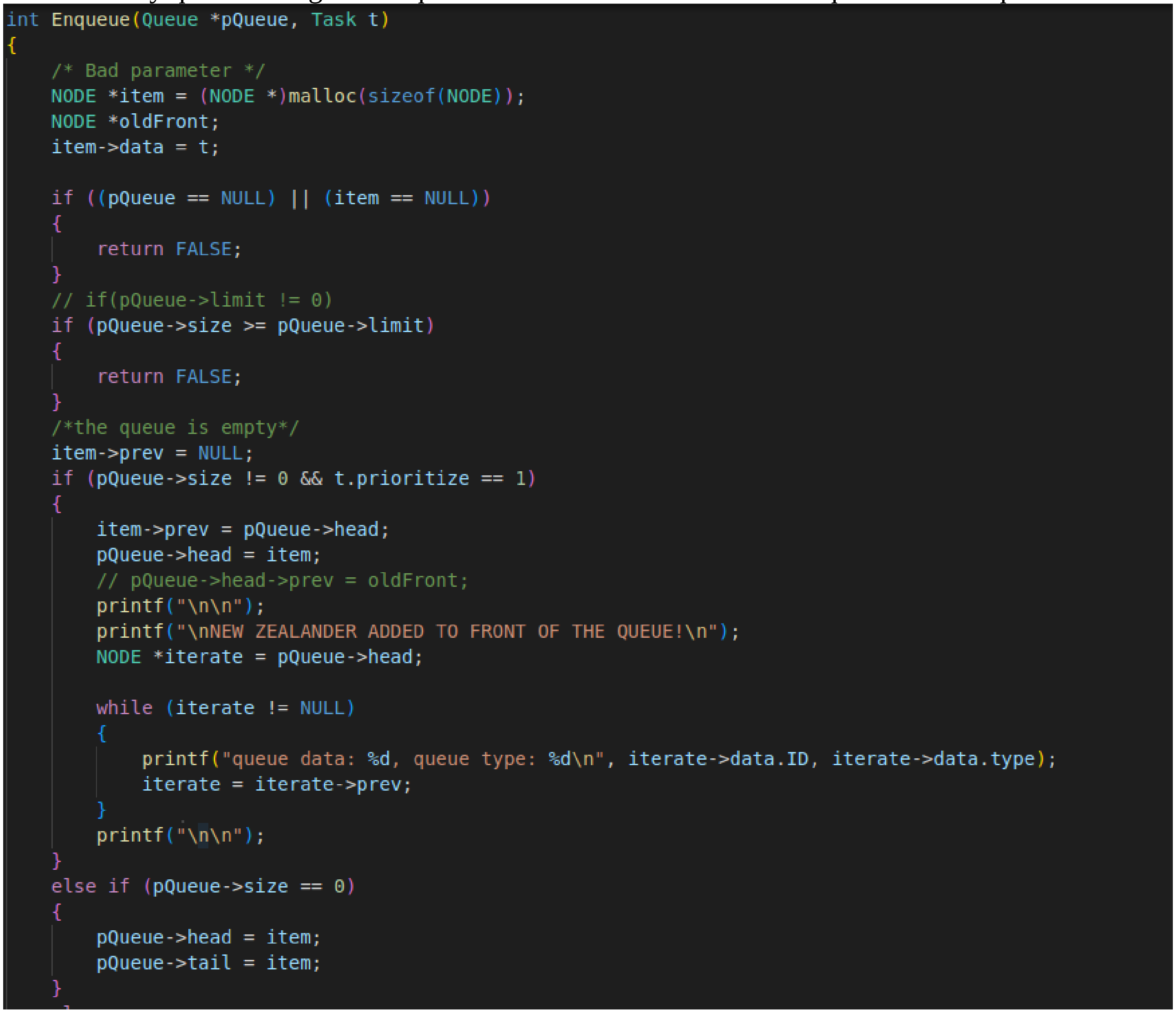
PART2:



```c
//PRIORITIZE QA IF DELREQ IS EMPTY (already implemented) OR QA HAS MORE THAN 3 INPUTS
pthread_mutex_lock(&mutexQA);
if(qaReq->size>=3){
    tired=-1;
    printf("I GAVE PRIORITY TO QA\n");
}
pthread_mutex_unlock(&mutexQA);
// do delivery
pthread_mutex_lock(&mutexDel);
if (!isEmpty(deliveryReq)&tired!=-1)
{
    Task delivered = Dequeue(deliveryReq); // DO DELIVERY

    GiftID = delivered.ID;
    GiftType = delivered.type;
    TaskType = "D";
    RequestTime = delivered.reqTime;
    TaskArrival=delivered.taskArr;
    printf("SANTA DID DELIVERY FOR ID: %d OF TYPE: %d \n", delivered.ID, delivered.type);
    tired = 1;
    pthread_mutex_unlock(&mutexDel);
    printf("SANTA SLEEPS 1\n");
    pthread_sleep(1); // TAKE YOUR TIME
    pthread_mutex_lock(&mutexFile);
    FILE *file;
    file = fopen("events2.log", "a+");
    gettimeofday(&current_time, NULL);
    TaskID = TaskID + 1;
    TT = current_time.tv_sec - TaskArrival-start_time.tv_sec;
    // TaskID GiftID GiftType TaskType RequestTime TaskArrival TT Responsible
    fprintf(file, "%d\t %d\t %d\t %s\t %d\t %d\t %d\t %s\t\n", TaskID, GiftID, GiftType, TaskType, RequestTime, TaskArrival, TT, "S");
    fclose(file);
    pthread_mutex_unlock(&mutexFile);
}
else
{
    pthread_mutex_unlock(&mutexDel);
}
// if not tired do qa
if (tired!=1)
{
    // do QA
    pthread_mutex_lock(&mutexQA);
    if (!isEmpty(qaReq))
    {
        Task qaed = Dequeue(qaReq); // DO QUALITY CONTROL
```

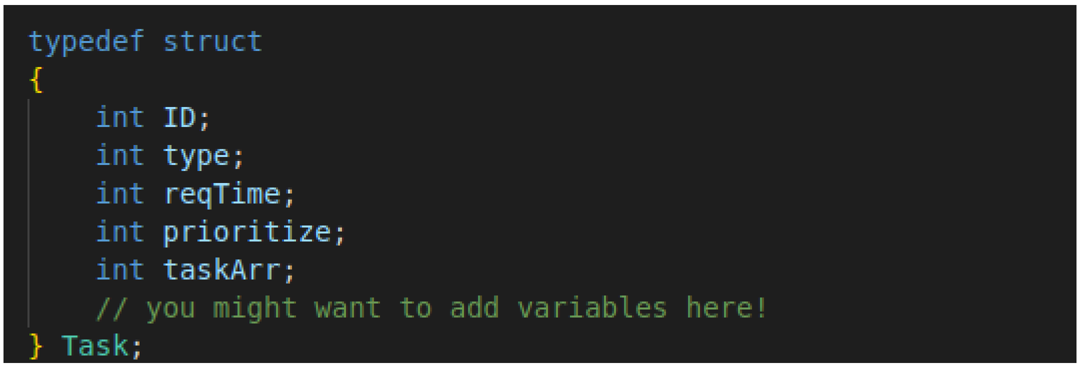
To achieve the required deadlock prevention, I used tired variable more portably. Originally tired was a variable which is used by a thread to regulate its jobs. For instance for santa, tired was started of with a 0 value, then if it gets into delivery its tired variable is updated to 1. This way once the delivery job is done and if block is exited, the next block of qa task wont be entered with the prevention of its tired not being a nonzero value. For priority giving, In the beginning of every santa while loop cycle, we added an additional condition; if the size of the qa queue is bigger than or equal to 0, the tired value will be assigned to -1. Queue size of qaReq being bigger than 3 means it has at least 3 tasks waiting for qa job, so the queue being not empty is already checked here. After that we only changed the condition of entering to delivery job's if block to not having an empty delivery queue and not having tired as -1, from only not having an empty delivery queue. The emptiness of the queues were already checked before each job start, and the priority was given to delivery queue in terms of order, so check of delivery queue being empty or not was already a condition which will give the priority to the qa queue in case of having an empty delivery queue.

PART3:
For this part, we needed to introduce new new zealanders to queues. These new zealanders are prioritized for each task their gifts require. To solve this, the most logical way was to add them in front of every queue their gift is enqueued into. So we modified the enqueue code in queue class.



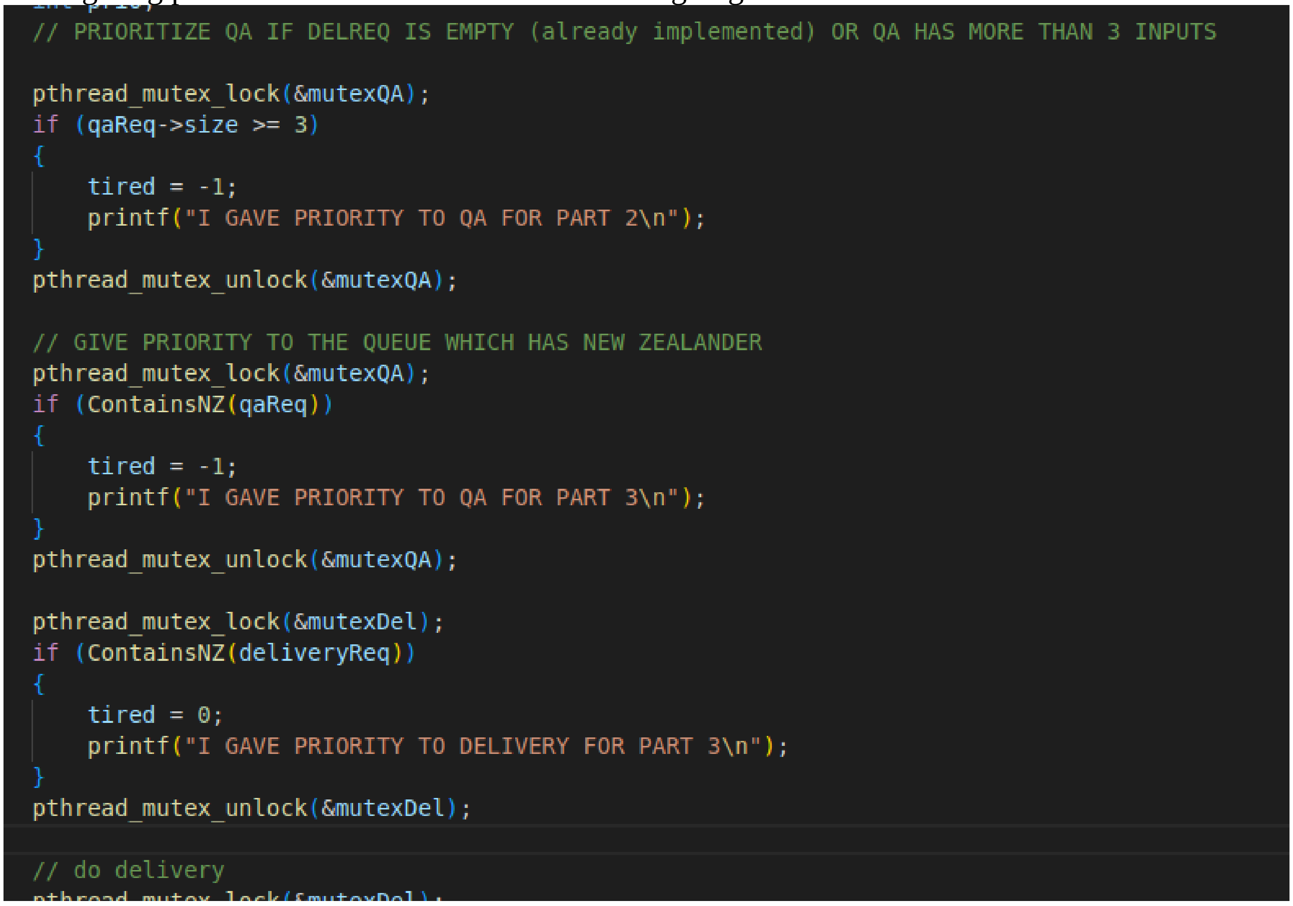
```c
int Enqueue(Queue *pQueue, Task t)
{
    /* Bad parameter */
    NODE *item = (NODE *)malloc(sizeof(NODE));
    NODE *oldFront;
    item->data = t;

    if ((pQueue == NULL) || (item == NULL))
    {
        return FALSE;
    }
    // if(pQueue->limit != 0)
    if (pQueue->size >= pQueue->limit)
    {
        return FALSE;
    }
    /*the queue is empty*/
    item->prev = NULL;
    if (pQueue->size != 0 && t.prioritize == 1)
    {
        item->prev = pQueue->head;
        pQueue->head = item;
        // pQueue->head->prev = oldFront;
        printf("\n\n");
        printf("\nNEW ZEALANDER ADDED TO FRONT OF THE QUEUE!\n");
        NODE *iterate = pQueue->head;

        while (iterate != NULL)
        {
            printf("queue data: %d, queue type: %d\n", iterate->data.ID, iterate->data.type);
            iterate = iterate->prev;
        }
        printf("\n\n");
    }
    else if (pQueue->size == 0)
    {
        pQueue->head = item;
        pQueue->tail = item;
    }
```

When a new task arrives the queue, first we check the size and it being prioritized or not. If the queue is not empty and the task is a new zealander prioritized task, we enqueue the item at the beginning of the queue as a head. We added some debugging prints just to debug this part easier. Prioritized is a feature of task structure.



```c
typedef struct
{
    int ID;
    int type;
    int reqTime;
    int prioritize;
    int taskArr;
    // you might want to add variables here!
} Task;
```

We also added another contains function with name ContaninsNZ to check if the queue has a new zealander to give priority to the queue which has a new zelander as a head when we try to assign elf A, elf B and Santa to their jobs. For instance, if Santa has nonempty delivery queue and non empty qa queue, originally delivery was prioritized by if-else ordering, but in part 2 the qa queue was prioritized if it has more than 2 jobs waiting, when asked the TA priority of being a new zealander was above having 3 or more qa jobs waiting in the queue. So, in the beginning of every while cycle when giving priorities we ordered tired variable assigning.



```c
// PRIORITIZE QA IF DELREQ IS EMPTY (already implemented) OR QA HAS MORE THAN 3 INPUTS

pthread_mutex_lock(&mutexQA);
if (qaReq->size >= 3)
{
    tired = -1;
    printf("I GAVE PRIORITY TO QA FOR PART 2\n");
}
pthread_mutex_unlock(&mutexQA);

// GIVE PRIORITY TO THE QUEUE WHICH HAS NEW ZEALANDER
pthread_mutex_lock(&mutexQA);
if (ContainsNZ(qaReq))
{
    tired = -1;
    printf("I GAVE PRIORITY TO QA FOR PART 3\n");
}
pthread_mutex_unlock(&mutexQA);

pthread_mutex_lock(&mutexDel);
if (ContainsNZ(deliveryReq))
{
    tired = 0;
    printf("I GAVE PRIORITY TO DELIVERY FOR PART 3\n");
}
pthread_mutex_unlock(&mutexDel);

// do delivery
```

We first check qa queue size because it can be overwritten by new zealander priority. Then we checked each queue having a new zealander or not, to give the priority to the job which has a new zealander in its task queue, then we enter job blocks.

Keeping logs is already explained in part1. It is a file writing task. But since the file can be handled by multiple threads, the mutex lock safety is applied to it as well.