CS:3620 — Fall 2022 — Homework 4

This homework is about the pthread API in Linux. It consists of two tasks. In Task 1, you need to change some lines of the provided c file, and make it able to find solutions for the assigned problems. In Task 2, you will see a multi-threaded program with some bugs. These bugs will lead to race conditions and/or dead locks. You are supposed to find all the bugs from the program code and describe where they are, how to modify them, and why you make these changes.

This homework is due on Nov 21th, 2022 at 11:59 PM CDT.

General Requirements Submit your homework as a *single* `tar` file on ICON. When unpacked, the `tar` file must have the following directory structure:

```
⟨your_HawkID⟩
    task1.c
    task2.pdf
```

Notice that, for Task 1, you do not have to provide a compilation script. Your code will be compiled using the following command:

```
gcc -ggdb -O0 -o task task.c -lpthread -lcrypto
```

To compile correctly your code, it may be necessary (in Ubuntu 16.04 64bit) to install the package libssl-dev:

```
sudo apt-get install libssl-dev
```

`task.c` needs to be a modified version of the file `task.c` provided in the tar file of this homework, as explained below. For Task 2, we expect you to provide the answer by reading the code. `You do not have to run the program.`

Task1 (60 Points)

Your overall goal is to change the provided task.c file to make it able to find solutions for the assigned problems using multiple parallel threads. I will now describe what the provided code does. Your code is not to re-implement what the provided code is already doing, but just to change it, as described in the section Your modifications.

The provided task.c code

The provided code implements a program that takes two or more command line arguments: ⟨nthreads⟩ ⟨challenges...⟩. ⟨nthreads⟩ specifies the number of parallel worker threads the program uses to solve the given challenges (as better explained below). ⟨challenges...⟩ is a list of one or more integer numbers in base 10.

```
Examples:
./task 1 12
```

The program will solve the challenge 12, using 1 worker thread.

```
./task 1 12 1000
```

The program will solve the challenges 12 and 1000, using 1 worker thread.

```
./task 3 12 1000 123
```

The program will solve the challenges 12, 1000, and 123, using 3 worker threads.

For every given challenge, the program output a line to stdout, with the following format:
⟨challenge⟩ ⟨solution1⟩ ⟨solution2⟩ ⟨solution3⟩ ⟨solution4⟩ ⟨solution5⟩ ⟨solution6⟩ ⟨solution7⟩ ⟨solution8⟩

For instance, when run in this way:

```
./task 1 12 1000 123
```

The output is:

```
12 1720 15424 184447 262378 431435 461772 526096 812453 1000
62444 91459 120212 147426 189835 281930 338887 542638 123
113731 150733 534218 618885 677519 694956 861114 944627
```

The 8 solutions outputted for each challenge satisfy the following 3 conditions:

1. For all the solutions, applying the algorithm SHA256 (To solve this homework, you are not required to know the details of this algorithm.) to a solution returns a hash starting with the given challenge.1 Every solution is considered as a little-endian unsigned 64bit number, and every challenge is considered as a little-endian unsigned 16bit number.

2. All the solutions have a different last digit (considering them in base 10).

3. All the solutions are not divisible for any number in the range [1000000,1500000].

The provided code generates correct solutions when running with ⟨`nthreads`⟩ equal to 1. However, since it does not use any locking mechanism, it fails when running with ⟨`nthreads`⟩ bigger than 1. The code may fail in different ways: entering an infinite loop, returning solutions which do not follow the 3 conditions above, triggering a `Segmentation fault`, ...

Your modifications

Your goal is to modify the provided code so that it respects the following properties.

1. Your modified code outputs solutions in the same format than the original one.

2. Your code needs to create and use a number of worker threads equal to ⟨`nthreads`⟩.

3. When running with ⟨`nthreads`⟩ equal to 1, your modified code must not run more than 20% slower than the original code.

4. For each challenge, your modified code has to print 8 solutions, still respecting the 3 conditions explained above. This property must be true when your code is run with any number of ⟨`nthreads`⟩ between 1 and 100 (included).

5. Assuming that you are using a machine with at least 4 vCPUs2, when the value of ⟨`nthreads`⟩ is 2, 3, 4, or 5, your code needs to run at least 25% faster than when the value of ⟨`nthreads`⟩ is 1.

6. On Line 56 in task.c `first_tried_solution` is initiated as 0 for each thread. You need to make some modifications in the intiation of `first_tried_solution` such that each thread is solving challenge in a different solution space. More specifcially, initation value of `first_tried_solution` will be different for each thread.

The provided `test.py` script tries to automatically verify property 1, property 4, and property 5. You can run it with the following comand:. `python test.py ./task`.

This script prints to `stdout` `ERROR` when a test fails, and `SUCCESS` when it succeeds, alongside debugging output.

If you run `test.py` with the original provided code for task, some tests will succeed, but any test involving a value of ⟨`nthreads`⟩ bigger than 1 will likely fail.

The script `test.py` assumes that your machine has at least 4 vCPUs and it is not under heavy load.

Hints and Suggestions

Do not modify any provided code verifying the 3 conditions mentioned above. Your goal is just to make the code working with multiple worker threads (i.e., values of ⟨`nthreads`⟩ bigger than 1), not to implements numeric checks.

`You can solve this homework by adding/modifying less than 20 lines of code.`.

Before starting writing any code, spend your time understanding the provided code and the homework description. Also, spend some time testing your code, since, when working with threads, code can work correctly sometimes and incorrectly other times, in unexpected ways.

There are 3 main aspects that you need to consider to fix the provided code.

First of all, in the provided code, all the worker threads start looking for solutions from the value 0. This needs to be changed so that every worker thread checks different candidate solutions.

Then, two global variables are read/written by different worker threads: `found_solutions` and `solutions`. Accessing them should be protected using an appropriate locking mechanism. I suggest to use a `read-write lock`, but there may be other ways to solve this homework. The lock can be acquired for reading when just checking the value of one of these variables, and acquired for writing when modifying it.

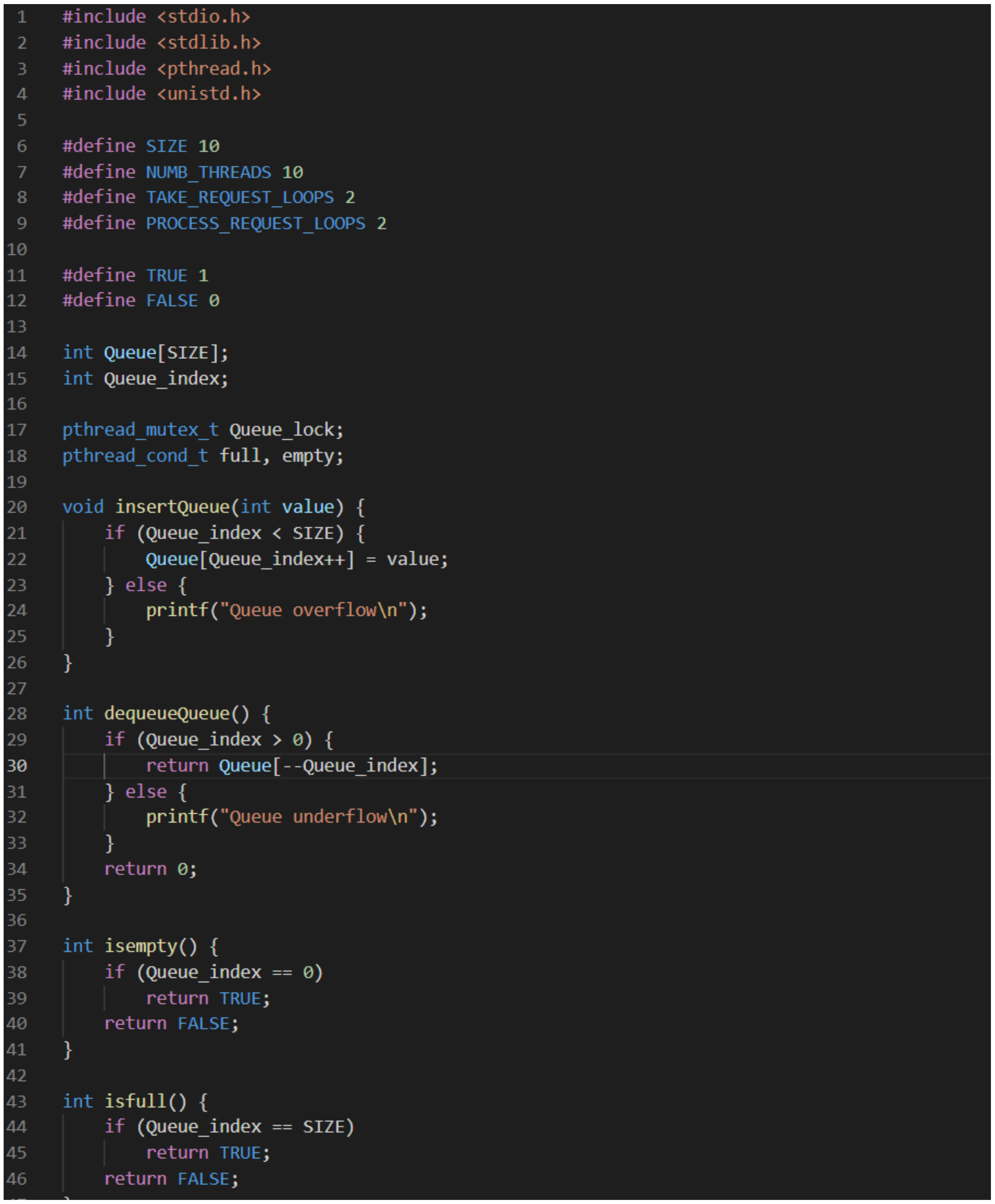
Finally, any worker thread should check if all the required 8 solutions have been found, and, if this is the case, terminate. This can happen at any time during the execution of the code in `worker_thread_function`. You may need to add instructions checking this.

Remember always to unlock an acquired lock. It is easy to make mistakes and write code that, under certain conditions, forgets to unlock a lock. This will most likely cause your code to stall.

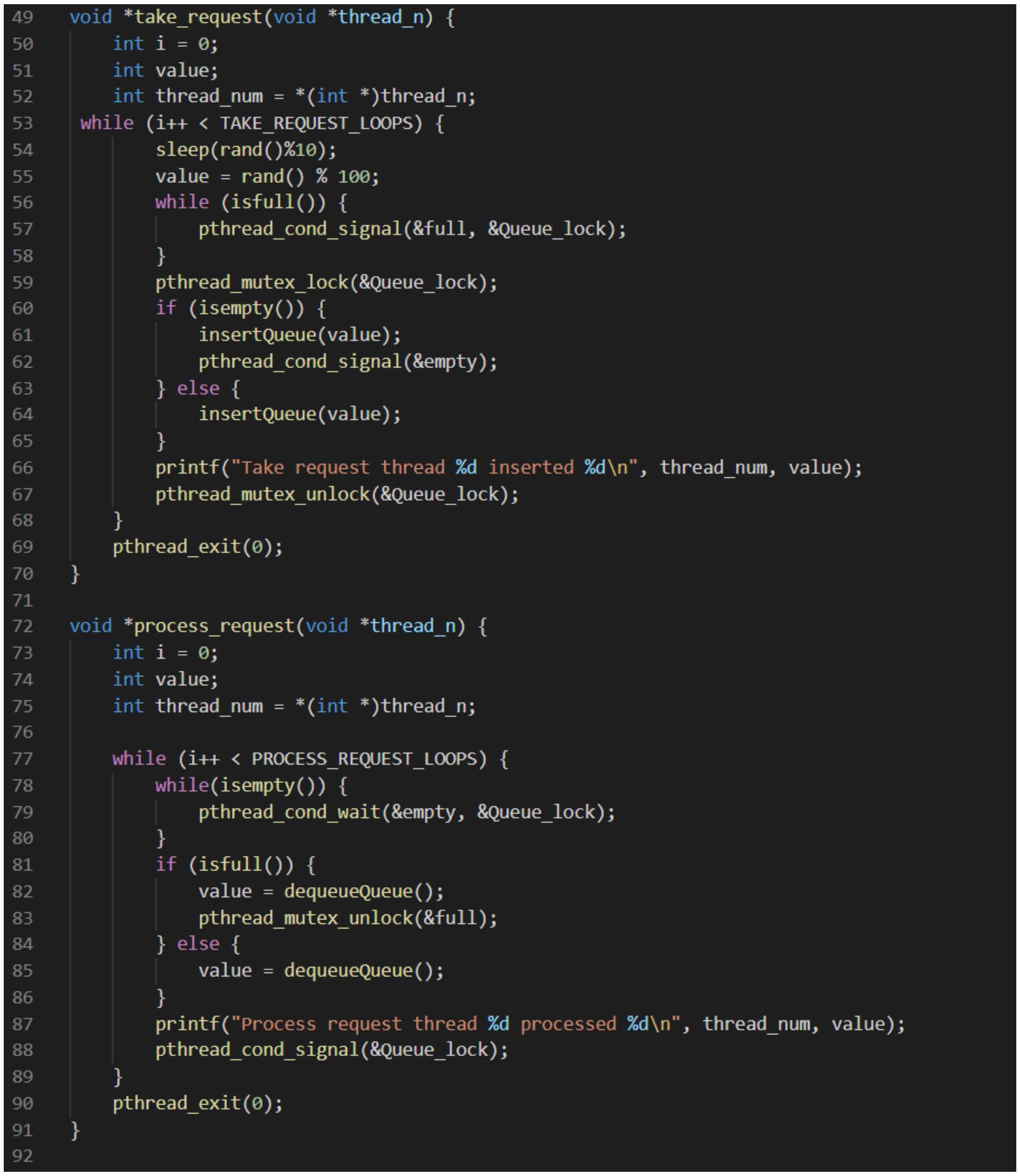
Task2 (40 Points)

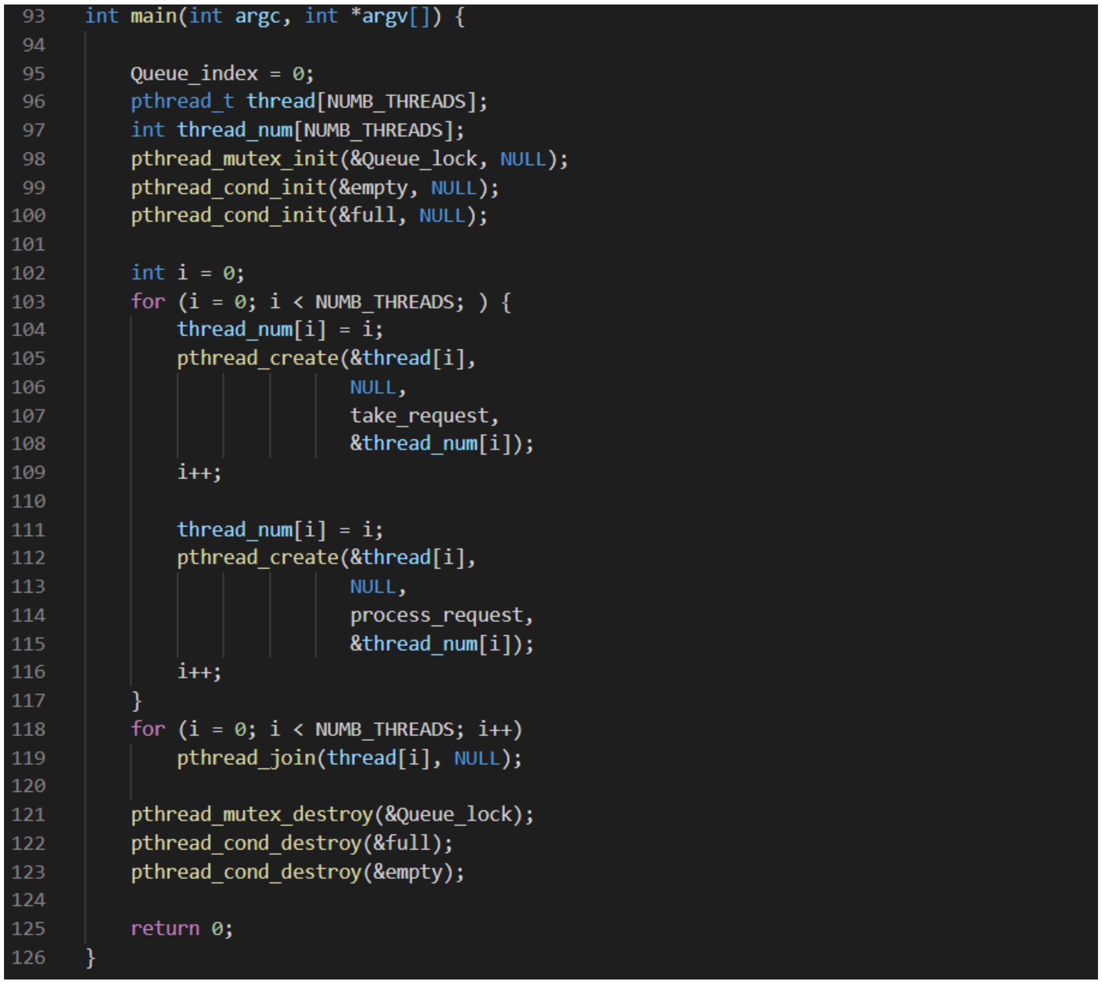
In multi-threaded programs, the producer and consumer problem is a classic topic. There is a student who want to simulate how producer and consumer work in the following code segment. But he made some mistakes when implementing it. You need to find all the mistakes he made, and try to describe them in details. Please at least include the following information for each bug you find in your answer: (1) The line number where the bug is; (2) The modification you want to make to fix the bug; (3) The rationale why you do so.

After your fix, the code should correctly implement the producer and consumer problem without race conditions or dead locks.



```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define SIZE 10
#define NUMB_THREADS 10
#define TAKE_REQUEST_LOOPS 2
#define PROCESS_REQUEST_LOOPS 2

#define TRUE 1
#define FALSE 0

int Queue[SIZE];
int Queue_index;

pthread_mutex_t Queue_lock;
pthread_cond_t full, empty;

void insertQueue(int value) {
    if (Queue_index < SIZE) {
        Queue[Queue_index++] = value;
    } else {
        printf("Queue overflow\n");
    }
}

int dequeueQueue() {
    if (Queue_index > 0) {
        return Queue[--Queue_index];
    } else {
        printf("Queue underflow\n");
    }
    return 0;
}

int isempty() {
    if (Queue_index == 0)
        return TRUE;
    return FALSE;
}

int isfull() {
    if (Queue_index == SIZE)
        return TRUE;
    return FALSE;
```


```c
void *take_request(void *thread_n) {
    int i = 0;
    int value;
    int thread_num = *(int *)thread_n;
 while (i++ < TAKE_REQUEST_LOOPS) {
        sleep(rand()%10);
        value = rand() % 100;
        while (isfull()) {
            pthread_cond_signal(&full, &Queue_lock);
        }
        pthread_mutex_lock(&Queue_lock);
        if (isempty()) {
            insertQueue(value);
            pthread_cond_signal(&empty);
        } else {
            insertQueue(value);
        }
        printf("Take request thread %d inserted %d\n", thread_num, value);
        pthread_mutex_unlock(&Queue_lock);
    }
    pthread_exit(0);
}

void *process_request(void *thread_n) {
    int i = 0;
    int value;
    int thread_num = *(int *)thread_n;

    while (i++ < PROCESS_REQUEST_LOOPS) {
        while(isempty()) {
            pthread_cond_wait(&empty, &Queue_lock);
        }
        if (isfull()) {
            value = dequeueQueue();
            pthread_mutex_unlock(&full);
        } else {
            value = dequeueQueue();
        }
        printf("Process request thread %d processed %d\n", thread_num, value);
        pthread_cond_signal(&Queue_lock);
    }
    pthread_exit(0);
}
```



```c
int main(int argc, int *argv[]) {

    Queue_index = 0;
    pthread_t thread[NUMB_THREADS];
    int thread_num[NUMB_THREADS];
    pthread_mutex_init(&Queue_lock, NULL);
    pthread_cond_init(&empty, NULL);
    pthread_cond_init(&full, NULL);

    int i = 0;
    for (i = 0; i < NUMB_THREADS; ) {
        thread_num[i] = i;
        pthread_create(&thread[i],
                       NULL,
                       take_request,
                       &thread_num[i]);
        i++;

        thread_num[i] = i;
        pthread_create(&thread[i],
                       NULL,
                       process_request,
                       &thread_num[i]);
        i++;
    }
    for (i = 0; i < NUMB_THREADS; i++)
        pthread_join(thread[i], NULL);

    pthread_mutex_destroy(&Queue_lock);
    pthread_cond_destroy(&full);
    pthread_cond_destroy(&empty);

    return 0;
}
```