Operating Systems (0368-2162)

Parallel Queue Assignment

Individual work policy

The work you submit in this course is required to be the result of your individual effort only. You may discuss concepts and ideas with others, but **you must program individually**. **You should never observe another student's code**, from this or previous semesters.

Students violating this policy will receive a **250 grade in the course** ("did not complete course requirements").

1 Introduction

The goal of this assignment is to gain experience with threads. In this assignment, you will create a generic concurrent FIFO queue that supports enqueue and dequeue operations.

2 Assignment description

Implement the following library in a file named `queue.c`. The following details the specification of the library.

```
void initQueue(void);
```

This function will be called before the queue is used. This is your chance to initialize your data structure.

```
void destroyQueue(void);
```

This function will be used for cleanup when the queue is no longer needed. It is possible for `initQueue` to be called afterwards.

```
void enqueue(void*);
```

Adds an item to the queue.

```
void* dequeue(void);
```

Remove an item from the queue. Will block if empty.

```
bool tryDequeue(void**);
```

Try to remove an item from the queue. If succeeded, return it via the argument and return true. If the queue is empty, return false and leave the pointer unchanged.

```
size_t size(void);
```

Return the current amount of items in the queue.

```
size_t waiting(void);
```

Return the current amount of threads waiting for the queue to fill. This call should not be blocked due to concurrent operations, i.e., you may not take a lock at all.

```
size_t visited(void);
```

Return the amount of items that have passed inside the queue (i.e., inserted and then removed). This should not block due to concurrent operations, i.e., you may not take a lock at all.

Thread interface: You must use C threads, as described in the recitation. If you've heard of or know pthreads—you may not use them in this assignment.

Error handling & termination:

- You can assume malloc does not fail.
- No need to check for errors in calls to `mtx_*` and `cnd_*` functions.

Correctness requirements:

- You may assume no calls will be made to the queue before initQueue or after destroyQueue (unless initQueue is called again).
- The library should be thread-safe. For example, an item added to the queue once, should not be returned from two dequeue calls.
- Make sure no resource deadlocks are possible in your program.
- Threads should run in parallel. Only accesses to shared data should be synchronized with locks.
- The queue must satisfy the following:
  1. Dequeue should be distributed to sleeping threads in FIFO order. If the queue is empty and there are k threads sleeping, then the next k items added to the queue should be handled by these threads according to the order in which they went to sleep (i.e., the first item added should be handled by the first thread that went to sleep, etc.). Specifically, if the queue is empty and there are k threads sleeping, inserting into the queue must make sure that the oldest sleeping thread will process the inserted item.
  2. A thread may put itself to sleep (to wait for an item) only if there are no items available for it when it **arrives** at the queue. (That is, if the queue is empty or if every item in the queue has to be handled by a different thread, due to the FIFO requirement.) If a thread goes to sleep after arriving at the queue, it should be woken up only once there is an item available for it to process (according to the FIFO requirement). A sleeping thread **must not** wake up only to go back to sleep again. (Assume that spurious wakeups are not possible.)
- For the functions that return size_t and are not allowed to lock. It is OK if the number is not exact due to concurrent threads taking their place. If no concurrent actions take place the number should be exact. Undefined behavior is not allowed.

3 Relevant functions & system calls

1. Learn about and (possibly) use the following: `thrd_create()`, `thrd_join()`, `cnd_init()`, `cnd_wait()`, `cnd_signal()`, `cnd_broadcast()`, `thrd_exit()`, `mtx_init()/lock()/trylock()/unlock()`,

4 Submission instructions

1. Submit just your `queue.c` file. Document your code with explanations for every non-trivial part of your code. Help the grader understand your solution and the flow of your code.

2. The program must compile cleanly on the course udocker (no errors or warnings) when the following command is run in a directory containing the `queue.c` file:

```
gcc -O3 -D_POSIX_C_SOURCE=200809 -Wall -std=c11 -pthread -c queue.c
```

IMPORTANT: This assignment should run on the udocker image on the `nova` server.