Thread-Safe Queue

This homework asks you to modify the attached program to make it thread safe. You will create two POSIX threads: one to produce information and one to consume the information. You must then modify the queue to make it thread safe and efficient (with mutexes and condition variables).

1. Your program **shall** build with the supplied makefile. You should not modify the function of the program. You should not need to modify the makefile. Your program must be able to run in my VM. It should not crash and it should print all the numbers correctly.
2. Your program **shall** create two threads:
   (a) One thread **shall** execute the `producer` method.
   (b) The other thread **shall** execute the `consumer` method.
3. Your program **shall** modify the queue methods to add thread safety:
   (a) The `push` method **shall** only access the queue if it holds a lock on the queue.
   (b) The `pop` method **shall** only access the queue if it holds a lock on the queue.
4. Your program **shall** modify the queue methods to add efficiency:
   (a) The `pop` method **shall** efficiently wait for items to be added to the queue. If the queue is empty, it should block until an item is added.
   (b) The `push` method **shall** notify threads waiting in the pop method when an item is added to the queue.

The `queue` header file and `main` source file can be downloaded from blackboard and these are provided below for reference:

```
#include <stdlib.h>
#include <stdbool.h>

typedef struct element {
   void* data;
   size_t size;
   struct element* next;
} element_t;

typedef struct {
   element_t* head;
   element_t* tail;
} queue_t;
```

```
void queue_init(queue_t* queue)
{
   queue->head = NULL;
   queue->tail = NULL;
   return;
}

void queue_destroy(queue_t* queue)
{
   if (queue != NULL)
   {
      while(queue->head != NULL)
      {
         element_t* temp = queue->head->next;
         free(queue->head);
         queue->head = temp;
      }
   }
   return;
}

bool push(queue_t* queue, void* data, size_t size)
{
   element_t* next = malloc(sizeof(element_t));
   next->data = data;
   next->size = size;
   next->next = NULL;

   if (queue->head == NULL)
   {
      queue->head = next;
   }
   if (queue->tail != NULL)
   {
      queue->tail->next = next;
   }
   queue->tail = next;

   return true;
}

bool pop(queue_t* queue, void** data, size_t* size)
{
   element_t* top = queue->head;
```

```
    if (queue->head != NULL)
    {
        if (queue->tail == top)
        {
            queue->tail = NULL;
        }
        queue->head = top->next;
        *data = top->data;
        *size = top->size;
        free(top);
        return true;
    }
    return false;
}
```

```
#include "stdio.h"
#include "stdlib.h"
#include "pthread.h"
#include "string.h"
#include "unistd.h"
#include "queue.h"

void* producer(void* arg)
{
    queue_t* queue = (queue_t*)arg;
    size_t i = 0;
    for(i = 0; i < 100000; ++i)
    {
        push(queue, (void*)i, sizeof(i));
        if ((i % 10000) == 0) { sleep(1); }
    }
    return NULL;
}

void* consumer(void* arg)
{
    queue_t* queue = (queue_t*)arg;
    size_t i = 0;
    for(i = 0; i < 100000; ++i)
    {
        size_t i = 0;
        size_t i_size = 0;
        if (pop(queue, (void**)&i, (void*)&i_size))
        {
```

```
            /*assert(i_size == sizeof(i));*/
            fprintf(stderr, "got %d from queue\n", i);
        }
    }
    return NULL;
}

int main(int argc, char* argv[])
{
    pthread_t pro_th, con_th;
    queue_t queue;
    queue_init(&queue);

    producer(&queue);  // run on other thread
    consumer(&queue);  // run on other thread

    queue_destroy(&queue);

    return EXIT_SUCCESS;
}
```