CSC 501
P4: Thread Library

General Description:

The code implements the routines used for a non pre-emptive user level thread library. All basic thread routines and semaphore routines have been covered. Deadlocks have been handled by passing the control back to the calling UNIX thread.

Design:

Data Structures:

The queues:
For the threads, two queues have been used- the rdy_q (ready threads) and the blk_q (blocked threads). In addition to these, each instantiated semaphore has a queue to keep track of the threads waiting on it. Each thread keeps links to its children threads and links to threads waiting for it in queues.
All these queues have the same structure. Every node has a pointer each to the previous and next nodes (doubly linked queue) and a pointer to the structure of the thread it represents. Finally these are circular queues- the first and the last node have links to each other. Care has been take to make the pointer to the head of each queue a NULL pointer whenever the queue is empty.

```
struct mythread_q
{
  struct mythread *my_thread;     //pointer to the thread represented by this node
  struct mythread_q *prev, *next; //pointers to previous and next nodes
};
```

The thread structure:
Each thread needs to maintain some data about its own state. The thread maintains the data that is initially passed to MyThreadInit during its creation, ie., the function pointer and the arguments. The thread also needs to keep track of its context data to enable context switching.
In addition to these, the thread structure maintains its own Identity number (long int) for identification purposes, a queue for keeping track of its children, a queue for keeping track of threads that are waiting for it and it must free before exiting, a pointer to its parent thread and a count of the threads that the thread is itself waiting for (this essentially is binary).

```
struct mythread //tcb
{
        struct mythread *parent;
        long int my_id;
        struct mythread_q *children;
        void (*thread_fxn)(void*);
        void *args;
        struct mythread_q *waitq; //who all are waiting for me
        int wait_for_count;       //how many people am i waiting for
        ucontext_t context;
};
```

The semaphore structure:
This structure contains the initial count of the semaphore, the present count, validity of semaphore and a queue structure to keep track of threads waiting on the semaphore.

```
typedef struct sem_str
{
        int initial;
        int count;
        int valid;
        struct mythread_q *sem_q;
} sem_str;
```

Major Functions:

In all the functions below, care has been taken to check if inputs to functions are invalid/NULL and if an operation makes the ready queue empty. Necessary actions have been taken under these conditions.

1. MySemaphoreWait: The waiting thread is taken from the ready queue and put on the semaphore's queue (context is then swapped to the next ready thread) if the semaphores present count is zero, else, the semaphore's present count is decremented and the thread stays on the ready queue.
2. MySemaphoreSignal: If there are threads waiting on the semaphore, one is released, else, the present count is increased if it is less than the initial count.
3. MyThreadInit: Calls the MyThreadCreate function. This will be the main thread and this function can be called only by the UNIX process.
4. MyThreadCreate: Instantiates a new thread and its context structure.
5. MyThreadYield: The presently executing thread is put at the back of the ready queue and the context is passed on to the thread next in queue.
6. MyThreadJoin: If the thread to be joined on (joinee) is not a child of the present thread (joiner), -1 is returned straight away. If the joinee is a child of the joiner, but the joinee has terminated, a 0 is returned without any waiting for the joiner. If the joinee is an active child (either in the ready queue or in the blocked queue), the joiner is put in the blocked queue and the joinee is notified. The joinee keeps track of threads that are waiting for it.
7. MyThreadJoinAll: The joiner thread goes to all its child threads that are still active and informs them that it will be waiting on them. Finally the joiner thread is put at the back of the blocked queue and context is passed to the thread at the front of the ready queue.
8. MyThreadExit: The exiting thread informs all threads that are waiting on it that it is exiting. If a thread is waiting only on the exiting thread, that thread is brought from the blocked queue to the ready queue. The thread also informs all its child threads and its parent that it is exiting.

Testing:

The following different scenarios have been cleared with the software:
1. Threads with functions with arguments and threads with functions with no arguments.
2. Valid joins: Joinee is a child of Joiner. Joinee may or may not be active.
3. Invalid joins: Joinee not a child of Joiner.
4. Recursive creation of threads.
5. MyThreadJoin and MyThreadJoinAll on the above recursively created threads.
6. Semaphores: Create semaphores with different initial count values (including zero). Multiple threads (including the main thread) are made to wait for these semaphores. Combinations like parent waits and child signals the semaphore, child waits and parent signals and some other thread signals have been covered.
7. Test with invalid semaphores
8. Individual testing of the que.h module has been done.
Most other general conditions have been found and accounted for.

Evaluation:

All thread related processes have been implemented in an efficient manner. Wherever possible, searching through a queue of threads to find a thread has been avoided to improve the order of complexity. At the same time effort has been made to make the routines failsafe in the event of some process behaving in an unexpected way.
An improvement would be to make the queues for the threads single directional. That would perhaps improve the efficiency further.
Most possible failures in the program have been checked for at multiple points of execution and the processes terminate with an error message on finding illegal values of flags or variables. The code has been sufficiently modularized and commented.