CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 3 (document version 1.2)
Multi-threading in C using Pthreads

Overview

- This homework is due by 11:59:59 PM on Tuesday, April 10, 2018.
- This homework will count as 8% of your final course grade.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with absolutely no warning messages when the `-Wall` (i.e., warn all) compiler option is used. We will also use `-Werror`, which will treat all warnings as critical errors.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v16.04.3 LTS. Note that the `gcc` compiler is version 5.4.0 (`Ubuntu 5.4.0-6ubuntu1~16.04.5`).
- Remember, to compile your code, use `-pthread` to include the Pthread library.

Homework Specifications

In this third assignment, the goal is to work with threads and focus on synchronization. You will use C and the POSIX thread (Pthread) library to implement a single-process multi-threaded system that solves the knight's tour problem from Homework 2.

As with Homework 2, your program investigates whether a valid solution is possible for the knight's tour problem on an $m \times n$ board. In this assignment, you must keep track of a global variable called `max_squares` that maintains the maximum number of squares covered by the knight.

Also, a global shared array called `dead_end_boards` is used to maintain a list of "dead end" board configurations. Child threads add their detected "dead end" boards to this array, which therefore requires proper synchronization.

As with Homework 2, your program simulates valid moves. And for a given board configuration, when multiple moves are detected, each possible move must be assigned to a new child thread, thereby forming a thread tree of possible moves.

A valid move constitutes relocating the knight two squares in direction D and then one square 90° from D, where D is up, down, right, or left. Key to this problem is the restriction that a knight may not land on a square more than once in its tour. Also note that the knight starts in the upper-left corner of the board.

When a dead end is encountered (i.e., no more moves can be made), the leaf node thread compares the number of squares covered to the global maximum, updating the global maximum, if necessary. Once all child threads have terminated, the main thread reports the number of squares covered, which is equal to product mn if a full knight's tour is possible. The main thread also displays either all of the "dead end" boards or all "dead end" boards with at least k squares covered, where k is an optional (third) command-line argument.

Dynamic Memory Allocation

As with the previous two homework assignments, your program must use `calloc()` to dynamically allocate memory for the $m \times n$ board. More specifically, use `calloc()` to allocate an array of m pointers, then for each of these pointers, use `malloc()` or `calloc()` to allocate an array of size n.

Of course, your program must also use `free()` and have no memory leaks. Note that you do not need to use `realloc()` for this assignment.

Given that your solution is multi-threaded, you will need to be careful in how you manage your child threads and the board; i.e., you will need to allocate (and free) memory for each child thread that you create.

Command-Line Arguments and Error Handling

There are two required command-line arguments; both are integers n and m, which together specify that the size of the board is $m \times n$, where m is the number of rows and n is the number of columns in the board.

As noted above, a third optional command-line argument, k, indicates that the main thread should display all "dead end" boards with at least k squares covered.

Validate the inputs m and n to be sure both are integers greater than 2. Further, if present, validate input k to be sure it is a positive integer no greater than $m \times n$. If invalid, display the following error message to `stderr`:

```
ERROR: Invalid argument(s)
USAGE: a.out <m> <n> [<k>]
```

If an incorrect number of command-line arguments is given, display the above error message to `stderr`, then return `EXIT_FAILURE`.

Handling System Call Errors

In general, if a system call fails, use `perror()` to display the appropriate error message on `stderr`, then exit the program and return `EXIT_FAILURE`. If a system or library call does not set the global `errno`, use `fprintf()` instead of `perror()` to write an error message to `stderr`. See the various examples on the course website and corresponding `man` pages.

Note that error messages must be one line only and use the following format:

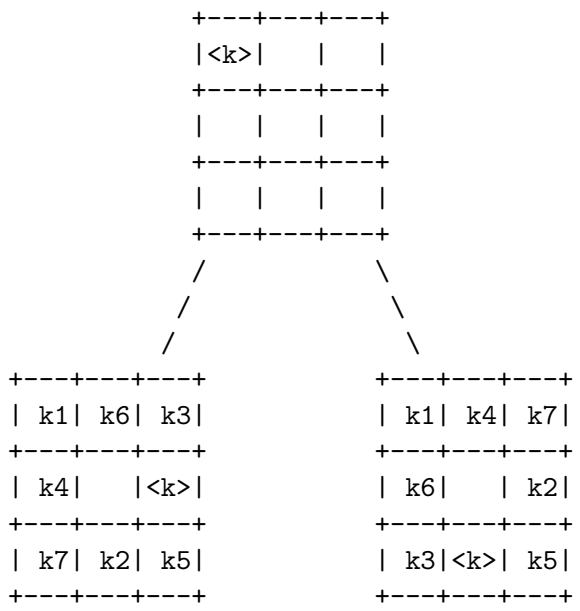
```
ERROR: <error-text-here>
```

Program Execution

To illustrate using an example, you could execute your program and have it work on a 3×3 board as follows:

```
bash$ ./a.out 3 3
```

This will generate the thread tree shown below, with <k> indicating the current position of the knight. For clarity on the order of moves, this diagram also shows the order in which each knight visits each square.



Note that the center square is not visited at all in this example. Also note that each of the two "dead end" boards would be added to the global shared array and displayed by the main thread once all child threads have completed.

Required Output

When you execute your program, you must display a line of output each time you create a new thread and each time you encounter a dead end.

Below is example output to illustrate the required output format. In this example, thread ID (`tid`) `1000` is the top-level main thread, with threads `1001` and `1002` being child threads to thread `1000`.

```
bash$ ./a.out 3 3
THREAD 1000: Solving the knight's tour problem for a 3x3 board
THREAD 1000: 2 moves possible after move #1; creating threads
THREAD 1001: Dead end after move #8
THREAD 1002: Dead end after move #8
THREAD 1000: Best solution found visits 8 squares (out of 9)
THREAD 1000: > kkk
THREAD 1000:   k.k
THREAD 1000:   kkk
THREAD 1000: > kkk
THREAD 1000:   k.k
THREAD 1000:   kkk
```

Match the above output format **exactly as shown above**, though note that the `tid` values will vary. Further, interleaving of the output lines may occur, though the first and last lines must be first and last, respectively.

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server. The specific URL is on the course website.

Note that this assignment will be available on Submitty a few days before the due date. Please do not ask on Piazza when Submitty will be available, as you should perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, as discussed in class (on 1/18), output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `fflush()` after every set of `printf()` statements, as follows:

```
printf( ... );     /* print something out to stdout */
fflush( stdout );  /* make sure that the output is sent to a */
                   /*  redirected output file, if specified  */
```

Second, also discussed in class (on 1/18), use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!" );
    fflush( stdout );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE homework3.c -pthread
```