CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 3 (document version 1.4)
Multi-threading in C using Pthreads

Overview

- This homework is due by 11:59:59 PM on Tuesday, April 2, 2019.
- 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 v18.04.1 LTS. Note that the `gcc` compiler is version 7.3.0 (`Ubuntu 7.3.0-27ubuntu1~18.04`).
- You must use the Pthread library. Remember, to compile your code, use `-pthread` to link the Pthread library.

Homework specifications

In this third assignment, 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, i.e., can a knight move over all squares exactly once on a given board? Sonny plays the knight in our assignment.

In brief, your program must determine whether a valid solution is possible for the knight's tour problem on an $m \times n$ board. To accomplish this, your program simulates valid moves. And for each board configuration you encounter, when multiple moves are detected, each possible move is allocated to a new child thread, thereby forming a tree of possible moves.

You must keep track of a global variable called `max_squares` that maintains the maximum number of squares covered by Sonny; when a dead end is encountered in a thread, that thread checks the `max_squares` variable, updating it if a new maximum is found.

Further, 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 the end of this array, which therefore requires proper synchronization.

(v1.3) Note that a "dead end" does not include a fully covered board, i.e., a full knight's tour.

A valid move constitutes relocating Sonny 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 Sonny may not land on a square more than once in its tour. Also note that Sonny starts in the upper-left corner of the board.

When a dead end is encountered (i.e., no more moves can be made) **(v1.4)** or a fully covered board is achieved, 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 for a given parent thread, the parent thread reports (i.e., returns to its own parent thread) the number of squares covered. When the top-level main thread joins all of its child threads, it displays the final maximum result, which is equal to product $m \times n$ if a full knight's tour is possible.

The top-level main thread also displays either all of the "dead end" boards or all "dead end" boards with at least x squares covered, where x is an optional (third) command-line argument.

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, x, indicates that the main thread should display all "dead end" boards with at least x squares covered.

Validate the inputs m and n to be sure both are integers greater than 2. Further, if present, validate input x 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> [<x>]
```

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>
```

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. **(1.2)** Note that you do not need to use `realloc()` for the individual $m \times n$ boards, but you definitely need to use `realloc()` to expand the global `dead_end_boards` array, as necessary.

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.

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 `<S>` indicating the current position of Sonny. There are two dead end boards (i.e., the leaf nodes). For clarity on the order of moves, this diagram also shows the order in which Sonny visits each square.

```
            +---+---+---+
            |<S>|   |   |
            +---+---+---+
            |   |   |   |
            +---+---+---+
            |   |   |   |
            +---+---+---+
            /           \
           /             \
          /               \
+---+---+---+           +---+---+---+
| 1 | 4 | 7 |           | 1 | 6 | 3 |
+---+---+---+           +---+---+---+
| 6 |   | 2 |           | 4 |   |<S>|
+---+---+---+           +---+---+---+
| 3 |<S>| 5 |           | 7 | 2 | 5 |
+---+---+---+           +---+---+---+
```

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.

And note that child threads are only created if a given board configuration has multiple possible moves for Sonny the knight.

(v1.1) To ensure a deterministic order of thread creation, if Sonny the knight is in row `r` and column `c`, start looking for moves at `(c-2)` and `(r-1)`, then check for possible moves going clockwise from there. **(v1.3)** And note that row `0` and column `0` identify the upper-left corner of the board.

Required Output

When you execute your program, you must display a line of output each time you detect multiple possible moves and each time you encounter a dead end. Note that you only display the dead end boards in the main thread once all child threads have ended (and been joined back in).

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 Sonny's 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: Thread [1001] joined (returned 8)
THREAD 1000: Thread [1002] joined (returned 8)
THREAD 1000: Best solution(s) found visit 8 squares (out of 9)
THREAD 1000: Dead end boards:
THREAD 1000: > SSS
THREAD 1000:   S.S
THREAD 1000:   SSS
THREAD 1000: > SSS
THREAD 1000:   S.S
THREAD 1000:   SSS
```

(v1.3) If a full knight's tour is found, display the following line of output:

```
THREAD 2010: Sonny found a full knight's tour!
```

Match the above output format **exactly as shown above**, though note that the `tid` values will vary. Further, interleaving of the output lines of the child threads may occur.

To simplify the problem and help you test, you are also required to add support for an optional `NO_PARALLEL` flag that could be defined at compile time (i.e., via `-D NO_PARALLEL`). If defined, your program should join each child thread immediately after calling `pthread_create()` to be sure that you do not run child threads in parallel. This will also provide fully deterministic output that can more easily be matched on Submitty.

(v1.3) To compile this code in `NO_PARALLEL` mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D NO_PARALLEL hw3.c -pthread
```

Submission instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server.

Note that this assignment will be available on Submitty a minimum of three 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 1/10), 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 hw3.c -pthread
```

Second, as discussed in class (on 1/14), output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure correctness on Submitty, this is a good technique to use.