CSCI 4210 — Operating Systems
Homework 3 (document version 1.1)
Multi-threading in C using Pthreads

- This homework is due by 11:59PM ET on Friday, April 9, 2021
- This homework is to be done individually, so **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 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
- You must use the Pthread library; therefore, compile your code using the `-pthread` flag with `-gcc` to link in the Pthread library
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v18.04.5 LTS and `gcc` version 7.5.0 (`Ubuntu 7.5.0-3ubuntu1~18.04`)

Hints and reminders

As we continue to pay close attention to the lower-level details of C, keep allocating exactly the number of bytes you need regardless of whether you use static or dynamic memory allocation. Deallocate dynamically allocated memory at the earliest possible point in your code; also close any and all open file descriptors as soon as you are done with them. Specifically for multi-threaded programming, remember that all threads share the same memory space (i.e., one runtime stack, one runtime heap).

Always read and re-read the corresponding `man` pages for library functions, system calls, etc. And continue to make use of the posted code examples, video lectures, and Submitty's Discussion Forum.

Homework specifications

In this third assignment, you will use C and the POSIX thread (Pthread) library to implement a single-process multi-threaded program that attempts to solve the knight's tour problem, i.e., can a knight make valid moves to cover all squares exactly once on a given board? Sonny plays the knight in our assignment.

Goal

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. Note that a new child thread is created **only if multiple moves are possible** at that given point of the simulation.

Valid moves and child threads

A valid move constitutes relocating Sonny the knight two squares in direction D and then one square 90° from D (in either direction), where D is up, down, right, or left.

Key to this problem is the further restriction that Sonny may not land on a square more than once in his tour.

For consistency, row `0` and column `0` identify the upper-left corner of the board. Sonny starts at the square identified by row r and column c (which are given as command-line arguments).

As noted above, a new child thread is created only if multiple moves are possible at that given state of the board. Further, if a dead end is encountered (i.e., no more valid moves can be made) or a fully covered board is achieved, the leaf node thread compares the number of squares covered to a global maximum (`max_squares`), updating this global maximum as necessary.

Once all child threads have terminated for a given parent thread, the parent thread reports the number of squares covered to its own parent thread. In general, threads are blocked on `pthread_join()` waiting for their child threads to terminate.

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 all "dead end" boards with at least x squares covered, where x is another command-line argument.

Global variables and synchronization

The given `hw3-main.c` source file contains a short `main()` function that initializes three global variables (as described below), then calls the `simulate()` function, which you must write in your own `hw3.c` source file. Submitty will compile your `hw3.c` code as follows:

```
bash$ gcc -Wall -Werror hw3-main.c hw3.c -pthread
```

You are required to make use of the three global variables in the given `hw3-main.c` source file. To do so, declare them as external variables in your `hw3.c` code as follows:

```
extern int next_thread_id;
extern int max_squares;
extern char *** dead_end_boards;   /* (v1.1) do NOT free this memory in hw3.c */
```

The three global variables are described below. And since multiple threads will be both accessing and changing these global variables, synchronization is required.

1. Given that Pthread IDs can vary, use the global `next_thread_id` variable to assign each thread its own unique ID. This variable is initialized to 1. Using this global variable, each child thread that is created must be assigned the next available thread ID in sequence (i.e., the first child thread created is assigned ID 1, the second child thread created is assigned ID 2, etc.); this requires synchronization.

2. Initialized to zero, the global `max_squares` variable tracks the maximum number of squares covered by Sonny so far. When a dead end is encountered in a child thread, that thread checks the `max_squares` variable, updating it if a new maximum has been found.

3. The global `dead_end_boards` array is used to maintain a list of "dead end" board configurations. This array is initialized by dynamically allocating an array of size 8 of `char**`. Child threads add their detected "dead end" boards to the end of this array. Note that a "dead end" board does not include a fully covered board, i.e., a full knight's tour.

   You will be responsible for keeping track of the size and maximum size of the `dead_end_boards` array. This will require you to reallocate more memory (via `realloc()`) for this array. **(v1.1)** Do **not** free up this memory; we will do so in `hw3-main.c` (see new version online).

Command-line arguments

There are five required command-line arguments. First, integers m and n together specify the size of the board to be $m \times n$, where m is the number of rows and n is the number of columns. Rows are numbered $0 \ldots (m-1)$ and columns are numbered $0 \ldots (n-1)$. The next pair of command-line arguments, r and c, indicate the starting square on which Sonny starts his attempted tour. The fifth command-line argument, x, indicates that the main thread should display all "dead end" boards with at least x squares covered. **Only store "dead end" boards with x or more squares covered.**

Validate inputs m and n to be sure both are integers greater than 2, then validate inputs r and c accordingly. Also validate input x to be sure it is a positive integer no greater than $m \times n$. If invalid, display the following to `stderr` and exit with `EXIT_FAILURE`:

```
ERROR: Invalid argument(s)
USAGE: a.out <m> <n> <r> <c> <x>
```

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. Use `calloc()` here to allocate an array of m pointers, then for each of these pointers, use `calloc()` to allocate an array of size n. Be sure your program has no memory leaks.

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

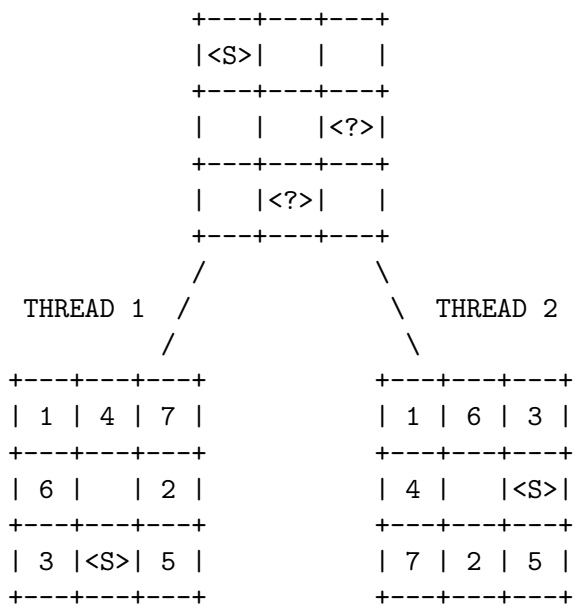
Given that your solution is multi-threaded, you will need to be careful in how you manage your child threads and the board; i.e., you will need to allocate (and free) memory for each child thread that you create.

Program execution

To illustrate using an example, you could execute your program and have it work on a 3×3 board with Sonny starting at row 0 and column 0 as follows:

```
bash$ ./a.out 3 3 0 0 4
```

This will generate the thread tree shown below, with `<S>` indicating the current position of Sonny and `<?>` indicating multiple possible moves from `<S>` that cause child threads to be created. There are two dead end boards (i.e., two leaf nodes). For clarity on the order of moves, this diagram also shows the order in which Sonny visits each square.



It is not possible to visit the center square in this example. Each of the two "dead end" boards would be added to the global shared array and displayed by the main thread at program completion.

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

To ensure a deterministic order of thread creation, if Sonny is in row `a` and column `b`, start looking for moves at `(a-2)` and `(b-1)`, checking for possible moves clockwise from there.

Required output and the "no parallel" mode

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 have been joined back in.

Below is an example that illustrates the required output format.

```
bash$ ./a.out 3 3 0 0 4
MAIN: Solving Sonny's knight's tour problem for a 3x3 board
MAIN: Sonny starts at row 0 and column 0
MAIN: 2 possible moves after move #1; creating 2 child threads...
THREAD 1: Dead end at move #8
THREAD 2: Dead end at move #8
MAIN: Thread 1 joined (returned 8)
MAIN: Thread 2 joined (returned 8)
MAIN: All threads joined; best solution(s) visited 8 squares out of 9
MAIN: Dead end boards covering at least 4 squares are:
MAIN: >>SSS
MAIN:   S.S
MAIN:   SSS<<
MAIN: >>SSS
MAIN:   S.S
MAIN:   SSS<<
```

If a full knight's tour is found, use the output format below and do **not** display "dead end" boards.

```
bash$ ./a.out 3 4 0 0 4
MAIN: Solving Sonny's knight's tour problem for a 3x4 board
MAIN: Sonny starts at row 0 and column 0
...
THREAD 5: Sonny found a full knight's tour!
...
MAIN: All threads joined; full knight's tour of 12 achieved
```

Match the above output format **exactly as shown above**. Given the parallel processing required for this assignment, some interleaving of the output is expected to 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 (see below). If defined, your program should join each child thread **immediately** after calling `pthread_create()`; this will ensure that you do not run child threads in parallel. This will also provide deterministic output that can more easily be matched on Submitty.

To compile this code in `NO_PARALLEL` mode, use the `-D` flag as follows:

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

Error handling

In general, if an error is encountered in any thread, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then abort further thread execution by calling `pthread_exit()` with 0 as a return value.

Remember to only use `perror()` if the given library function or system call sets the global `errno` variable. Otherwise, use `fprintf()`.

In general, error messages must be one line only and use the following format:

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

Submission instructions

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

Note that this assignment will be available on Submitty a minimum of three days before the due date. Please do not ask when Submitty will be available, as you should first 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, make use of the `DEBUG_MODE` technique to make sure that Submitty does not execute any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
  printf( "the value of q is %d\n", q );
  printf( "here12\n" );
  printf( "why is my program crashing here?!\n" );
  printf( "aaaaaaaaaaaaaggggggggghhhh!\n" );
#endif
```

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

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

Second, 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 output is not buffered, this is a good technique to use.