CSCI 4210 — Operating Systems
Homework 3 (document version 1.0)
Multi-Threaded Programming and Synchronization

- This homework is due in Submitty by 11:59PM EST on Thursday, March 31, 2022

- You can use at most three late days on this assignment

- This homework is to be done individually, so **do not share your code with anyone else**

- You **must** use C for this assignment, and all submitted 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** also use the POSIX thread (Pthread) library, appending the `-pthread` flag to `gcc`

- All submitted code **must** successfully compile and run on Submitty, which uses Ubuntu v20.04.3 LTS and `gcc` version 9.3.0 (`Ubuntu 9.3.0-17ubuntu1~20.04`)

Hints and reminders

To succeed in this course, do **not** rely on program output to show whether your code is correct. Consistently allocate exactly the number of bytes you need regardless of whether you use static or dynamic memory allocation. Further, deallocate dynamically allocated memory via `free()` at the earliest possible point in your code. And as covered in initial class videos, make use of `valgrind` to check for errors with dynamic memory allocation and use. Also close any open file descriptors or `FILE` pointers as soon as you are done using them.

Another key to success in this course is to always read (and re-read!) the corresponding `man` pages for library functions, system calls, etc.

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 again plays the knight in our assignment.

Goal

The fundamental goal of this homework is to use `pthread_create()` and `pthread_join()` to achieve a fully synchronized parallel multi-threaded solution to the knight's tour problem.

In brief, your program must determine whether a valid solution is possible for the knight's tour problem on an $m \times n$ board, and if so, how many solutions exist. To accomplish this, your program uses a brute force approach and simulates all valid moves. For each board configuration, 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. Remember that all threads will run within one process.

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. Also note that Sonny can start and finish on different squares; we are not looking for a cycle.

When a dead end is encountered (i.e., no more moves can be made), the leaf node thread knows the number of squares it was able to cover, which might be a full knight's tour. The leaf node thread compares the number of squares covered to a global maximum (`max_squares`), updating this global maximum as necessary. And if a full knight's tour is achieved (i.e., the last move of a knight's tour has been made), the child thread also increments a global count (`total_tours`).

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.

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. Further, if a full knight's tour is possible, the number of tours found is also reported.

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 int total_tours;
```

The three global variables are described below. Feel free to use additional global variables. Since multiple threads will be 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. Also initialized to zero, the global `total_tours` variable tracks the number of full tours that are found. When a full knight's tour is encountered in a child thread, that thread increments the `total_tours` variable.

Command-line arguments

There are four 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.

Validate inputs m and n to be sure both are integers greater than 2, then validate inputs r and c accordingly. If invalid, display the following to `stderr` and return `EXIT_FAILURE`:

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

Dynamic memory allocation

As with Homework 2, 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.

You must also use `free()` and have no memory leaks when your program terminates.

Do **not** use `malloc()` or `realloc()`. Be sure your program has no memory leaks.
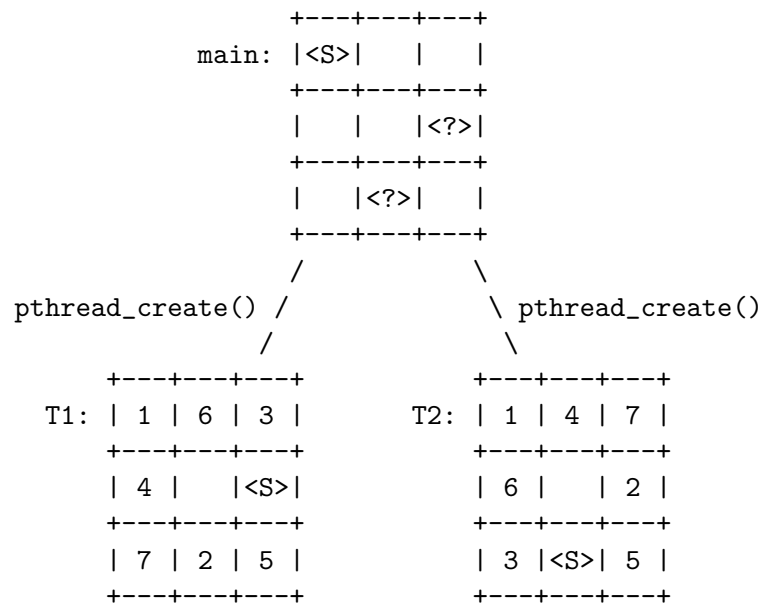
Given that your solution is multi-threaded, you will need to be careful in how you manage your child threads and the boards, i.e., you will need to allocate (and free) memory for each child thread that you create.

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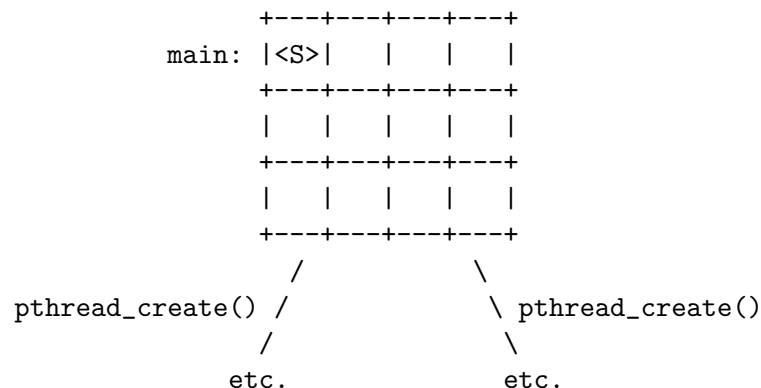
```
bash$ ./a.out 3 3 0 0
```

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. The numbers in this diagram show the order in which Sonny visits each square.

```
                  +---+---+---+
          main:   |<S>|   |   |
                  +---+---+---+
                  |   |   |<?>|
                  +---+---+---+
                  |   |<?>|   |
                  +---+---+---+
                   /         \
   pthread_create() /         \ pthread_create()
                 /             \
      +---+---+---+        +---+---+---+
   T1:| 1 | 6 | 3 |     T2:| 1 | 4 | 7 |
      +---+---+---+        +---+---+---+
      | 4 |   |<S>|        | 6 |   | 2 |
      +---+---+---+        +---+---+---+
      | 7 | 2 | 5 |        | 3 |<S>| 5 |
      +---+---+---+        +---+---+---+
```

Note that the center square is not visited at all in this example. Also note that both of these child threads will simultaneously try to set the global `max_squares` to 8 before terminating.

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

As with Homework 2, try writing out the tree by hand using the 3×4 board below. Remember that child threads are created only when multiple moves are possible from a given board configuration.

```
                  +---+---+---+---+
          main:   |<S>|   |   |   |
                  +---+---+---+---+
                  |   |   |   |   |
                  +---+---+---+---+
                  |   |   |   |   |
                  +---+---+---+---+
                   /         \
   pthread_create() /         \ pthread_create()
                 /             \
              etc.              etc.
```

Required output

When you execute your program, you must display a line of output each time you detect multiple possible moves, each time you reach a dead end or full knight's tour, each time you update the global `max_squares` or `total_tours` variables, and each time a child thread is joined back in to its parent thread.

Below is example output that shows the required output format.

```
bash$ ./a.out 3 3 0 0
MAIN: Solving Sonny's knight's tour problem for a 3x3 board
MAIN: Sonny starts at row 0 and column 0 (move #1)
MAIN: 2 possible moves after move #1; creating 2 child threads...
T1: Dead end at move #8; updated max_squares
T2: Dead end at move #8
MAIN: T1 joined
MAIN: T2 joined
MAIN: Search complete; best solution(s) visited 8 squares out of 9
```

If a full knight's tour is found, use the output format below.

```
bash$ ./a.out 3 4 0 0
MAIN: Solving Sonny's knight's tour problem for a 3x4 board
MAIN: Sonny starts at row 0 and column 0 (move #1)
...
T5: Sonny found a full knight's tour; incremented total_tours
...
MAIN: Search complete; found 2 possible paths to achieving a full knight's tour
```

Match the above output format **exactly as shown above**, though note that the assigned thread IDs will vary. Further, interleaving of the output lines is expected, though the first few lines and the last line must be first and last, respectively.

Running in "no parallel" mode

To simplify the problem and help you test, you are also required to add support for an optional `NO_PARALLEL` flag that may 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, which will therefore provide deterministic output that can more easily be matched on Submitty.

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

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

NOTE: This problem grows extremely quickly, so be careful in your attempts to run your program on boards larger than 4×4.

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()`. Only use `perror()` if the given library function or system call sets the global `errno` variable.

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( "aaaaaaaaaaaaggggggggghhhh!\n" );
#endif
```

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

```
bash$ gcc -Wall -Werror -g -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 good results on Submitty, this is a good technique to use.