CSCI 4210 — Operating Systems
Homework 3 (document version 1.1)
Multi-Threaded Programming and Synchronization

- This homework is due in Submitty by 11:59PM EST on Wednesday, April 5, 2023
- 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** use the POSIX thread (Pthread) library by appending the `-pthread` flag to `gcc`
- All submitted code **must** successfully compile and run on Submitty, which uses Ubuntu v20.04.5 LTS and `gcc` version 9.4.0 (`Ubuntu 9.4.0-1ubuntu1~20.04.1`)

Hints and reminders

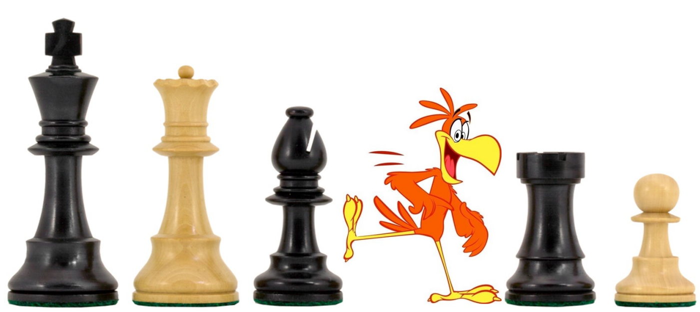
To succeed in this course, do **not** rely on program output to show whether your code is correct. And no guesswork! Instead, consistently allocate exactly the number of bytes you need regardless of whether you use static or dynamic memory allocation.

Further, deallocate dynamically allocated memory via `free()` at the earliest possible point in your code. Consider using `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. To better understand how `man` pages are organized, check out the `man` page for `man` itself!

Homework specifications

In this third assignment, you will use C and the POSIX thread (Pthread) library to implement a single-process multi-threaded solution to the classic 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.



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 open and closed 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. Specifically, a new child thread is created **only if multiple moves are possible** at that given point of the simulation.

Remember that all threads 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.

A dead end is reached if no more moves can be made and there is at least one unvisited square. When a dead end is encountered, the leaf node thread knows the number of squares it was able to cover.

The leaf node thread compares the number of squares covered to the global maximum, `max_squares`, updating this global variable if necessary.

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 the appropriate global counter, either `total_open_tours` or `total_closed_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.

To synchronize the threads, each parent thread calls `pthread_join()` for each of its child threads, then exits.

When the top-level `main` thread joins all of its child threads, it reports the number of squares covered, which is equal to product mn if a knight's tour is possible. If at least one knight's tour is possible, the top-level parent thread reports the number of open and closed tours found.

Global variables and synchronization

The given `hw3-main.c` source file contains a short `main()` function that initializes four 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 four 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 long next_thread_number;
extern int max_squares;
extern int total_open_tours;
extern int total_closed_tours;
```

These global variables are described below. Feel free to use additional global variables in your own code. Since multiple threads will be accessing and changing these global variables, synchronization is crucial.

1. Given that Pthread IDs are implementation-specific, we will use our own thread numbering scheme. Therefore, use the global `next_thread_number` variable to assign each thread its own unique number. This variable is initialized in `hw3-main.c`.

   Using this global variable, each child thread that is created must be assigned the next available thread number in sequence (e.g., the first child thread created is number 1, the second child thread created is number 2, etc.); this requires synchronization.

2. Initialized to zero in `hw3-main.c`, the global `max_squares` variable tracks the maximum number of squares covered by Sonny. When a dead end is encountered in a child thread, that thread checks this variable, updating it if a new maximum has been found.

3. Also initialized to zero, the global `total_open_tours` and `total_closed_tours` variables track the number of open and closed tours found, respectively. When a knight's tour is encountered in a child thread, that thread increments one of these two variables.

Dynamic memory allocation

As with Homework 3, you 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 `calloc()` to allocate an array of size n.

Of course, 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.

Command-line arguments

There are four required command-line arguments.

First, integers m and n together specify the size of the board as $m \times n$, 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: hw3.out <m> <n> <r> <c>
```

No square brackets allowed!

As we continue to practice the use of pointers and pointer arithmetic, once again, **you are not allowed to use square brackets** anywhere in your code!

If a `'['` or `']'` character is detected, including within comments, that line of code will be removed before running `gcc`.

To detect square brackets, consider using the command-line `grep` tool as shown below.

```
bash$ grep '\[' hw3.c
...
bash$ grep '\]' hw3.c
...
```

You can also combine this into one `grep` call as follows:

```
bash$ grep '\(\[\|\]\)' hw3.c
...
```

Program execution

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

```
bash$ ./hw3.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. The numbers in this diagram show the order in which Sonny visits each square.

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

The center square is not visited at all in this example. Further, 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 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 thread:  |   |   |   |   |
                      +---+---+---+---+
                      |<S>|   |   |   |
                      +---+---+---+---+
                      |   |   |   |   |
                      +---+---+---+---+
                         /             \
      pthread_create()  /               \  pthread_create()
                       /                 \
                     etc.               etc.
```

Required output and program execution modes

When you execute your program, display a line of output for each of the following cases: (1) when you detect multiple possible moves; (2) when you reach a dead end; (3) when you find a knight's tour; (4) when you update `max_squares`; and (5) when you join a child thread. (Note that a valid knight's tour is not considered a dead end.)

Below is example output that shows the required output format. **(v1.1)** Output has been updated to match actual program execution.

```
bash$ ./hw3.out 3 3 0 0        /* e.g., in parallel mode */
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...
T2: Dead end at move #8; updated max_squares
T1: 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$ ./hw3.out 3 4 1 0        /* e.g., in NO_PARALLEL mode */
MAIN: Solving Sonny's knight's tour problem for a 3x4 board
MAIN: Sonny starts at row 1 and column 0 (move #1)
...
T3: Sonny found an open knight's tour; incremented total_open_tours
...
MAIN: Search complete; found 4 open tours and 0 closed tours
```

Be sure to indicate whether an open or closed tour is found. For an open tour, use the example above. For a closed tour, use the following:

```
bash$ ./hw3.out 3 10 0 0        /* e.g., in NO_PARALLEL mode */
MAIN: Solving Sonny's knight's tour problem for a 3x10 board
MAIN: Sonny starts at row 0 and column 0 (move #1)
...
T580926: Sonny found a closed knight's tour; incremented total_closed_tours
...
MAIN: Search complete; found 416 open tours and 32 closed tours
```

Match the above output format **exactly as shown above**, though note that the assigned thread numbers may 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 "quiet" mode

To help scale your solution up to larger boards, you are required to support an optional `QUIET` flag that may be defined at compile time (see below). If defined, your program displays only the first two lines and the final line of output in the top-level thread.

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

```
bash$ gcc -Wall -Werror -o hw3.out -D QUIET hw3-main.c hw3.c -pthread
bash$ ./hw3.out 3 4 1 0
MAIN: Solving Sonny's knight's tour problem for a 3x4 board
MAIN: Sonny starts at row 1 and column 0 (move #1)
MAIN: Search complete; found 4 open tours and 0 closed tours
```

In your code, use the `#ifdef` and `#ifndef` directives as follows:

```
#ifndef QUIET
      printf( "T%ld: Dead end at move #%d\n", ... );
#endif
```

(v1.1) Note the correction above to use `%ld` for the assigned thread number, since the global `next_thread_number` variable is a `long` integer.

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

```
bash$ gcc -Wall -Werror -o hw3.out -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( "aaaaaaaaaaaaagggggggghhhh!\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.