CSCI 4210 — Operating Systems
Homework 2 (document version 1.2)
Processes, Pipes, and IPC

- This homework is due in Submitty by 11:59PM on **(v1.1)** Thursday, February 13, 2025
- You can use at most three late days on this assignment
- This homework is to be done individually, so **please do not share your code with others**
- Place your code in `hw2.c` for submission; you may also include your own header files
- 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
- All submitted code **must** successfully compile and run on Submitty, which currently uses Ubuntu v22.04.5 LTS and `gcc` version 11.4.0 (`Ubuntu 11.4.0-1ubuntu1~22.04`)
- You will have **eight** penalty-free submissions on Submitty, after which points will slowly be deducted, e.g., `-1` on submission #9, etc.
- You will have at least **three** days before the due date to submit your code to Submitty; if the auto-grading is not available three days before the due date, the due date will be 11:59PM three days after auto-grading becomes available

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.

Make use of `valgrind` (or `drmemory` or other dynamic memory checkers) to check for errors with dynamic memory allocation and dynamic memory usage. As another helpful hint, close open file descriptors as soon as you are done using them.

Finally, always read (and re-read!) the `man` pages for library functions (section 3), system calls (section 2), etc.

Homework specifications

In this homework, you will use C to implement a multi-process 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 $m \times n$ board? Specifically, you are to use `fork()`, `waitpid()`, and `pipe()` to achieve a fully synchronized parallel solution to the knight's tour problem.

Your implementation will consider both *open* and *closed* solutions. Here, a closed solution is a knight's tour that starts and ends on the same square, essentially forming a cycle; therefore, an open solution is any solution that does not form a cycle.

Your program must determine and count how many open and closed solutions exist. To do so, your program must use a *brute force* approach that simulates **all valid moves**.

Multiple moves and pipe communication

For each board configuration, when multiple moves are detected, each possible move is assigned to a **new child process**, thereby forming a process tree of possible moves. Specifically, a new child process is created **only if multiple moves are possible** at that given point of the simulation.

HINT: review the `fork-pass-by-fork.c` example; remember that when you call `fork()`, the state (i.e., variables) of the parent process is copied to the child.

To communicate between processes, the top-level parent process creates a single pipe that all child processes will use to report when a knight's tour is detected. In other words, each child process will share a unidirectional communication channel back to the top-level parent process.

Further, each child process will communicate with its immediate parent process by returning as its exit status the maximum coverage achieved through to that point of the tree. For each leaf node, this will simply be the number of squares covered. For each intermediate node, this will be the *maximum* of values returned by its child nodes.

Valid moves and dead ends

A valid move consists of relocating 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 the knight may not land on a square more than once in this tour.

A dead end is reached if no more moves can be made and there is at least one unvisited square. For a dead end, the leaf node process returns the number of squares covered.

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

If a dead end is encountered or a knight's tour is achieved, the leaf node process returns the number of squares covered as its exit status.

(Be sure to count the start and end square of the tour only once.)

Before terminating, if a valid knight's tour is detected, i.e., if the last move of a knight's tour has been made, the child process writes to the pipe to notify the top-level parent process.

Each intermediate node of the tree must wait until all child processes have reported a value. At that point, the intermediate node returns (to its parent) the maximum of these values (i.e., the best possible solution below that point) as its exit status.

Once all child processes in the process tree have terminated, the top-level node 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 process reports the number of open and closed tours found.

Dynamic Memory Allocation

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 through all running (and terminating) processes.

Do **not** use `malloc()`, `realloc()`, or `memset()`. **(v1.1)** Similar to square brackets, any line containing any of these substrings will be removed before compiling your code.

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 the knight starts the 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 error message to `stderr` and return `EXIT_FAILURE`:

```
ERROR: Invalid argument(s)
USAGE: ./hw2.out <m> <n> <r> <c>
```

No square brackets allowed!

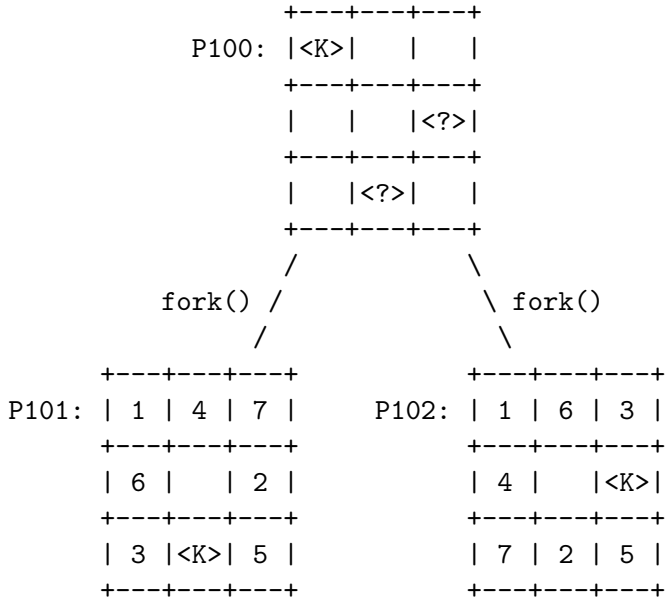
To emphasize the use of pointers and pointer arithmetic, **you are not allowed to use square brackets** anywhere in your code! As with our first lecture exercise, if a `'['` or `']'` character is detected, including within comments, Submitty will completely remove that line of code before running `gcc`.

Program Execution

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

```
bash$ ./hw2.out 3 3 0 0
```

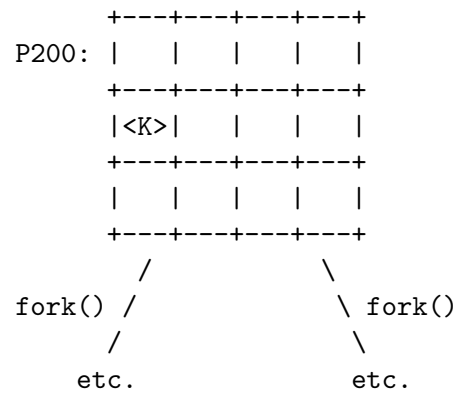
This will generate the process tree shown below starting with process `P100`, with `<K>` indicating the current position of the knight and `<?>` indicating multiple possible moves. The numbers in this diagram show the order in which the knight visits each square.



Note that the center square is not visited at all in this example. Also note that both of these child processes return `8` as their exit status, which represents the number of squares visited.

To ensure a deterministic order of process creation, if the knight 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.

Try writing out the tree by hand using the 3×4 board below. Remember that child processes are created only when multiple moves are possible from a given board configuration.



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; and (3) when you notify the top-level parent process of a tour via the pipe. (Note that a valid knight's tour is not considered a dead end.)

Below is example output that shows the required output format. In the example, process `P100` is the top-level parent process, with processes `P101` and `P102` as child processes of process `P100`. Use `getpid()` to display actual process IDs.

```
bash$ ./hw2.out 3 3 0 0
PID 100: Solving the knight's tour problem for a 3x3 board
PID 100: Starting at row 0 and column 0 (move #1)
PID 100: 2 possible moves after move #1; creating 2 child processes...
PID 101: Dead end at move #8
PID 102: Dead end at move #8
PID 100: 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$ ./hw2.out 3 4 1 0
PID 200: Solving the knight's tour problem for a 3x4 board
PID 200: Starting at row 1 and column 0 (move #1)
...
PID 205: Found an open knight's tour; notifying top-level parent
...
PID 200: 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 shown above. For a closed tour, use the following:

```
bash$ ./hw2.out 3 10 0 0
PID 200: Solving the knight's tour problem for a 3x10 board
PID 200: Starting at row 0 and column 0 (move #1)
...
PID 222: Found a closed knight's tour; notifying top-level parent
...
PID 200: Search complete; found 416 open tours and 32 closed tours
```

Match the above output format **exactly as shown above**, though note that the `PID` values 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 "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 parent process.

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

```
bash$ gcc -Wall -Werror -o hw2.out -D QUIET hw2.c
bash$ ./hw2.out 3 4 1 0
PID 200: Solving the knight's tour problem for a 3x4 board
PID 200: Starting at row 1 and column 0 (move #1)
PID 200: Search complete; found 4 open tours and 0 closed tours
```

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

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

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 uses a blocking `waitpid()` call **immediately** after calling `fork()`; this will ensure that you do not run child processes 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 hw2.out -D NO_PARALLEL hw2.c
```

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

Error handling

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution, **(v1.2)** i.e., refer to the **Command-line arguments** section on page 3 for specific requirements.

In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution. Only use `perror()` if the given library 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 perform final testing of your code), we will use Submitty.

To help make sure that your program executes properly, use the techniques below.

First, make use of the `DEBUG_MODE` preprocessor technique that helps avoid accidentally displaying extraneous output in Submitty. 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 square brackets!\n" );
#endif
```

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

```
bash$ gcc -Wall -Werror -g -D DEBUG_MODE hw2.c
```

Second, output to standard output (`stdout`) is buffered. To disable buffered output, use `setvbuf()` as follows:

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

You would not generally do this in practice since this can slow down your program, but to ensure you see as much output as possible in Submitty, this is a good technique to use.