CSCI 4210 — Operating Systems
Homework 2 (document version 1.0)
Process Management and Synchronization

- This homework is due in Submitty by 11:59PM EST on Thursday, February 17, 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
- 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 second 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 board? Sonny plays the knight in our assignment.

Goal

The fundamental goal of this homework is to use `fork()` and `waitpid()` to achieve a fully synchronized parallel 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 assigned to a **new child process**, thereby forming a process tree of possible moves.

Note that a new child process is created **only if multiple moves are possible** at that given point of the simulation. And remember that when you call `fork()`, the state of the parent process is copied to the child process.

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 reporting 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, i.e., the maximum coverage achieved.

Valid moves

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 starts and finishes 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 process reports the number of squares covered, including in that count the start and end squares of the 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.

If a dead end is encountered or a fully covered board is achieved, the leaf node process reports the number of squares covered as its exit status. Before terminating, if a fully covered board is detected (i.e., 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 full knight's tour is possible. And if a full knight's tour is possible, the number of tours found is also reported.

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()` or `realloc()`. Be sure your program has no memory leaks.

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 error message to `stderr` and return `EXIT_FAILURE`:

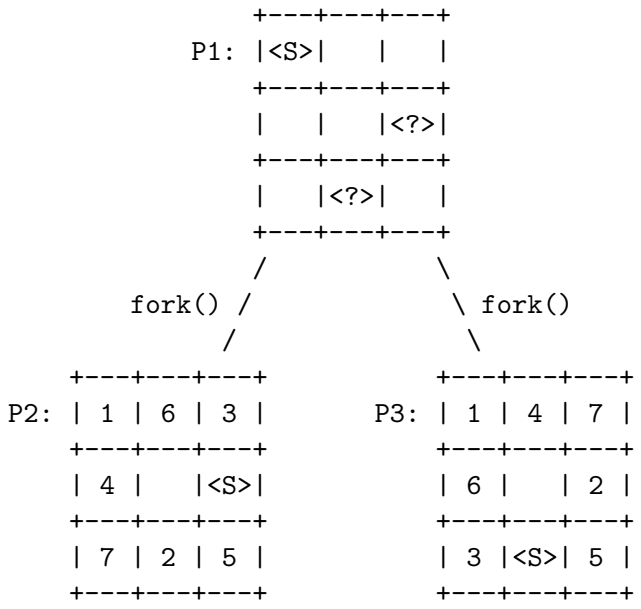
```
ERROR: Invalid argument(s)
USAGE: hw2.out <m> <n> <r> <c>
```

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
```

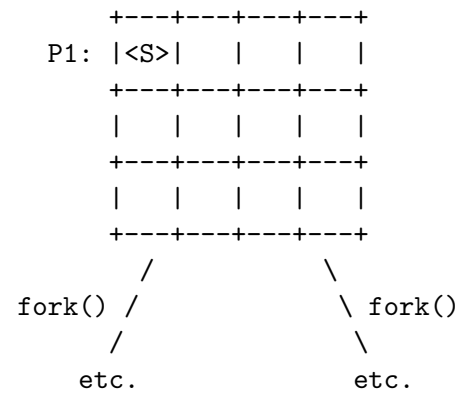
This will generate the process tree shown below starting with process `P1`, with `<S>` indicating the current position of Sonny and `<?>` indicating multiple possible moves. The numbers in this diagram show the order in which Sonny visits each square.



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 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.

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 the "no parallel" mode

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, and each time you notify the top-level parent process of a full tour via the pipe. Note that a full tour is not considered a dead end.

Below is example output that shows the required output format. In the example, process (`PID`) `1` is the top-level parent process, with processes `2` and `3` being child processes of process `1`. Use `getpid()` to display actual process IDs.

```
bash$ ./a.out 3 3 0 0
PID 1: Solving Sonny's knight's tour problem for a 3x3 board
PID 1: Sonny starts at row 0 and column 0 (move #1)
PID 1: 2 possible moves after move #1; creating 2 child processes...
PID 2: Dead end at move #8
PID 3: Dead end at move #8
PID 1: 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
PID 1: Solving Sonny's knight's tour problem for a 3x4 board
PID 1: Sonny starts at row 0 and column 0 (move #1)
...
PID 5: Sonny found a full knight's tour; notifying top-level parent
...
PID 1: 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 `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 "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 -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. 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>
```

If a child process encounters an error, return `EXIT_FAILURE`. In general, a parent process should terminate and return `EXIT_FAILURE` if one of its child processes returns `EXIT_FAILURE`.

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 -g -D DEBUG_MODE hw2.c
```

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.