CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 2 (document version 1.5)
Process Creation and Pipes in C

Overview

- This homework is due by 11:59:59 PM on Monday, March 5, 2018.
- This homework will count as 8% of your final course grade.
- This homework is to be completed **individually**. 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 absolutely 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.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v16.04.3 LTS. Note that the `gcc` compiler is version 5.4.0 (`Ubuntu 5.4.0-6ubuntu1~16.04.5`).

Homework Specifications

In this second homework, you will use C to implement a multi-process solution to the classic knight's tour problem. The goal is to practice using `fork()` and `pipe()` in support of interprocess communication (IPC).

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 a given board configuration, when multiple moves are detected, each possible move is allocated to a new child process, thereby forming a process tree of possible moves.

To communicate between processes, pipes connect all parent/child process pairs, enabling child processes to report to their respective parent processes.

A valid move constitutes relocating the knight two squares in direction D and then one square 90° from D, where D is up, down, right, or left. Key to this problem is the restriction that a knight may not land on a square more than once in its tour. Also note that the knight starts in the upper-left corner of the board.

When a dead end is encountered (i.e., no more moves can be made), the leaf node process reports the number of squares covered, which includes the start and end squares of the tour.

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

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.

Dynamic Memory Allocation

Similar to the first homework assignment, your program 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 `malloc()` or `calloc()` to allocate an array of size n.

Of course, your program must also use `free()` and have no memory leaks through all running (and terminating) processes. Note that you do not need to use `realloc()` for this assignment.

Command-Line Arguments and Error Handling

There are two required command-line arguments; both are integers n and m, which together specify that the size of the board is $m \times n$, where m is the number of rows and n is the number of columns in the board.

Validate the inputs m and n to be sure both are integers greater than 2. If invalid, display the following error message to `stderr`:

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

If an incorrect number of command-line arguments is given, display the above error message to `stderr`, then return `EXIT_FAILURE`.

If any other error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()` as appropriate, then abort further program execution by returning `EXIT_FAILURE`.

Error messages must be one line only and use the following format:

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

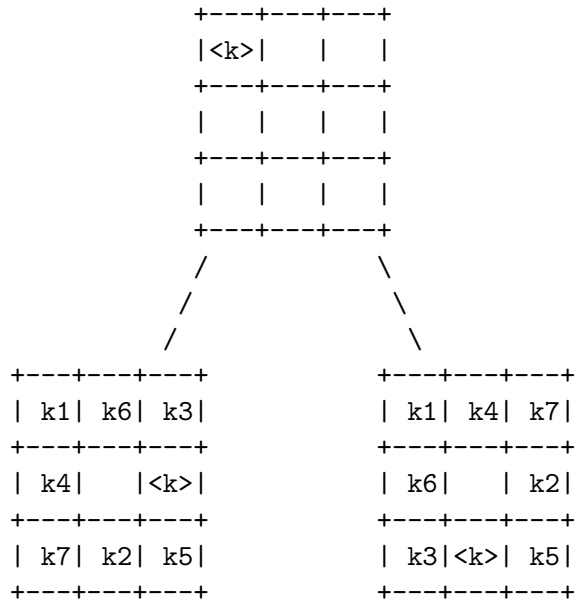
If a child process encounters an error, return `EXIT_FAILURE`. In general, a parent process should terminate and return `EXIT_FAILURE` if one of its child processes returns `EXIT_FAILURE`.

Program Execution

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

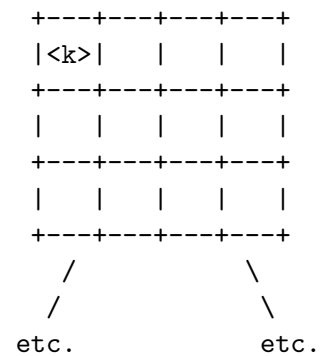
```
bash$ ./a.out 3 3
```

This will generate the process tree shown below, with `<k>` indicating the current position of the knight. For clarity on the order of moves, this diagram also shows the order in which each knight visits each square.



Note that the center square is not visited at all in this example. Also note that each of the two child processes report `8` as the number of squares visited.

Now try writing out the tree by hand using a 3×4 board. Remember that child processes are created when multiple moves are possible from a given board configuration. This is started for you below.



Required Output

When you execute your program, you must display a line of output each time you create a new process and each time you send data back to the parent process via a pipe.

Given the large amount of output that may result beyond rather trivial cases, only display the board if `DISPLAY_BOARD` is defined (i.e., much like how `DEBUG_MODE` works). If you do display the board, only display it after each move in which your program is about to call `fork()` or reaches a dead end.

Below is example output to illustrate the required output format. In this example, process ID (`pid`) `1000` is the top-level parent process, with processes `1001` and `1002` being child processes to process `1000`.

```
bash$ ./a.out 3 3
PID 1000: Solving the knight's tour problem for a 3x3 board
PID 1000: 2 moves possible after move #1
PID 1001: Dead end after move #8
PID 1001: Sent 8 on pipe to parent
PID 1002: Dead end after move #8
PID 1002: Sent 8 on pipe to parent
PID 1000: Received 8 from child
PID 1000: Received 8 from child
PID 1000: Best solution found visits 8 squares (out of 9)
```

Match the above output format **exactly as shown above**, though note that the `pid` values will vary. Further, interleaving of the output lines may occur, though the first and last lines must be first and last, respectively.

For an intermediate node, use the following output format when all of its child processes have terminated (noting that this example is contrived):

```
...
PID 1001: Received 5 from child
...
PID 1001: Received 7 from child
...
PID 1001: Received 3 from child
PID 1001: All child processes terminated; sent 7 on pipe to parent
...
```

When run in `DISPLAY_BOARD` mode (i.e., when compiled with `DISPLAY_BOARD` defined), as noted above, the output should include the board. Repeating the example above in this mode, the output would be as follows.

```
bash$ ./a.out 3 3
PID 1000: Solving the knight's tour problem for a 3x3 board
PID 1000: 2 moves possible after move #1
PID 1000:   k..
PID 1000:   ...
PID 1000:   ...
PID 1001: Dead end after move #8
PID 1001:   kkk
PID 1001:   k.k
PID 1001:   kkk
PID 1001: Sent 8 on pipe to parent
PID 1002: Dead end after move #8
PID 1002:   kkk
PID 1002:   k.k
PID 1002:   kkk
PID 1002: Sent 8 on pipe to parent
PID 1000: Received 8 from child
PID 1000: Received 8 from child
PID 1000: Best solution found visits 8 squares (out of 9)
```

(v1.3) Note that the above output may also interleave, so the boards displayed may not be clear. That is okay for this assignment.

NOTE: This problem grows extremely quickly, so do not attempt to run your program on boards larger than 4×4 (or else you might run out of resources and break something).

(v1.3) Given the above, you are also required to add support for an optional `NO_PARALLEL` flag that could be defined at compile time (i.e., via `-D NO_PARALLEL`). If defined, your program should call `wait()` or `waitpid()` directly after each `fork()` call to be sure that you do not run processes in parallel. See the `octuplets-copy-variables.c` example on the course website for an example.

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server. The specific URL is on the course website.

Note that this assignment will be available on Submitty a few days before the due date. Please do not ask on Piazza when Submitty will be available, as you should 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, as discussed in class (on 1/18), output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `fflush()` after every set of `printf()` statements, as follows:

```
printf( ... );     /* print something out to stdout */
fflush( stdout );  /* make sure that the output is sent to a */
                   /*  redirected output file, if specified  */
```

Second, also discussed in class (on 1/18), use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

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

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

```
bash$ gcc -Wall -Werror -D DEBUG_MODE homework2.c
```