CSCI 4210 — Operating Systems
Homework 1 (document version 1.1)
Processes, Pipes, and an Abridged (m, n)-Queens Problem

- This homework is due in Submitty by 11:59PM EDT on Thursday, June 20, 2024
- 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**
- 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 v22.04.4 LTS and `gcc` version 11.4.0 (`Ubuntu 11.4.0-1ubuntu1~22.04`)
- You will have **five** penalty-free submissions on Submitty, after which points will slowly be deducted, e.g., `-1` on submission #6, 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 EDT 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 other dynamic memory checkers) to check for errors with dynamic memory allocation and dynamic memory usage. As another helpful hint, close open file descriptors, **including pipe descriptors**, as soon as you are done using them.

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

No square brackets allowed!

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

Homework specifications

In this second homework, you will use C to implement a multi-process solution to a variation of the n-Queens problem. In the classic n-Queens problem, the goal is to place n Queens on an $n \times n$ chessboard such that no Queens attack any other Queens.

For our homework, we generalize the problem to an $m \times n$ chessboard and will attempt to answer the following questions:

- What is the maximum number of non-attacking Queens that we can place on an $m \times n$ board? Note that if a solution exists, this maximum will be the minimum of m and n.
- If a solution exists, how many distinct solutions exist?
- **(v1.1)** Using the abridged algorithm described below, how many dead ends do we encounter?

(v1.1) Call our problem the Abridged (m, n)-Queens problem. The fundamental goal of this homework is to use `fork()`, `waitpid()`, and `pipe()` to achieve a fully synchronized parallel solution.

To accomplish this, your program uses a *brute force* approach to explore **(v1.1)** a subset of all possible valid boards. Here, each subsequent placement of a Queen occurs via `fork()` to generate a (potentially very large) process tree.

For consistency, row `0` and column `0` identify the upper-left corner of the $m \times n$ board.

If m and n are equal or $m < n$, the algorithm that you must implement attempts to place a Queen in each row, starting at row `0`. For each square within that row in which a Queen can be placed, call `fork()` to assign a child process the task of exploring that "move"; here, a valid move is the placement of a Queen such that no Queen attacks any other Queen currently on the board.

Continue to row `1`, row `2`, etc., until an *end-state* is reached. An end-state is either (1) a valid solution or (2) a dead-end in which a Queen cannot be placed in the given row. In both of these cases, the child process is a leaf node in the generated process tree.

Specifically, a new child process is created **only if a move is possible** with the given board configuration. As a hint to implementing this, when you call `fork()`, the state of the parent process is copied to the child, i.e., all variables within scope are copied.

If $n < m$, then $m - n$ rows are empty; therefore, to simplify the problem and use the above algorithm, just swap m and n before your first line of output.

Inter-process communication (IPC)

To communicate between processes, the top-level parent process creates **exactly one pipe** that all child processes will use to report their results. Each child process will share this unidirectional communication channel back to the top-level parent process.

Do not create more than one pipe!

The pipe is written to for each end-state encountered, i.e., when a final Queen is placed and therefore either a solution is found or a dead-end is reached. When this occurs (in a leaf node of the process tree), the child process writes to the pipe to notify the top-level parent process how many Queens were successfully placed.

Use whatever protocol you would like to accomplish this, noting that the values written to the pipe will range from `1` to `m`. And you may assume that `m` will be no larger than `256`.

Each intermediate node of the tree must wait until all of its child processes have terminated. At that point, the intermediate node terminates. **(v1.1)** The exit status of each process should be `EXIT_SUCCESS` or, if an error occurs, use `abort()` to stop the process. Once all child processes in the process tree have terminated, the top-level process outputs a summary of the results.

If an error occurs (e.g., `fork()` fails), use `abort()` to stop the running process. In the parent process, if a child process terminates abnormally, have the parent process also call `abort()`. The goal is to have an error cascade back to the top-level process.

Dynamic Memory Allocation

You must use `calloc()` to dynamically allocate memory for the $m \times n$ board. Use the two-layer structure approach from Homework 1, i.e., use `calloc()` to allocate an array of m pointers, then for each of these pointers, use `calloc()` to allocate an array of n characters.

Of course, you must also use `free()` and have no memory leaks or invalid reads/writes through all running and terminating processes.

Do **not** use `malloc()`, `realloc()`, or `memset()`.

Command-line arguments

There are two required command-line arguments. 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 $0 \ldots (n-1)$.

Validate m and n to be sure that both are integers greater than zero. If invalid, display the following error message to `stderr` and return `EXIT_FAILURE`; **(v1.1)** do not use `abort()` in this case.

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

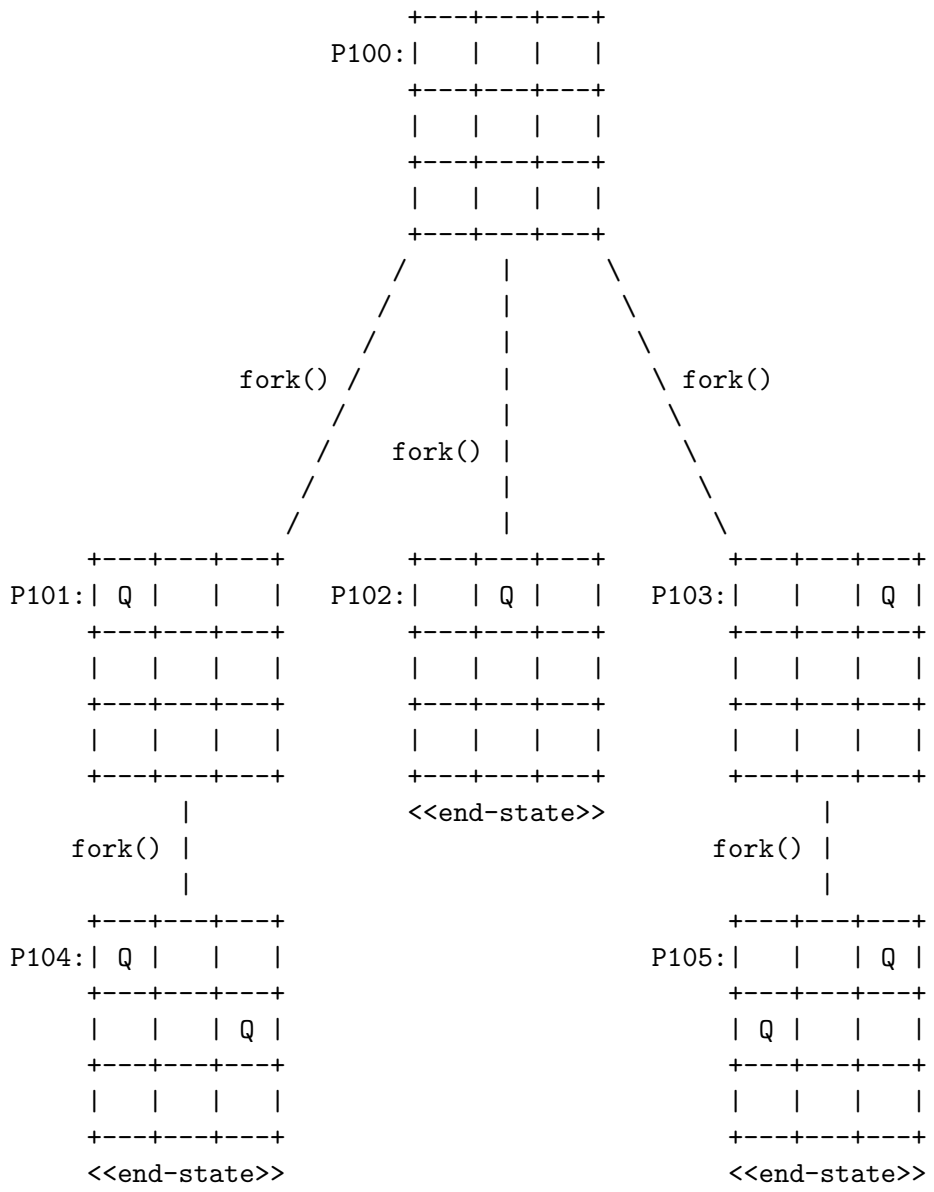
As a reminder, if $n < m$, swap m and n before your first line of output.

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
```

This will generate the process tree shown below starting with process `P100`.



To ensure a deterministic order of process creation at each row, iterate through the given row from column 0 to column $(n-1)$. In the above example, `P100` creates processes `P101`, `P102`, and `P103` in that order.

Leaf nodes `P104` and `P105` both write `2` to the pipe, whereas `P102` writes `1`.

Required output

Display a line of output when you determine the number of moves for a given row and when you reach an end-state. **(v1.1)** Note that sample output below and on the next page has been updated by adding "`Abridged`" to the output.

Corresponding to the example on the previous page, below is sample parallelized output that shows the required output format. Use `getpid()` to display actual process IDs.

```
bash$ ./hw2.out 3 3
P100: solving the Abridged (m,n)-Queens problem for 3x3 board
P100: 3 possible moves at row #0; creating 3 child processes...
P101: 1 possible move at row #1; creating 1 child process...
P102: dead end at row #1; notifying top-level parent
P103: 1 possible move at row #1; creating 1 child process...
P104: dead end at row #2; notifying top-level parent
P105: dead end at row #2; notifying top-level parent
P100: search complete
P100: number of 1-Queen end-states: 1
P100: number of 2-Queen end-states: 2
P100: number of 3-Queen end-states: 0
```

If a solution is found, use the output format below.

```
bash$ ./hw2.out 4 4
P200: solving the Abridged (m,n)-Queens problem for 4x4 board
P200: 4 possible moves at row #0; creating 4 child processes...
P201: 2 possible moves at row #1; creating 2 child processes...
...
P208: found a solution; notifying top-level parent
...
P200: search complete
P200: number of 1-Queen end-states: 0
P200: number of 2-Queen end-states: 2
P200: number of 3-Queen end-states: 2
P200: number of 4-Queen end-states: 2
```

Match the above output format **exactly as shown**, though note that the process IDs will vary. Further, interleaving of the output lines is expected, though the first two lines and the last `m`+1 lines must always 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 `m`+1 lines 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 8 8
P200: solving the Abridged (m,n)-Queens problem for 8x8 board
P200: 8 possible moves at row #0; creating 8 child processes...
P200: search complete
P200: number of 1-Queen end-states: 0
P200: number of 2-Queen end-states: 0
P200: number of 3-Queen end-states: 0
P200: number of 4-Queen end-states: 18
P200: number of 5-Queen end-states: 150
P200: number of 6-Queen end-states: 256
P200: number of 7-Queen end-states: 220
P200: number of 8-Queen end-states: 92
```

In your code, use the `#ifdef` and `#ifndef` (i.e., if not defined) preprocessor directives as follows:

```
#ifndef QUIET
      printf( "P%d: dead end at row #%d; notifying top-level parent\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.

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 -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.