Systems Programming Coursework 2020/2021

Konrad Dabrowski

Deadline: 2p.m. Thursday 21st January 2021

General Instructions

Please remember that you should not share your work or make it available where others can find it, as this can facilitate plagiarism and you could be penalised. This requirement applies until the assessment process is completed, which does not happen until the exam board meets in June 2021.

Please also note that submitted code will be checked for plagiarism.

In case anything is not clear, please email konrad.dabrowski@durham.ac.uk. A Coursework FAQ is available at Duo → Programming Paradigms (20/21) → Systems Programming → Coursework FAQ.

In your C code, you may only use functions present in the C11 standard (a copy of this is available on Duo), those you have written yourself or that have been provided as part of the coursework. You may not use any external libraries and may only use the header files `connect4.h` (provided), `add.h` (which you will create), `subtract.h` (which you will create) and those in the C standard (these are as follows: `assert.h`, `complex.h`, `ctype.h`, `errno.h`, `fenv.h`, `float.h`, `inttypes.h`, `iso646.h`, `limits.h`, `locale.h`, `math.h`, `setjmp.h`, `signal.h`, `stdalign.h`, `stdarg.h`, `stdatomic.h`, `stdbool.h`, `stddef.h`, `stdint.h`, `stdio.h`, `stdlib.h`, `stdnoreturn.h`, `string.h`, `tgmath.h`, `threads.h`, `time.h`, `uchar.h`, `wchar.h` and `wctype.h`).

All your code must compile with no warnings when compiled with the `gcc` options `-Wall -Wextra -pedantic -std=c11`. Your code must run on Linux. **In particular, you should compile & test it on** `mira`, as that is where it will be marked (see Practical 1 for instructions on how to access mira.) Unless your program is exiting with an error, all `malloc()`ed memory must be `free()`d (Hint: use `valgrind` to check). **Do not implement any features or output any text that has not been asked for.**

Submission Instructions

Submit a single zip file on DUO called `code.zip`. This must contain your `Makefile`, `add.c`, `add.h`, `subtract.c`, `subtract.h`, `maths_prog.c`, `connect4.c`, `report.pdf` and `sort.c`. Do not add any other files or organise files into subfolders.

A: Makefiles and dynamic-linked libraries (15%)

Consider the code in `maths_prog.c`. Refactor this code by moving the `add()` function definition out of `maths_prog.c` and into its own file `add.c` and moving the `subtract()` function definition out of the `subtract()` function into its own file `subtract.c`. Construct suitable header files `add.h` and `subtract.h` and add suitable `#include`s where appropriate. You will combine the code in your `add.c` and `subtract.c` files into a dynamically-linked library `libmaths.so` and build a `maths_prog` program using the remaining code in `maths_prog.c`, and this program will be dynamically linked to this library. To do this, you will create a `Makefile` to compile your library and your command-line program. In your `Makefile`, use the `-Wall -Wextra -pedantic -std=c11` options each time you call `gcc`. Have rules for each of: creating an object file from each `.c` file, creating the dynamically-linked library `libmaths.so` and creating the command-line program `maths_prog`. Using the command `make` should compile your program to a file called `maths_prog` and your library `libmaths.so`, both in the current folder. Running `make clean` should delete all files produced during compilation.

B: Connect 4 Twist & Turn (30% Correctness + 15% Robustness and Memory-efficiency)

Connect 4 Twist & Turn is a game played by two players (`'x'` and `'o'`) on a rectangular grid that is wrapped round a cylinder (so the "leftmost" column is one to the right of the "rightmost" column). Rows are numbered $1, \ldots, n$ bottom-to-top and columns are numbered $1, \ldots, m$ left to right. Each cell of the grid can either contain a token of a player (either `'x'` or `'o'`) or it can be empty (denoted `'.'`). The cylinder stands upright, so if a token is in a column, but the position underneath is empty, then the token will automatically drop down into that space by gravity. For example the following position:

```
.........
.........
..x......
..x......
..o......
.....xo..
```

would immediately turn into the following position:

```
.........
.........
.........
..x......
..x......
..o..xo..
```

A *move* consists of two parts:

1. The player places one of their tokens (`'x'` or `'o'`) into the top position of a non-full column of their choice (so this token immediately drops down to the lowest unoccupied row in the column).
2. The player can, *if they choose*, rotate one of the rows one place to the right or left. (They may also choose to *not* rotate any rows.) Tokens then drop down by gravity, so, again, no token will have an empty position underneath it.

For example (rows and columns numbered for clarity), consider the following position:

```
6........
5.x.....x
4.o.....o
3.x.....x
2.o.....o
1.xo.xo.x
 12345678
```

If `'o'` plays in Column 3, and rotates Row 4 one place to the right, the resulting position is as follows:

```
6........
5........
4.x.....x
3.xo....x
2.oo....o
1oxo.xo.x
 12345678
```

Players take it in turns to make a move, starting with player `'x'`. The aim of the game is for a player to get four of their own tokens together consecutively (i.e. one-after-the-other-with-no-gaps) in a straight line, either vertically, horizontally or diagonally, without their opponent doing the same. Note that the grid is on a cylinder, so the grid wraps round, and a line going off the left hand side can continue on the right hand side. For example, the following would be a winning position for `'x'`.

```
........
........
.x......
xo......
ox....ox
ox....xo
```

If a player gets a line of four before the other player, that player wins the game. If the players both get a line of four simultaneously, the game is a draw. If the board becomes full without either player achieving a line of four, the game is a draw. If a player wins, or the game becomes a draw, the game ends immediately and there are no further valid moves.

Your task is to implement the game logic. You will initialise the game from a file that contains the grid for a game that is in play. The file format for *valid* input files: each line represents one row of the grid and each character represents one position on that row: `'.'` represents an empty position and `'x'` or `'o'` represents a position taken up by the respective player's token. In a valid file, gravity will have already caused all tokens to fall to the lowest possible row, and you will be able to tell whose turn it is by how many of each token is present. For example in the following input, it is `'x'`'s turn to play next:

```
........
........
........
........
....o...
...xxo...
```

In any valid input file, there will be at most 512 columns; the number of rows is not limited. The format for output files is the same, except that some `'x'` or `'o'` characters may be capitalised (as explained later).

You have been provided with a `connect4.h`, which you may not edit. Its contents are as follows:

```
typedef struct board_structure *board;

struct move{
int column; // column to play in
int row; // row to rotate:
/* row=0 indicates no rotating
   row>0 indicates rotating the corresponding row one place to the right
   row<0 indicates rotating the corresponding -row one place to the left*/
};

board setup_board();
void cleanup_board(board u);

void read_in_file(FILE *infile, board u);
void write_out_file(FILE *outfile, board u);

char next_player(board u);
char current_winner(board u);

struct move read_in_move(board u);
int is_valid_move(struct move m, board u);
char is_winning_move(struct move m, board u);
void play_move(struct move m, board u);
```

You have also been provided with a `connect4.c` file, in which you must define the things declared in `connect4.h`. For this part of the coursework, submit only your `connect4.c` file.

1. `struct board_structure` is an appropriate structure (which you must devise) to hold the grid in memory. Note that `board` has been `typdef`ed to be a pointer to such a structure.

2. `setup_board()` dynamically allocates a `struct board_structure` and returns a pointer to it. It also does any initial setup, if necessary. It does not assume any particular size of board.

3. `cleanup_board(board u)` frees the memory that `u` points at, along with any memory that was allocated inside `u`.

4. `read_in_file(FILE *infile, board u)` reads in the file from the file pointer `infile` and stores the grid in the `board u`, which has been previously set up by `setup_board()`. You **must** use dynamic memory allocation to set aside memory for storing the grid. (Hint: how will you work out how many cells are in each row? You may want to `fscanf()` with `%c`. Recall that `fscanf()` returns `EOF` when the file ends.)

5. `write_out_file(FILE *outfile, board u)` writes the content of the grid pointed to by `u` into the file from the file pointer `outfile`. If a player has four tokens in a line, this function should print those tokens in capitals (note that this could apply to both players). If a player has more than one line of four tokens, only one of them may be printed in capitals (choose one arbitrarily).

6. `char next_player(board u)` returns `'x'` or `'o'` if it is the corresponding player's turn to play. Note that you can work out whose turn it is from what is on the board.

7. `char current_winner(board u)` returns `'x'` or `'o'` if the corresponding player has won the game on the board `u` and returns `'d'` if the game is a draw. Otherwise it returns `'.'`.

8. `struct move read_in_move(board u)` reads two integers representing a move from `stdin`. It does not need to check whether it is possible to make this move (see also `is_valid_move()`). This function must contain the following two lines, which can be found in the template `connect4.c` file:

   ```
   printf("Player %c enter column to place your token: ",next_player(u));
   ```

   The function then reads in an integer from the user. Note that columns are numbered $1, 2, \ldots$ from left to right.

   ```
   printf("Player %c enter row to rotate: ",next_player(u));
   ```

   The function then reads in an integer `r` from the user. Note that rows are numbered $1, 2, \ldots$ from bottom to top. (If `r==0`, this indicates that no row is to be rotated. If `r>0`, this indicates that row `r` is to be rotated one place to the right. If `r<0`, this indicates that row `-r` is to be rotated one place to the left.) The function must not modify the board `u`.

9. `int is_valid_move(struct move m, board u)` returns `1` if `m` is a valid move to make on the `board u`and `0` otherwise.

10. `char is_winning_move(struct move m, board u)` returns `'x'` or `'o'` if playing this move would cause the corresponding player to win. It returns `'d'` if playing the move would result in a draw. Otherwise it returns `'.'`. It does not actually play the move `m`.

11. `void play_move(struct move m, board u)` plays the move `m` and updates the board `u` correspondingly.

You code should support having more than one `board` in memory at a time. You may add additional functions to `connect4.c` if you wish. You may not use any global variables.

Consider appropriate error checking and make sure that your code can be used robustly. In case of any errors, print an appropriate message to `stderr` and exit the current program with a non-zero error code (e.g. use `exit(1)`). Try to make your code as memory-efficient as possible.

An example `main.c` file is included, with which you should be able to playtest the game. It starts with the board specified in `initial_board.txt`, which contains a board of five rows and six columns with no tokens present. You can compile it by doing `gcc -Wextra -Wall -pedantic -std=c11 -o main main.c connect4.c` and then running `./main`.

You can check some of the basic functionality of your program by running the provided `test_script.sh` file. It uses the files `test1.c`, `output1.txt` and `test_input1.txt`. You should try putting all of these files in the same directory as `connect4.h` and your `connect4.c` and running `sh test_script.sh`. **Note: if your code does not pass this test script, you will get zero marks for this part of the coursework.**

C: Reflective Report (25%)

Write a 1-2 page report describing and justifying the approaches you used for the above code and on how the code in the `main.c` and `connect4.h` could be improved. In particular, you should cover:

- The data structure you have chosen for `struct board_structure` to store the grid.
- What methods you have used to make your code robust.
- What methods you have used to make your code memory-efficient.
- How the robustness of `main.c` and `connect4.h` could be improved.

Submit this report as a pdf file called `report.pdf`. This can be typed or handwritten, but in the latter case please make sure your handwriting is clear and easily readable.

D: command-line sort program (15%)

In this part of the coursework, you will implement a simplified version of the UNIX `sort` command. Your code **must** use the C function `qsort()` to do the actual sorting and you **must** use a `switch` statement to parse the command-line options (in particular, you may not use `getopt` or `qsort_r()`, as they are not part of the C11 standard). Your code should:

- read from standard input by default and should also be able to read from files specified on the command-line. The syntax for which file(s) to read from should be the same as the UNIX command (run the command `man sort` for more details of what the `sort` command does).
- support the `-o`, `-n` and `-r` command-line options, in the same way as UNIX's `sort`.
- support an `-h` option that outputs usage information and briefly explains how much of this part of the coursework you have managed to implement.
- be robust.

The behaviour of your `sort` program should match UNIX `sort`'s as closely as possible. Submit a `sort.c` file containing your source code.