CSE 220: Systems Fundamentals I

Homework #4

Spring 2017

Assignment Due: Sunday April 9, 2017 by 11:59 pm

(This assignment will be worth 125 pts.)
(There will be limited PIAZZA assistance from the TAs/Prof on April 8/9th. Plan accordingly!)

Assignment Overview

The focus of this homework assignment is on reading data from files, memory organization, and working with multidimensional arrays in memory. This assignment also reinforces MIPS function calling and register conventions.

In this homework you will be implementing the functions to play Connect Four. For those unfamiliar, Connect Four is played by inserting colored chips in one of several columns, which then fall to the next unfilled slot in that column. It is a two-player game, and the first player to get four chips of their color in a row horizontally, vertically, or diagonally wins. Players use their color chip not only to get four in a row, but to block the other player by dropping chips to interrupt the other color's progress.

To implement the game, we will be using a 2D array to track the state of the board. As Connect Four is offered in several sizes, we will generalize the functions to support any board dimension. To succeed in this assignment, you should become familiar with loading from and storing to 2D arrays of arbitrary dimensions, as well as performing address arithmetic on 2D arrays containing objects of arbitrary size.

You **MUST** implement all of the functions in the assignment as defined. It is OK to implement additional helper functions of your own in `hw4.asm`.

⚠ **You MUST follow the MIPS calling and register conventions. If you do not, you WILL lose points.**

⚠ Do not submit a file with the functions/labels `main` or `_start` defined. You will obtain a ZERO for the assignment if you do this.

ℹ If you are having difficulties implementing these functions, write out the pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS.

ℹ When writing your program, try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.

Understanding the Game Board

The Slot Object

Each slot of the game board will take up two bytes (a half-word) in memory. A slot can hold exactly one game piece. The higher addressed byte contains the ASCII character representing the piece in that slot. Valid values are `'R'` for a red piece, `'Y'` for a yellow piece, and `'.'` for a blank slot. The lower addressed byte contains the (unsigned) turn number specifying the turn when the piece was placed in that slot. As we are constrained by the range of a byte, this means turn numbers will be in the range of `[0, 255]`. Empty slots will have a turn number of `0`. Slots with pieces will have a turn in the range `[1, 255]`.

The Board in Memory

Recall that 2D arrays can be stored in two ways: row-major order and column-major order. Our game board will be stored in row-major order. Because 2D arrays are implemented as an array of 1D arrays in memory, this means that the right-most slot in a row is followed in memory by the left-most slot of the next row. The 2D array will also order the rows such that the bottom of the board comes first. Thus, memory addresses will increase as we progress from the left-most slot to the right-most slot, and as we progress from the bottom-most row to the top-most row of the game board.

Notation and Terminology

The board will have `n` rows and `m` columns, which are input parameters.

Throughout this document we will refer to slots of the board in (row, col) format. Position `(0,0)` of the board is the bottom-left corner. Position `(n-1, m-1)` is the top-right corner of a board.

For example, the 42 slots in a board with 6 rows and 7 columns would be indexed as in the figure below.

Top of Board

(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)

Bottom of Board

Recall also that the bits of a byte, half-word, word, etc., are numbered from 0 to to k-1 starting from the right end of the k-bit unit. Thus, the byte of the slot object with the ASCII character uses bits `[8, 15]` and the byte with the turn number uses bits `[0, 7]`. For this reason, we may refer to the ASCII field of the slot object with the term "upper byte" and the turn number field as the "lower byte." Note that referring to bytes in upper/lower fashion is common parlance, and is good to know beyond just the context of this assignment.

Getting Started

From the Resources section of Piazza download the files `hw4.asm` which you need for the assignment. At the top of your `hw4.asm` file in comments put your name and SBU ID number.

```
# Homework #4
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to Test Your Functions

To perform **basic** tests on your functions, we suggest you write your own main functions **in separate files** which call each function independently. Remember to include your `hw4.asm` file in each of your main files using the line:

```
.include "hw4.asm"
```

⚠ Your assignment will not be graded using any of your mains!

Your main files will not be graded. No code that you include inside of the main files will be submitted. You will only submit your `hw4.asm` file via Sparky. Make sure that all code required for implementing your functions (`.text` and `.data`) are included in the `hw4.asm` file!

To make sure that your code is self-contained, try assembling your `hw4.asm` file by itself in MARS. If you get any errors (such as a missing label), this means that you need to refactor (reorganize) your code, possibly by moving labels you inadvertently defined in your mains to `hw4.asm`.

All functions implemented in the assignment must be placed in `hw4.asm` and follow the standard MIPS register conventions for functions taught in lecture. If you write your own helper functions, these also must be included in `hw4.asm`.

Note, there is no need to create any variables or memory storage for this homework in your `hw4.asm`. Therefore you should not declare a `.data` section in your `hw4.asm` file. Your functions should be implemented using ONLY of the information provided in the arguments. (Your main files will need to declare .data sections in order to call your functions with inputs.)

⚠ Make sure to initialize your registers within your own functions! Never assume registers or memory will hold any particular values!

⚠ Each function will be tested independently and therefore should not rely on any information other than the state of the arguments provided to your function. Do not write any functions that expect certain labels to be available in the main file.

Part I: Slots

We will start with getting and setting the values of an individual slot of the 2D array that represents the board. In order to access a slot on the board, we must use the general formula for working with 2D arrays in memory. Generalizing is not only good practice, but necessary in a situation like this assignment where we will be working with boards of arbitrary dimensions. Consider the following scenario:

Suppose you would like to access the `obj` in the 2D array `obj_arr`. The dimensions of `obj_arr` is `n` rows by `m` cols, and the location of `obj` is at `(i, j)` in the 2D array, where `i` is in the range [0, `n`-1], and `j` is in the range [0, `m`-1]. The address of `obj` is then given by:

```
obj_arr[i][j] = base_address + (row_size * i) + (size_of(obj) * j)
```

where

```
row_size = num_cols * size_of(obj)
```

Note that `obj_arr[i][j]` will be the computed address of where the desired object starts in memory, and not the object itself. Also, consider why the size of a row is `num_cols * size_of(obj)`. Why does this make sense? How would this calculation vary if it were column-major order instead? What is the size of an `obj` for us?

Now go ahead and implement the following functions:

a. `int set_slot(slot[][] board, int num_rows, int num_cols, int row, int col, char c, int turn_num)`

   This function takes in a 2D array `num_rows` by `num_cols` in size, calculates the address of a particular slot given by (`row`, `col`), then stores the given `c` and `turn_num` into the appropriate fields of the two-byte slot object in memory.

   You may assume the `board` is the appropriate size when given valid `num_rows` and `num_cols`. You may also assume the board is in a valid game state.

   Arguments:

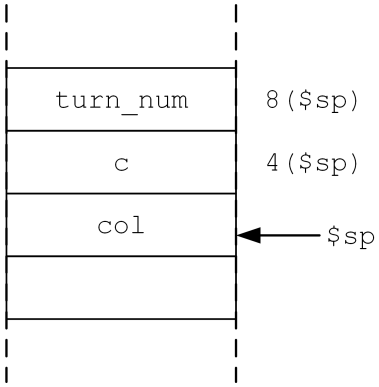
   - `board`: The starting address of a 2D array holding the state of the game board.

- `num_rows` : The number of rows in the board.
- `num_cols` : The number of columns in the board.
- `row` : The row number of the slot being set.
- `col` : The column number of the slot being set.
- `c` : The character being stored in the upper byte of the slot.
- `turn_num` : The (unsigned) turn number being stored in the lower byte of the slot.
- *returns*: 0 for success, -1 for error

Return -1 for error in any of the following cases:

- `num_rows` or `num_cols` is less than zero
- `row` is outside the range of [0, `num_rows` -1]
- `col` is outside the range [0, `num_cols` -1]
- `c` is not characters 'R', 'Y' or '.'
- `turn_num` is outside the range [0, 255]

Because the function has more than 4 arguments, the remaining arguments must be placed on the stack. The caller function will place the arguments `col` , `c` and `turn_num` on the stack prior to calling `set_slot` . At the start of the `set_slot` function, the stack pointer and arguments will be set as follows:



Remember, it is the responsibility of the function that PLACES arguments on the stack before calling an inner function to also REMOVE those arguments when that inner function returns. As the inner function, `set_slot` will read the arguments, but it SHOULD NOT shrink the stack to remove them.

b. `(char piece, int turn) get_slot(slot[][] board, int num_rows, int num_cols, int row, int col)`

This function takes the board, which is a 2D array `num_rows` by `num_cols` in size, calculates the address of a particular slot given by (`row` , `col`), then retrieves the ASCII character and turn number from the appropriate fields of the slot in memory.

You may assume the `board` is the appropriate size when given non-negative `num_rows` and `num_cols` . You may also assume the board is in a valid game state.

Arguments:

- `board` : The starting address of a 2D array holding the state of the game board.
- `num_rows` : The number of rows in the board.
- `num_cols` : The number of columns in the board.
- `row` : The row number of the slot being fetched.
- `col` : The col number of the slot being fetched.
- *returns*: The (ASCII character read from the slot in `$v0` , the turn number read from the slot in `$v1`) on success, or (-1,-1) on error

Return (-1,-1) for error in any of the following cases:

- `num_rows` or `num_cols` is less than zero
- `row` is outside the range [0, `num_rows` -1]
- `col` is outside the range [0, `num_cols` -1]

Now that we are able to get and set individual slots, we can do much more powerful manipulations. The first test of our ability to change the state of the board will be to use `set_slot` to clear the board.

Never assume that memory is in the state you want it to be in! The part of memory our board occupies may have previously been used by another program and thus have garbage data in it. To ensure a clean, appropriate state, we will have to explicitly set it.

c. `int clear_board(slot[][] board, int num_rows, int num_cols)`

This function will clear the board, removing all pieces. Loop over all cells of the 2D array and call `set_slot` for each entry. Set each slot to the default state: upper byte (ASCII field) to `'.'` and the lower byte (turn number) to 0.

You may assume the `board` is the appropriate size when given non-negative `num_rows` and `num_cols` . You may NOT assume the board is in a valid game state.

Arguments:

- `board` : The starting address of a 2D array holding the state of the game board.

- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- *returns*: 0 for success, -1 for error.

Return -1 for error if `num_rows` or `num_cols` is less than zero.

❗ `clear_board` MUST call `set_slot`

Part II: File Operations for Loading and Saving the Game

To assist with reading and writing files, MARS defines several system calls. As with the other system calls you know, the system call number is placed in `$v0`.

- **syscall 13: open file**

  Arguments:

  - `$a0`: address of null-terminated string containing filename
  - `$a1`: flags
  - `$a2`: mode (unused)

  Returns:

  - `$v0`: the *file descriptor*, which is an integer used internally to identify an open file (negative if error)

  Note: For syscall 13, MARS implements three `flag` values: 0 for read-only, 1 for write-only with create, and 9 for write-only with create and append. It ignores `mode`. The returned file descriptor will be negative if the operation failed.

- **syscall 14: read file**

  Arguments:

  - `$a0`: file descriptor (the integer returned by syscall 13)
  - `$a1`: address of input buffer
  - `$a2`: maximum # of characters to read

  Returns:

  - `$v0`: the # of characters read (0 if end-of-file, negative if error)

- **syscall 15: write file**

  Arguments:

- `$a0`: file descriptor (the integer returned by syscall 13)
- `$a1`: address of output buffer
- `$a2`: maximum # of characters to write

Returns:

- `$v0`: the # of characters written (0 if end-of-file, negative if error)

The underlying file I/O implementation uses `java.io.FileInputStream.read()` to read and `java.io.FileOutputStream.write()` to write. MARS maintains file descriptors internally and allocates them starting with 3. File descriptors 0, 1 and 2 are always open for: reading from standard input, writing to standard output, and writing to standard error, respectively. An example of these system calls in action can be found on the MARS website.

Using these syscalls, write the following functions to support loading and saving the game state to a file.

d. `(int num_rows, int num_cols) load_board(slot[][] board, char[] filename)`

This function opens a file referred to by `filename` as read only and initializes `board` with the information. The file is stored in UNIX ASCII format (meaning each line is ended with `\n` only). On Windows machines, newlines are stored as `\r\n`. Create all your test files using MARS to prevent any issues.

The file is formatted as:

```
rrcc
rrccpttt
....
```

ℹ You may assume each line is always properly formatted, and the file contains at least the first line. You may also assume board points to memory of sufficient size to hold the game state you are loading.

The first line of the file describes the dimensions of the board, where `rr` is a two-digit number specifying the number of rows in the board and `cc` is a two-digit number specifying the number of columns in the board.

The remaining lines in the file represent the state of the game board. To save space in the file when saving, empty slots are omitted. Therefore, only slots with pieces will be in the file on load. Each subsequent line corresponds to a single slot in the board. Each line is formatted as `rrccpttt`, where `rr` is the two-digit row for the piece, `cc` is the two-digit column for the piece, `p` is the piece color (`'R'` or `'Y'`), and `ttt` is the three-digit turn number for the placement of the piece. All numbers are expressed in base 10.

Example input file for a valid 4 row by 5 column game which has 5 moves:

```
0405
0000R003
0001Y002
0002R001
0101Y004
0201R005
```

In this example input file, Red started by placing a piece at position (0,2). Followed by Yellow placing a piece at position (0,1). Red continued with a piece at position (0,0). Yellow at position (1,1). Finally, Red at (2,1).

Example input file for a valid 9 row by 12 column game which has 0 moves:

```
0912
```

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `filename`: The name and path of the file which contains Connect Four game data.
- *returns*: (number of rows in `$v0`, the number of columns in `$v1`) on success, (-1,-1) on error

Return (-1,-1) for error in any of the following cases:

- any file error occurs
- the number of rows or columns in the board is equal to zero
- the turn number for any piece is outside the range [1,255]
- `row` for a piece is outside the range [0, `num_rows` -1]
- `col` for a piece is outside the range [0, `num_cols` -1]

ℹ This function only checks for valid input data as specified above. It DOES NOT error check the validity of the pieces placed in the board or the game rules. You will implement a another function, `validate_board` to check for valid game state/board to play.

❗ `load_board` MUST call `set_slot`

e. `int save_board(slot[][] board, int num_rows, int num_cols, char[] filename)`

This function will open the file specified by `filename` for write-only with create, output the current state of the board in the proper format, and close the file.

Proper output file format requires the following:

- Each line of the file is terminated with `'\n'`
- The first line of the file is present and is formatted `rrcc`
- The first line is followed by a line in the file per piece in the board in the format `rrccpttt`. The pieces in the board MUST BE outputted in row-major order to the file, `(0,0)` to `(n-1, m-1)`

One way to check if you outputted correctly, is to read the file created with `load_board`.

Arguments:

- `board`: The starting address of the 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- `filename`: The name and path of the file to store the Connect Four game data.
- *returns*: number of slot in the board which contain pieces (ie. total number of piece played) on success, -1 on error

Return -1 for error in any of the following cases:

- any file error occurs
- if `num_rows` or `num_cols` are less than zero
- the file can not be opened or written to

❗ `save_board` MUST call `get_slot`

f. `byte validate_board(slot[][] board, int num_rows, int num_cols)`

This function will check for the validity of the game board.

Recall from earlier that the size of the turn number field places some constraints on the range of a turn number. This field size likewise places some constraints on the size of our board. Because the maximum turn number is 255, the max number of slots must also be 255. Additionally, for someone to win in game play the board must be of sufficient size to allow 4 pieces to be placed in a row. Thus, the minimum number of rows and columns must each be set at 4.

It is possible that the board is invalid in multiple ways. Therefore, we will use a bit vector to represent the ways the board is invalid.

For each of the following possible error situations, set the specified bit of `$v0` to 1 if the error is present, and 0 if the error is not present. Bit #0 is the least significant (rightmost) bit of the byte, and bit #7 is the most significant (leftmost) bit.

- **Bit 0:** `num_rows` is less than 4.

- **Bit 1:** `num_cols` is less than 4.
- **Bit 2:** `num_rows` * `num_cols` is greater than 255.
- **Bit 3:** The absolute difference between the number of placed red pieces and yellow pieces is greater than 1.
- **Bit 4:** The red and yellow pieces DO NOT alternate turn numbers.
- **Bit 5:** An empty slot exists below a piece.
- **Bit 6:** A piece with a lower turn number resides above a piece with a higher turn number.
- **Bit 7:** There are 2 or more pieces with the same turn number or the turn numbers do not start at 1.

You may assume the `board` is the appropriate size when given non-negative `num_rows` and `num_cols`. You may NOT assume the board is in a valid game state.

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- *returns*: The bit vector (byte) as specified above.

❗ `validate_board` MUST call `get_slot` .

ℹ We have provided sample input files to test each of the error situations above independatly. We encourage you to make your own input files which test them in combination.

Part III: Gameplay

g. `int display_board(slot[][] board, int num_rows, int num_cols)`

In order for players to see the state of the board, it must be printed on request. This function will take the state of the board and print it in a grid `num_rows` by `num_cols` in size. The characters printed shall be `'R'` for red pieces, `'Y'` for yellow pieces, and `'.'` for blank slots.

Given the following game state (as seen in the `load_board` description), `display_board` would print the corresponding output:

```
Game state                Output

0405                      .....
0000R003                  .R...
0001Y002                  .Y...
0002R001                  RYR..
0101Y004
0201R005
```

You may assume the `board` is the appropriate size when given non-negative `num_rows` and `num_cols`. You may also assume the board is in a valid game state.

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- *returns*: number of slots which contain pieces, or -1 on error.

Return -1 if `num_rows` or `num_cols` are less than zero.

❗ `display_board` MUST call `get_slot` .

h. `int drop_piece(slot[][] board, int num_rows, int num_cols, int col, char piece, int turn_num)`

This function drops a piece into column `col` of the board. The piece will be placed in the lowest empty slot of `col`.

You may assume the `board` is the appropriate size when given non-negative `num_rows` and `num_cols`. You may also assume the board is in a valid game state.

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- `col`: The column number to drop the piece into.
- `piece`: `'R'` or `'Y'`.
- `turn_num`: The turn number being stored in the lower byte of the slot. All values `> 255` result in error.

- *returns*: 0 if the piece was successfully placed, or -1 on error.

`piece` will be passed as `0($sp)`, and `turn_num` will be passed as `4($sp)`.

Return -1 in any of the following cases:

- if `num_rows` or `num_cols` are less than zero
- `col` is outside the range of [0, `num_col` -1]
- `piece` is not `'R'` or `'Y'`
- `turn_num` is greater than 255
- there is no available slots in `col` to place the piece

❗ `drop_piece` MUST call `get_slot` and `set_slot`

i. `(char piece, int turn_num) undo_piece(slot[][] board, int num_rows, int num_cols)`

If you've ever played Connect Four, you've probably dropped a piece in the wrong column or generally wanted to take back a move. This function will check the board for the last piece dropped and reset the corresponding slot in the board back to default (`'.'` and 0). The color of the piece removed is returned with the turn number.

If there is no turn to undo (meaning the first move hasn't happened yet) then the function will return (`'.'`, -1).

While there is more than one way to undo a move, consider this: the state of the board is always passed to the function, therefore it should only consider the board it was given. DO NOT track the last move by storing it in memory. The functions are designed to get all necessary information from their parameters and theoretically could be used to maintain different games simultaneously. This means you should not store values across function calls in labels in your `hw4.asm`'s `.data` section.

Additionally, there is also algorithmic efficiency to consider. Where will you find the most recent dropped pieces on the board? You can get away with not checking all of the slots because in any column the most recent piece will be at the top.

You may assume the `board` is the appropriate size when given non-negative `num_rows` and `num_cols`. You may also assume the board is in a valid game state.

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.

- *returns*: for `piece` return `'.'` if no move can be undone, `'R'` if a red piece was removed or `'Y'` if a yellow piece was removed. Return the `turn_num` of the piece removed, or -1 if no moves have been made.

Return (`'.'`, -1) in any of the following cases:

- if `num_rows` or `num_cols` are less than zero
- there are no available pieces to remove (ie. board is empty)

❗ `undo_piece` MUST call `get_slot` and `set_slot`

j. `char check_winner(slot[][] board, int num_rows, int num_cols)`

This function searches the **entire board** to determine if a player has won the game. In the real game, a player can win by placing 4 in a row on a horizontally, vertically, or diagonally within the board. We will ONLY be handling horizontal and vertical placement. NO DIAGONALS!

You may assume the `board` is the appropriate size given `num_rows` and `num_cols`, and that they are greater than 4. You may also assume the board is in a valid game state (there is only 1 winner, but they could win in multiple ways).

ℹ We highly suggest you write out pseudo code or HLL code for your algorithm PRIOR to implementing this function!

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- *returns*: `'.'` if no winner yet, `'R'` if a red player has won or `'Y'` if the yellow player has won.

❗ `check_winner` MUST call `get_slot`

Extra Credit (15pts possible)

For those who want to implement and play the true connect4 game, additionally implement a function to check for diagonal wins for extra credit.

k. `char check_diagonal_winner(slot[][] board, int num_rows, int num_cols)`

This function searches the **entire board** to determine if a player has won the game with a diagonal ONLY.

You may assume the `board` is the appropriate size given `num_rows` and `num_cols`, and that

they are greater than 4. You may also assume the board is in a valid game state (there is only 1 winner, but they could win in multiple ways).

Arguments:

- `board`: The starting address of a 2D array holding the state of the game board.
- `num_rows`: The number of rows in the board.
- `num_cols`: The number of columns in the board.
- *returns*: `'.'` if no winner yet, `'R'` if a red player has won on a diagonal line or `'Y'` if the yellow player has won on a diagonal line.

Hand-in Instructions

See Sparky Submission Instructions on Piazza for hand-in instructions.

❗ There is no tolerance for homework submission via email. Work must be submitted through Sparky. Please do not wait until the last minute to submit your homework. If you are struggling, stop by office hours for additional help.