CSE 220: Systems Fundamentals I

Stony Brook University

Programming Assignment #3

Fall 2020

Assignment Due: Sunday, October 25, 2020 by 11:59 pm EDT

Updates to this Document

- 10/16/2020: Added a small clarification to Part 9.
- 10/15/2020: Added a small clarification to Part 7.
- 10/13/2020: The Part 10 function should return the number of moves performed.

Learning Outcomes

After completion of this programming project you should be able to:
- Implement non-trivial algorithms that require conditional execution and iteration.
- Design and implement functions that implement the MIPS assembly function calling conventions.
- Implement algorithms that process 2D arrays of values.
- Read files from disk using MIPS system calls.

Getting Started

Visit the course website and download the files hwk3.zip and MarsFall2020.jar. Fill in the following information at the top of hwk3.asm:
1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)

Having this information at the top of the file helps us locate your work. If you forget to include this information but don't remember until after the deadline has passed, don't worry about it – we will track down your submission.

Inside hwk3.asm you will find several function stubs that consist simply of `jr $ra` instructions. Your job
in this assignment is to implement all the functions as specified below. Do not change the function names, as the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, but they must be saved in hwk3.asm. Helper functions will not necessarily be graded, but might be spot-checked to ensure you are following the MIPS function calling conventions.

If you are having difficulty implementing these functions, write out 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 into MIPS assembly code.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

Finally, do not define a `.data` section in your hwk3.asm file. A submission that contains a `.data` section will probably result in a score of zero.

Important Information about CSE 220 Programming Projects

- Read the entire project description document twice before starting. Questions posted on Piazza whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- You must use the Stony Brook version of MARS posted on the course website. Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you might need to complete the homework assignments.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. It is much easier for your TAs and the professor to help you if we can quickly figure out what your code does.
- You personally must implement the programming assignments in MIPS Assembly language by yourself. You may not write or use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS assembly code you submit as part of the assignments.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- Do not submit a file with the function/label main defined. You are also not permitted to start your label names with two underscores (__). You will obtain a zero for the assignment if you do this.
- Submit your final .asm file to the course website by the due date and time. Late work will be penalized as described in the course syllabus. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

With few exceptions, all aspects of your programming assignments will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this homework assignment you will be writing functions in assembly language. The functions will be tested independently of each other. This is very important to note, as you must take care that no function you write ever has side-effects or requires that other functions be called before the function in question is called. Both of these are generally considered bad practice in programming.

Some other items you should be aware of:

- Each test case must execute in 1,000,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed. To find the instruction count of your code in MARS, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.
- Any excess output from your program (debugging notes, etc.) will impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

You must follow the register conventions taught in lecture and reviewed in recitation. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any $s registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save $ra before calling the callee. In addition, if the caller wants a particular $a, $t or $v register's value to be preserved across the secondary function call, the best practice would be to place a copy of that register in an $s register before making the function call.
- A function which allocates stack space by adjusting $sp must restore $sp to its original value before returning.
- Registers $fp and $gp are treated as preserved registers for the purposes of this course. If a function modifies one or both, the function must restore them before returning to the caller. There really is no reason for your code to touch the $gp register, so leave it alone.

The following practices will result in loss of credit:

- “Brute-force” saving of all $s registers in a function or otherwise saving $s registers that are not overwritten by a function.
- Callee-saving of $a, $t or $v registers as a means of “helping” the caller.
- “Hiding” values in the $k, $f and $at registers or storing values in main memory by way of offsets to $gp. This is basically cheating or at best, a form of laziness, so don’t do it. We will comment out any such code we find.

How to Test Your Functions

To test your implemented functions, open the provided “main” files in MARS. Next, assemble the main file and run it. MARS will include the contents of any .asm files referenced with the .include directive(s) at the end of the file and then append the contents of your hwk3.asm file before assembling the program.

Each main file calls a single function with one of the sample test cases and prints any return value(s). You will need to change the arguments passed to the functions to test your functions with the other cases. To test each of your functions thoroughly, create your own test cases in those main files. Your submission will not be graded using the examples provided in this document or using the provided main file(s). Do not submit your main files with your hwk3.asm file – we will delete them. Also please note that these testing main files will not be used in grading.

Again, any modifications to the main files will not be graded. You will submit only your hwk3.asm for grading. Make sure that all code required for implementing your functions is included in the hwk3.asm file. To make sure that your code is self-contained, try assembling your hwk3.asm file by itself in MARS. If you get any errors (such as a missing label), this means that your hwk3.asm file is attempting to reference labels in the main file, which will not have the same names during grading!

A Final Reminder on How Your Work Will be Graded

It is imperative (crucial, essential, necessary, critically important) that you implement the functions below exactly as specified. Do not deviate from the specifications, even if you think you are implementing the program in a better way. Modify the contents of memory only as described in the function specifications!

Preliminaries

For this assignment you will be implementing a version of the classic game of Snake in MIPS. Briefly, the snake consists of several body segments (represented by ASCII characters) that can move around a 2D game board, eating apples. Each time the snake eats an apple, its length increases by one. In the original game, the game ends when the snake runs into a wall, runs into itself, or runs out of bounds. We will implement the same basic logic in our version.

The head of the snake is represented by the ASCII character ‘1’, with each following segment in the body given by consecutive digit characters (i.e., ‘2’, ‘3’, ‘4’, etc.) After ‘9’, body segments are denoted with uppercase letters, starting with ‘A’ and going all the way through the alphabet to ‘Z’. Thus, the snake can be at most 9 + 26 = 35 body segments in length.

The game board consists of a 2D grid of ASCII characters (“slots”), with the period, ‘.’, representing an empty slot; a pound sign, ‘#’, representing a wall; the lowercase letter ‘a’ representing an apple, and the digits and uppercase letters representing parts of the snake. Below is an example of an in-progress game on a 15x30 grid (15 rows, 30 columns).

```
..............................
..............................
......######......######......
......#................#......
......#................#......
......#................#......
..............................
....a..........123............
.................4............
......#..........56....#......
......#................#......
......#................#......
......######......######......
..............................
..............................
```

The player scores points as he/she moves the snake about the game grid and eats apples. Scoring is explained later.

Data Structures

To characterize the state of the game we track several pieces of data in the form of a *struct*, which is like an object with data fields only (no methods). We will call this data type `GameState` throughout this document.

- `num_rows`: the number of rows in the grid (an unsigned byte)
- `num_cols`: the number of columns in the grid (an unsigned byte)
- `head_row`: the row of the snake’s head (an unsigned byte)
- `head_col`: the column of the snake’s head (an unsigned byte)
- `length`: the length of the snake (an unsigned byte)
- `grid`: the contents of the game board (a null-terminated string that stores the board in row-major order)

The topmost row of the grid is row #0, and the leftmost column is column #0. Similarly, the bottom-most row is `num_rows-1`, and the rightmost column is `num_cols-1`. The maximum grid size is 99x99.

As an example, the game board shown above would be represented in MIPS as follows:

```
.align 2  # ensures that the data structure is on a word-aligned boundary
state:    # name of the data structure (game state)
.byte 15  # num_rows (byte #0)
.byte 30  # num_cols (byte #1)
.byte 7   # head_row (byte #2)
.byte 15  # head_col (byte #3)
.byte 6   # length   (byte #4)
# Game grid: (bytes #5-end)
.asciiz
"..................................................................######.
.....######............#................#............#................#...
.........#................#........................................a......
....123.............................4..................#..........56....#.
...........#................#............#................#............###
###......######...........................................................
......."
```

To access the fields of the struct, we need to be provided its starting address (i.e., the address of the `num_rows` field). From there we can use `lbu` with appropriate offsets to access the field values, including the contents of the grid.

Part 1: Load a Game Board from Disk

```
int, int load_game(GameState* state, string filename)
```

The `load_game` function reads the contents of a file that defines a game board and initializes the referenced `GameState` data structure. The notation `GameState*` indicates that `state` is the starting address of a `GameState` struct. You may assume that `state` points to a block of memory large enough to store the struct represented inside the file. If the file exists, you may assume that the file's contents are valid. If the file does not exist, the function simply returns -1, -1.

The file format is very simple:

```
# of rows in the game grid
# of columns in the game grid
contents of grid in which each row is on its own line
```

Important note: on Microsoft Windows, a line ends with the character combination `\r\n`, whereas on Mac and other Unix-like operating systems like Linux, a line ends only with `\n`. Your code must be able to handle both line-ending styles. All lines of the file, including the final row of the grid, are guaranteed to end with `\n` or `\r\n`. As an example, the above game grid would be saved in a Microsoft Windows-generated file as:

```
15\r\n
30\r\n
..............................\r\n
..............................\r\n
......######......######......\r\n
......#................#......\r\n
......#................#......\r\n
......#................#......\r\n
..............................\r\n
....a..........123............\r\n
.................4............\r\n
......#..........56....#......\r\n
......#................#......\r\n
......#................#......\r\n
......######......######......\r\n
..............................\r\n
..............................\r\n
```

Again, your code must be able to contend with `\r\n` line-endings and with `\n` line-endings.

To assist with reading and writing files, MARS provides several system calls:

Service	**Code in $v0**	**Arguments**	**Results**
open file	13	$a0 = address of null-terminated filename string $a1 = flags $a2 = mode	$v0 contains file descriptor (negative if error)
read from file	14	$a0 = file descriptor $a1 = address of input buffer $a2 = maximum # of characters to read	$v0 contains # of characters read (0 if end-of-file, negative if error)
write to file	15	$a0 = file descriptor $a1 = address of output buffer (negative if error) $a2 = maximum # of characters to write	$v0 contains # of characters written
close file	16	$a0 = file descriptor	

Service 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. 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 how to use these syscalls can be found on the MARS syscall web page.

Some advice: read the contents of the file one character at a time using system call #14. This system call requires a memory buffer to hold the character read from the disk. You should allocate four bytes of memory on the stack (by adjusting $sp) to store that byte temporarily. Discard newline characters (both `\r` and `\n`) as you read them and do not store them in the `GameState` struct. Finally, remember to reset $sp once you have finished reading the file contents and to close the file with system call #16.

MARS can be a little buggy when it comes to opening files. Therefore, either:

- put all your .asm and .txt game files in the same directory as the MARS .jar file, or
- use absolute path names when giving the filename in your testing mains.

The function `load_game` takes the following arguments, in this order:

- `state`: a *pointer* to (i.e., starting memory address of) an uninitialized `GameState` struct large enough to hold the contents of the game represented by file to be loaded. Assume that the contents of the struct are filled with random garbage.
- `filename`: a string containing the filename to open and read the contents of

Returns in $v0:

- -1 if the input file does not exist; otherwise: 1 if a lowercase letter ‘a’ representing an apple was found in the file; 0, if not

Returns in $v1:

- -1 if the input file does not exist; otherwise: the number of wall characters, ‘#’, found in the file

Additional requirements:

- The function must not write any changes to main memory except as needed. If the file does not exist, no changes may be written to `state`.

Example #1:

`filename = “game01.txt”`
Return value in $v0: 1
Return value in $v1: 6
Contents of `state` must be updated to reflect the data structure defined in the file.

Example #2:

`filename = “game02.txt”`
Return value in $v0: 0
Return value in $v1: 16
Contents of `state` must be updated to reflect the data structure defined in the file.

Example #3:

`filename = "junk.txt"` (non-existent file)
Return value in $v0: -1
Return value in $v1: -1
Contents of `state` must be left unchanged.

Part 2: Get the Character Stored in the Game Grid

```
int get_slot(GameState* state, byte row, byte col)
```

The function `get_slot` returns the character stored in the slot at `state.grid[row][col]`. A recommendation: use `lbu` to read tile values from the game board. If `row` is outside the valid range or `col` is outside the valid range for the given `GameState`, the function returns -1. For instance, the valid range for the `row` argument is `[0, board.num_rows-1]`.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `row`: the row of the `grid` array from where we want to read a value. This is an 8-bit, two's complement value.
- `col`: the column of the `grid` array from where we want to read a value. This is an 8-bit, two's complement value.

Returns in $v0:

- the character located at `state.grid[row][col]`, or
- -1 for the error condition explained above

Additional requirements:

- The function must not write any changes to main memory.

Examples:

In the examples below, imagine that we initialized state by loading it with `game01.txt`:

```
5
12
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.....4567...
```

Return values other than -1 are the ASCII codes of the returned characters.

row	**col**	**Return Value ($v0)**
1	5	49
2	3	35
3	16	-1
-3	5	-1

Part 3: Set the Character Stored in the Game Grid

`int set_slot(GameState* state, byte row, byte col, char ch)`

The function `set_slot` sets the value stored in the slot at `state.grid[row][col]` to `ch`. A recommendation: use `sb` to write a character in the game grid. If `row` is outside the valid range or `col` is outside the valid range for the given `GameState`, the function returns -1. Otherwise,the function returns `ch`. The function does not need to validate that `ch` is a valid character for the Snake game.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `row`: the row of the `grid` array at which we want to write a character. This is an 8-bit, two's complement value.
- `col`: the column of the `grid` array at which we want to write a character. This is an 8-bit, two's complement value.
- `ch`: the character to write at `state.grid[row][col]`

Returns in $v0:

- `ch`, provided that `row` and `col` are both valid, or
- -1 for the error condition explained above

Additional requirements:

- The function must not write any changes to main memory except as necessary.

Examples:

In the examples below, imagine that we initialized state by loading it with `game01.txt`.

```
5
12
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.....4567...
```

Example #1:

```
row = 0
col = 5
ch = ‘Q’
```
Return value in $v0: 81
Updated game state:
```
.....Q......
.a.#.1..#...
...#.2..#...
...#.3..#...
.....4567...
```

Example #2:
```
row = 2
col = 11
ch = ‘Z’
```
Return value in $v0: 90
Updated game state:
```
............
.a.#.1..#...
...#.2..#..Z
...#.3..#...
.....4567...
```

Example #3:
```
row = 3
col = 7
ch = ‘?’
```
Return value in $v0: 63
Updated game state:
```
............
.a.#.1..#...
...#.2..#...
...#.3.?#...
.....4567...
```

Example #4:
```
row = -5
col = 2
ch = ‘*’
```
Return value in $v0: -1

The game state should be unchanged.

Part 4: Place an Apple in the Game Grid

```
int, int place_next_apple(GameState* state, byte[] apples,
                          int apples_length)
```

The function `place_next_apple` searches the `apples` array for a valid apple to place in the game grid and places it. The `apples` array consists of `apples_length` pairs of integers stored in row-major order that represent `(row, col)` coordinates where an apple hasn't yet been placed in the game grid. The pair `(-1, -1)` in the array indicates that an apple has already been placed at that position in the grid. For example, perhaps that apple was placed during a prior call to `place_next_apple`.

As an example, suppose `apples` begins `[4, 2, 7, 3, 1, 9, -1, -1, ...]`. This indicates that potential locations in the game grid where an apple could be placed include (4, 2), (7, 3), (1, 9), among others. To find an apple to place in the grid, the function will start with (4, 2) and see if that position in the grid is empty (i.e., `state.grid[4][2]` contains '.'). If so, `state.grid[4][2]` is changed to 'a' with `set_slot`, and then `apples[0]` and `apples[1]` are both changed to -1. On the other hand, suppose `state.grid[4][2]` contains a wall or a part of the snake or another apple. In this case, `place_next_apple` skips over the (4, 2) pair and tries (7, 3). The search continues in this fashion until a valid apple can be placed. When the pair (-1, -1) is encountered in `apples`, `place_next_apple` skips over it and continues to the next pair. You may safely assume that `place_next_apple` will always be able to find a valid apple to place in the game grid.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `apples`: an array of `apples_length` pairs of 8-bit, two's complement integers
- `apples_length`: the number of pairs of integers in `apples`

Returns in $v0:

- the row of the apple that was placed

Returns in $v1:

- the column of the apple that was placed

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `get_slot` and `set_slot`.

Examples:

In the examples below, imagine that we initialized state by loading it with `game01.txt`:

```
5
12
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.....4567...
```

Example #1:

```
apples = [3, 2, 1, 4, 4, 3]
apples_length = 3
```
Return value in $v0: 3
Return value in $v1: 2
Updated `apples`: `[-1, -1, 1, 4, 4, 3]`
Updated game state:
```
............
.a.#.1..#...
...#.2..#...
..a#.3..#...
.....4567...
```

Example #2:

```
apples = [2, 5, 3, 5, 1, 4, 4, 3, 2, 9]
apples_length = 5
```
Return value in $v0: 1
Return value in $v1: 4
Updated `apples`: `[2, 5, 3, 5, -1, -1, 4, 3, 2, 9]`
Updated game state:
```
............
.a.#a1..#...
...#.2..#...
...#.3..#...
.....4567...
```

Example #3:

```
apples = [2, 5, 3, 5, -1, -1, -1, -1, 4, 1, 4, 3, 2, 9]
apples_length = 7
```
Return value in $v0: 4

Return value in $v1: 1
Updated `apples: [2, 5, 3, 5, -1, -1, -1, -1, -1, -1, 4, 3, 2, 9]`
Updated game state:

```
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.a...4567...
```

Part 5: Find the Next Body Segment of the Snake

```
int, int find_next_body_part(GameState* state, byte row, byte col,
                             char target_part)
```

The function `find_next_body_part` inspects the four grid slots up, down, to the left and to the right of `state.grid[row][col]` for the character `target_part`. If `target_part` is found in one of those locations, then the coordinates of `target_part` are returned. Otherwise, the function returns -1, -1. The function also returns -1, -1 if `row` or `col` is outside its valid range for `state.grid`. This function is called by other functions inside of a loop to find all of the body segments of the snake.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `row`: the row of the `grid` array slot around which we search for `target_part`. This is an 8-bit, two's complement value.
- `col`: the column of the `grid` array slot around which we search for `target_part`. This is an 8-bit, two's complement value.
- `target_part`: the part of the snake's body segment we are looking for

Returns in $v0:

- the row of `target_part`, if it was found; -1, otherwise

Returns in $v1:

- the column of `target_part`, if it was found; -1, otherwise

Additional requirements:

- The function must not write any changes to main memory.
- The function must call `get_slot`.

Examples:

In the examples below, imagine that we initialized state by loading it with `game01.txt`:

```
5
12
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.....4567...
```

`row`	`col`	`target_part`	**Return Values**
3	5	'4'	4, 5
4	7	'7'	4, 8
3	5	'9'	-1, -1

Part 6: Slide (Slither) the Snake Forward One Slot

```
int slide_body(GameState* state, byte head_row_delta, byte head_col_delta,
               byte[] apples, int apples_length)
```

The function `slide_body` attempts to slide (slither) the snake one place forward in the direction `(head_row_delta, head_col_delta)`:

- (1, 0): move the snake down one row, starting from the head and with the body following, one segment at a time
- (-1, 0): move the snake up one row
- (0, 1): move the snake right one column
- (0, -1): move the snake left one column

This function does not merely slide the entire snake up, down, left or right. Rather, the pair `(head_row_delta, head_col_delta)` indicates how the snake's head should *lead* the movement. The fields `state.head_row` and `state.head_row` must be updated, as appropriate.

For instance, suppose the snake is in the configuration below:

```
..........
.a........
.....1....
.....234..
.......56.
```

and we call `slide_body` with (0, -1) for `(head_row_delta, head_col_delta)`. The snake is now positioned as follows:

```
..........
.a........
....12....
.....345..
.......6..
```

Suppose we call `slide_body` with (0, -1) for `(head_row_delta, head_col_delta)` two more times. Now we have this:

```
..........
.a........
..1234....
.....56...
..........
```

Notice how the body slithers along with the head leading the way.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `head_row_delta`: the vertical direction in which to move the snake's head. This is an 8-bit, two's complement value.
- `head_col_delta`: the horizontal direction in which to move the snake's head. This is an 8-bit, two's complement value.
- `apples`: an array of `apples_length` pairs of 8-bit, two's complement integers
- `apples_length`: the number of pairs of integers in `apples`

You may assume that the pair `(head_row_delta, head_col_delta)` is one of the four combinations of -1, 0, 1 listed above.

The function returns a code in $v0 which indicates what happened during the attempted move:

- 0 if the snake's head moved onto an empty slot (i.e., '.') and slithered forward without any trouble
- 1 if the snake's head moved onto a slot containing an apple (i.e., 'a') and slithered forward without any trouble

- -1 if the snake could not move forward because doing so would take it outside the game board, would intersect a wall slot, or would intersect with a body part of the snake. Return -1 without making any changes to the state of the game grid.

To make the snake slither forward, first move the head one slot forward in the desired direction. Then, call `find_next_body_part` repeatedly in a loop to iteratively move each other part forward one place. Specifically, body segment ‘2’ will take the place of ‘1’ (the head), ‘3’ will take the place of ‘2’, ‘4’ will take the place of ‘3’, and so on, until the final body segment of the snake has moved.

In the case that the snake is about to move onto an apple, call `place_next_apple` before moving any segment of the snake forward. Then have the entire snake slither forward in the desired direction, with the head replacing the apple. Note that `slide_body` does not increase the length of the snake when the snake consumes an apple. That task is handled by a different function (`increase_snake_length`).

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `get_slot`, `set_slot`, `place_next_apple` and `find_next_body_part`.

Examples:

In the examples below, imagine that we initialized state by loading it with `game03.txt`:

```
8
14
..............
......##......
..............
..#..a.....#..
..#..1234..#..
........56...E
......##.7..CD
.........89AB.
```

Example #1:

```
head_row_delta = 0
head_col_delta = -1
apples = [1, 7, 3, 2, 1, 4, 4, 3]
apples_length = 4
```

Return value in $v0: 0

Expected `apples: [1, 7, 3, 2, 1, 4, 4, 3]`

Updated game state:

```
..............
......##......
..............
..#..a.....#..
..#.12345..#..
........67....
......##.8..DE
.........9ABC.
```

Example #2:

```
head_row_delta = -1
head_col_delta = 0
apples = [1, 7, 3, 2, 1, 4, 4, 3]
apples_length = 4
```

Return value in $v0: 1
Updated `apples: [1, 7, 3, 2, -1, -1, 4, 3]`
Updated game state:

```
..............
....a.##......
..............
..#..1.....#..
..#..2345..#..
........67....
......##.8..DE
.........9ABC.
```

Example #3:

```
head_row_delta = 0
head_col_delta = 1
apples = [1, 7, 3, 2, 1, 4, 4, 3]
apples_length = 4
```

Return value in $v0: -1
`apples` remains unchanged.
Game state remains unchanged.

Part 7: Add a New Body Segment to the Tail of the Snake

```
int add_tail_segment(GameState* state, char direction, byte tail_row,
                     byte tail_col)
```

The function `add_tail_segment` takes the coordinates of the snake's tail at `(tail_row, int tail_col)` and attempts to add a new segment to the tail of the snake in the given direction (`char` direction). Note that this function is essentially a helper function for increase_snake_length. For example, suppose that direction = 'U' and tail_row = 3, tail_col = 6. These arguments indicate that the function should try to add a tail segment at index state.grid[2][6]. You may assume that state.grid[tail_row][tail_col] is the valid, actual tail of the snake. The new tail segment may only be placed on an empty grid slot, denoted by the period character, '.'. If the new tail segment is successfully added, the length field of the state object must be incremented by 1. That new length is returned by add_tail_segment. If the tail segment cannot be added (e.g., due to a collision or because the snake's length is already 35), the function makes no changes to main memory and returns -1.

The function takes the following arguments, in this order:

- state: a pointer to a valid GameState struct
- direction: the direction relative to the tail of the snake to append a new segment. Must be one of 'U', 'D', 'L' or 'R'. For example, 'D' indicates that the new tail segment should be placed in the row immediately below the current tail segment (if possible). As another example, 'R' indicates that the new tail segment should be placed in the column immediately to the right of the current tail segment (if possible).
- tail_row: the row number of the final segment in the tail. You may assume that this value is valid. This is an 8-bit, two's complement value.
- tail_col: the column number of the final segment in the tail. You may assume that this value is valid. This is an 8-bit, two's complement value.

Returns in $v0:

- The updated value of state.length if the new tail segment was placed. If the tail segment could not be placed or if direction is not one of 'U', 'D', 'L' or 'R', then $v0 is -1.
- If direction is invalid, the function must not make any changes to memory and return -1.

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call get_slot and set_slot.

Examples:

In the examples below, imagine that we initialized state by loading it with game01.txt:

```
5
12
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.....4567...
```

Example #1:

```
tail_row = 4
tail_col = 8
direction = 'R'
```
Return value in $v0: 8
Updated game state:
```
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.....45678..
```

Example #2:

```
tail_row = 4
tail_col = 8
direction = 'D'
```
Return value in $v0: -1
Game state remains unchanged.

Example #3:

```
tail_row = 4
tail_col = 8
direction = 'U'
```
Return value in $v0: -1
Game state remains unchanged.

Part 8: Increase the Length of the Snake's Body

```
int increase_snake_length(GameState* state, char direction)
```

The function `increase_snake_length` calls upon `find_next_body_part` (iteratively) and then `add_tail_segment` to find the tail of the snake in the game and append a new tail segment in the *opposite* direction that the snake's head is moving. For example, if `direction = 'L'`, then `increase_snake_length` will attempt to add a new segment to the *right* of the current tail. Likewise, if `direction = 'D'`, then `increase_snake_length` will attempt to add a new segment *above* the current tail segment. If this attempt is not successful, the function will search the cells in the neighborhood of the tail in counterclockwise order to find a position for the new tail segment. This aspect of the function will be made clear with a set of examples below. Note that `increase_snake_length` does not change the value of the `length` field in `state`.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct

- `direction`: the direction that the snake's head is moving. Must be one of 'U', 'D', 'L' or 'R'.

Returns in $v0:

- `state.length` if a new tail segment was successfully added by `add_tail_segment`, or -1 if not.
- If `direction` is invalid, the function must not make any changes to memory and return -1.

Additional requirements:

- The function must not write any changes to the main memory itself. Any changes written to memory must be made via `add_tail_segment`.
- The function must call `find_next_body_part` and `add_tail_segment` .

Example #1: The snake's head is moving to the left (`direction = 'L'`), so `increase_snake_length` calls `add_tail_segment` to try appending a segment to the *right* end of the current tail. `add_tail_segment` returns 8, indicating success.

The game state has been initialized by loading it with `game01.txt`:

```
5
12
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.....4567...
```

`direction = 'L'`
Return value in $v0: 8
Updated game state:

```
............
.a.#.1..#...
...#.2..#...
...#.3..#...
.....45678..
```

Example #2: The snake's head is moving to the up (`direction = 'U'`), so `increase_snake_length` calls `add_tail_segment` to try appending a segment *below* the current tail. `add_tail_segment` returns -1 because a body segment is in the way. So, `increase_snake_length` calls `add_tail_segment` to try appending a segment to the *right* of the current tail. `add_tail_segment` once again returns -1 because that slot is out of bounds. `increase_snake_length` continues by calling `add_tail_segment` to try appending a segment to the *above* the end of the current tail. `add_tail_segment` returns 15, indicating success.

The game state has been initialized by loading it with `game04.txt`:

```
8
14
..............
......##......
..............
..#........#..
a.#..1234..#..
........56...E
......##.7..CD
.........89AB.
```

```
direction = 'U'
```
Return value in $v0: 15
Updated game state:

```
..............
......##......
..............
..#........#..
a.#..1234..#.F
........56...E
......##.7..CD
.........89AB.
```

Example #3: The snake's head is moving down (`direction = 'D'`), so `increase_snake_length` calls `add_tail_segment` to try appending a segment *above* the current tail. `add_tail_segment` returns -1 because an apple is in the way. So, `increase_snake_length` calls `add_tail_segment` to try appending a segment *left* of the current tail. `add_tail_segment` returns 14, indicating success.

The game state has been initialized by loading it with `game05.txt`:

```
8
14
..............
......##......
..............
..#........#..
..#..1234..#a.
........56..D.
......##.7..C.
.........89AB.
```

```
direction = 'D'
```
Return value in $v0: 14

Updated game state:

```
..............
......##......
..............
..#........#..
..#..1234..#a.
........56.ED.
......##.7..C.
.........89AB.
```

Example #4: The snake's head is moving up (`direction = 'U'`), so `increase_snake_length` calls `add_tail_segment` to try appending a segment below the current tail. `add_tail_segment` returns -1, indicating failure. Next, `increase_snake_length` will `add_tail_segment` three more times, attempting to append tail segments to the right, above and to the left of the current tail segment. In each case, `add_tail_segment` returns -1, indicating failure. Because in every case `add_tail_segment` returned -1, `increase_snake_length` returns -1.

The game state has been initialized by loading it with `game06.txt`:

```
5
5
....a
..1..
.#2#.
.#3#.
.###.
```

`direction = 'U'`
Return value in $v0: -1
The grid after call to `increase_snake_length` remains unchanged.

Part 9: Move the Snake Forward One Slot

```
int, int move_snake(GameState* state, char direction, byte[] apples,
                    int apples_length)
```

The function `move_snake` is a top-level function of the game that moves the snake through the game grid one slot at a time. Given the value of `direction`, it makes an appropriate call to `slide_body` in an attempt to move the snake forward one slot in the grid.

- If the return value of `slide_body` is -1, then `move_snake` returns (0, -1) to indicate, respectively, that no points were scored during this call to `move_snake` and that ultimately the movement was unsuccessful.

- On the other hand, if `slide_body` returned 1, this means the snake ate an apple. In response, `move_snake` calls `increase_snake_length` to attempt to increase the length of the snake, using the direction of the head's movement as the value for `direction`. The call to `increase_snake_length` returns either -1, indicating failure to increase the length of the body, or a value other than -1 to indicate success. If `increase_snake_length` failed, then `move_snake` returns (0, -1). Otherwise, it returns (100, 1) indicating that 100 points were scored.
- The last possibility for `slide_body` is that it returned 0, indicating that the snake's head (and body) moved forward successfully and the head moved into an empty slot. In this case, `move_snake` returns (0, 1) to indicate that no points were scored, but that the snake successfully moved forward.

Note that if `direction` is invalid, `move_snake` returns (0, -1).

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `direction`: the direction that the snake's head is moving. Must be one of 'U', 'D', 'L' or 'R'.
- `apples`: an array of `apples_length` pairs of (signed) integers
- `apples_length`: the number of pairs of integers in `apples`

Returns in $v0:

- The number of points scored during this function call. Must be 0 or 100.

Returns in $v1:

- 1 if `increase_snake_length` successfully added a new body segment to the tail of the snake or the snake simply moved forward into an empty slot;
- or -1 if `increase_snake_length` does not add a segment or if `direction` is invalid.

Additional requirements:

- The function must not write any changes to the main memory itself. Any changes written to memory must be made via `slide_body` and `increase_snake_length`.
- The function must call `slide_body` and `increase_snake_length`.

Example #1:

Suppose that we initialized state by loading it with `game03.txt`:

```
8
14
..............
......##......
..............
..#..a.....#..
..#..1234..#..
........56...E
```

```
......##.7..CD
.........89AB.

direction = ‘U’
apples = [1, 2, 2, 9, 0, 5, 1, 7, 6, 10, 3, 10, 3, 11, 2, 10, 2, 1, 2, 4,
2, 5, 1, 13]
apples_length = 12
```
Return value in $v0: 100
Return value in $v1: 1
Updated `apples`: `[-1, -1, 2, 9, 0, 5, 1, 7, 6, 10, 3, 10, 3, 11, 2, 10, 2, 1, 2, 4, 2, 5, 1, 13]`
Updated game state:
```
..............
..a...##......
..............
..#..1.....#..
..#..2345..#..
........67....
......##.8..DE
.........9ABCF
```

Example #2:

Suppose that we initialized state by loading it with `game03.txt`:

```
8
14
..............
......##......
..............
..#..a.....#..
..#..1234..#..
........56...E
......##.7..CD
.........89AB.

direction = ‘D’
apples = [1, 2, 2, 9, 0, 5, 1, 7, 6, 10, 3, 10, 3, 11, 2, 10, 2, 1, 2, 4,
2, 5, 1, 13]
apples_length = 12
```
Return value in $v0: 0
Return value in $v1: 1
`apples` remains unchanged.
Updated game state:

```
..............
......##......
..............
..#..a.....#..
..#..2345..#..
.....1..67....
......##.8..DE
.........9ABC.
```

Example #3:

Suppose that we initialized state by loading it with `game07.txt`:

```
5
12
............
.a.#....#...
...#12..#...
...#.3..#...
.....4567...

direction = 'L'
apples = [4, 4, 2, 7, 3, 5, 1, 8, 1, 7, 3, 11, 1, 11, 0, 4]
apples_length = 8
```

Return value in $v0: 0
Return value in $v1: -1
`apples` remains unchanged.
Game state remains unchanged.

Example #4:

Suppose that we initialized state by loading it with `game07.txt`:

```
5
12
............
.a.#....#...
...#12..#...
...#.3..#...
.....4567...

direction = 'Z'
apples = [4, 4, 2, 7, 3, 5, 1, 8, 1, 7, 3, 11, 1, 11, 0, 4]
```

`apples_length = 8`
Return value in $v0: 0
Return value in $v1: -1
`apples` remains unchanged.
Game state remains unchanged.

Part 10: Simulate a Game of Snake

```
int, int simulate_game(GameState* state, string filename, string
directions, int num_moves_to_execute, byte[] apples, int apples_length)
```

The function `simulate_game` simulates at most `num_moves_to_execute` moves of a game of Snake. The initial configuration of the game grid is given by the contents of the file with name `filename`. The null-terminated string `directions` consists of the uppercase letters 'L', 'R', 'U' and 'D' in any order and combination and represents the attempted moves of the player. The array `apples` contains the positions of potential apples that can be placed during the simulation. The simulation algorithm that you must implement is as follows:

1. Call `load_game` to attempt to initialize the uninitialized `state` struct with the contents of filename.
    a. If `load_game` fails to initialize the struct, `simulate_game` returns -1, -1 and makes no changes to main memory. Otherwise, we continue on to the next step.
2. If `load_game` did not find an apple while initializing the `state` struct, call `place_next_apple` with appropriate arguments.
3. Initial a variable to store the total score to 0.
4. While `num_moves_to_execute` is not yet 0 **and** the snake's length is still less than 35 **and** the player hasn't lost (e.g., by running into a wall or intersecting the snake itself) **and** there are still directions left to consider:
    a. Get the next direction to move from `directions`.
    b. Call `move_snake` with appropriate arguments.
        i. If `move_snake` returned a `score` of 100, then add `score * (state.length - 1)` to the total score.
        ii. On the other hand, if `move_snake` returned a result of -1, indicating failure to move the snake, immediately break out of the loop.
    c. Decrement `num_moves_to_execute` by 1.
5. Return the number of executed moves in $v0 and the total score in $v1.

The function takes the following arguments, in this order:

- `state`: a *pointer* to (i.e., starting memory address of) an uninitialized `GameState` struct large enough to hold the contents of the game represented by file to be loaded. Assume that the contents of the struct are filled with random garbage.
- `filename`: a string containing the filename to open and read the contents of

- `directions`: a null-terminated string of characters representing directions to move the snake. Each character is guaranteed to be one of 'U', 'D', 'L' or 'R'. You do not need to validate the contents of `directions`.
- `num_moves_to_execute`: the maximum number of characters to read from `directions`
- `apples`: an array of `apples_length` pairs of 8-bit, two's complement integers Available at 0($sp).
- `apples_length`: the number of pairs of integers in `apples`. Available at 4($sp).

Returns in $v0:

- The number of moves executed.

Returns in $v1:

- The total score at the end of the simulated game.

Additional requirements:

- The function must not write any changes to main memory.
- The function must call `load_game`, `place_next_apple` and `move_snake`

Examples:

Due to the complexity of the function arguments, examples of `simulate_game` are provided in `simulate_game_test1.txt`, `simulate_game_test2.txt`, etc., along with function arguments.

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment may be discovered and penalized after the grades for that homework have been recorded.

8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.

How to Submit Your Work for Grading

To submit your hwk3.asm file for grading:

1. Go to the course website.
2. Click the Submit link for this assignment.
3. Type your SBU ID# on the line provided.
4. Press the button marked **Add file** and follow the directions to attach your file.
5. Hit Submit to submit your file grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the steps again. We will grade only your last submission.