CSE 220: Systems Fundamentals I

Homework #2

Fall 2017

Assignment Due: October 8, 2017 by 11:59 pm via Sparky

⚠ **READ THE WHOLE DOCUMENT TWICE BEFORE STARTING!**

⚠ DO **NOT** COPY/SHARE CODE! We will check your assignments against this semester and previous semesters!

ℹ Download the Stony Brook version of MARS posted on Piazza. **DO NOT USE** the MARS available on the official webpage. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you will need to complete the homework assignments.

⚠ All assignments MUST be implemented in MIPS Assembly language.

⚠ All test cases MUST execute in 10,000 instructions or less. Efficiency is an important aspect of programming.

⚠ Any excess output from your program (debugging notes, etc) may impact your grading. Do not leave erroneous printouts in your code!

⚠ Do not submit a file with the functions/labels `main` or `_start` 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.

Introduction

In this assignment you will be creating functions. The goal is to understand passing arguments, returning values, and the role of register conventions. We will continue our exploration of IPv4 packets by calculating checksums and rebuilding the payload contents from an array of fragmented packets.

You **MUST** implement all the functions in the assignment as defined. It is OK to implement additional helper functions of your own in `hw2.asm`.

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.

Getting started

Download `hw2.zip` from Piazza in the Homework section of Resources. This file contains `hw2.asm` and multiple `hw2_main` files, which you need for the assignment. At the top of your `hw2.asm` program in comments put your name and SBU ID number.

```
# Homework #2
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to test your functions

To test your functions, simply open one of the provided `hw2_main` files in MARS. Next, assemble the `main` file and run. Mars will take the contents of the file referenced with the `.include` at the end of the file and add the contents of your `hw2.asm` file to the main file before assembling it. Once the contents have been substituted into the file, Mars will then assemble it as normal.

Each of the main files tests the functions you are to implement with one of the sample test cases. You should modify these files or create your own files in order to test your functions with more test cases.

⚠ Your assignment will not be graded using these tests!

Any modifications to the `main` files will not be graded. You will only submit your `hw2.asm` file via Sparky. Make sure that all code required for implementing your functions (`.text` and `.data`) are included in the `hw2.asm` file! To make sure that your code is self-contained, try assembling your `hw2.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 a `main` file to `hw2.asm`.

⚠ It is highly advised to write your own main programs (new individual files) to test each of your functions thoroughly.

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

Part 1: Basic Functions

a. `int replace1st(char[] string, char toReplace, char replaceWith)`

This function finds the FIRST occurrence of the ASCII character `toReplace` in null-terminated `string` and replaces it with the ASCII character `replaceWith`.

ℹ We WILL NOT test the function with an invalid `string` argument address.

ℹ The function MUST modify `string` directly in memory.

- `string`: starting address of the character array (aka string).
- `toReplace`: ASCII character to find in `string`.
- `replaceWith`: ASCII character to insert into `string`.
- *returns*: the address of the character in the string AFTER the replaced character, or 0 if no occurrence of the `toReplace` character was found in `string`.

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

- `toReplace` or `replaceWith` is not a valid ASCII character, outside the range [`0x00`,`0x7F`].

❗ Your function may not modify `string` except as required to implement the above specification.

Examples:

Sample input strings are provided in `hw2_examples.asm`. Assume the starting address of the `string` argument is 0x400 for each example.

Code	**Return Value**	Modified**string**
`replace1st("Funny Funny", ‘F’,‘B’)`	0x401	"Bunny Funny"
`replace1st("Funny Bunny", ‘B’,‘s’)`	0x407	"Funny sunny"
`replace1st("Funny\tBunny", ‘\t’,‘_’)`	0x406	"Funny_Bunny"
`replace1st("Funny Funny", ‘t’,‘S’)`	0	"Funny Funny"
`replace1st("Funny Funny", 0x80, ‘A’)`	-1	"Funny Funny"
`replace1st("Funny Funny", ‘F’, 0xFF)`	-1	"Funny Funny"
`replace1st("Funny Funny", 0x80, 0xE1)`	-1	"Funny Funny"

b. `int printStringArray(String[] sarray, int startIndex, int endIndex,`
`int length)`

This function prints out the null-terminated strings specified by the addresses stored in `sarray` in the range `sarray[startIndex]` through `sarray[endIndex]` (inclusive).

Each string is printed to the screen followed by two newline character "\n\n".

The function MUST NOT modify the array in memory.

We WILL NOT test the function with an invalid `sarray` argument address.

- `sarray` : starting address of the array in memory (indexed by 0).
- `startIndex` : index of array to begin printing from (inclusive).
- `endIndex` : index of array to stop printing at (inclusive).
- `length` : number of elements in the array.
- *returns*: the number of strings printed, or -1 for error.

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

- `length` is less than 1
- `startIndex` or `endIndex` is less than zero or greater than or equal to the array length
- `endIndex` is less than `startIndex`

Your function must not modify `sarray` in any manner.

Examples:

Assume the starting address of `sarray_ex1` is 0x400 and each element in `sarray_ex1` contains the starting address of the specified strings in memory.

```
sarray_ex1 = [ “Stony Brook”, “Computer Science”, “MIPS is amazing!!”,
               “I\nlove\nprogramming”, “FarBeyond”]
```

Code	Return Value	Prints

`printStringArray(sarray_ex1, 0, 2, 4)`	3	`Stony Brook`
		
		`Computer Science`
		
		`MIPS is amazing!!`

`printStringArray(sarray_ex1, 3, 4, 5)`	2	`I`
		`love`
		`programming`
		
		`FarBeyond`

`printStringArray(sarray_ex1, 1, 1, 5)`	1	`Computer Science`
`printStringArray(sarray_ex1, 0, 4, -2)`	-1	
`printStringArray(sarray_ex1, -1, 2, 5)`	-1	
`printStringArray(sarray_ex1, 0, 5, 5)`	-1	
`printStringArray(sarray_ex1, 3, 2, 5)`	-1	

c. `int verifyIPv4Checksum(byte[] header)`

This function verifies that the header of an IPv4 packet was transmitted correctly by adding all half-words of the packet header (the size of header is specified by the `Header length` field) together, *including the checksum field*. If the summation is greater than or equal to 2^{16}, perform end-around-carry. Flip all the bits of the value. If the result is 0, the packet has no errors. If it is non-zero, an error occurred. See Homework #1 if you need a refresher of how an IPv4 packet is structured.

The function DOES NOT modify the bytes in memory.

- `bytes` : starting byte address of the IPv4 header.
- *returns*: 0 if the checksum in the packet is correct. Otherwise, if the checksum in the packet is incorrect, return the calculated checksum.

❗ Your function must not modify `bytes` in any manner.

Examples:

These examples use the sample packets provided in the sample file.

Code	Return Value
`verifyIPv4Checksum(valid_header_ex1)`	`0`
`verifyIPv4Checksum(valid_header_ex2)`	`0`
`verifyIPv4Checksum(invalid_header_ex1)`	`0xff1f`
`verifyIPv4Checksum(invalid_header_ex2)`	`0xe612`

Part 2: Handling a set of IPv4 packets

Each IPv4 packet begins with the IPv4 header followed by the payload (data to be transmitted). In Homework #1, we set the `Total Length` field of the header to denote the entire packet size in bytes, including header and data.

In this part, the goal is to extract the payloads from an ordered array of IPv4 packets to rebuild the correctly transmitted message.

The maximum IPv4 packet length is 65,535 bytes. For this assignment, we will assume that each packet has a maximum size of 60 bytes to keep things small and simple. The IPv4 header is 20 bytes. Therefore, the maximum payload size is 40 bytes (actual size is determined by the `Total Length` field of the header).

A 1D array of IPv4 packets will be stored in the `.data` section of a main file.

d. `(int,int) extractData(Packet[] parray, int n, byte[] msg)`

This function extracts the payload of each packet in `parray` and sequentially writes the payloads into memory starting at the address referenced by the `msg`.

The `Total Length` field of each packet header must be used to determine the size of the packet payload. The `Total Length` field specifies the full length of the packet in bytes. To determine the length of the packet payload, subtract the size of the header (20 bytes) from this value. You do not need the flag or fragment offset fields.

- The first packet's payload is stored starting at `msg[0]`.
- Each subsequent packet's payload is stored starting at the byte after the last byte of the previous packet's payload.

`parray` provides the starting address of the Packet array in memory. Each element of the array is treated as type `Packet`. Therefore, each element of the array is 60 bytes of memory, whether the bytes are used by the packet or not.

It is guaranteed there is enough space allocated in memory to store the full datagram at the `msg` argument.

You may assume all `n` packets are provided **sequentially** in the array.

Upon successful extraction of all payloads into the `msg` array, the function returns (`0`, `M`), where `M` is the total number of bytes that were written by the function starting at `msg`.

If any packet fails checksum verification, the function returns (`-1`, `k`) where `k` is the first index of `parray` whose packet had a checksum error (i.e., `0≤k≤n-1`). Changes to `msg` upon failure are ignored.

Function parameter and return value summary:

- `parray`: starting address of the 1D array of ordered IPv4 Packet(s).
- `n`: number of packets in `parray`.
- `msg`: starting address of the 1D array of byte for the `msg`.
- *returns*: (`0`, `M`) upon success, (`-1`, `k`) upon failure. `M` is the total number of bytes stored in `msg`. `k` is the first array index with a checksum error.

❗ The function must not modify `parray` in any manner.

❗ Your function may not modify `msg` except as required to implement the above specification.

❗ Your function MUST CALL `verifyIPv4Checksum`.

Examples:

Code	Return Value
`extractData(pktArray_ex1, 1, msg_buffer)`	`(0, 32)`
`msg_buffer` is modified to `This is a single packet!\nHello!\n`	
`extractData(pktArray_ex2, 2, msg_buffer)`	`(0, 27)`
`msg_buffer` is modified to `a\nbb\nccc\ndddd\neeeee\nffffff\n`	
`extractData(pktArray_ex3, 4, msg_buffer)`	`(0, 130)`
`msg_buffer` see below table	
`extractData(pktArray_ex4, 4, msg_buffer)`	`(-1, 2)`
`msg_buffer` is ignored due to checksum error.	

```
I’m a shooting star leaping through the sky\nLike a tiger defying the laws of gravity\nI’m a racing car passing by like Lady Godiva\n
```

Once you have extracted the data from the packet, the application typically parses the message for use. To experiment with this type of operation, we will parse the message into strings which can then be printed with our `printStringArray` function. We will explore this functionality in the `processDatagram` function now.

e. `int processDatagram(byte[] msg, int M, String[] sarray)`

This function parses `msg`, replacing any occurrences of `\n` (if any) with `\0` using `replace1st`. The starting address of each null-terminated string created by the function is stored sequentially into `sarray`. The function also writes a `\0` at position `msg[M]`.

- `msg`: starting byte address of the message in memory.
- `M`: total number of bytes stored in `msg`.
- `sarray`: starting address of the array to hold the addresses of ASCII character strings in memory.
- *returns*: The number of string addresses written to `sarray`, or `-1` if `M ≤ 0`

❗ Your function MUST CALL `replace1st`.

❗ Your function may not modify `msg` or `sarray` except as required to implement the above specification.

ℹ It is guaranteed there is enough space allocated in memory to store all string addresses into the `sarray` argument. It is also guaranteed that the length of `msg` it at least `M` so that there is space to write the null-terminator at index `M`.

Examples:

Sample input is provided in `hw2_examples.asm`. Assume the starting address of `msg` is 0x400. `msg` is `a\nbb\nccc\ndddd\neeeee\nffffff\n` for all examples. `msg2` and `msg3` can be found in `hw2_examples.asm` as well.

Code	Return	`abcArray`
`processDatagram(msg,8,abcArray)`	3	[0x400,0x402,0x405]
`msg` is modified to `a\0bb\0ccc\0dddd\neeeee\nffffff\n`		
`processDatagram(msg,3,abcArray)`	2	[0x400,0x402]
`msg` is modified to `a\0b\0\nccc\ndddd\neeeee\nffffff\n`		
`processDatagram(msg,26,abcArray)`	6	[0x400,0x402,0x405,0x409,0x40E,0x414]
`msg` is modified to `a\0bb\0ccc\0dddd\0eeeee\0ffffff\0`		
`processDatagram(msg2,15,abcArray)`	3	[0x400,0x403,0x404]
`msg2` is modified to `hi\0\0howareyou?\0`		
`processDatagram(msg3,10,abcArray)`	1	[0x400]
`msg3` is modified to `helloworld\0`		
`processDatagram(msg,0,abcArray)`	-1	unmodified
`msg` is unmodified.		

Part 3: Putting it all together

Using the functions we have written so far we can build a function `printDatagram` to print out the contents of an array of packets.

f. `int printDatagram(Packet[] parray, int n, byte[] msg, String[] sarray)`

This function takes an array of IPv4 packets and begins by calling `extractData`. If `extractData` returns `0` to indicate that all packets contain valid checksums, `printDatagram` proceeds to print out the contents of the message by calling `processDatagram` and `printStringArray`, in order. Otherwise, if `extractData` returns `-1` to signify an incorrect checksum, `printDatagram` returns `-1`. Likewise, if `processDatagram` returns `-1`, then `printDatagram` returns `-1`.

- `parray`: starting address of a 1D array of ordered IPv4 Packet(s).
- `n`: the number of packets in `parray`.
- `msg`: starting byte address of the message in memory.
- `sarray`: starting address of the array to hold the addresses of ASCII character strings in memory.
- *returns*: `-1` if `(n ≤ 0)` or if any function returns an error; returns `0`, otherwise.

❗ Your function MUST CALL `extractData`, `processDatagram`, and `printStringArray`.

❗ Your function may not modify `parray`, `msg` or `sarray` except as required to implement the above specification.

ℹ It is guaranteed there is enough space allocated in memory to store all strings and the null-terminator in the `sarray` argument.

ℹ It is guaranteed there is enough space allocated in memory to store the full message at the `msg` argument.

Examples:

Code	Return	Prints

`printDatagram(pktArray_ex1, 1, msg_buffer, abcArray)`	0	`This is a single packet!`
		` `
		`Hello!`
		` `

`printDatagram(pktArray_ex2, 1, msg_buffer, abcArray)`	0	`a`
		
		`bb`
		
		`ccc`
		
		`dddd`
		
		`e`

`printDatagram(pktArray_ex3, 1, msg_buffer, abcArray)`	0	`I'm a shooting star leaping through the sky`
		
		`Like a tiger defying the laws of gravity`
		
		`I'm a racing car passing by like Lady Godiva`

`printDatagram(pktArray_ex4, 1, msg_buffer, abcArray)`	-1	

Hand-in instructions

Do not add any miscellaneous printouts, as this will probably make the grading script give you a zero. Please print out the text exactly as it is displayed in the examples, one output line ONLY.

See Sparky Submission Instructions on Piazza for hand-in instructions. There is no tolerance for homework submission via email. They 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.

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.