CSE 220: Systems Fundamentals I

Homework #3

Fall 2017

Assignment Due: November 10, 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!

🛈 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.

⚠ You personally must implement the assignment in MIPS Assembly language by yourself. You may not 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 assignment. You may also not write a code generator in MIPS Assembly that generates MIPS Assembly.

⚠ All test cases MUST execute in 100,000 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 solidifying your understanding of MIPS register conventions and writing more complex functions. We will build upon homework 2 and write a recursive function.

You **MUST** implement all the functions in the assignment as defined. It is OK to implement additional helper functions of your own in `hw3.asm`.

⚠ **You MUST follow the MIPS calling and register conventions appropriately (i.e brute-forcing by saving all registers does not count). If you do not, you WILL lose points.**

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

You can use the files you already have for hw2, but you **MUST** copy all functions from your `hw2.asm` file into `hw3.asm`. Don't forget to change the comments at the top of your asm file appropriately.

You can get additional test files from `hw3.zip` which is available on Piazza in the homework section of Resources.

You will only submit your `hw3.asm` file via Sparky. Make sure that all code required for implementing your functions (`.text` and `.data`) are included in the `hw3.asm` file! To make sure that your code is self-contained, try assembling your `hw3.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 `hw3.asm`.

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

Part 1: Correct your HW2 functions

- Update the four functions listed below you wrote for the the previous homework so that they follow proper register conventions. Also see the note about an update to `verifyIPv4Checksum` that you may need to make in your code. If you didn't complete or get a perfect score in the previous homework, you can use the grading tests to help debug your existing code.

  The following functions are needed for Part 2 of this assignment:

  - `replace1st`
  - `printStringArray`
  - `processDatagram`

    ◦ `verifyIPv4Checksum`

    Your implementation MUST work with **ANY** `Header length` value in the range of [5,15].

    ℹ We realize that "real" IPv4 packets do not have headers of this length, but there is no reason why our checksum verifier can't be generalized to do so.

- We will check your register conventions AGAIN! Make sure ALL functions for this homework follow register conventions. Refer to the lecture notes or ask your TA to review in recitation or office hours if you are still unclear about the conventions.

Part 2: Handling an unordered set of IPv4 packets

In homework 1, we set the `Flag` and `Fragment Offset` fields of an IPv4 packet to specify if the packet was the first packet, the only packet, or a member of a sequence of packets. In homework 2, we extracted data from an ordered sequence of packets stored in an array and ignored these fields.

In this homework, we will create a function which will perform similarly to the homework 2's `extractData` function. It will handle larger packet lengths, perform checks on the array of packets to ensure all packets were received, and account for the packets being out of order in the array.

The goal is to extract the payloads from an **unordered array** of generic IPv4 packets to rebuild the correctly transmitted message.

We will assume that:

- Each packet has a set maximum size of `packetentrysize` bytes, instead of 60 bytes as in homework 2. Therefore, the MAXIMUM payload size of a packet is `packetentrysize - (4* header length field )` bytes. the actual paload size of a packet must be computed using the `Total Length` and `Header Length` fields.

- Each packet has a variable length header in the range of [5,15] (as indicated by the `Header length` field in the packet).

Implement the following new function:

a. `(int,int) extractUnorderedData(Packet[] parray, int n, byte[] msg, int packetentrysize)`

- `parray`: starting address of the 1D array of unordered IPv4 packet(s).
- `n`: number of packets in `parray`.
- `msg`: starting address of the 1D array of bytes for the `msg`.
- `packetentrysize`: the number of bytes for each `parray` entry.
- *returns*: (`0`, `M+1`) upon success, (`-1`, `k`) upon failure. All error cases are described below.

`parray` represents 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 `packetentrysize` bytes of memory.

This function validates `parray` for the following properties:

- each packet in `parray` was transmitted correctly using `verifyIPv4Checksum`
- if `n>1`, `parray` contains a SINGLE beginning packet and `parray` contains a SINGLE last fragment packet
- if `n==1`, the single packet in `parray` is not fragmented. A packet is not fragmented if:
    - `Flags` equals `010`, or
    - `Flags` equals `000` with a `Fragment Offset` of 0.

The function is not responsible for checking that the combined payloads build a contiguous data block into `msg`.

If `parray` is valid, then the payload of each packet is extracted and placed into the memory space referenced by the `msg` argument (using the proper `Fragment Offset`). The function uses the `Total Length`, `Flags` and `Fragment Offset` field of each packet header to determine the size and position to copy the packet payload into `msg`.

The `Total Length` field specifies the full length of the packet. Note, this is different from `packetentrysize` which is the max possible `Total Length` value. To determine the length of the packet payload, subtract the size of the header (4*`Header Length`) from this value.

Use `Flags` to determine how to store each packet payload into `msg`.

- If `Flags` equals `100` and `Fragment Offset` equals 0, the packet is the beginning packet. Store the payload into `msg[0]`.

- If `Flags` equals `100` and `Fragment Offset` does not equal 0, the packet is an intermediate packet. Store the payload into `msg[Fragment Offset]`.
- If `Flags` equals `000` and `Fragment Offset` does not equal 0, the packet is the last fragment packet. Store the payload into `msg[Fragment Offset]`.
- If `Flags` equals `000` and `Fragment Offset` equals 0, the packet is the ONLY packet. Store the payload into `msg[0]`.
- If `Flags` equals `010`, this means the message is not fragmented. Ignore the `Fragment Offset` field. Store the payload into `msg[0]`.

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

ℹ You may assume `n` packets are provided in the array and all fragment offsets are unique.

ℹ It is NOT an error if an intermediate packet is missing from the array of packets. If this happens, there will be "holes" in `msg`, where sections of the message will be missing.

Upon successful extraction of all payloads into the `msg` array, the function returns (`0`, `M+1`), where `M` is the index of the last (highest index) byte stored in `msg`.

The function returns (`-1`, `k`), where `k` is the array index of the first error, if:

- `Total Length` for a packet $>$ `packetentrysize` for packet `k`
- checksum verification fails for packet `k`

The function returns (`-1`, `-1`) if:

- `n < 1`
- `n == 1` and the packet in `parray` has `Flags` equals `100`
- `n == 1` and the packet in `parray` has `Flags` equals `000` and `Fragment Offset` does not equal 0
- `n != 1` and any packet in `parray` has `Flags` equals `010`
- `n > 1` and the `parray` does not have exactly one beginning packet (`Flags` equals `100` and `Fragment Offset` equals 0)
- `n > 1` and the `parray` does not have exactly one last fragment packet (`Flags` equals `000` and `Fragment Offset` does not equal 0)

❗ The function DOES NOT modify the Packets in `parray`.

❗ Your function MUST CALL `verifyIPv4Checksum`.

Examples:

See `hw3_examples.asm` for packet arrays and comments about each example.

Code	Return Value
`extractUnorderedData(pm_checksum,1,msg_buffer,200)`	(-1, 0)
`extractUnorderedData(pm_flagerr,1,msg_buffer,200)`	(-1,-1)
`extractUnorderedData(pm_3pkterr,3,msg_buffer,200)`	(-1,-1)
`extractUnorderedData(pm_1pkt,1,msg_buffer,200)`	(0,7)
`extractUnorderedData(queen_all,5,msg_buffer,80)`	(0,153)
`extractUnorderedData(queen_all_unsorted,5,msg_buffer,80)`	(0,153)
`extractUnorderedData(marypoppins,4,msg_buffer,100)`	(0,190)
`extractUnorderedData(marypoppins_unsorted,4,msg_buffer,100)`	(0,190)
`extractUnorderedData(queen_holes,4,msg_buffer,80)`	(0,153)
`extractUnorderedData(queen_holes_unsorted,4,msg_buffer,80)`	(0,153)

Part 3: Create new printUnorderedDatagram Function

Implement the following new function:

b. `int printUnorderedDatagram(Packet[] parray, int n, byte[] msg,`
`String[] sarray, int packetentrysize)`

This function is identical to `printDatagram` from homework 2, except that you replace the call to `extractData` with `extractUnorderdData`.

In addition, this function takes the `packetentrysize` as the fifth argument. As discussed in lecture and recitation, this argument is stored on the stack by the caller function and accessed by `printUnorderedDatagram` from the stack. It is the responsibility of the caller function to return the space to the stack after the function call.

Part 4: Recursive Function

Although most recursive algorithms tend to look simple in a high-level language, they can be tricky to write at an assembly level. In this part you will be given the code for a recursive version of an Edit Distance algorithm in Java and will implement it in MIPS.

The Edit Distance algorithm calculates the minimal number of steps needed to be taken in order to transform one String into another String over a given length for each.

c. `int editDistance(String str1, String str2, int m, int n)`

This function calculates the edit distance between two Strings `str1` and `str2`, given lengths `m` and `n`.

- `str1`: starting address of the first string.
- `str2`: starting address of the second string.
- `m`: length of `str1`.
- `n`: length of `str2`.
- *returns*: the number of edits to transform `str1` into `str2`. -1 if `m` or `n` is less than 0.

Below is the algorithm that you **MUST** follow to implement this function:

```
int editDistance(String str1 , String str2 , int m ,int n) {
  if(m < 0 || n < 0) return -1;

  System.out.print("m:" + m + ",n:" + n + "\n");

  // If first string is empty, just insert n
  if (m == 0) return n;

  // If second string is empty, just remove m
  if (n == 0) return m;

  // If last chars of the two strings are the same ignore
  // the last character of each
  if (str1.charAt(m-1) == str2.charAt(n-1))
    return editDistance(str1, str2, m-1, n-1);

  // If last characters are not same, consider all three
  // operations on last character of first string. Recursively
  // compute minimum cost for all three operations and take the
  // minimum of the three values.
  int insert = editDistance(str1, str2, m, n-1);
  int remove = editDistance(str1, str2, m-1, n);
  int replace = editDistance(str1, str2, m-1, n-1);

  //You may implement the min of 3 values in any way you want
```

```
    return 1 + Math.min(insert, Math.min(remove, replace));
}
```

❗ We know dynamic programming is a more efficient solution. The goal of this exercise is to test your register conventions and stack management.

❗ We will grade your recursive output to ensure that you implemented the function recursively.

ℹ We will ensure all test can be completed within the stated instruction limit.

Examples:

See `hw3_example_outputs.txt` for outputs for each example.

Code	Return Value
`editDistance("Bunny", "Funny", 5, 5)`	1
`editDistance("SuN", "Mon", 3, 3)`	3
`editDistance("Cat", "Toad", 3, 4)`	3
`editDistance("1234567890", "1", 10, 1)`	9
`editDistance("", "1", 0, 1)`	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.