Homework 5

Learning Outcomes

After completion of this assignment, you should be able to:

- Dynamically allocate memory.
- Manage allocated memory using linked lists.

Getting Started

To complete this homework assignment, you will need the MARS simulator. Download it from Blackboard. You can write your programs in the MARS editor itself. You can choose to use other text editors if you are not comfortable with the MARS editor. At any point, if you need to refer to instructions click on the *Help* tab in the MARS simulator.

Read the rest of the document carefully. This document describes everything that you will need to correctly implement the homework and submit the code for testing.

You should have already setup Git and configured it to work with SSH. If you haven't then do Homework 0 first!

The first thing you need to do is download or clone this repository to your local system. Use the following command:

`$ git clone <ssh-link>`

After you clone, you will see a directory of the form *cse220-hw5-username*, where *username* is your GitHub username.

In this directory, you will find *hw5.asm*. This file has function stubs that you will need to fill up. At the top of the file you will find hints to fill your full name, NetID, and SBU ID. Please fill them up accurately. This information will be used to collect your scores from GitHub. If you do not provide this information, your submission may not be graded. The directory also has template test files ending with *test.asm*. Use the files for testing. You can change the data section or the text section in these files to test different cases for each part (described later). You may also create your own *_test.asm* files if necessary. Don't push these additional *_test.asm* files to the repository.

Note the hw5.asm file doesn't have a .data section. Do not add a .data section.

Don't forget to add you name and IDs at the top of hw5.asm. Follow the exact format, that is, replace the hints with the correct information. You will be penalized if you do not follow the format.

Assembling and Running Your Program in MARS

To execute your MIPS programs in MARS, you will first have to assemble the program. Click on the *assemble* option in the *Run* tab at the top of the editor. If the instructions in your program are correctly specified, the MARS assembler will load the program into memory. You can then run the program by selecting the *Go* option in the same *Run* tab. To debug your program, add breakpoints. This is done after assembling the program. Select the *execute* tab, you will see the instructions in your program. Each instruction will have a checkbox associated with it. Clicking on the checkbox will add a breakpoint, that is, when the program is run, control will stop at that instruction allowing you to inspect the registers and memory up to that point. The execute tab will show you the memory layout in the bottom pane. The right hand pane shows the list of registers and their values.

Always assume that memory and registers will have garbage data. When using memory or registers, it is your responsibility to initialize it correctly before using it. You can enable the *Garbage Data* option under *Settings* in MARS to run your programs with garbage data in memory.

Testing

Unlike previous homework assignments, in this homework assignment, we do not provide any executable test cases. You can take inspiration from previous homeworks and the specifications (described later) to first come up with a comprehensive set of test cases and then begin writing the code for the implementation. This approach will help you systematically test your code. To test your implementation, we have provided several *_test.asm* files, a test file for each part. Write your test cases in those files. You are allowed to add additional test files if necessary. However, your test files will be ignored during grading and the grading scripts will use a different set of tests. It is your responsibility to test for all possible scenarios w.r.t the specifications defined later.

Dynamic Memory Allocation

In this homework assignment, we will need to allocate memory in the heap. To store data in the system heap, MARS provides system call 9, which is called sbrk. For example, to allocate N bytes of memory, where N is a positive integer literal, we would write this code:

```
li $a0, N
li $v0, 9
syscall
```

When the system call completes, the address of the newly-allocated memory buffer will be available in `$v0`. The address will be on a word-aligned boundary, regardless of the value passed through $a0. Unfortunately, there is no way in MARS to de-allocate memory to avoid creating memory leaks. The run-time systems of Java, Python and some other languages take care of freeing unneeded memory blocks with garbage collection, but assembly languages, C/C++, and other languages put the responsibility on the programmer to manage memory. You will learn more about this in CSE 320.

Problem Specification

In this assignment, we will develop a basic polynomial calculator that can do addition and multiplication of polynomials.

A single variable polynomial is an algebraic expression of the form: $P(x) = a_n x^{e_n} + a_{n-1} x^{e_{n-1}} + ... + a_1 x^{e_1}$, where $a_i x^{e_i}$ is a term in the polynomial so that a_i is a non-zero coefficient and e_i is a positive exponent. We will assume an ordering of the terms in the polynomial such that $e_n > e_{n-1} > ... > e_2 > e_1$.

An application of linked lists is to represent polynomials and their manipulations. The primary advantage of this representation is that it can accommodate polynomials of growing sizes. One way to represent a polynomial as linked lists is to think of the terms in the polynomial as nodes in a linked list. Hence, a term can be represented as **<coeff, exp, link>**, where **coeff** is the coefficient and **exp** is the exponent of a term, and **link** is a pointer to the next term. An additional data block called the head term is used to represent the polynomial itself. Hence, a polynomial $P(x) = 3x^8 - 7x^6 + 14x^3 + 10x - 5$ is represented as:

P_x_ptr --> <3,8> --> <-7,6> --> <14,3> --> <10,1> --> <-5,0> --> NULL.

Our goal in this assignment is to implement functions that will implement polynomials as linked lists and perform basic operations on them such as addition and multiplication. But before that, we define the data structures that we will use for this purpose. To this end, we will make use of the following data structure:

```
struct Term {
   int coeff          // coefficient of a term (4 bytes)
   int exp                    // exponent of a term (4 bytes)
   Term* next_term    // Pointer to the next term in the polynomial //(4 bytes)
}
```

A Term with *next_term* NULL is assumed to be the last term in a polynomial.

The polynomial itself is a simple structure that holds a pointer to the first term in the polynomial.

```
 struct Polynomial {
   Term* head_term    // (4 bytes)
   int no_of_terms  // (4 bytes)
}
```

Implement the functions defined in the following parts to build a basic polynomial arithmetic calculator in MIPS.

Note whenever we use the term pointer to X, we mean the address of X. We will use both terms interchangeable in this README.

Part 1: Create A Term

Term* create_term(int coeff, int exp)

The *create_term* function initializes a single-variable polynomial term with the inputs coefficient (coeff) and exponent (exp), both of which are signed integers. It allocates 12 bytes of memory in the heap to hold the term structure. The structure is initialized as follows: The first 4 bytes holds the coefficient, The next 4 bytes holds the exponent, and The final 4 bytes holds the address of another term. Set this to 0 at this point. You must use *syscall sbrk* to allocate 12 bytes for the term.

The function returns the address of the term in `$v0`. If the coefficient is 0 or the exponent is negative, then return -1 in `$v0`.

Part 2: Create A Polynomial

Polynomial* create_polynomial(int[] terms)

The function takes as argument an array of pairs (coefficient, exponent). Assume that the array will always be terminated by the pair (0,-1). The array of pairs may contain pairs with the same exponent. In such cases, terms are merged by adding the coefficients creating one term with that exponent. The position of the merged term in the polynomial must be the same as that of the first occurrence of the exponent in the array.

The function must create a polynomial structure (defined above) with a pointer to the linked list of terms and the no. of terms in the linked list. It must return a pointer to the structure in `$v0`.

If the *terms* array is empty then return NULL in *$v0*.

Sample Test Cases:

1. **create_polynomial([(5,1), (2,7), (3,2), (0,-1)])** returns in `$v0` a pointer to a polynomial with 3 terms and *head_term* pointing to the linked list *Term(5,1) --> Term(2,7) --> Term(3,2) --> NULL*.
2. **create_polynomial([(2,2), (5,1), (3,2), (2,7), (0,-1)])** returns in `$v0` a pointer to a polynomial with 3 terms and *head_term* pointing to the linked list *Term(5,2) --> Term(5,1) --> Term(2,7) --> NULL*.
3. **create_polynomial([(5,1),(3,2),(2,7),(5,1),(0,-1)])** should return in `$v0` a pointer to a polynomial with 3 terms and *head_term* pointing to the linked list *Term(10,1) --> Term(3,2) --> Term(2,7) --> NULL*.
4. **create_polynomial([(5,1),(3,2),(2,7),(5,1),(4,7),(4,4),(0,-1)])** should return in `$v0` a pointer to a polynomial with 4 terms and *head_term* pointing to the linked list *Term(10,1) --> Term(3,2) --> Term(6,7) --> Term(4,4) --> NULL*.

Part 3: Sort Polynomial

void sort_polynomial(Polynomial* p)

The function takes as argument a polynomial structure and sorts it in descending order of exponents. Assume that the polynomial has only unique exponents and every term is valid as defined in part 1. The function returns nothing. Note the function changes the polynomial passed as argument.

Sample Test Cases:

1. Suppose `p.head_term --> Term(2,3) --> Term(3,1) --> Term(14,5) --> Term(7,0) --> NULL` and `p.no_of_terms = 4`.

The function changes *p* such that `p.head_term --> Term(14,5) --> Term(2,3) --> Term(3,1) --> Term(7,0) --> NULL` and `p.no_of_terms = 4`.

2. Suppose `p.head_term --> Term(3,2) --> Term(1,1) --> Term(-3,0) --> NULL`.

The function changes *p* such that `p.head_term --> Term(3,2) --> Term(1,1) --> Term(-3,0) --> NULL` and `p.no_of_terms = 3`.

Part 4: Add Polynomials

Polynomial* add_polynomial(Polynomial* p, Polynomial* q)

The function takes two polynomial structure pointers as arguments adds them and returns a pointer to a new polynomial structure that holds the result of the adding the input polynomials. The input polynomials may not be sorted by decreasing order of exponents. The rules of adding polynomial p and q are as follows:

- If a term in `p` and `q` have exponent `e` and coefficients `c1` and `c2`, then the resulting polynomial should have a term with exponent e and coefficient `(c1 + c2)`.
- If a term with exponent `e` and coefficient `c` exists in `p` but not in `q`, then the resulting polynomial should have a term with exponent `e` and coefficient `c`.
- If a term with exponent `e` and coefficient `c` exists in `q` but not in `p`, then the resulting polynomial should have a term with exponent `e` and coefficient `c`.

The function returns a pointer to the result polynomial structure in **$v0**. If either argument is NULL, then the function returns a pointer to the non-null polynomial. If both arguments are NULL, the function returns a pointer to an empty polynomial. An empty polynomial has it's head term set to NULL and the no. of terms set to 0.

The resulting polynomial must always be sorted by their exponents in decreasing order.

Sample Test Cases:

1. Suppose, `p.head_term --> Term(2,3) --> Term(3,1) --> NULL` and `q.head_term --> Term(4,2) --> Term(7,0) --> NULL`

`add_polynomial(*p,*q)` returns `*r`, where `r.head_term --> Term(2,3) --> Term(4,2) --> Term(3,1) --> Term(7,0) --> NULL` and `r.no_of_terms = 4`.

2. Suppose, `p.head_term --> Term(2,3) --> Term(3,1) --> NULL` and `q.head_term --> Term(4,2) --> Term(7,1) --> NULL`

`add_polynomial(*p,*q)` returns `*r`, where `r.head_term --> Term(2,3) --> Term(4,2) --> Term(10,1) --> NULL` and `r.no_of_terms = 3`.

3. Suppose, `p.head_term --> Term(2,3) --> Term(3,1) --> NULL` and `q.head_term --> Term(-2,3) --> Term(7,1) --> NULL`

`add_polynomial(*p,*q)` returns `*r`, where `r.head_term --> Term(10,1) --> NULL` and `r.no_of_terms = 1`.

4. Suppose, `p.head_term --> Term(2,3) --> Term(-7,1) --> NULL` and `q.head_term --> Term(-2,3) --> Term(7,1) --> NULL`

`add_polynomial(*p,*q)` returns `*r`, where `r.head_term --> NULL` and `r.no_of_terms = 0`.

Part 5: Multiply Polynomials

Polynomial* mult_polynomial(Polynomial* p, Polynomial* q)

The function takes two polynomial structure pointers as arguments, multiplies them, and returns a pointer to a new polynomial structure pointer with the product in **$v0**. The input polynomials may not be sorted. Assume that the exponent and coefficient for each term in the resulting polynomial will be 32 bits. Polynomials are multiplied as follows:

1. Each term in `p` is multiplied with each term in `q`. The resulting terms are added to produce the result polynomial.
2. When a term with coefficient `c1` and exponent `e1` is multiplied with a term with coefficient `c2` and exponent `e2`, the result is a term with coefficient `(c1*c2)` and exponent `(e1+e2)`.
3. When a term is multiplied with another term and NO term with the resulting exponent exists in the polynomial then a new term with the resulting exponent and coefficent is created.
4. When two terms are multiplied to produce a term with exponent `e` and coefficient `c1` and a term with exponent `e` exists in the polynomial then the existing term with exponent `e` should be updated with the coefficent `c1+c2`.
5. If multiplying two terms results in a coefficent `c` and a term with the coefficient `-c` exists in the polynomial or vice-versa, then the term should be canceled and not be part of the polynomial.

The function returns a pointer to an empty polynomial if either of the polynomials is NULL. If all terms cancel each other out as a result of the multiplication then also the function returns a pointer to an empty polynomial.

The resulting polynomial must always be sorted by descending order of exponents.

Sample Test Cases:

1. Suppose, `p.head_term --> Term(2,3) --> Term(3,1) --> NULL` and `q.head_term --> Term(4,2) --> Term(7,0) --> NULL`

`mult_polynomial(*p,*q)` returns `*r`, where `r.head_term --> Term(8,5) --> Term(26,3) --> Term(21,1) --> NULL` and `r.no_of_terms = 3`.

2. Suppose, `p.head_term --> Term(2,3) --> Term(3,1) --> NULL` and `q.head_term --> Term(4,2) --> Term(7,1) --> NULL`

`mult_polynomial(*p,*q)` returns `*r`, where `r.head_term --> Term(8,5) --> Term(14,4) --> Term(12,3) --> Term(21,2) --> NULL` and `r.no_of_terms = 4`.

3. Suppose, `p.head_term --> Term(2,3) --> Term(7,1) --> NULL` and `q.head_term --> Term(-2,3) --> Term(7,1) --> NULL`

`mult_polynomial(*p,*q)` returns `*r`, where `r.head_term --> Term(-4,6) --> Term(49,2) --> NULL` and `r.no_of_terms = 2`.

4. Suppose, `p.head_term --> Term(2,3) --> Term(-7,1) --> NULL` and `q.head_term --> Term(-2,3) --> Term(7,1) --> NULL`

`mult_polynomial(*p,*q)` returns `*r`, where `r.head_term --> Term(-4,6) --> Term(28,4) --> Term(-49,2)` and `r.no_of_terms = 3`.

Submitting Code to GitHub

You can submit code to your GitHub repository as many times as you want till the deadline. After the deadline, any code you try to submit will be rejected. To submit a file to the remote repository, you first need to add it to the local git repository in your system, that is, directory where you cloned the remote repository initially. Use following commands from your terminal:

`$ cd /path/to/cse220-hw5-<username>` (skip if you are already in this directory)

`$ git add hw5.asm`

To submit your work to the remote GitHub repository, you will need to commit the file (with a message) and push the file to the repository. Use the following commands:

`$ git commit -m "<your-custom-message>"`

`$ git push`

After you submit your code on GitHub. Enter your GitHub username in the Blackboard homework assignment and click on Submit. This will help us find your submission on GitHub.