**Laboratory 4**

**Data Structures Representation**

*Computer Science 240*

We have seen how primitive data like integers, floating point numbers, and characters are stored in memory (as contiguous numerical values, accessed by location/address in memory). How do we represent more complicated data/data structures in memory at the machine level? Different high-level languages may use different representations. You will learn more about this in lecture, and will experiment with some of these representations in lab today.

**One-dimensional arrays of numerical values**

*Exercise 1:* Let’s begin with perhaps the simplest data structure, a one-dimensional array (similar to the one you worked with on the assignment). When all the elements of an array are the same size (such as an array of byte values), the elements of the array can be stored in their indexed order in memory, and accessed using:

**base address of array + (size of element in bytes \* index)**

Here is the MIPS code to define an array, and a main program to invoke a procedure getElement, which takes as parameters the base address and the index of an array element, and returns the value of the specified array element:

.data

elements: .word 7 # number of elements in the array

.word 1 # size of element in the array

.byte 1,5,19,22,4,7,3

prompt: .asciiz ‘Enter an array index: ‘

result: .asciiz ‘The value of the array element is: ‘

.text

.globl main

main: li $v0,4 #prompt for an index

la $a0,prompt

syscall

li $v0,5 # read in the index and store in $a0

syscall

move $a0,$v0

la $a1,elements # put the base address of array in $a1

jal getElement # read in the value

move $t0,$v0 # move returned value to $t0

li $v0,4 # output the result string

la $a0,result

syscall

move $a0,$t0 # output the result

li $v0,1

syscall

endmain: li $v0,10

syscall

1. What registers are used for the parameters to the procedure?

2. In what register is the element value returned?

3. Add the procedure *getElement* to the above code (you can copy and paste the starting code into MARS, but you may have to re-type the quotes used in the strings).

Your code should assume that the size of the elements in the array is 1 byte. Test by using various values in the range 0 – 6 for the index, and verify that you get the correct values.

Paste the code for your procedure here:

4. Add some new array declarations to your data segment, including an array of halfwords and one of words. Modify your main program so that it calls the procedure with the correct parameters to access your new arrays, and test to be sure the procedure works for different size elements. Demonstrate to the instructor

Paste the new array declarations here:

5. Write a procedure *printArray*, which will print out all the elements of an array, by using the *getElement*  procedure. For the byte array *elements* defined above, it should print (include the square brackets and commas):

**[1,5,19,22,4,7,3]**

**One-dimensional arrays of strings**

*Exercise 2:* On the lab assignment for today, the strings in the array were all of equal length. This made it easy to access the elements of the array in the same way you did for the first exercise (base address + size \* index). What if the strings are of variable length? There are a variety of techniques which might be used to represent the data in memory.

Assume you have declared the following array of strings in Java:

**String[] words =** **{‘I’,’do’,’not’,’like’,’green’,’eggs’,’and’,’ham’}**

2. Show the data declarations you would use to define this array using:

Variable-length strings prefixed by bytes describing length, stored contiguously in memory:

Variable-length null-terminated strings stored (not necessarily contiguously) in memory, where the array contains pointers to (addresses of) the strings:

3. For the first definition above, implement a procedure *getAddressOf,* which takes as parameters the base address of the array and theindex, and returns the address of the string at the given index. To test your procedure, examine the returned value using MARS, and verify that it is the correct address of the string element you are accessing.

Paste your code for *getAddressOf* here:

Notice how much harder it is to index elements of variable size! This strategy is not actually used as a construct in higher-level languages. It can be coded in a language like C, but it does not really fit C’s array support.

The second strategy (array of pointers to strings) is usually used in higher-level languages, and would be used to define memory for the Java declaration given above.

4. Implement a new version of the procedure *getAddressOf,* assuming the array elements are addresses of strings. Use the address returned by the procedure to print the null-terminated string stored at the address which is returned from the procedure. Demonstrate to the instructor.

Paste your second version of *getAddressOf* here:

**Two-dimensional arrays**

*Exercise 3:* While Java allows two-dimensional arrays only as arrays of addresses of arrays (much like the arrays of addresses of strings from the last exercise), in C, nested array of arrays are used, where each row is stored contiguously in memory (*row-major* format), and the address of an element can be calculated by the following formula:

**address of element[x][y] =**

**base address of array +**

**(x \* number of columns \* size of element) +**

**(y \* size of element)**

**-or-**

**base address of array +**

**(x\*columns + y)\*size of element**

Assume that the size of the elements in bytes will be 1, 2, or 4. This will not always be true in real-life data structures, but it makes the calculation more efficient here. Why?

1. Add the procedure *getElement* to the following code, which contains a declaration for a 4x4 array of integers, and some test code to allow the user to enter the indices to access an element of the array.

.data

twodi: .word 4 #size in bytes of each element

.word 4 #number of rows

.word 4 #number of columns

.word 1,3,5,7

.word 2,4,6,8

.word 9,11,13,15

.word 10,12,14,16

prompt1: .asciiz "\nTo access an element A[x][y], enter x: "

prompt2: .asciiz " also enter y: "

.text

.globl main

main: li $v0,4 #prompt for an index

la $a0,prompt1

syscall

li $v0,5 # read in the row index and store in $a0

syscall

move $a0,$v0

li $v0,4 #prompt for an index

la $a0,prompt2

syscall

li $v0,5 # read in the column index and store in $a1

syscall

move $a1,$v0

la $a0,twodi # put the base address of the array in $a0

jal getElement

move $a0,$v0 # move the value of the element from $v0 to $a0 for printing

li $v0,1 # print the result

syscall

li $v0,10 # exit

syscall

Paste your code for *getElement* here:

2. Write a procedure *sumAll* which uses a nested loop to iterate and sum all the elements. Demonstrate to the instructor.

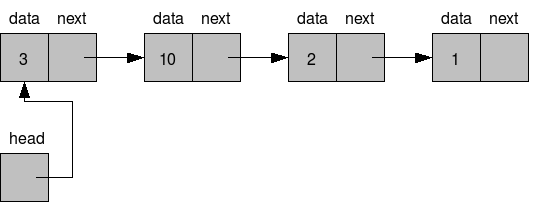
Paste your code for *sumAll* here:

3. Write a second version of sumAll, using a single loop ( exploit contiguous row layout and calculate total array size to determine loop bound).

Paste your code for the second version of *sumAll* here:

**Linked Lists**

*Exercise 4:* Linked lists are represented by sequences of *nodes* or *elements*, each containing a value in the list, as well as a reference to the *next* node in the list following this one.  The first node in the list is called the *head* of the list:



How are dynamically allocated data structure such as linked lists implemented at the machine level?

In MIPS, allocation of the heap (the area of memory used for dynamic data storage, which starts at address 0x10040000) is done via sbrk (syscall 9), $a0 = number of bytes to allocate, $v0 = address of word-aligned block of at least $a0 bytes.

In our implementation for lab, a list node is a word-aligned 8-byte memory block that holds:

- at offset 0: the address of the next node in the list

- at offset 4: an integer element

The actual memory layout in MIPS for this structure might look something like this:

(head) 0x10040000: .word 0x10040008

.word 00 (not used, head node has no value)

.

.

.

(node1) 0x10040008: .word 0x10040030 (address of next node)

.word 03 (value)

.

.

.

(node2) 0x10040030: .word 10040084 (address of next node)

.word 10 (value)

.

.

.

(node3) 0x10040084: .word 10040220 (address of next node)

.word 2 (value)

.

.

.

(node4) 0x10040220: .word 10040300 (address of next node)

.word 1 (value)

1. Download the two files for use in this exercise from the CS240 Lab Google Group, and examine the code.

***LinkedList.java*** defines a simple ListNode representation as well as methods to manipulate lists built of ListNodes  (note that these methods are written to emulate C and avoid more complicated features of Java’s object orientation). This code includes:

***add(ListNode head, int value)*** *//add value to the end of the list starting with node head*

***insertIterative(ListNode head, int index, int value)*** //insert value at position index in the list starting with node head

***show(ListNode head)*** // print the elements of a list in order, separated by spaces.

A recursive version of insert is also included in the code.

***LinkedList.asm*** is a translation of the methods of LinkedList.java to MIPS code, using a simple memory layout to represent ListNode objects, described in comments in LinkedList.asm. The MIPS implementations of *add(head, value)* and *show(head)* are provided.

2. Implement the insert operation by translating LinkedList.java’s insertIterative method to MIPS.

Paste your code for *insertIterative* here:

As you learn about the ListNode representation and design and implement your insertIterative procedure, consider the following questions.

Why must add and insertIterative allocate new ListNode objects on the heap?  What could happen if these methods/procedures allocated space for ListNodes on the stack?

More generally, how long is data allocated on the stack valid?

3. Implement the same sequence of adds and inserts in the MIPS main program as given in the Java main program:

***head = add(head, 2);***

***head = insertIterative(head, 0, 0);***

***head = insertIterative(head, 2, 8);***

***head = insertIterative(head, 2, 4);***

***head = insertIterative(head, 3, 6);***

***head = add(head, 10);***

***show(head);***

// Should print "0 2 4 6 8 10"

Copy and paste the heap memory display from MARS showing the memory layout after this sequence of operations.