Lab Assignment 5: Memory Management and Static Control Flow

Due Oct 12, 2018 by 11:59pm **Points** 100 **Available** after Oct 6, 2018 at 12pm

Goal

This assignment is divided into four parts. In the first part, you will get additional practice reading assembly code, figuring out what C program it came from, and counting memory references.

Second, you will examine dynamic allocation and deallocation in C. The goal here is to help you clarify your understanding of this topic, particularly dangling pointers, memory leaks, and how to avoid them. You will do this by examining a program that contain dangling pointer bugs to identify the bugs and fix them.

Third, you will extend the SM213 implementation to add the instructions we have discussed in class that support static control flow, including static procedure calls and procedure return (which is dynamic, actually).

Finally, you will write a fairly substantial assembly-language program that uses all of the language concepts we have discussed so far.

All provided code is given in **a5code.zip**. It contains the following files:

- `q1.c` and `q1.s`: used in question 1;
- `stack.c`: used in question 2;
- `Makefile`: allows for easier compilation of the stack program in question 2;
- `CPU.java`: a starting point to extend the ISA in question 3.

Question 1 — Reading Assembly Code [10%]

Read the included `q1.c` and `q1.s` files. You will see that `q1.c` declares two structs, allocates some objects from those structs and initializes them. It has a procedure named `q1()` that is blank. The code listed in q1.s implements the code in this procedure (without the procedure call itself). Read `q1.s` carefully and run it through the simulator to figure out how it manipulates these structs. Then do the following.

1. [2%] Comment every code line of `q1.s` with a high-level comment (i.e, a C-like comment).
2. [6%] Modify the procedure `q1()` in `q1.c` to perform the same computation as `q1.s`. Do not modify the other parts of this C file; they are used to test and mark your code. Note that you can optionally provide different values for the structs on the command line. The C code you write must work for arbitrary values.
3. [2%] Examining the code you write for `q1()` in `q1`. Count the minimum number of memory reads and writes that are required to execute these five lines. Note these numbers may be different from `q1.s`, which takes each line individually (e.g., reading the value of `i` each time, which is not really necessary). Ignore register allocation for this questions; i.e., you can assume that you have an infinite number of registers available.

   Place the number of reads and the number of writes, in this order, in the file `q1.txt`, on two separate lines (just these numbers and nothing else). Then carefully explain your answer by listing each of the reads and writes that are required (give line number and describe the access using variable names) in the file `q1-desc.txt`.

Question 2 – Dynamic Allocation (Part 1) [25%]

The file `stack.c` contains a program that implements a stack data structure and tests it. A stack has two operations: push and pop. Push adds strings to the stack and pop removes them, in last-in-first-out order. So, if you push “one”, “two”, and then “three”, in that order, three pops will give you “three”, “two”, “one”, in that order. You can use the included Makefile to build the program by typing

```
make stack
```

This will produce an executable file named stack that you can run by typing:

```
./stack
```

This program tests the stack by pushing two elements, popping one, pushing two more, popping all three, and then printing the four popped values in the order they were popped. The values pushed are “A”, “B”, “C”, and “D”, in this order and so the output should be:

```
B D C A
```

But, as you will see the program does not produce this output. It has a bug. Find the bug and write a careful description of it in the file `q2.txt`. The bug is a dangling pointer somewhere. If you can’t find it, you might try commenting out calls to `free()`. This should fix the dangling pointer bug by replacing it with a memory leak.

Once you have found and described the bug, you need to fix it. Update the file `stack.c` so that it has neither dangling pointers nor memory leaks. Your fix should change the existing program as minimally as possible. In particularly, the program must still call `malloc` where it does and use this dynamically allocated memory as it does.

Test your program by running it to verify that you get the correct output. If you do, this is a good indication that you have fixed the dangling pointer. But you are not done. You must also ensure that your program does not have a memory leak. To do so, you must run the program through `valgrind`, which is available on the undergrad servers, though it may not be available on other platforms you might use for other parts of the assignment. The test you need to run is the following:

```
valgrind ./stack
```

If you want more details about memory leaks, you can run it like this:

```
valgrind --leak-check=yes ./stack
```

If you have a memory leak you will see an error that looks something like this:

```
==31419== LEAK SUMMARY:
==31419== definitely lost: 240 bytes in 2 blocks
==31419== indirectly lost: 240 bytes in 2 blocks
```

If you have a dangling pointer you may see an error that looks something like this:

```
==31993== Invalid read of size 1
==31993== at 0x4E7D000: vfprintf (vfprintf.c:1629)
```

If you eliminate both bugs (your goal) then you should see output something like this:

```
==1272== HEAP SUMMARY:
==1272== in use at exit: 0 bytes in 0 blocks
==1272== total heap usage: 4 allocs, 4 frees, 480 bytes allocated
==1272==
==1272== All heap blocks were freed -- no leaks are possible
==1272==
==1272== For counts of detected and suppressed errors, rerun with: -v
==1272== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Or you might see (depending on which version of valgrind you use):

```
==9402== LEAK SUMMARY:
==9402== definitely lost: 0 bytes in 0 blocks
==9402== indirectly lost: 0 bytes in 0 blocks
==9402== possibly lost: 0 bytes in 0 blocks
==9402== still reachable: 4,096 bytes in 1 blocks
==9402== suppressed: 25,084 bytes in 373 blocks
```

There are a few different ways to fix the bug. Any of them are fine as long as you still allocate stack elements dynamically using `malloc`.

Question 3: Extending the ISA [15%]

Implement the following six control-flow instructions in the simulator. The code provided with this assignment includes the file CPU.java that you can use as the starting point for this assignment. Alternatively you can use the version you implemented yourself for Assignment 2.

Instruction	Assembly	Format	Semantics
branch	`br a`	`8-pp`	pc ← (a = pc + pp*2)
branch if equal	`beq rc, a`	`9cpp`	pc ← (a = pc + pp*2) if r[c]=0
branch if greater	`bgt rc, a`	`Acpp`	pc ← (a = pc + pp*2) if r[c]>0
jump	`j a`	`B--- aaaaaaaa`	pc ← a
get pc	`gpc $o, rd`	`6Fpd`	r[d] ← pc + (o = p*2)
indirect jump	`j o(rt)`	`Ctpp`	pc ← r[t] + (o = pp*2)

Note that the indirect-jump offset is unsigned, and so for indirect jump, pp ranges from 0 to 255. On the other hand, the branch pc-relative value is signed and so for branches, pp ranges from -128 to 127.

Test your implementation by creating a file named `test.s` that contains tests for each of the new instructions.

Question 4: Write a program in assembly [50%]

Implement an assembly-language program that examines a list of student grades and finds the student with the median average grade; this student's ID should be placed in the variable `m`.

Your program must correspond to a C program where the input list of students and the output student ID are given as follows.

```
struct Student {
  int sid;         // the student's ID
  int grade[4];    // the student's 4 grades
  int average;     // this is computed by your program
};

int n;             // number of students
int m;             // you store the median student's id here
struct Student* s; // a dynamic array of n students
```

Your assembly file must format student records exactly as C would; i.e., a struct with 6 integers. You should assume no padding is used by the struct, i.e., each student instance uses exactly 24 bytes. You must use the following assembly declarations for this input and output data. Note: this example is for an array with one student. You should obviously test your code on larger arrays.

```
n:    .long 1     # just one student
m:    .long 0     # put the answer here
s:    .long base  # address of the array
base: .long 1234  # student ID
      .long 80    # grade 0
      .long 60    # grade 1
      .long 78    # grade 2
      .long 90    # grade 3
      .long 0     # computed average
```

Put your solution in a file named `q4.s`.

Work Incrementally – Do each of these steps separately!

This is a very challenging programming problem. It will stretch and strengthen your ability manage a large number of programming details, as is required when writing assembly. To be successful with this you must be very disciplined to program and test incrementally.

Observe that the program breaks down into several sub-problems. You should tackle them one at a time and thoroughly test each step before moving to the next.

Here's a list of the key sub-problems. You might want to further sub-divide things, but do not try to combine steps.

1. Compute the average grade for a single student and store it in the struct. For simplicity, you can ignore the fractional part of the average; i.e., you do not need to round.
2. Iterate through the list of students, compute their average grades and store them.
3. Swap the position of two students in the list.
4. Compare the average grades of two students and swap their position conditionally, using your code from Step 3.
5. Now consider creating a procedure to encapsulate either Step 3 or Step 4 as described in the Use Procedures section below. This step is optional but highly recommended.
6. Sort the list by average grade in ascending order. See the discussion below regarding sort algorithms.
7. Find the median entry in the sorted list and store that student's ID in `m`. For simplicity you can assume the list contains an odd number of students.

Sorting Algorithms

You are free to use any sort algorithm you like, but Bubble Sort is the probably the simplest. Here's a version of bubble (sinking) sort in C that you might consider.

```
void sort (int* a, int n) {
  for (int i=n-1; i>0; i--)
    for (int j=1; j<=i; j++)
      if (a[j-1] > a[j]) {
        int t = a[j];
        a[j] = a[j-1];
        a[j-1] = t;
      }
}
```

Take this step by step and incorporate your work from Step 3-5 above. For Bubble Sort, here are two sub-problems.

- First, write a loop that iterates through the list once, bubbling the student with the lowest average to the top (or sinking the lowest student to the bottom). If you created a procedure in Step 5, then the inner part of this loop is a call to this procedure; otherwise it is the code from Step 4.
- Then, add the outer loop that repeats the inner loop on the unsorted sub-list repeatedly until the entire list is sorted.

If you want a slightly different challenge, consider using other algorithms like Selection Sort or Insertion Sort instead.

Ordering and Combining Steps

Notice that it is not necessary to do these steps in the order listed above. Steps 1 and 2, for example, are completely independent from the other steps. Step 7 is also independent from the other steps. You could do Step 6 before Steps 3-5 and then incorporate the conditional swap once the other parts of Step 6 are working. Similarly, you can do Step 5 before Steps 3 or 4 and incorporate them into the procedure later. The key thing here is work on each piece independently and then carefully combine steps to eventually produce the program. The program itself will be on the order of 70 to 100 assembly-language instructions. That sounds like a lot … and it is. But if you think of this as seven steps each with 15 or so instructions (some will have less), it's not impossible.

Be sure to comment every line of your code and to separate sub-problems with comments and labels so that it is easy for you to see what each part of the code does without having to read the code.

Multiplying and Dividing by 24

You will have noticed that the variable `s` is a dynamic array of type `struct Student` and that `sizeof(struct Student)` is 24. And so you might find that, depending on how you are implementing your code, you'd like to multiply or divide by 24 (i.e., to convert between an array index and a byte-offset), but you'll recall that you can't do this with a single instruction in our ISA. Then you'll notice that x*24 = x*16 + x*8 and you'll see that you can multiple by 24 using 3 instructions (or 4 depending on how you count).

Register Allocation

Keeping track of registers is going to be a challenge. Note that this program is too big for you to use a distinct register for every value in the program. You are going to have to re-use registers to store different things in different parts of the program.

This will add complexity, for example, when combining two steps or when adding a step to the rest of the program. For example, if your code for Step 4 uses register r0 and one of the loops you write in Step 6 also uses r0, you're going to have to change one of them to use a different register (or use a procedure as described in the next section). So, for parts of the code that connect like this, some pre-planning will help.

Another useful strategy is to group sections of code (one or maybe a few steps) and treat them as independent stages with respect to registers. At the beginning of a stage you assume nothing about the current value of registers and you are free to use any registers you like within that stage. At the end of the stage, any register values that are needed by subsequent stages should be written to memory. You may want to create some additional variables to store these temporary values.

Using Procedures

One way to divide code into stages is to use procedures. Step 5 suggests that you do this to encapsulate the code that conditionally swaps two adjacent elements of the list (i.e., Steps 3 and 4). You can use this approach to simplify register management by having the procedure save registers to memory before it starts and restore them from memory before it returns (e.g., using temporary variables). Any register the procedure saves in this way is a register it is free to use without interfering with the registers used in the two loops in Step 6 that call it.

Assuming you only swap adjacent elements, this swap procedure needs one argument / parameter: i.e., the index of one of the array elements to swap. The index of the other element is this value plus or minus one, depending on how you do it. The caller should pass this value in a register; for example if the loop has the index in r4 then the procedure should just use r4 to get this value.

Use `gpc` and `j` to call the procedure and use the indirect jump to return. Do not worry about creating a stack.

All of this is optional, but good practice for learning how to implement procedure calls in assembly which you will be tested on. You can do this for other parts of your program as well.

Work Incrementally

Finally, partial marks (if you need them) will be awarded based on the number of the steps listed above that you've written correctly. Submitting a big blob of assembly that doesn't do any of the steps right won't be worth anything. So, work incrementally. Test each step separately. Once you have a step working, save it in an auxiliary file just in case.

Deliverables

Use the handin program. The assignment directory is `~/cs213/a5`, it should contain the following plain-text files.

1. `README.txt`: the name and student number of you and your partner.
2. `PARTNER.txt`: your partner's CS login id and nothing else (i.e., the 4- or 5-digit id in the form a0z1). As usual your partner should not submit anything.
3. `q1.s`: commented as described in Question 1.
4. `q1.c`: the translation of `q1.s` as described in Question 1.
5. `q1.txt`, and `q1-desc.txt`: the count and description of memory accesses in the code of Question 1.
6. `q2.txt`: your description of the bug in Question 2.
7. `stack.c`: updated as needed to fix the bug in Question 2.
8. `CPU.java`: your implementation of the new instructions listed in Question 3 as well as a correct implementation of the part of the ISA you implemented previously.
9. `test.s`: your test code for the new instructions in `CPU.java`.
10. `q4.s`: your answer to Question 4.