Lab Assignment 6: The Stack

Due Oct 19, 2018 by 11:59pm **Points** 100 **Available** after Oct 14, 2018 at 12am

Goal

This assignment has four parts.

First, you will examine how programs use the runtime stack to store local variables, arguments and the return address. You will do this using a set of snippets and you will answer questions about their execution. This part is not for marks.

Then, you will examine two SM213 programs that contain procedure calls to determine what they do, and write equivalent C programs that perform the same task.

Then, you will examine an SM213 program that contains a recursive function using tail recursion. You will be required to change this program so that it performs the same operations, but without any recursive calls.

Finally, you will mount a buffer-overflow, stack-smash attack on an SM213 program.

All files required for this program are available in **a6code.zip**.

As usual, you are encouraged to do this in groups of two. Just be sure that you both do all the work, helping each other.

Part 1: Understanding the Stack (not for marks)

The file provided code contains the following files.

- `S7-static-call.java`, `S7-static-call.c`, `S7-static-call-stack.s`, `S7-static-call-reg.s`
- `S8-locals-args.java`, `S8-locals-args.c`, `S8-locals-args.s`
- `S9-args.java`, `S9-args.c`, `S9-args-stack.s`, `S9-args-regs.s`

Familiarize yourself with the snippets above by running them in the simulator and asking yourself the following questions. Although these questions won't be graded, answering them can help you with the other parts of the assignment and with the concepts in general for the midterm and final.

Carefully examine the execution of `S7-static-call-stack` in the simulator and compare it to `S7-static-call-regs`. Run the snippets and look carefully at what happens. Ask yourself these questions.

- What is the difference between the two approaches?
- What is one benefit of the approach followed in stack?
- What is one benefit of the approach followed in regs?

Carefully examine the execution of `S8-locals-args` in the simulator. Ask yourself these questions.

- What lines of foo and b allocate b's stack frame?
- What lines of foo and b de-allocate b's stack frame?
- What changes would be required in b to have one additional argument (for a total of 3 arguments) and two additional local variables (for a total of 4 arguments)? Add the arguments and locals; note that you do not actually use these new variables in any way.

- What changes would be required in foo to call `b(0,1,2)`?

Carefully examine both versions of `S9-args-stack` and `S9-args-regs` in the simulator. Ask yourself these questions.

- What memory accesses does stack make that regs doesn't make?
- How many more memory accesses does stack make, compared to regs?

Part 2: Interpreting Assembly functions [30%]

This part uses the files `q2a.s` and `q2b.s`, found in the provided code.

Answer the next two questions by modifying the .s files and by writing .c files. The .c files must compile and execute. When they execute they must perform the same computation as the .s file and print out the value of its static variables, one per line as a decimal number (nothing other than that number on each line). This means that `q2a.c` must print 10 lines of numbers and `q2b.c` must print 16.

1. [15%] Examine `q2a.s` and its execution in the simulator. Add a comment to every line that explains what that line does in as high-level a way as possible. Then, write an equivalent C program called `q2a.c` that is the most likely candidate for the file that was compiled into `q2a.s`. Use the same variable and procedure names in this program that you used in the assembly-file comments. Ensure that there is a correspondence between lines in the assembly file and lines of the C program, but do not include the start procedure in `q2a.c`; this procedure is added automatically by the C compiler to initialize the stack and call main. The end of your main procedure should print the value of the program's ten static variables as described above.

   You may notice a register that is used in a way that is best explained by saying that it is a local variable, even if its value is never read from or written to the stack. Avoiding these memory accesses is a common optimization compilers make for local variables.

2. [15%] Do the same for `q2b.s`.

Part 3: Tail Recursion [20%]

This part uses the file `q3.s`, found in the provided code. This file contains the implementation, in SM213, of a function called `search`, which searches for a value in a sorted array using binary search, returning the address (as a pointer) to the location where the element is found, or 0 (NULL) if it is not in the array. The file also includes: a start function; a main function, that calls `search` with arguments provided in global variables, and saves the result into a global variable `ret`; and a sample array and value for testing.

Run the provided code in the simulator, and observe how it works. Run it several times, changing the values of `val` and `size`. In particular, take note of the arguments taken by function `search`, and how the function uses the stack.

Note that the function is tail recursive, and as soon as the element is found a quick succession of call epilogues (deallocating space in the stack) and returns is executed, without any real processing involved. This makes the code a good candidate for optimization, by converting the recursive calls with a loop.

Modify the file `q3.s` to convert the tail recursion to a loop. In your modification, you are only allowed to:

- add comments (optional, but strongly encouraged);
- remove instructions from function `search`;

- change the offset in memory accesses involving the stack pointer (r5) as the base address.

Note that you are not allowed to modify the start or main functions, or to add or modify instructions except as described above. Also note that your function must continue to return the same value as before for any array, size and value to search. To allow the handin testing script to verify your code, you should not change the size of the stack.

Also note that, typically, a loop uses branches, and that the recursive call uses absolute jumps. Although you are not allowed to replace the `j` instruction with a `br` instruction, this change would not provide a meaningful change in behaviour.

In your final submission, the function `search` should not allocate any space in the stack beyond the space that is already provided for its own arguments (the space allocated in `main`), not even to save the return address. In other words, function `search` should not modify r5 or r6 in any way.

Part 4: Stack Smash Attack [50%]

For the next two questions, refer to the file `copy.c` provided in the code files.

1. [10%] Using `copy.c` as a guide, write a simple SM213 assembly-language program that copies a null-terminated array of integers (use Snippets 8 or 9 from part 1, or the code from parts 2 and 3, as a guide). Call this program `copy.s`. In `copy.c`, the input array is stored in a global variable named src and the destination array is in a local variable (i.e., stored on the stack). Your assembly code must do the same.

   As in `copy.c`, you will need two procedures: one that copies the array and one that initializes the stack pointer and calls the copy procedure. Ensure that the copy procedure saves r6 (the return address) on the stack in its prologue and restores it from the stack in its epilogue, as shown in class.

   Note that this code contains a buffer-overflow bug. That is intentional. Be sure your assembly code has this bug so that you will be able to attack it in Question 2. Another thing you'll want to do is to keep the value of i in a register in the body of the loop. If you were to read/write it from/to the stack on every iteration, you'll find that the buffer overflow will overwrite the value of i and thus change the way the attack string is written to the stack.

2. [40%] Modify `copy.s` to devise a buffer-overflow attack on this program. The arbitrary code to be executed by the attack should set the value of every register to -1 and then halt.

   You are stuck with a similar set of restrictions that a real attacker confronts. You may not modify the program you have just written in any way other than to change its input (i.e., `src`). Change `src` to make it bigger to contain virus program and other values as needed so that copy executes the virus program when it returns, instead of actually returning to main.

   You must specify the attack string (the value of `src`) using a sequence of `.long` directives. Recall that each `.long` specifies the value of 4 bytes of memory. The string will contain the virus program as machine instructions, which are either 2 bytes or 6 bytes. You will thus need to compact multiple instructions into a single `.long` and possibly also split a 6-byte instruction across two `.long`s.

   Run your attack in the simulator to be sure that it works.

What to Hand In

Use the handin program. The assignment directory is `~/cs213/a6`, 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). Your partner should not submit anything.
3. `q2a.s`, `q2a.c`, `q2b.s`, and `q2b.c`: your answers to Part 2.
4. `q3.s`: your answer to Part 3.
5. `copy.s`: your answer to Part 4.