C track: assignment 8: Implementing a virtual machine

Goals

This lab is the last assignment in this track. By now you know almost all the important aspects of the C language, with the exception of a few features like function pointers which aren't really suitable for an introductory class (see the advanced C track for more on those). Therefore, this assignment is more of a "fun" assignment where you can use your newly-acquired programming skill to do an interesting project. Although fun, this lab is somewhat more involved than the previous assignments, so we're supplying you with a lot of the code pre-written so that you can concentrate on the interesting parts. If you're having trouble, please come and see us before the assignment is due, or ask questions in class.

The program you are to write for this lab will implement a *virtual machine* (also known as a VM), which is a simulation of a hypothetical microprocessor. Virtual machines, once a computer science curiosity, have become big business; the java and C# languages both use virtual machines to execute their code, as do nearly all scripting languages. The virtual machine you will implement will be extremely simple but will still be powerful enough to do complex computations.

This program is interesting because it demonstrates that there is nothing magical about the programs that compile and execute other programs; you can write programs in C that interpret programs in a different programming language than C, and you can invent the programming language yourself if you want. Programs that execute other programs -- what a concept! Also, the more ambitious of you might have fun writing your own programs for the VM or extending the instruction set of the VM (suggestions for this are given below).

Language concepts covered this week

- Different integer types (`int`, `unsigned int`, `short`, `unsigned short`, `char`, `unsigned char`).
- The `switch` statement.

Other concepts covered this week

Stacks and virtual machines.

The stack data type

A *stack* is a fundamental data type that is frequently used in the implementation of programming languages. A stack is a data structure that, like an array, is used to store a group of values. Unlike an array, a stack can be arbitrarily large and access to the elements of a stack is only at one end of the stack. A stack consists of the following:

- A region of memory.
- A location in memory that is called the *bottom* of the stack. This location never changes.
- A location in memory that is called the *top* of the stack, or *TOS*. This is the only location on the stack which is directly accessible. This location varies as the stack grows or shrinks.
- Two operations: *push*, which pushes a data value onto the top of the stack (making the stack one larger), and *pop*, which removes the data element from the top of the stack (making the stack one smaller). Some implementations of stacks discard the popped value and some use it; our implementation will just discard it.
- Optionally, other operations that access or manipulate the contents of the stack. Our implementation won't have any such operations.
- Most stack implementations also have a maximum size.

The name *stack* refers to the fact that data elements on the stack are like the plates in a stack of plates sometimes found in restaurants: you only have access to the top element of the stack, which was the last element placed (pushed) on top of the stack. This is referred to as a last-in, first-out (LIFO) data structure. In our case the stack will store integers, so if you push the integer 10 onto the stack and then push 20, the top of the stack is 20. If you pop the top of the stack (and that's the only thing you can pop), the new top of the stack is 10 again. This might seem useless, but we will have other operations that use the stack as well. The stack is well suited for storing intermediate results of computations, because it can grow to be very large (in principle, arbitrarily large, but of course it's limited by the total amount of memory on your machine). The stack is particularly useful when implementing function calls, but our simple VM will not have function calls (although you might want to try implementing function calls for fun).

The implementation of the stack is usually done as an array which is the size of the maximum number of elements that can be in the stack, as well as a "pointer" to the location in the array which is **one location above** the top of the stack. This is called the **stack pointer**. It's a bit of a misnomer in that it isn't a

C pointer; it's just an index into the array. We repeat that the stack pointer is **one location above** the top of the stack; for some reason, many people seem to think that the stack pointer is the index of the top of the stack, when it's in fact one more than this.

The contents of the array above the top of the stack don't matter (they are considered to be garbage). When you push an element onto the stack, you have to increment the stack pointer and also check that the stack is not already completely full (that's called a *stack overflow*). When you pop an element from the stack, you check that the stack isn't already empty (which would result in a *stack underflow* if you tried to pop it) and then you decrement the stack pointer. You don't have to clear the value that used to be at the top of the stack, because it doesn't matter anymore; it will be overwritten when you push a new element onto the stack.

The virtual machine

Machine architecture

The virtual machine is implemented as a struct which contains the following:

1. A stack which can be at most 256 elements large
2. A stack pointer
3. A group of 16 *registers* (implemented as an array). Unlike the stack, you can access any register at any time.
4. A memory region where VM *instructions* are stored (see below for the instruction set and their format). This is also implemented as an array. There can be at most 2^16 or 65536 bytes worth of instructions in a program, and each instruction can be from 1 to 5 bytes in length (including operands). We call this memory region the *instruction array*.
5. An *instruction pointer* which tells us where we are in the instruction array.

Instruction set

The heart of the VM is the *instruction set* which it implements. There are only 14 instructions, and here they are. The word in capital letters is the instruction name while the hexadecimal number in parentheses is the one-byte *byte code* of the instruction, which is how the instruction is represented in the computer's memory (and in a disk file).

- NOP (0x00)

  "no operation"; do nothing.

- PUSH (0x01)

  PUSH <n> means to push <n> onto the top of the stack (TOS). <n> is a standard 4-byte integer (int).

- POP (0x02)

  Pop the top of the stack.

- LOAD (0x03)

  LOAD <r> means to load the value in register <r> to the TOS. Here <r> is a one-byte unsigned integer (see below).

- STORE (0x04)

  STORE <r> means to store the TOS to register <r> and pop the TOS. Again, <r> is a one-byte unsigned integer.

- JMP (0x05)

  "Jump"; JMP <i> means to go to location <i> in the instruction array (*i.e.* change the instruction pointer to <i>). <i> is a two-byte unsigned integer (see below). This is also the case for the next two instructions.

- JZ (0x06)

  "Jump if top of stack is zero"; JZ <i> means that if the TOS is zero, pop the TOS and go to location <i> in the instruction array. If the TOS is not zero, just pop the TOS and continue with the next instruction.

- JNZ (0x07)

  "Jump if top of stack is nonzero"; JNZ <i> means that if the TOS is nonzero, pop the TOS and go to location <i> in the instruction array. If the TOS is zero, just pop the TOS and continue with the next instruction.

- ADD (0x08)

  Pop the top two elements on the stack and replace them with their sum (S2 + S1, where S1 is the TOS and S2 is the element on the stack below the TOS).

- SUB (0x09)

  Pop the top two elements on the stack and replace them with their difference (S2 - S1).

- MUL (0x0a)

  Pop the top two elements on the stack and replace them with their product (S2 * S1).

- DIV (0x0b)

  Pop the top two elements on the stack and replace them with S2 / S1 (integer division).

- PRINT (0x0c)

  Print the TOS to stdout (followed by a newline) and pop the TOS.

- STOP (0x0d)

  Halt the program.

Some instructions (PUSH, LOAD, STORE, JMP, JZ, JNZ) take *operands* which are integers of different sizes (1, 2, or 4 bytes). These operands are part of the instructions that are read in from the program file. Next, we'll talk about how to get these integer values.

Integer sizes

Integers in a C program do not all have to be of the same size in bytes. In addition, they can be unsigned or signed. Usually we use signed integers, but in some cases a sign value would not mean anything, so we can use unsigned integers instead. Unsigned integers can hold larger values than signed integers, because one bit of the signed integer is taken up by the sign (actually it's a bit more complicated than that, but never mind).

Here are the different integer sizes you can use in a C program:

- `int` -- a signed integer, usually 32 bits on most machines
- `unsigned int` -- same as `int` but unsigned (duh)
- `short` -- a signed integer, usually 16 bits on most machines
- `unsigned short`
- `long` -- a signed integer which is at least as big as an `int` and sometimes bigger (implementation-dependent)
- `unsigned long`

- `unsigned char` -- you can use this as an unsigned one-byte value

In addition, many compilers (including `gcc`) support a `long long` type which is "longer than long", but we won't need that here. For this program, we will only need the `int`, `unsigned short` and `unsigned char` types.

We will use these types as follows:

- The program instructions will be a series of one-byte values (`unsigned char`). If an instruction has an operand (which could be a one-, two-, or four-byte value, depending on the instruction), it comes after the instruction itself in the instruction array.
- Programs will be a maximum of 2^16 (65536) bytes long, so the instruction pointer will be an `unsigned short`, since the largest value that can be stored in two bytes is 65536. Note that the instruction pointer should always point to the start of a valid instruction, or the results will be unpredictable.
- The stack will be at most 256 values large, so the stack pointer will be an `unsigned char`.
- All other integer values will be plain `int`.

All of these values can be used like regular `ints`; you can add, subtract, increment, decrement them etc. to your heart's content. You have to be careful when using unsigned values, because if you subtract 1 from 0, you will get a large value (255 for an unsigned char) since there is no way to represent -1 in an unsigned value (and the compiler won't warn you that you are doing this). Similarly, if you add two shorts or unsigned chars you have to be extra careful about not overflowing, or the results will not be what you want. This is in line with the C philosophy that you have to know exactly what you're doing at all times.

At the beginning of the program, the contents of the VM program file are read into the instruction array `vm.inst`. As mentioned above, each instruction consists of a one-byte bytecode which identifies which instruction it is, and some instructions contain an operand which can be 1, 2, or 4 bytes long. We've supplied a function called `read_n_byte_integer` which will grab the next n bytes from the instruction array (where n = 1, 2, or 4) and convert them into a regular `int`. Once you have the integer value, it shouldn't be hard to figure out what to do with it.

VM assembly language and VM machine language

There are two parts to getting a VM program to run: writing the program and executing the program on the VM. You are only responsible for the second task. For the first, We've written a simple VM program which computes 10 factorial (10 x 9 x 8 x ... x 2 x 1) and prints it to the terminal. The VM program is a binary file. Binary programs are very difficult to write directly, because you are writing directly in the "machine language" of the VM, which is not meant to be directly usable by humans (and not very friendly for your text editor either, although emacs can handle it if you use "hexl-mode").

To make it easier to write machine language programs, we're supplying you with an "assembler" program that takes a slightly higher-level textual description of a VM program and translates it into the binary form actually executed by the VM. The text version of the program is said to be written in the VM's "assembly language". There is a one-to-one relationship between instructions in the assembly language and instructions in the machine language, so you don't lose any control writing programs in assembly language. However, the programs are a lot easier to read and write, and you can add comments etc.

The VM assembly language program is called `factorial.bca` and is located here. The `.bca` suffix means "byte-code assembler". Take a look at it now. Comments start with the character "#" and go to the end of the line. Empty lines are ignored. The other instructions are the same as the machine instructions described above, with one exception: they may have a numeric *label* at the beginning of the instruction. This label indicates a memory location where a *jmp*, *jz*, or *jnz* instruction may jump to. The arguments of these instructions are also labels of this form. These are converted to absolute memory locations (in the VM's memory) by the assembler program, which is called `bca` (for "byte-code assembler") and which is located here. We wrote this in python because it was too tedious to write in C :-(You are not expected to understand how it works (although if you're interested, you can take the python track of CS 11 and find out :-)). You don't need to use this program at all unless you want to write your own VM programs, since we're supplying you with the machine language program as well (see below).

If you want to write your own programs for the VM, the procedure is as follows:

- Write the program is the assembler format, and make sure the file ends in `.bca`. Let's say it's called `myprog.bca`.
- Run the assember: `bca myprog.bca`. This will create a VM machine language file called `myprog.bcm`.
- Run the VM on your program: `bci myprog.bcm`. This will execute your VM program, assuming that you've implemented the VM correctly (see below).

Program to write

You have to implement the VM by writing a program called `bci`. This program takes a single argument, which is the name of a VM machine language program file. The program will set up the VM, read in and execute the program, and exit.

Supporting files

To simplify your task, we're supplying you with a lot of supporting files. They are:

- The Makefile.
- `factorial.bca`, a sample program in the VM "assembly language" that computes 10 factorial (10x9x8x...x2x1). You don't actually need this file, but it is interesting to read and you can use it as a template to write your own VM programs if you like.
- `bca`, the "byte code assembler", which takes the VM assembly language programs and converts them to VM machine language. You don't need this either, unless you want to write some VM programs of your own. If you do download it, make it executable before trying to use it.
- `factorial.bcm`, a sample program in the VM "machine language" that computes 10 factorial. Use the "save as" feature of your browser to download this, because it's a binary file. This is the VM program that your program must execute.
- `bci.h`, the header file for the VM implementation.
- `bci.c`, the implementation of the VM. This is the only file you have to modify; the places in the code marked TODO are places where you have to add your own code.
- The `main.c` file, which contains the `main` function of the program.
- The test script.

These are the only files you need for this assignment. To reiterate: the only thing you have to do is to fill in the parts of `bci.c` marked with a TODO (but you have to fill in ALL of them). (Remove the TODO comments, of course). Don't change anything else. These implement the instructions of the virtual machine. This is not very much code; it can be done in less than 100 lines. The tricky part is to make sure that you update the stack pointer and instruction pointer correctly after each instruction (and it's not really that tricky). In addition, you should add error-checking code to check that you don't try to pop an empty stack or push onto a full stack (and similarly for all instructions which push or pop stack elements). You should also check that you don't read beyond the end of the program instruction memory (or before the beginning), and you should check that you don't try to load from or store to a non-existent register. If an error occurs, you should print an error message to `stderr` and exit the program with a nonzero exit value. Put error-checking code into the corresponding functions for machine operations.

When checking for stack overflow, the easiest way is to disallow any stack push that would fill up the last location on the stack (location 255 in our implementation). Even though this wastes some space, it's easy to write the code for it. You might think that you could allow the use of location 255 but disallow pushing onto a stack that has that location filled. In fact, you could, but you'd have to be quite careful, because the largest integer in an `unsigned char` is 255, so you can't increment the stack pointer and check that it's no more than 255 (here, 255 + 1 wraps around and the result is zero). The compiler will warn you if you try to do something like this.

For this assignment, you should run the code on an Intel x86 family microprocessor (*e.g.* a Pentium), because the way that integers are laid out in the machine language is not guaranteed to work on other processors.

Testing your program

You can run the program manually by typing

```
% bci factorial.bcm
```

Typing

```
% make test
```

will run the test script on your program, which just tests to see that the output of `bci factorial.bcm` is what it should be (which is 3628800). Don't hand in the program until it passes the test. As always, make sure the test script ("run_test") is executable before you try to run it.

Extensions to the VM

If you find this program interesting, you might try the following extensions to the VM:

- Adding support for function calls. This is actually quite tricky, because you have to put the arguments to the function call on the stack (in order to handle recursion properly) as well as the return address of the function. Or you could use a separate stack just to hold return addresses of functions.

- Adding a non-register region of memory to the VM, as well as instructions to move data from there to registers and back.
- Adding new instructions to the VM.

Note that these extensions could take quite a bit of programming, so you might want to wait until you have some free time ;-)

If you find this material interesting, you should definitely take a course on computer architecture (CS 24 is an excellent introductory course on this and other matters).

To hand in

The file `bci.c`.

Finally...

I hope you've enjoyed this course. If you have any suggestions for how the course can be improved, please let me know.
