Lecture notes: January 13, 2017

Topics:

1. Selection sort wrapup
   i. Pseudocode (see 1/11 notes)
   ii. How long does it take to run?
2. Program stack
3. Hexadecimal and Endianness
4. GDB

Selection sort wrapup

Readings: Forouzan and Gilberg
- pp 370–371
- p 534 [Note that their analysis of selection sort is a little different than ours]

We covered the pseudocode from 1/11

Some of you may have noticed that selection sort takes a very long time to run on large inputs. How long, exactly?

Let's count *iterations*: how many times does the inner loop (which searches for the minimum value in the *rest* of the array) run?

```
int input[N] = //input
cursor = 0 //initial position of the cursor
for (cursor = 0; cursor < N; cursor++)
      //sorted list from [0,cursor)
      //rest of the list from [cursor, N)
      for (i = cursor; i < N; i++)
            //search the rest of the list to find the smallest value
      //swap the smallest value with the value at input[cursor]
```

The inner loop runs *N* times, and each time it runs, it runs for (*N* - *cursor*) iterations. Cursor takes on every value from 0 to *N - 1*:

$$\sum_{i=0}^{N-1} N - i$$

That summation is the same as:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

Note, also, that most of the work happens in the inner loop, so how long selection sort takes is dominated by how long that inner loop takes. Trying to be precise about just how long the inner loop takes is tricky: depending on how you wrote it, it may take more or fewer instructions to

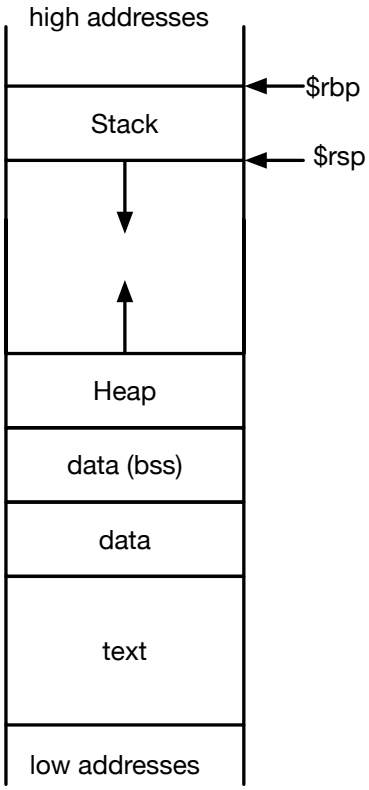
execute. But what matters is, no matter how you wrote that inner loop, *if we make N twice as big, the inner loop will run about four times as many times*! That's the dominating factor here: double the input, take four times as long. So all that really matters is the quadratic term. The next +N or /2 doesn't really matter.
Thinking about run time this way is called *asymptotic analysis*, and we'll come back to it later in class.

Program stack

When a program runs, your computer's memory is divided up into four *segments*:

1. The stack – this is where local variables for functions, function arguments, return values, and return addresses go. Every function has a *stack frame* that stores this information. When a function gets called, its *stack frame* is "pushed" onto the stack, and when it returns, the frame is "popped" off the stack.
2. The heap – this is where memory you allocate using `malloc` goes
3. The "text" – this is where the code of the program is stored
4. The "data" – this is where global variables of various sorts are stored. This space is broken up into smaller chunks:
    i. "data" holds global or static variables in the program that are initialized (e.g., if you declare a global variable `int x = 7;`)
    ii. "bss" holds global or static variables that are *uninitialized* (e.g., if you declare a global variable `int y;`) This whole segment is initialized to zero when the program starts (Why distinguish bss from data? Initialized variables need to have the correct values initialized for them, so they need to be stored in your program's binary. Uninitialized values don't need values stored, so the binary just tracks *how much space* the uninitialized variables take up.)

How are these segments placed in memory? One thing to note is that the text segment and the data segment(s) have fixed size, but the stack and the heap do not—as the program runs, you may call additional functions (requiring space on the stack) or allocate more memory using `malloc` (requiring space on the heap). To make room, the stack is organized as follows (higher addresses on top, lower addresses on the bottom):



When the program starts (i.e., we call `main()`), all of the local variables for main() are placed on the stack in a *stack frame*. We use the register $rbp (the *base pointer*) to mark the "bottom" of the stack frame, and the register $rsp (the *stack pointer*) to mark the "top" of the stack. (Note that when we draw the stack this way, the "top" of the stack has a lower address, and looks like it's lower than the base of the stack. Confusing, I know.)

If `main` calls `foo`, a bunch of things happen: the arguments to `foo`, and space for its return value, are "pushed" onto the stack (this automatically decrements $rsp to move the top of the stack). The address of the *next* instruction in `main` is pushed onto the stack (this is where execution will go to when foo returns). The current value of $rbp is pushed onto the stack, and then $rbp is moved to $rsp. $rsp is then moved down. This new space between $rbp and $rsp is `foo`'s stack frame: it's where any local variables for `foo` can get stored. When `foo` returns, the process is rewound, "popping" the frame off the stack, and the program resumes from the return address saved on the stack.

Essentially, as functions are called, we push stack frames for them onto the stack, so the stack keeps growing as long as we call more functions. Whatever function is *currently executing* has its frame at the top of the stack. When a function returns, its frame is popped off the stack.

(PA02 asks the following question: what if you find a way to overwrite the part of the stack that stores the return address of a function?)

Hexadecimal and Endianness

Readings: Forouzan and Gilberg, Appendix D (especially pp 1033–1037)

Computers don't store numbers in decimal (base 10). Instead, they store data in binary (base 2):

5 = 101
21 = 1 0101
1547 = 110 0000 1011

And so on. Because it's difficult to read base 2, we often instead write numbers in base 16, or hexadecimal. We use the letters A through F to represent 10 through 15. That lets us write groups of four binary digits as a single hexadecimal digit:

5 = 101 = 0x5
21 = 1 0101 = 0x15
1547 = 110 0000 1011 = 0x60B

Exercise: what are the following numbers in binary and in hexadecimal?

73
2918
206

One very confusing thing about the way data is stored in memory is *endianness*. When we write a single number:

1257

We put the most significant digit (the '1') at the left, and the least significant digit (the '7') on the right. When a program wants to store a number, it thinks of it as a series of bytes. In C, integers take up four bytes. So, the number 1257, when thought of as an integer, takes up four bytes. One *byte* in a computer is eight binary *bits* (digits), so each byte can be represented by two hexadecimal digits.

Written in hexadecimal, 1257 is:

0x00 00 04 E9

(256 * 4 + 16 * 14 + 9)

What order should those bytes be stored in memory? In *big endian* systems, we store the most significant byte (0x00) at the lowest address, and the least significant byte (0xE9) at the highest address. If the addresses are written left to right, from low to high, we get:

0x00 00 04 E9

Which matches the way we “normally” write numbers. In *little endian* systems, we store the most significant byte (0x00) at the *highest* address, and the least significant byte (0xE9) at the *lowest* address. If the addresses are written left to right, from low to high, we get:

0xE9 04 00 00

Which looks “backwards.” “Luckily” for us, x86 systems (like all the ones we use in this course) are little endian. When you’re reading data from memory, things will look backwards to you. Luckily, we usually don’t have to worry about endianness. The compiler and processor know about it, so when you read an integer from memory, you get the value you expect, and you don’t have to reverse any bytes.

The only time that endianness really matters is when you try to look at the contents of memory yourself. Like with GDB.

GDB

Demo of GDB using `wrongindex3.c` from PA02

1. Compiling with -g
2. Starting gdb
3. Adding breakpoints, running, continuing, stepping
4. Printing values, registers, addresses
5. Printing contents of memory