Lecture 9 Scribe Notes (10/2/12)

Control and Calling Convention

Announcements:

- Tech-Talk with GitHub this upcoming Thursday. All are welcomed - info to be posted on the web page.
- Assignment 2, Binary Bomb posted.

Control

-Addressing modes:

- `4(%esp)` in "C notation" corresponds to `((char *)%esp + 4)`
- Result treated as `(unsigned *)`

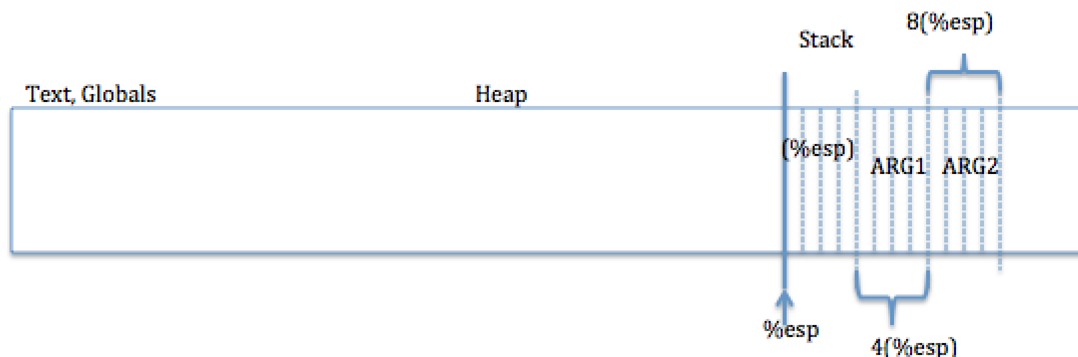
-In x86, return values are stored in the `%eax` register:

-Assembly code example:

```
movl (%esp) , %eax
addl 4(%esp), %eax
ret
```

`%eax` holds the return value.

`%eax = %eax + 4(%esp)`



Files: `F13.s` and `F14.s`, `F15.s` (available from the repo).

`F15.s`: function `f` takes `a...h` as arguments.

Calling Convention

- Agreements on how functions are represented in assembly
- Regardless of compiler, function should be represented the same way
- Sets up standards that enable interoperability

-Arguments are placed on the stack rather than on the registers.

`f()` <- Caller :

- Sets up the stack with arguments
- Saves its own local variables
- Jumps to callees first instruction (the entry point)
- Removes args from the stack once callee done.

`g()` <- Callee :

- Reads args and calculate,
- Puts the return value in %eax

-Since the caller removes the args from the stack, the callee doesn't have to know the precise number of args.

`subl $12, %esp` → move `%esp` 12 bytes back
`call g`
`addl $12, %esp` → move `%esp` 12 bytes back to the starting place.
^The above sets up the stack frame with args by subtracting 12 from stack pointer, gives control to `g`, and then moves the stack frame back after `g` is finished executing.

-Stack pointer remains the same throughout the function and only changes when another function is called.

`F18.s`:
`subl $28, %esp` → moves stack pointer 28 bytes back. "$" signifies a constant value.
`movl $.LC0, (%esp)` → stores the stack's args at `%esp` before calling `puts`
`call puts`

-Return Address - address of instruction after the call
-Running `f19` in gdb to figure out what happens in the call instruction:
`gdb f19`
`b f`
`r`
`x/3i $pc`
`info reg` → shows info on the registers
`x/w $esp` → print out 1 word (4 bytes in 32-bit OS) of `%esp`
`si` → step one instruction forward in the program's execution

`call` instruction pushes onto the stack the return address
`ret` instruction pops the address of the next instruction to execute off the stack

-Code, as well as globals, have static storage duration.

`f20.s`:
`jmp g`

The function `f` does nothing after `g` returns. Also `f` has no args and no local variables. Observing the `-O2` flag, which indicates a medium level of optimization, gcc just jumps to `g`, because it is faster. There is no reason to act as if `f` exists at all.

Tail Call Optimization → callee's return address utilized as the return address for the calling function.

`f25. s` -- 2 args: `d` and `a`
then `d = d-a`
then `d = d >> 1`
`a = a+d`
so: `a + (d - a) >> 1`

`movl`
`movl`
`subl`
`shrl` → shifting right
-Shift right without args shifts register to the right by 1.
-Compiler optimizes divisions through the use of right shift - right shift is way faster than a divide operation.
- Right shift by a number is same as dividing by 2 raised to that number. i.e

```
d >> 1  == d /(2^1).
```

- Calling convention maintains the stack as a 16 byte multiple for alignment.

An infinite number of functions can be expressed as the same assembly, and vice versa
For example,

```
f(uint a, uint b)
f(signed a, signed b)
```

`f(struct pair)` -- `int a, int b;`
`f(long long)` add 2 halves `(long) x + (x >> 32)`
`f(struct pair)` where struct pair has `int array[2];`
all produce the same assembly when compiled.

`leal` → load effective address. Usually appears in complicated code. Calculates address without dereferencing.
For example:
`leal (%edx, %eax, 8), %eax` → simply means `%eax = %edx + %eax*8`