CS 61 Scribe Notes Group 2
10/2/12
Collaborators: Andrew Bocskocsky Alexander Moore
Daniel Broudy, Zehra Naz

Machine Code (Continued)

Last time...

Figure out why

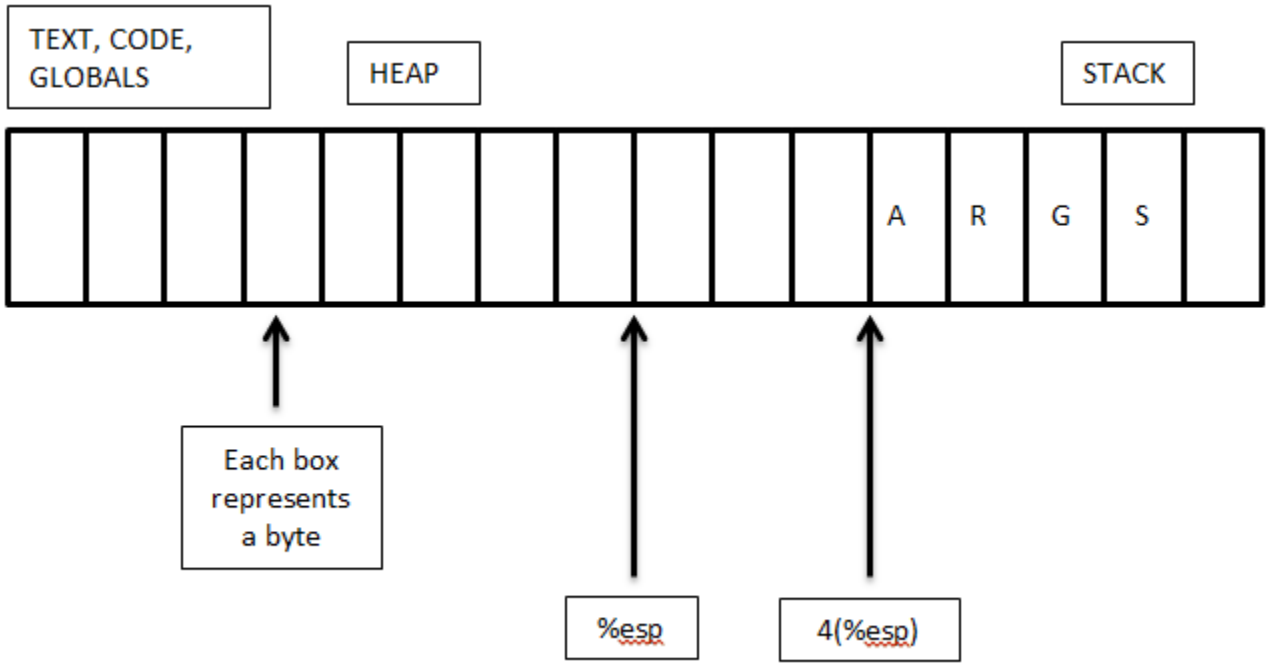
movl 4(%esp), %eax

corresponds to doing something like:

“C notation” *(((char*) %esp) + 4)
^^^^^^^^^^^^^^^^^
(unsigned *)

%eax is the return value. the argument is stored 4(%esp)

Arguments stored in a pointer to a location in memory.



Dereferencing %esp with (%esp) will give the value stored at %esp. You can put #'s outside of the parenthesis to "offset" from %esp by that many bytes. So, the argument there is stored at %esp offset by 4 bytes. This is in the STACK.
Arguments are stored starting from 4 off of %esp and go up in addresses.

%esp is a register. it doesn't have a type?
%eax takes on different values as the function goes. At the end %eax is the return value.

f13.s

Adds 2 arguments together.

```
movl 8(%esp), %eax     ← Move 8 off of %esp into %eax [   8(%esp) is the 2nd argument.   ]
addl 4(%esp), %eax     ← Add 4 off of %esp (the 1st argument)  to %eax (which is now holding 8(%eax))
ret                    ← %eax is the return value
```

Read as:
%eax = %eax + 4(%esp)

```
unsigned f(unsigned a, unsigned b){
        return a + b;
}
```

Types don't always make it through into the assembly.

f14.s

Assembly looks the same as it did in f13.s However, here we are doing SIGNED ADDITION in the C file.

```
int f( int a, int b){
        return a + b;
}
```

f15.s

Assembly looks the same as it did in f14.s. But look at the .c file:

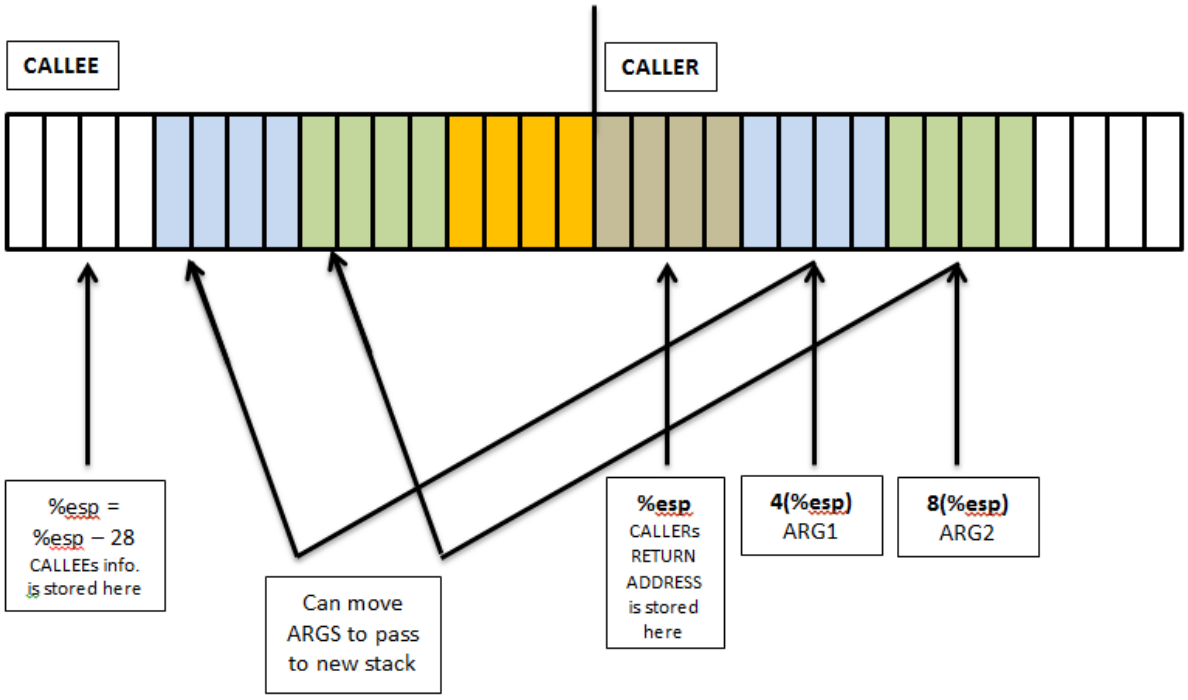
```
int f( int a, int b, int c, int d, int e, int f, int g, int f){
        return a + b;
}
```

The *calling convention* is optimizing the function by throwing out the unused variables.

CALLING CONVENTION

Agreements on how functions are represented in assembly.
Standards that allow interoperability.

The number of arguments passed to the function are known to the caller but not always by the callee



Example of Caller and Callee:

```
f()
{
        g();
}
```

f() ← Caller:
- Responsibilities:
- *Set up stack with arguments
- *Save its own local vars
- *Jump to callee's first instruction

g() ← Callee:
- Calculate return value in %eax

At the end, the CALLER has to clean up the stack once the CALLEE is done. It has to remove its arguments from the stack. As a result, the CALLEE doesn't have to know precisely how many arguments there are.

Can think about this by looking at f15.c:

Extra arguments don't appear in the assembly at all b/c the assembly is for the callee, which doesn't need to know about the extra arguments since it doesn't do anything with them. He didn't compile the caller's assembly.

f16.s

```
f:
        subl    $12, %esp       ← Moves the stack pointer 12 bytes
        call    g               ← Calls g
        addl    $12, %esp       ← Moves the %esp 12 bytes forward again once the call is over.
        ret                     ← When it returns the stack pointer is in the same place
```

Moves the stack pointer 12 bytes to the left and calls g (external function, doesn't appear here). Moving 12 bytes to the left is allocating stack space for the function. g doesn't take arguments, b/c we're not putting anything in that stack space-- the function's arguments would be garbage values if it took any.
We are pushing additional space onto the stack.

f17.s

```
f:
        subl    $12, %esp       ← Moves the stack pointer 12 bytes
        call    g               ← Calls g
        call    g               ← Calls g
        call    g               ← Calls g
        addl    $12, %esp       ← Moves the %esp 12 bytes forward again once the call is over.
        ret
```

f17.c

```
extern void g(void);

void f(void){
 g();
 g();
 g();
```

}

Question: How does the first call of g know to return to where the 2nd will be called? How does the 2nd call of g know to return to where the 3rd will be called?

$ always means constant! eg $.LC0 means a pointer to a constant string.

```
f:
        subl    $28, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $.LC1, (%esp)
        call    puts
        movl    $.LC2, (%esp)
        call    puts
        addl    $28, %esp
        ret
```

Call appears to be changing the stack pointer, subtracting 4 from the stack pointer and puts something there. The object that gets put there is the return address. This is the address of the instruction after the call.

f19.c

Want to look at which calls are being made. Do this by setting breakpoints in gdb on the function you're looking at.

Using gdb
adding breakpoints
x command means examine.
x/3i $pc means print off the three instructions after the program counter.
si ← walks one instruction at a time through the function.
info reg ← tells you all the registers of the machine (we will never discuss the last 6. Look at %esp now-- it's the stack pointer.)
x/w $esp ← look at the current value on the stack. Where address gets stored when we call si
x/3i f means print the instructions around f

"call" is, indeed, moving the stack pointer. It's jumping to a different address. It moved the stack pointer back by 4 bytes.

“call” instruction pushes the return address onto the stack
The ret instruction undoes the call instruction -- it pops it off the stack
Value in %esp has static storage duration

Break!

f20.s

```
f:
        jmp     g
```

Calls g just like f16.c, but we type:
//! -02
to make the compiler optimize more. Compiler notices that function “f” doesn’t do anything after g returns. No arguments for g. No need to change the stack. All “jmp” does is go straight to g’s instructions. It doesn’t create any new stack space for g, because g doesn’t need it. When g returns, the return address was already set up by f, so it goes to f’s return address.

At %esp, f’s RETURN ADDRESS is stored. This only happens once we use the “call” instruction. That’s why the arguments start 4 off of %esp. When we move the stack pointer for another function q, q’s return address is stored at the new %esp. Return address is where stack pointer should go back to after the instructions are done.

Unconditional jump to g. This is what optimizing compilers do. They change the assembly.

f21.s

```
f:
        subl    $28, %esp  ← creates space on the stack
        movl    36(%esp), %esx ← argument one is located 32 off %esp
        movl    %eax, 4(%esp)
        movl    32(%esp), %eax ← argument two is located 36 off of %esp
        movl    %eax, (%esp)
        call    sum
        addl    $28, %esp
        ret
```

We are passing to sum argument 1 and argument 2.
sum argument 1 is stored as %esp and argument 2 is stored at 4(%esp) but this is before space is allocated for the the address that the call instruction should return to (the return address).

Calling convention says that every stack frame is a multiple of 16 bytes

f24.s

```
f:
        movl    8(%esp), %eax
        addl    4(%esp), %eax
        shrl    %eax   ← Shift Right (shifts a register right 1 position)
        ret
```

This is implements (a + b) / 2 ← optimized to (a+b) >> 1

```
unsigned f(unsigned a, unsigned b){
        return (a+b)/2
}
```

f25.s

```
f:
        movl    4(%esp), %eax
        movl    8(%esp), %edx
        subl    %eax, %edx
        shrl    %edx
        addl    %edx, %eax
        ret
```

Can write C variable names instead of the registers to make it easier to visualize. Boils down to:

```
d = d - a
d = d >> 1
a = a + d
return a
```

It's actually a + (b - a) >> 1

This one is better than the (a + b) / 2 one. Why? Because it's more robust against integer overflow.

Consider:
a = 0x8000,0000

[1 followed by 31 zeroes]

What should midpoint be between a and b = 0x8000,0000? Should be 0x8000,0000, but it will return 0 because of integer overflow if you use the (a + b) / 2 method.
Integer overflow is a reason why compilers cannot optimize some code.

f30.s

```
f:
        movl    8(%esp), %eax
        addl 4(%esp), %eax
        ret
```

C Code:

```
struct pair
{
        int a;
        int b;
};

int f(struct pair x) {
        return x.a + x.b;
}
```

Arguments to a function are laid out almost exactly like a struct. so assembly looks the same if you pass it a struct pair or two separate arguments.

f31.c

Generates same assembly code as f30.c
List of various configurations that give the same assembly code:
1) f(unsigned a, unsigned b)
2) f(struct pair)
3) f(int a, int b)
4) f(long long) <- Add 2 halves of the long long. x + (x >> 32) ← x >> 32 is the other half of the long long
5) f(struct pair x)
        where “struct pair” has an array in it (int x[2]) and we add x.x[0] and x.x[1] in f

Arrays are different! arrays are passed as pointers to the first element of the array. There must be an extra layer of dereferencing in the assembly.

f34.c

```
        movl        4(%esp), %edx
        movl        8(%esp), %eax
        movl        (%edx, %eax, r), %eax
        movl        12(%esp), %ecx
        addl        (%edx, %ecx, 4), %eax
        ret
```

4(%esp) is a pointer
8(%esp) is an index

12(%esp) is an index

(%edx,%ecx,4) means take %edx as a pointer and add %ecx * 4 to it. Can only miltiply by 1, 2, 4 or 8. Limitation of x86 architecture

f35.c

```
movl    4(%esp), %edx
movl    8(%esp), %eax
movl    4(%edx, %eax, 8), %eax ← Offset by 4.
movl    12(%esp), %ecx
addl    4(%edx, %ecx, 8), %eax <- Scale is now 8 rather than 4. Also offsetting by 4
ret
```

f36.c

```
movl    8(%esp), %eax
movl    4(%esp), %edx
leal    (%edx, %eax, 8), %eax ← pointer arithmetic
ret
```

What it does:
lea = load effective address
This is an address calculation. in lea it doesn't dereference the address it just uses the address.
Pointer arithmetic turns into an lea instruction.

movl = Dereference an address (1st argument) and store its value in the second argument
addl = Reads what's in each argument, adds them and stores it in the second argument
leal = Takes the first argument and stores it in the second. It does NOT dereference!