CS 354 - Machine Organization & Programming
Tuesday April 9th, and Thursday April 11th, 2023

Last Week

C, Assembly, & Machine Code Low-level View of Data Registers Operand Specifiers & Practice L18-7 Instructions - MOV, PUSH, POP	Operand/Instruction Caveats Instruction - LEAL Instructions - Arithmetic and Shift ---------- END of Exam 2 Material ---------- Instructions - CMP and TEST, Condition Codes

This Week: From L18: Instructions - SET, Jumps, Encoding Targets, Converting Loops	The Stack from a Programmer's Perspective The Stack and Stack Frames Instructions - Transferring Control Register Usage Conventions Function Call-Return Example

Next Week: Stack Frames
B&O 3.7 Intro - 3.7.5
3.8 Array Allocation and Access
3.9 Heterogeneous Data Structures

The Stack from a Programmer's Perspective

stack frames are also called activation records
caller and callee each have their own stack frames
stack grows down
top of stack has stack frame of the currently executing function
stack will have: local variables, return address, arguments

Consider the following code:

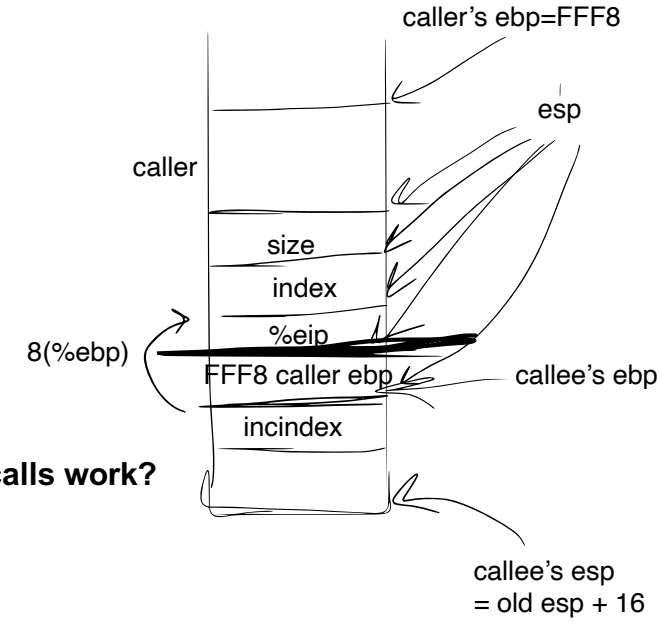
```
int inc(int index, int size) {
   int incindex = index + 1;
   if (incindex == size) return 0;
   return incindex;
}

int dequeue(int *queue, int *front,
        int rear, int *numitems, int size) {
   if (*numitem == 0) return -1;
   int dqitem = queue[*front];
   *front = inc(*front, size);
   *numitems -= 1;
   return dqitem;
}

int main(void) {
   int queue[5] = {11,22,33};
   int front = 0;
   int rear = 2;
   int numitems = 3;
   int qitem = dequeue(queue, &front, rear,
         &numitems, 5);
   ...
```

pushl %ebp (pushing caller stack frame to not mess up caller stack frame)
movl %esp, %ebp (setting up new stack frame)
subl $16, %esp (empty memory in stack for local variables, always in 16 bytes increments)

movl 8(%ebp), %eax (index into eax)
addl $1, %eax (add 1 to index)
movl %eax, -4(%ebp)



What does the compiler need to do to make function calls work?

- transfer to the callee
- handle passing arguments
- allocate and free stack frames
- make space for local variables
- handle return value
- preserve registers

how to return to caller

The Stack and Stack Frames

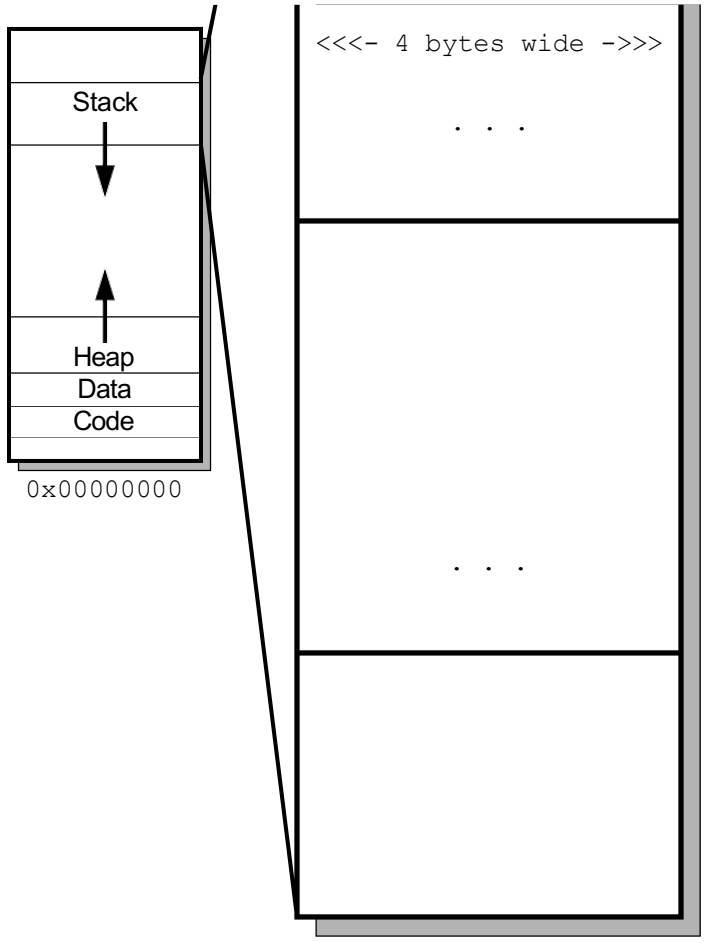
Stack Frame

IA-32:

%ebp extended base (stack) pointer: start of our stack frame

%esp extended stack pointer: tells us where the top of the stack is (eg push/pop will update it)

Stack Layout



Earlier Stack Frames (function X)

Caller's Stack Frame (function Y)

when we make a function call:
when call into callee, new stack frame is created
1. pushl %ebp (push old ebp onto stack)
2. mov %esp,%ebp (move curr stack pointer into ebp, move ebp to beginning of callee stack frame)
3. subl $12,%esp (move end of stack pointer, make room for 12 bytes in this stack frame)

Callee's Stack Frame (function Z)
(terminal function - doesn't call others)

args
return addr
caller ebp

A Callee's args

→ What is the offset from the %ebp to get to a callee's first argument?

8 bytes — jumping over return address and caller ebp

→ When are local variables allocated on the stack?

1. not enough registers
2. arrays, structs
3. reference the address of it

eg:
int i; (compiler can decide to keep in register and not make room on stack frame for it)
int *p = &i; (i has to have some mem addr, can be part of stack frame, cannot sit in register)

Instructions - Transferring Control

Flow Control

function call: like unconditional jump

```
call *Operand
call Label
```

steps (for both forms of call)

1. pushes on to the stack the return addr
   equivalent to something like: pushl %eip + 4 then jmp ____

call Label
eip pointer →

2.

function return:

`ret` popl return address then jmp ret addr

step

1.

Stack Frames

allocate stack frame:

make room for our local variables (subl $16, %esp)

free stack frame:

```
leave
```

steps

1. movl %ebp, %esp

2. popl %ebp (moves %ebp back to original and %ebp shifts up 4 as well)

Register Usage Conventions

Return Value %eax is how you return value to caller

%eax - return value

Frame Base Pointer `%ebp`

callee uses to access arguments 8(%ebp)

access local variables -4(%ebp)

Stack Pointer `%esp`

caller uses to store the callee's arguments
store the return address

push arguments, return address

callee uses to get return address
needs to setup its stack frame

Registers and Local Variables

→ Why use registers?

fast, but limited to 1,2, or 4 bytes

→ Potential problem with multiple functions using registers?

shared conflicts
caller/callee must agree

IA-32

caller-save: registers that caller is responsible for saving before making call (%eax, %ecx, %edx)

callee-save: %ebx, %edi, %esi callee has to gurantee these registers' values are preserved

Function Call-Return Example

```
int dequeue(int *queue, int *front, int rear, int *numitems, int size) {
   if (*numitem == 0) return -1;
   int dqitem = queue[*front];
   *front = inc(*front, size);
   *numitems -= 1;
   return dqitem;
}
int inc(int index, int size) {
   int incindex = index + 1;
   if (incindex == size) return 0;
   return incindex;
}
```

1ab setup calleE's args
2 call the calleE function
a save caller's return address
b transfer control to calleE
7 caller resumes, assigns return value

3 allocate callee's stack frame
a save calleR's frame base
b set callee's frame base
c set callee's top of stack
4 callee executes ...

5 free callee's stack frame
a restore calleR's top of stack
b restore calleR's frame base
6 transfer control back to calleR

CALL code in `dequeue`

1a `0x0_2C movl` index`,(%esp)`

b `0x0_2E movl` size`,4(%esp)`

2 `0x0_30 call inc`

a

b

RETURN code in `dequeue`

7 `0x0_55 movl %eax,(%ebx)`

CALL code in `inc`

3a `0x0_F0 pushl %ebp`

b `0x0_F2 movl %esp,%ebp`

c `0x0_F4 subl $12,%esp`

4 `0x0_F6` execute `inc` function's body

RETURN code in `inc`

5 `0x0_FA leave`

a

b

6 `0x0_FB ret`

Function Call-Return Example

Execution Trace of Stack and Registers