CS 354 - Machine Organization & Programming
Tuesday, November 12, 2019

Project p4b (~4%): DUE TOMORROW at 10 pm on Wednesday, November 13th

Last Time

The Stack from a Programmer's Perspective
The Stack and Stack Frames
Instructions - Transferring Control

Today

Register Usage Conventions
Function Call-Return Example
Recursion
Stack Allocated Arrays

Next Time

More Arrays, Structures, Pointers in Assembly
Read: B&O 3.8, 3.9
See pointers.pdf and recursion.pdf in Files section on course website

Register Usage Conventions

Return Value

%eax stores return value.

Frame Base Pointer %ebp

callee uses to access its argument 8(%ebp) offset > 0
access its locals -4(%ebp) offset < 0

Stack Pointer %esp

caller uses to set up arguments movl ARG, D(%esp)
save return address call

distance
D >= 0
0 : Arg 0
4 : Arg 1
8 : Arg 2

callee uses to restore return address ret
save/restore callee's %ebp pushl %ebp
popl %ebp.

Registers and Local Variables

→ Why use registers?
speed, but size is limited to 1, 2, 4 byte(s)

→ Potential problem with multiple functions using registers?
since registers are shared, conflicts can result.
caller & callee must have a consistent approach for saving & restoring registers.

IA-32

caller-save: %eax, %ecx, %edx
caller must save before calling a function.

callee-save: %ebx, %esi, %edi
callee must save before using these.

Function Call-Return Example

CALLER

```
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);
```

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

```
   *numitems -= 1;
   return dqitem;
}
```

CALLEE

```
int inc(int index, int size) {
```

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 ...

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

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

CALLER

CALL code in `dequeue`

1a `0x_07C movl` index, `(%esp)`

b `0x_07E movl` size, `4(%esp)`

2 `0x_080 call inc`

a pushl %eip (push program counter)

b jump 0x110 inc( )

RETURN code in `dequeue`

7 `0x_085 movl %eax,(%ebx)`

CALLEE

CALL code in `inc`

3a `0x_110 pushl %ebp`

b `0x_112 movl %esp,%ebp`

c `0x_114 subl $12,%esp`

4 `0x_116` execute `inc` function's body

RETURN code in `inc`

5 `0x_11A leave`

a movel %ebp, %esp

b popl %ebp

6 `0x_11B ret`

Function Call-Return Example

Execution Trace of Stack and Registers

Stack bottom

0x_F90	main's frame
.	
.	
.	
0x_F70	0x_F90
0x_F6C	dequeue's frame
0x_F68	
0x_F64	
0x_F60	
0x_F5C	1b size Arg
0x_F58	1a index Arg
0x_F54	2a 0x_085 Return value.
0x_F50	3a 0x_F70 Svd ebp
0x_F4C	
0x_F48	
0x_F44	

%ebp (B)

32 bytes

%esp (S)

16 bytes

2a 3a 3b 3c 5a 5b

update to next addr in instruction

%eip 0x_07C

- 1a 0x_07c
- 1b 0x_080
- 2 0x_085
- 2b 0x_110
- 3a 0x_112
- 3b 0x_114
- 3c 0x_116
- 4 inc() execute. 0x_11A
- 5 0x_11B
- 6 0x_085
- 7 dequeue() Resume saving RF? val in %eax

%ebp 0x_F70

- 3b 0x_F50.
- 5b 0x_F70

%esp 0x_F58

- 2a 0x_F54
- 3a 0x_F50
- 3c 0x_F44
- 5a 0x_F50
- 5b 0x_F54
- 6 0x_F58

Recursion

Use a stack trace to determine the result of the call `fact(3)`: $3! = 3\times2\times1 = 6$

$N! = N * \underbrace{(N-1)!}_{\text{Recurse}}$

1 if $n <= 1$

```
int fact(int n) {
   int result;
   if (n <= 1) result = 1;      Base
   else        result = n * fact(n - 1);
   return result;
}
```

⑥ $\leftarrow 3! = 3 * 2!$

② $2 * 1!$

①

<u>direct recursion</u> when a func call itself

<u>recursive case</u> repeats the recursion

<u>base case</u> stop recursion.

<u>"infinite" recursion</u> is like an infinite loop and results from a missing or bad base case

Assembly Trace

```
fact:
   pushl %ebp
   movl %esp, %ebp
   pushl %ebx
   subl $4,%esp

   movl 8(%ebp),%ebx
   movl $1,%eax

   cmpl $1,%ebx
   jle .L1

   leal -1(%ebx),%eax
   movl %eax,(%esp)
   call fact

   imull %ebx,%eax

.L1:
   addl $4,%esp
   popl %ebx
   popl %ebp
   ret
```

frame set up — callee save register

Arg = movl 8(%ebp),%ebx — n is in %ebx

local = movl $1,%eax — result is in %eax.

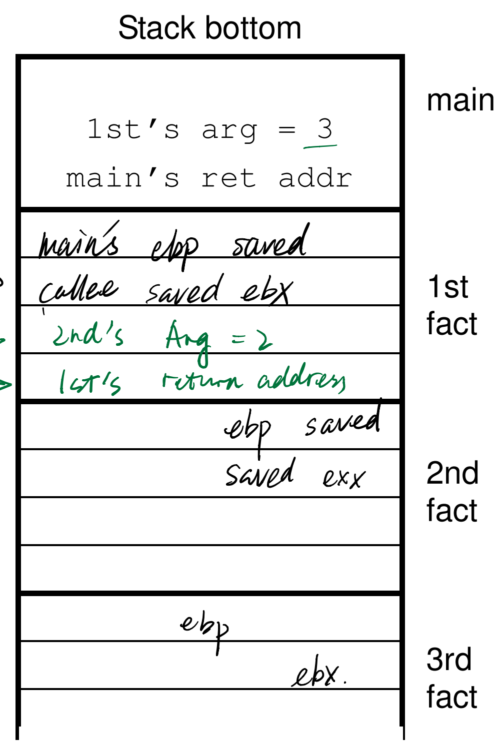
func body — cmpl / jle: > if n <= 1

leal: n-1

call fact: Recursive call.

imull: n * return value

frame break down



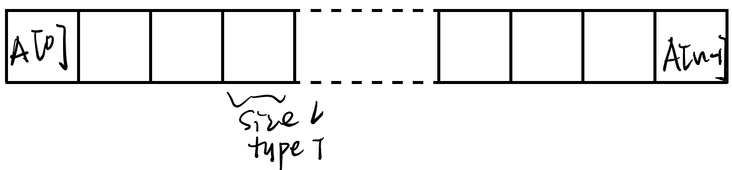
✵ "Infinite" recursion causes stack overflow since each recursive call consumes a bit more of stack

* when tracing func in assembly code, often you can skip over the instructions that manage stack frame. and focus on the func's value. (body)

Stack Allocated Arrays in C

Recall Array Basics

T `A[`*N*`];` where *T* is the element datatype of size *L* bytes and *N* is the number of elements



1. A contiguous region of stack of $L \times N$ bytes that is allocated but not initialized.
2. the identifier "A" is associated with the starting address of array.

✵ *The elements of* `A` are accessed addr arith which are implemented as mem type operands in ASM

Recall Array Indexing and Address Arithmetic

`&A[i]` $= A + i = X_A + L * i$
where X_A is starting addr of array
L is element size.

→ For each array declarations below, what is L (element size), the address arithmetic for the ith element, and the total size of the array?

C code	L	address of ith element	total array size
1. `int I[11]`	4	$X_I + 4*i$	44
2. `char C[7]`			
3. `double D[11]`			
4. `short S[42]`	2	$x_S + 2*i$	84
5. `char *C[13]`			
6. `int **I[11]`	4	$x_I + 4*i$	44
7. `double *D[7]`			

Stack Allocated Arrays in Assembly

Arrays on the Stack

→ How is an array laid out on the stack? Option 1 or 2?

higher addresses	
earlier frames	
1.	**2.**
A[0]	A[N-1]
A[1]	...
...	A[1]
A[N-1]	A[0]
Stack Top	

❋ *The first element (index 0) of an array*
is closed to the top of the stack.

Accessing 1D Arrays in Assembly

IA-32 address modes are designed simply array access.

Assume array's start address in %edx and index is in %ecx

```
movl (%edx, %ecx, 4), %eax
```

X_A i L

→ Assume `I` is an `int` array, `S` is a `short int` array, for both the array's start address is in %edx, and the index `i` is in %ecx. Determine the element type and instruction for each:

	C code	type	assembly instruction to move C code's value into %eax
→ 1.	`I`	int*	X_I movl %edx, %eax
→ 2.	`I[0]`	int	$M[X_I]$ movl (%edx), %eax first storing address.
3.	`*I`		
→ 4.	`I[i]`	int	$M[X_I + 4i]$ movl (%eax, %ecx, 4), %eax
5.	`&I[2]`		
6.	`I+i-1`		
– 7.	`*(I+i-3)`	int	$M[X_I + 4i - 4*3]$ movl -12(%eax, %ecx, 4), %eax
8.	`S[3]`		
9.	`S+1`		
– 10.	`&S[i]`	short*	$X_S + 2i$ leal (%edx, %ecx, 2), %eax.
11.	`S[4*i+1]`		
12.	`S+i-5`		