

CMSC 330: Organization of Programming Languages

Function Calls, Tail Recursion, Short Circuiting

Overview

- ▶ Function calls
- ▶ Tail recursion
- ▶ Short circuiting

CMSC 330

2

How Function Calls Really Work

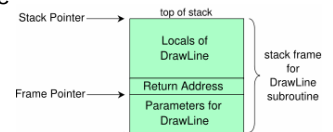
- ▶ Function calls are important
 - Usually have direct instruction support in hardware
 - Detail important for assembly language programming
 - See CMSC 212, 311, 412, or 430
- ▶ Will just provide quick overview here
- ▶ Key point to remember
 - Function calls generally require allocating stack frames

CMSC 330

3

Stack Frame / Activation Record

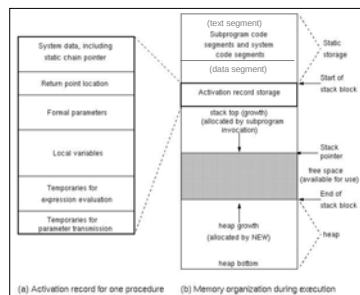
- ▶ Machine-dependent data structure containing state information for each function invocation
 - Allocated on stack at function invocation
 - Freed upon function return (by popping stack)
- ▶ Contents may include
 - Local variables
 - Return address
 - Actual parameters
 - Return value
 - Address of frame of calling function
 - Address of frame of lexically enclosing function



CMSC 330

4

Machine Model (Generic UNIX)



- ▶ The **text segment** contains the program's source code
- ▶ The **data segment** contains global variables, static data (data that exists for the entire execution and whose size is known), and the heap
- ▶ The **stack segment** contains the activation records for functions

CMSC 330

5

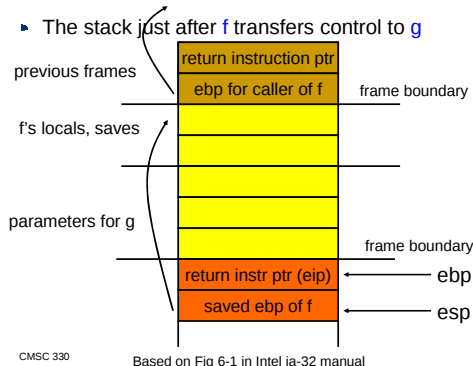
Machine Model (x86)

- ▶ The CPU has a fixed number of **registers**
 - Think of these as memory that's really fast to access
 - For a 32-bit machine, each can hold a 32-bit word
- ▶ Important x86 registers
 - **eax** generic register for computing values
 - **esp** pointer to the top of the stack
 - **ebp** pointer to start of current stack frame
 - **eip** the program counter (points to next instruction in text segment to execute)

CMSC 330

6

The x86 Stack Frame/Activation Record



x86 Calling Convention

- To call a function
 - Push parameters for function onto stack
 - Invoke `CALL` instruction to
 - Push current value of `eip` onto stack
 - I.e., save the program counter
 - Start executing code for called function
 - Callee pushes `ebp` onto stack to save it

CMSC 330

8

x86 Calling Convention (cont.)

- When a function returns
 - Put return value in `eax`
 - Invoke `LEAVE` to pop stack frame
 - Set `esp` to `ebp`
 - Restore `ebp` that was saved on stack and pop it off the stack
 - Invoke `RET` instruction to load return address into `eip`
 - I.e., start executing code where we left off at call

CMSC 330

9

Example

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

int main(void) {
    int x;

    x = f(3, 4);
}
```

`gcc -S a.c`

```
f:
    pushl    %ebp
    movl     %esp, %ebp
    movl     12(%ebp), %eax
    addl     8(%ebp), %eax
    leave    %eax
    ret

main:
    ...
    subl     $8, %esp
    pushl    $4
    pushl    $3
    call     f
1:  addl     $16, %esp
    movl     %eax, -4(%ebp)
    leave    %eax
    ret
```

CMSC 330

10

Lots More Details

- A whole lot more to say about calling functions
 - Local variables are allocated on stack by the callee as needed
 - This is usually the first thing a called function does
 - Saving registers
 - If the callee is going to use `eax` itself, you'd better save it to the stack before you call
 - Passing parameters in registers
 - More efficient than pushing/popping from the stack
 - Etc...
- Details covered in other courses

CMSC 330

11

Tail Calls

- A **tail call** is a function call that is the last thing a function does before it returns
 - Not just function call in last line of code in function

```
let add x y = x + y
let f z = add z z (* tail call *)
```

```
let rec len = function
[] -> 0
| (_::t) -> 1 + (len t) (* not tail call, performs +1 *)
```

```
let rec len a = function
[] -> a
| (_::t) -> len (a + 1) t (* tail call *)
```

CMSC 330

12

Tail Recursion

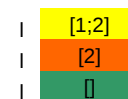
- Recall that in OCaml, all looping is via recursion
 - Seems very inefficient
 - Needs one stack frame for each recursive call
- A function is **tail recursive**
 - If it is recursive
 - And recursive call is a tail call
- If function is tail recursive
 - Can reuse stack frame for each recursive call

CMSC 330

13

Tail Recursion (cont.)

```
let rec len l = match l with
  [] -> 0
  | (_::t) -> 1 + (len t)
len [1; 2]
```



eax: 2

- Function is not tail recursive
 - Use stack frame store return value
 - Add 1 to return value, use as new return value

CMSC 330

14

Tail Recursion (cont.)

```
let rec len a l = match l with
  [] -> a
  | (_::t) -> (len (a + 1) t)
len 0 [1; 2]
```



eax: 2

- Function is tail recursive
 - Same stack frame can be reused for the next call
 - Since we'd just pop it off and return anyway

CMSC 330

15

Short Circuiting

- Will OCaml raise a **Division_by_zero** exception?

```
let x = 0
if x != 0 && (y / x) > 100 then
  print_string "OCaml sure is fun"
if x == 0 || (y / x) > 100 then
  print_string "OCaml sure is fun"
```

- No: **&&** and **||** are short circuiting in OCaml
 - e1 && e2 evaluates e1. If false, it returns false. Otherwise, it returns the result of evaluating e2
 - e1 || e2 evaluates e1. If true, it returns true. Otherwise, it returns the result of evaluating e2

CMSC 330

16

Short Circuiting (cont.)

- C, C++, Java, and Ruby all short-circuit **&&**, **||**
- But some languages don't, like Pascal (although Turbo Pascal has an option for this):

```
x := 0;
...
if (x <> 0) and (y / x > 100) then
  writeln('Sure OCaml is fun');
```

- So this would need to be written as

```
x := 0;
...
if x <> 0 then
  if y / x > 100 then
    writeln('Sure OCaml is fun');
```

CMSC 330

17