CMSC 313
Introduction to Computer Systems
Lecture 26
Procedure Implementation

Alan Sussman
als@cs.umd.edu

Administrivia

- Project 6 due Thursday
  - questions?
- Practice final exam posted soon, with answers later
- Read Section 3.7 of Bryant and O'Hallaron
- Please do course evaluation, at www.CourseEvalUM.umd.edu

Section 3.7, Bryant and O'Hallaron

PROCEDURE IMPLEMENTATION

Y86 procedure-related instructions

Instruction	Effect	Description
`call Label`	push PC on stack PC ← `Label`	Call function
`ret`	PC ← pop value from stack	Return from function

- These two instructions are used to handle function calls in Y86
- **`call`** instruction pushes the return address on the stack, then jumps to the destination address
- **`ret`** instruction returns from a call
- functions are responsible for ensuring the stack is properly adjusted before returning

Implementing functions

- When calling a function, the computer needs to know where execution will resume once the function returns (the *return address*)
- The **`call`** instruction stores this address on the stack; when the called function (the *callee*) is done, we'll need to undo any changes made to the stack so that we can get back to the caller

Call stack

- **`%esp`** contains the address of the top of the stack (the address of the element most recently pushed)
- The stack grows downward, from a maximum address of 0x1000
- To use it, an assembly program must first initialize the stack pointer:
  `irmovl $0x1000, %esp`
- Then, it can push and pop as necessary

Simple function call example

```
int n = 2;

void f();
void g();

int main() {
  f();
  n = 1;
  g();
  return 0;
}
void f() {
  g();
}
void g() {
  while (n > 0)
    printf("%d", n--);
}
```

```
main:    irmovl $0x1000, %esp
         call   f
         irmovl $1, %ebx
         rmmovl %ebx, n
         call   g
         halt
f:       call   g
         ret
g:       mrmovl n, %ecx
Loop:    irmovl $0, %eax
         addl   %eax, %ecx
         jle    EndLoop
         wrint  %ecx
         irmovl $1, %edx
         subl   %edx, %ecx
         rmmovl %ecx, n
         jmp    Loop
EndLoop: ret
         .align 4
n:       .long 2
```


Simple function call example

```
0x000: 308400100000 | main:    irmovl $0x1000, %esp
0x006: 801d000000   |          call   f
0x00b: 308301000000 |          irmovl $1, %ebx
0x011: 40384c000000 |          rmmovl %ebx, n
0x017: 8023000000   |          call   g
0x01c: 10           |          halt
0x01d: 8023000000   | f:       call   g
0x022: 90           |          ret
0x023: 50184c000000 | g:       mrmovl n, %ecx
0x029: 308000000000 | Loop:    irmovl $0, %eax
0x02f: 6001         |          addl   %eax, %ecx
0x031: 714b000000   |          jle    EndLoop
0x036: f318         |          wrint  %ecx
0x038: 308201000000 |          irmovl $1, %edx
0x03e: 6121         |          subl   %edx, %ecx
0x040: 40184c000000 |          rmmovl %ecx, n
0x046: 7029000000   |          jmp    Loop
0x04b: 90           | EndLoop: ret
0x04c:              |          .align 4
0x04c: 02000000     | n:       .long 2
```

Simple recursion example

```
int n = 3;

void f();

int main() {
  f();
  return 0;
}
void f() {
  if (n > 0) {
    printf("%d", n--);
    f();
  }
}
```

```
main:   irmovl $0x1000, %esp
        call   f
        halt
f:      mrmovl n, %ecx
        irmovl $0, %eax
        addl   %eax, %ecx
        jle    EndIf
        wrint  %ecx
        irmovl $1, %eax
        subl   %eax, %ecx
        rmmovl %ecx, n
        call   f
EndIf:  ret
        .align 4
n:      .long 3
```

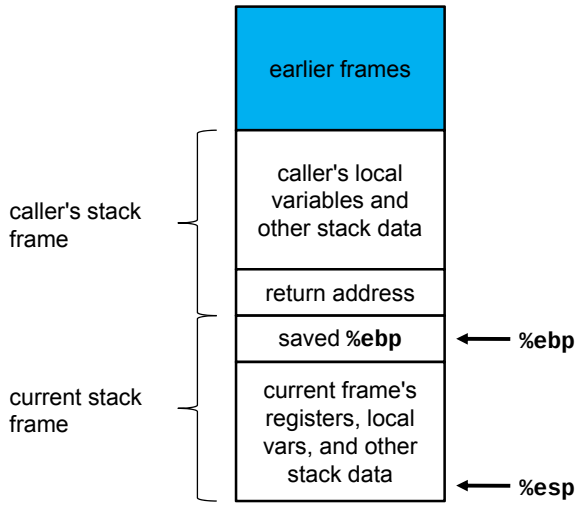

Simple recursion example

```
0x000: 308400100000 | main:   irmovl $0x1000, %esp
0x006: 800c000000   |         call   f
0x00b: 10           |         halt
0x00c: 501838000000 | f:      mrmovl n, %ecx
0x012: 308000000000 |         irmovl $0, %eax
0x018: 6001         |         addl   %eax, %ecx
0x01a: 7134000000   |         jle    EndIf
0x01f: f318         |         wrint  %ecx
0x021: 308001000000 |         irmovl $1, %eax
0x027: 6101         |         subl   %eax, %ecx
0x029: 401838000000 |         rmmovl %ecx, n
0x02f: 800c000000   |         call   f
0x034: 90           | EndIf:  ret
0x038:              |         .align 4
0x038: 03000000     | n:      .long 3
```


Local automatic variables

- These are also kept on the stack
- Multiple versions of the "same" variable can be kept; think of local variables in a recursive function
- We organize the stack into *frames* - one frame exists for each active (not yet returned) function call
  - We use the register **%ebp** to point to the first element in the current frame (the *frame pointer*)

Stack frames



Leaving a function

- x86 has a **`leave`** instruction, to make things easier for assembly programmers just before a return
- The equivalent Y86 code is:

```
rrmovl %ebp, %esp  # give up current frame
popl %ebp  # restore caller's frame pointer
```

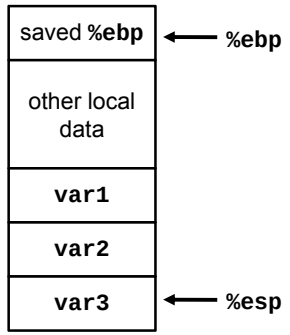
- This puts the stack pointer in the correct position so that a subsequent **`ret`** instruction will return to the correct location

Storing local variables

- These are stored at the top of the stack (nearest to **`%esp`**)
- Inside the current frame:

```
void f() {
  int var1, var2, var3;
  ...
}
```

- Accessed by offsets from **`%esp`**



Local variable example

```
void f();

int main() {
  f();
  f();
  return 0;
}

void f() {
  int a, b, c, d;
  a = 30;
  b = 25;
  c = a + b;
  printf("%d", c);
  d = 300;
}
```

```
main:   irmovl $0x1000, %esp  # init stack ptr
        call   f
        call   f
        halt

f:      pushl  %ebp           # save old frame ptr
        rrmovl %esp, %ebp     # set new frame ptr
        irmovl $16, %eax      # sub 16 (4 ints)
        subl   %eax, %esp     #  from stack ptr
        irmovl $30, %eax      # a = 30
        rmmovl %eax, 12(%esp) # store a in stack
        irmovl $25, %eax      # b = 25
        rmmovl %eax, 8(%esp)  # store b in stack
        mrmovl 12(%esp), %eax # load a
        mrmovl 8(%esp), %ecx  # load b
        addl   %eax, %ecx     # c = a + b
        rmmovl %ecx, 4(%esp)  # store c in stack
        mrmovl 4(%esp), %eax  # load c
        wrint  %eax           # print c
        irmovl $300, %eax     # d = 300
        rmmovl %eax, 0(%esp)  # store d in stack
        rrmovl %ebp, %esp     # reset stack ptr
        popl   %ebp           # restore old frame
        ret
```


Passing parameters

- Similar to local variables, but are pushed on the stack in the calling frame
- Accessed by offsets of **`%ebp`**; a function has to reach into the calling frame to access these
- If you're just passing one or two parameters, you may be able to just store the values in registers (faster than stack storage), but we'll just cover stack usage here

Parameter example

```
void f(int);

int main() {
  f(72);
  printf("\n");
  return 0;
}

void f(int i) {
  printf("%d", i);
}
```

```
main: irmovl $0x1000, %esp  # init stack ptr
      irmovl $72, %eax      # load arg to reg
      pushl  %eax           # push arg on stack
      call   f
      irmovl $10, %eax      # printf("\n");
      wrch   %eax
      halt                  # return 0;
f:    pushl  %ebp           # save old frame ptr
      rrmovl %esp, %ebp     # set new frame ptr
      mrmovl 8(%ebp), %eax  # load i param
      wrint  %eax           # print i
      rrmovl %ebp, %esp     # reset stack ptr
      popl   %ebp           # restore frame ptr
      ret
```