UNM SCHOOL *of* ENGINEERING

Department of Computer Science

Lect #12 – part 1 Wrap up Procedures (3.7) + Arrays and structs (3.8)

Prof. Soraya Abad-Mota, PhD

Previously

- Covered the stack discipline to allow procedure calls and returns
- For a harmonious transition between function calls there needs to be a protocol for using the registers so that we don't lose information stored in them → caller-saved and callee-saved registers
- To reinforce these concepts, review slides 39 to 44 from last lecture looking at what happens to the stack, and 45 to 50 for register saving conventions.

Can you answer these questions?

- Where do registers “live”?
- Why do we need to classify registers in caller-saved or callee-saved?
- Where are local variables stored ? When? Why?
- Can registers be used for temporary storage?
- “When finished executing a function, the top of the stack holds the next instruction.” (True or false)

From questions in the index cards

- Size of the stack → a few mega bytes (Mbytes)
  - In Linux, you may use `ulimit -s` to see & change it
- Why didn't the `am I` function called itself the second time around? (On the sequence of slides 25-36)
- `%rsp` always points to the top of the stack (whatever is there, what can be in the stack?)
- The stack is storage in main memory, handled in a special way (stack discipline). You don't call the stack, it is not a function.
- Confusion about caller and callee saved, the registers don't establish the rules.

Recall Example

Maybe main called multstore;
%rbx, callee- or caller-saved?

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx            # Save %rbx
  400541: mov    %rdx,%rbx       # Save dest
  400544: callq  400550 <mult2>  # mult2(x,y)
  400549: mov    %rax,(%rbx)     # Save at dest
  40054c: pop    %rbx            # Restore %rbx
  40054d: retq                   # Return
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

Today (wrap-up 3.7 + sec. 3.8)

Part 1:

- Begin with Problem 3.34 (have you done it?)
- Emphasize stack discipline and use of registers
- Study on your own slides 57 to 64
- Conclusions about recursion and procedures (65 and 66)

Part 2:

- Arrays and pointer arithmetic

Part 1: Wrap-up Procedures (sec. 3.7)

1. Finish Problem 3.34 from textbook (p. 252)
   - look at the assembly code for a portion of P
   - draw the stack from the portion shown immediately before call to Q
   - show what happens in the stack at the call to Q
2. Study slides 55 to 61 to see an illustration of how recursive procedures are handled (on your own, slowly)
3. Conclusions about recursion (slide 65)
4. Summary about procedure handling (slide 66)

Practice

- Problem 3.34 (p. 252)
  - typo in this problem: there are eight local values, named a0 to a7 (not a8 as stated in the book)

Problem 3.34 (p. 252)

A function P, generates local value a0-a7. Calls function Q using these values as arguments.

```
long P(long x)  /* x in %rdi */
```

`P` has 20 lines of assembly code.

A. Identify which local values get stored in callee-saved registers.
B. Identify which local values get stored on the stack.
C. Explain why the program could not store all of the local values in callee-saved registers.
D. Draw the stack before and after each call to Q (not required in the textbook, but very relevant)

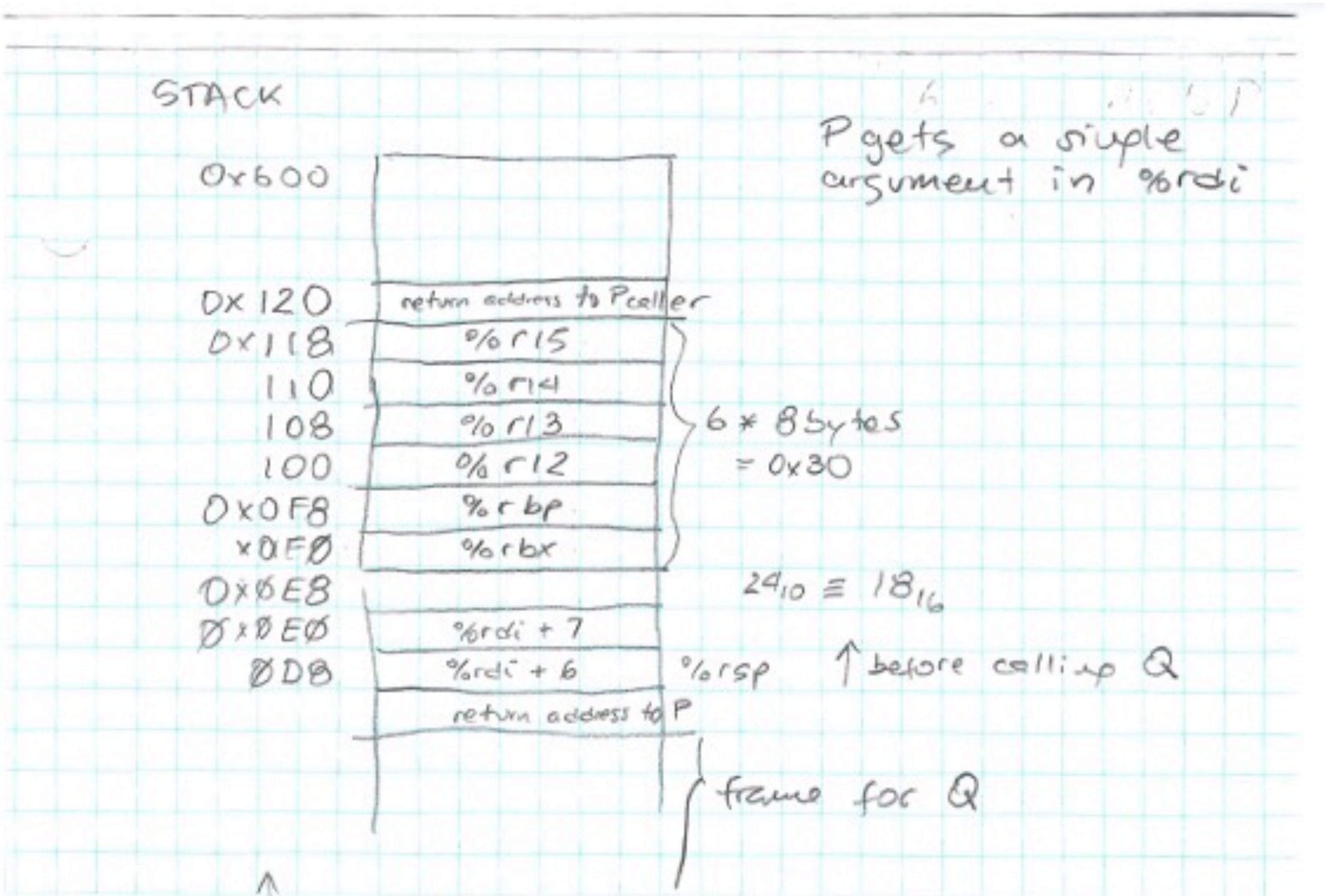
P:

(PracP. 3.34 p. 253)

```
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %rbp
pushq %rbx
subq  $24,%rsp
movq  %rdi,%rbx
leaq  1(%rdi), %r15
leaq  2(%rdi), %r14
leaq  3(%rdi), %r13
leaq  4(%rdi), %r12
```

```
leaq 5(%rdi),%rbp
 leaq 6(%rdi),%rax
movq %rax, (%rsp)
leaq 7(%rdi),%rdx
movq %rdx,8(%rsp)
movl $0, %eax
call  Q
```

x is in %rdi param of P


STACK
P gets a single argument in %rdi
0x600
0x120
return address to Pcaller
0x118
%r15
110
%r14
108
%r13
100
%r12
0x0F8
%rbp
x0F0
%rbx
6 * 8 bytes = 0x30
0x0E8
$24_{10} \equiv 18_{16}$
0x0E0
%rdi + 7
0D8
%rdi + 6
%rsp
before calling Q
return address to P
frame for Q

Subsection 3.7.6 (Recursion)

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - **Illustration of Recursion (Sec 3.7.6)**
    - please follow the next 7 slides (63 to 69) on your own, (example of recursive procedures and how they are handled with the same stack discipline)
    - last one is a summary slide on observations about recursion
      - JUMP to slide 68

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

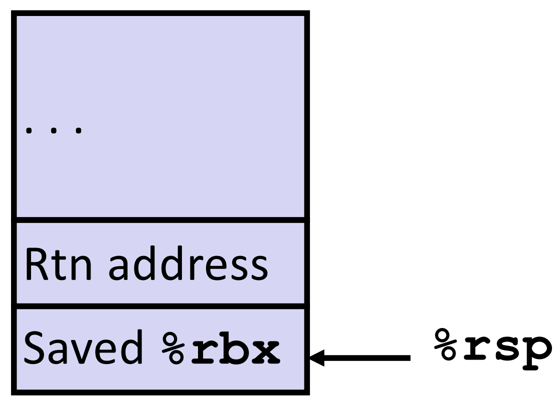
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
`%rbx`	`x & 1`	Callee-saved
`%rax`	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

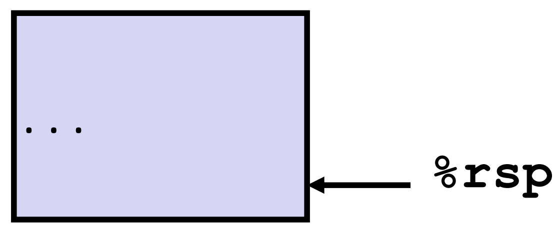
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value

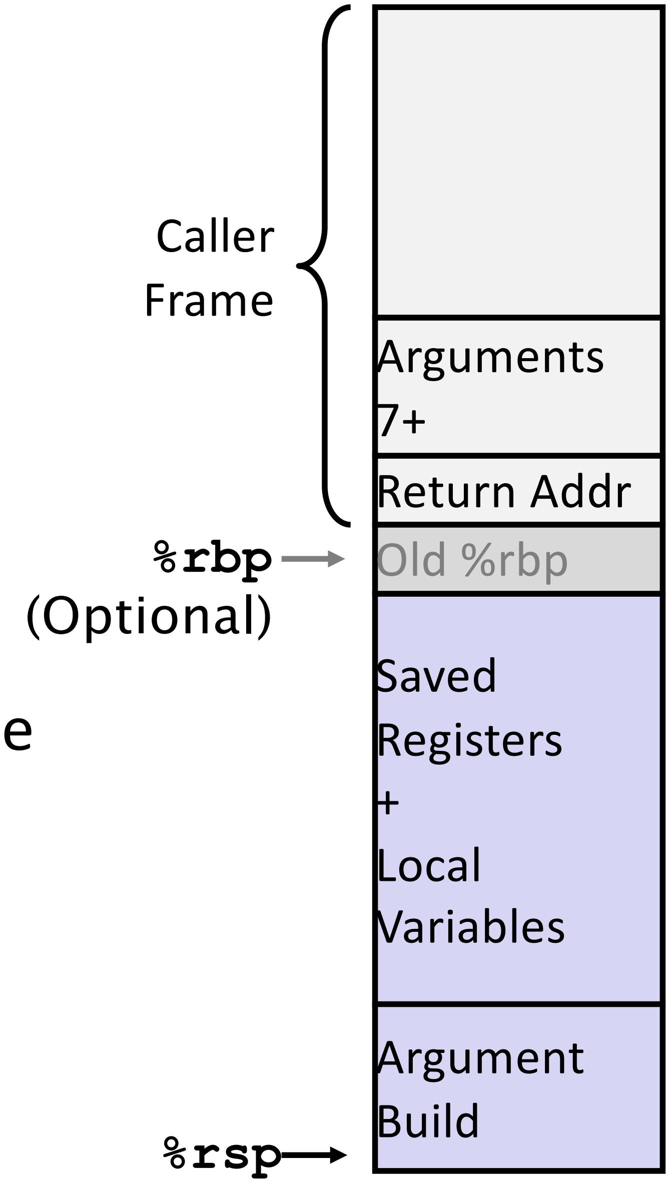


Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow)
    - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P

x86-64 Procedure Summary

- Important Points
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in **`%rax`**
- Pointers are addresses of values
  - On stack or global



End of section 3.7