IF2130 – Organisasi dan Arsitektur Komputer

sumber: Greg Kesden, CMU 15-213, 2012

Machine-Level Programming: Procedure

Today

- Switch statements
- ► IA 32 Procedures
 - Stack Structure
 - Calling Conventions
 - Illustrations of Recursion & Pointers



```
long switch eg
   (long x, long y, long z)
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break:
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w = z;
        break;
    default:
        w = 2;
    return w;
```

Switch Statement Example

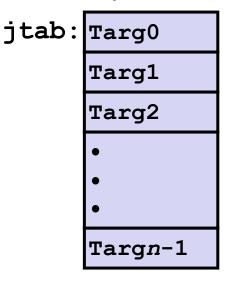
- Multiple case labels
 - Here: 5 & 6
- ▶ Fall through cases
 - Here: 2
- Missing cases
 - Here: 4

Jump Table Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n-1
}
```

Jump Table



Jump Targets

Targ0: Code Block 0

Targ1: Code Block

Targ2: Code Block 2

Approximate Translation

```
target = JTab[x];
goto *target;
```

Targn-1:

Code Block n-1

Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

What range of values takes default?

Setup:

```
switch eg:
  pushl
          %ebp
                           Setup
  movl
          %esp, %ebp
                           Setup
          8 (%ebp), %eax
                         # %eax = x
  movl
          $6, % eax
  cmpl
                         # Compare x:6
          .L2
                           If unsigned > goto default
  jа
                         # Goto *JTab[x]
          *.L7(,%eax,4)
  jmp
                                             Note that w not
                                             initialized here
```

Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

Indirect

jump

jmp

```
switch_eg:
    pushl %ebp # Setup
    movl %esp, %ebp # Setup
    movl 8(%ebp), %eax # eax = x
    cmpl $6, %eax # Compare x:6
    ja .L2 # If unsigned > goto default
```

*.L7(,%eax,4) # Goto *JTab[x]

Jump table

```
.section
           .rodata
  .align 4
.L7:
  .long
           .L2 \# x = 0
           .L3 \# x = 1
  .long
           .L4 \# x = 2
  .long
  .long
           .L5 \# x = 3
 .long
           .L2 \# x = 4
  .long
           .L6 \# x = 5
  .long
           .L6 \# x = 6
```

Assembly Setup Explanation

Table Structure

- Each target requires 4 bytes
- Base address at .L7

Jumping

- Direct: jmp .L2
- Jump target is denoted by label . L2
- Indirect: jmp *.L7(,%eax,4)
- Start of jump table: . L7
- Must scale by factor of 4 (labels have 32-bits = 4 Bytes on IA32)
- Fetch target from effective Address .L7 + eax*4
 - Only for $0 \le x \le 6$

Jump table

```
.section
          .rodata
 .align 4
.L7:
  .long .L2 \# x = 0
          .L3 \# x = 1
 .long
          .L4 \# x = 2
 .long
          .L5 \# x = 3
  .long
          .L2 \# x = 4
  .long
  .long
          .L6 \# x = 5
          .L6 \# x = 6
  .long
```

Jump Table

Jump table

```
.section .rodata
  .align 4
.L7:
  .long .L2 # x = 0
  .long .L3 # x = 1
  .long .L4 # x = 2
  .long .L5 # x = 3
  .long .L2 # x = 4
  .long .L6 # x = 5
  .long .L6 # x = 6
```

```
switch(x) {
case 1: // .L3
   w = y*z;
   break;
case 2: // .L4
   w = y/z;
   /* Fall Through */
case 3: // .L5
   w += z;
   break;
case 5:
case 6: // .L6
   w -= z;
   break;
default: // .L2
   w = 2;
```

Handling Fall-Through

```
long w = 1;
                              case 3:
                                      w = 1;
switch(x) {
                                      goto merge;
case 2:
   w = y/z;
    /* Fall Through */
case 3:
                                              case 2:
   w += z;
                                                  w = y/z;
   break;
                                          merge:
                                                  w += z;
```



Code Blocks (Partial)

```
switch(x) {
case 1: // .L3
  w = y*z;
  break;
case 3: // .L5
 w += z;
 break;
. . .
default: // .L2
  w = 2;
```

```
.L2: # Default
mov1 $2, %eax # w = 2
 jmp .L8 # Goto done
.L5: \# x == 3
 movl $1, %eax # w = 1
  jmp .L9 # Goto merge
.L3: \# x == 1
 movl 16(%ebp), %eax # z
 imull 12 (%ebp), %eax \# w = y*z
  jmp .L8 # Goto done
```

Code Blocks (Rest)

```
switch(x) {
 case 2: // .L4
   w = y/z;
    /* Fall Through */
 merge: // .L9
   w += z;
   break;
 case 5:
 case 6: // .L6
   w -= z;
   break;
```

```
.L4: \# x == 2
  movl 12(%ebp), %edx
 movl %edx, %eax
 sarl $31, %edx
  idivl 16(%ebp) # w = y/z
.L9: # merge:
 addl 16(\%ebp), \%eax # w += z
  jmp .L8 # goto done
.L6: \# x == 5, 6
 movl $1, %eax # w = 1
  subl 16(\%ebp), \%eax # w = 1-z
```

x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- ▶ Table entries 64 bits (pointers)
- Cases use revised code

```
.L3:

movq %rdx, %rax

imulq %rsi, %rax

ret
```

Jump Table

```
.section .rodata
 .align 8
.L7:
 . quad
         .L2 \# x = 0
         .L3 \# x = 1
 . quad
 .quad .L4 \# x = 2
         .L5 \# x = 3
 .quad
         .L2 \# x = 4
 . quad
         .L6 \# X = 5
 . quad
               \# x = 6
         .L6
 . quad
```



IA32 Object Code

Setup

- Label . L2 becomes address 0x8048422
- Label . L7 becomes address 0x8048660

Assembly Code

Disassembled Object Code



IA32 Object Code (cont.)

- Jump Table
 - Doesn't show up in disassembled code
 - Can inspect using GDB
 - gdb switch
 - \rightarrow (gdb) x/7xw 0x8048660
 - ▶ Examine 7 hexadecimal format "words" (4-bytes each)
 - ▶ Use command "help x" to get format documentation

0x8048660: 0x08048422 0x08048432 0x0804843b 0x08048429

0x8048670: 0x08048422 0x0804844b 0x0804844b

IA32 Object Code (cont.)

Deciphering Jump Table

0x8048660: 0x8048670: 0x08048422

 0×08048432

0x0804843b

0x08048429

0x08048422

0x0804844b

0x0804844b

Address	Value	x
0x8048660	0x8048422	0
0x8048664	0x8048432	I
0x8048668	0x804843b	2
0x804866c	0x8048429	3
0x8048670	0x8048422	4
0x8048674	0x804844b	5
0x8048678	0x804844b	6

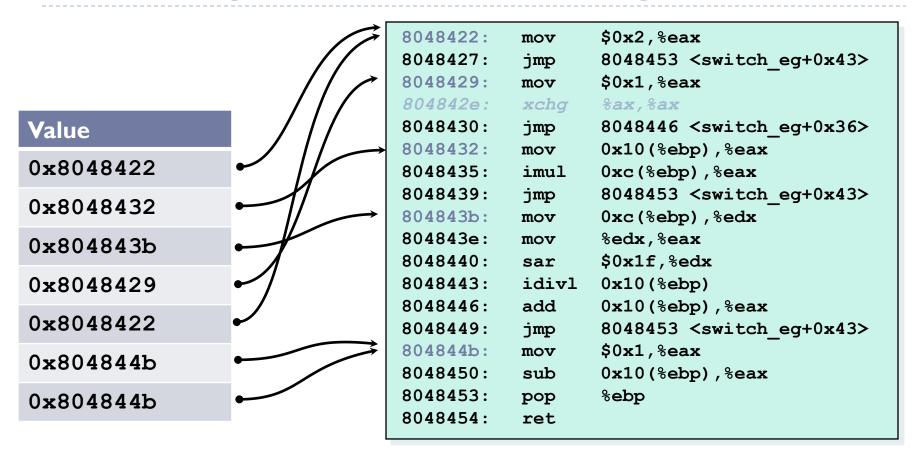


Disassembled Targets

```
8048422:
          ъ8 02 00 00 00
                                      $0x2, %eax
                               mov
8048427: eb 2a
                                      8048453 <switch eg+0x43>
                               jmp
8048429: b8 01 00 00 00
                                      $0x1, %eax
                               mov
804842e: 66 90
                               xchq %ax, %ax # noop
8048430: eb 14
                                      8048446 <switch eg+0x36>
                               jmp
8048432: 8b 45 10
                                      0x10(%ebp),%eax
                               mov
8048435: Of af 45 Oc
                               imul
                                      0xc(%ebp),%eax
8048439: eb 18
                                      8048453 < \text{switch eq} + 0x43 >
                               фmр
804843b: 8b 55 0c
                                      0xc(%ebp),%edx
                               mov
804843e: 89 d0
                                      %edx,%eax
                               mov
8048440: c1 fa 1f
                                      $0x1f,%edx
                               sar
8048443: f7 7d 10
                                      0x10(%ebp)
                               idivl
8048446: 03 45 10
                                      0x10(%ebp),%eax
                               add
8048449: eb 08
                                      8048453 <switch eg+0x43>
                               jmp
804844b: b8 01 00 00 00
                                      $0x1, %eax
                               mov
8048450: 2b 45 10
                                      0x10(%ebp),%eax
                               sub
8048453:
         5d
                                      %ebp
                               pop
8048454:
          c3
                               ret
```



Matching Disassembled Targets





Summarizing

C Control

- if-then-else
- do-while
- while, for
- switch

Assembler Control

- Conditional jump
- Conditional move
- Indirect jump
- Compiler generates code sequence to implement more complex control

Standard Techniques

- Loops converted to do-while form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees

Today

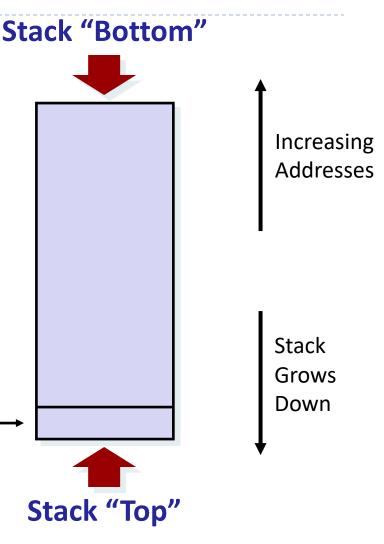
- Switch statements
- ▶ IA 32 Procedures
 - Stack Structure
 - Calling Conventions
 - Illustrations of Recursion & Pointers

IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register %esp contains lowest stack address
 - address of "top" element

Stack Pointer: %esp





IA32 Stack: Push

pushl Src

- Fetch operand at Src
- Decrement %esp by 4
- Write operand at address given by %esp

Stack Grows Down **Stack Pointer: %esp** Stack "Top"

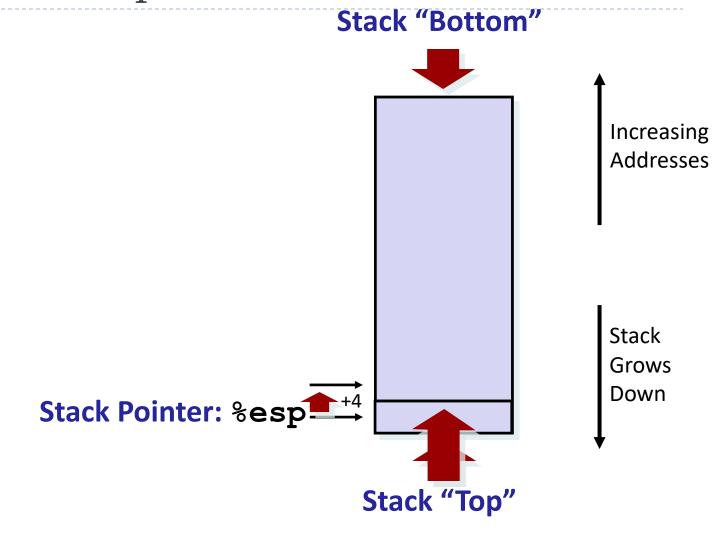
Stack "Bottom"

Increasing

Addresses



IA32 Stack: Pop





Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: call label
 - Push return address on stack
 - Jump to label
- Return address:
 - Address of the next instruction right after call
 - Example from disassembly

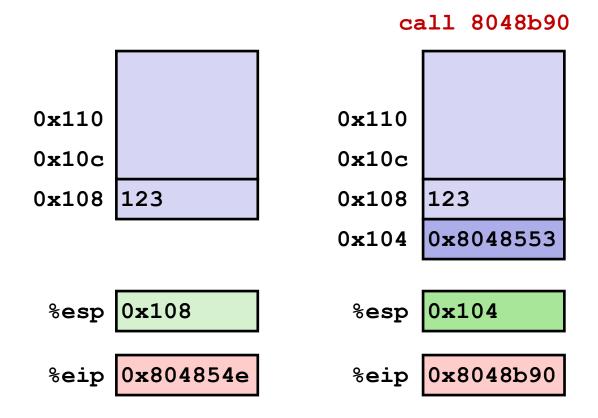
```
804854e: e8 3d 06 00 00 call 8048b90 <main> 8048553: 50 pushl %eax
```

- Return address = 0x8048553
- Procedure return: ret
 - Pop address from stack
 - Jump to address



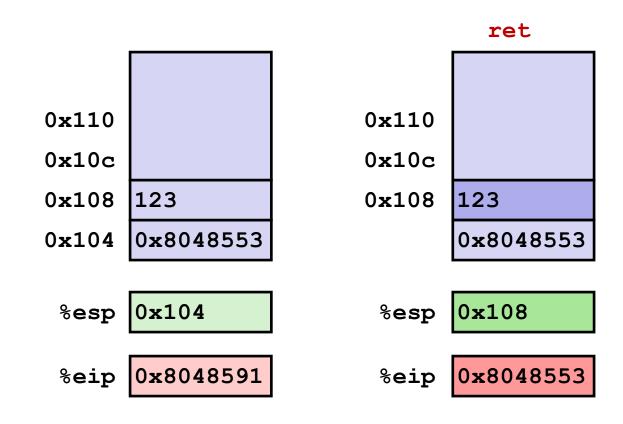
Procedure Call Example

804854e: e8 3d 06 00 00 call 8048b90 <main> 8048553: 50 pushl %eax



Procedure Return Example

8048591: c3 ret



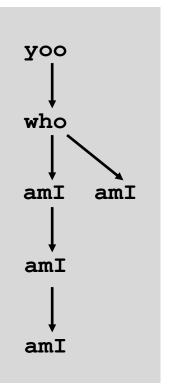
Stack-Based Languages

- Languages that support recursion
 - e.g., C, Pascal, Java
 - Code must be "Reentrant"
 - Multiple simultaneous instantiations of single procedure
 - Need some place to store state of each instantiation
 - Arguments
 - Local variables
 - Return pointer
- Stack discipline
 - State for given procedure needed for limited time
 - From when called to when return
 - Callee returns before caller does
- Stack allocated in *Frames*
 - state for single procedure instantiation

Call Chain Example

```
who (...)
{
    amI();
    amI();
    amI();
}
```

Example Call Chain



Procedure amI () is recursive



Stack Frames

- Contents
 - Local variables
 - Return information
 - Temporary space

Frame Pointer: %ebp

Frame for proc

Previous

Frame

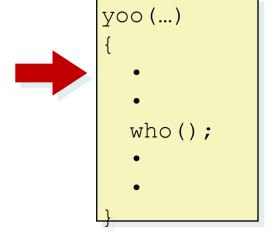
Stack Pointer: %esp

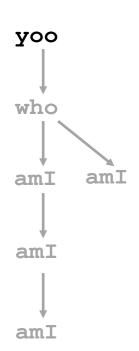


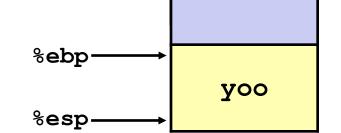
- Space allocated when enter procedure
 - "Set-up" code
- Deallocated when return
 - "Finish" code



Example







Stack

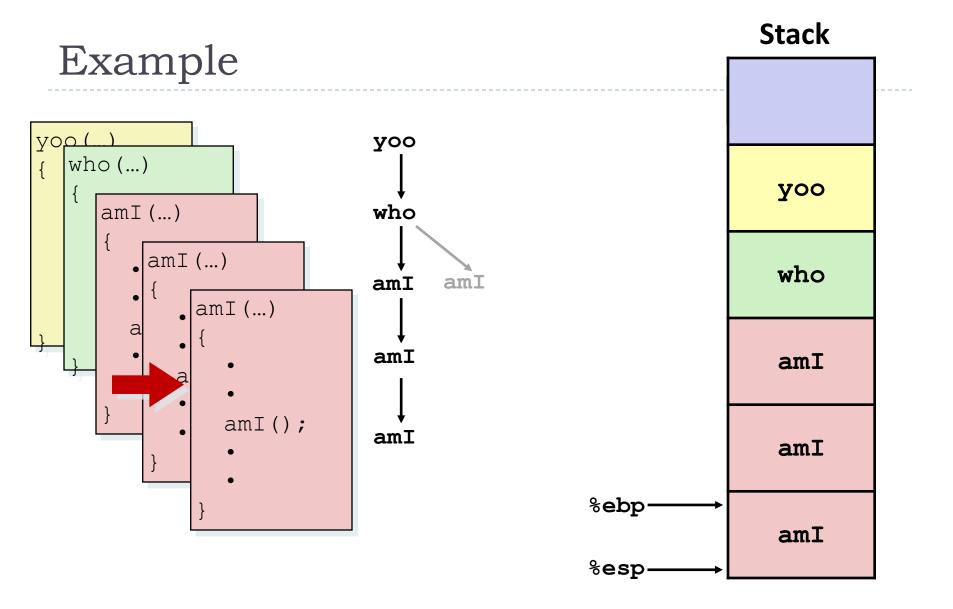
Example yoo() { who(...) { amI(); amI(); amI(); amI } esp Stack yoo who sebp who amI %esp

amI

Stack Example yop() yoo who (...) yoo amI (...) who who amIamI %ebpamI(); amI amI%esp-

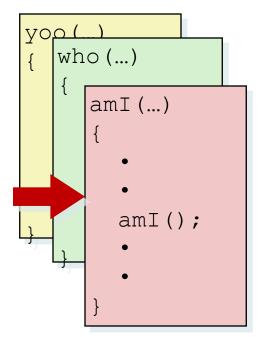
amI

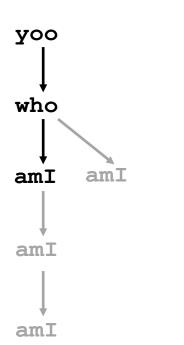
Stack Example yop () yoo who (...) yoo amI (...) who amI (...) who amIamI amI amIamI(); %ebp amI amI %esp

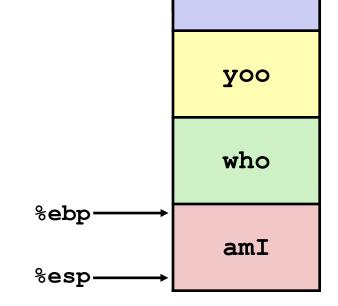


Stack Example yop() yoo who (...) yoo amI (...) who amI (...) who amIamI amI amIamI(); %ebp amI amI %esp

Example

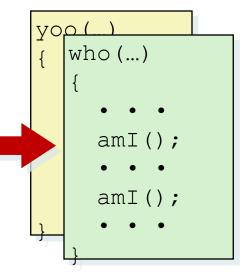


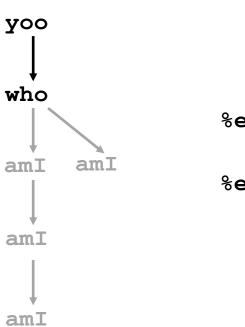




Stack

Example







yoo

%ebp-

%esp-

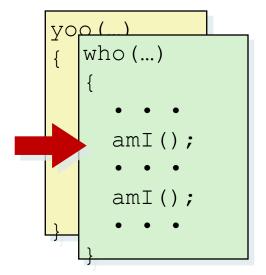
who

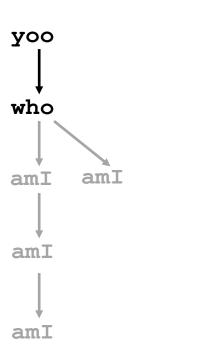
Example yoo () { who (...) } amI (); amI(); amI amI who amI amI

amI

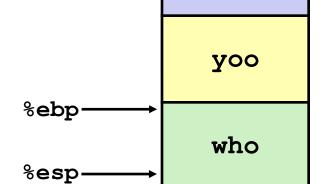
%esp-

Example

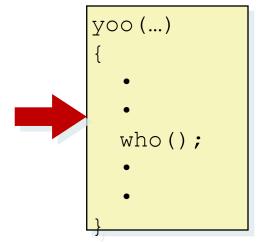




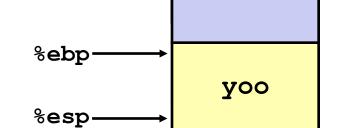




Example



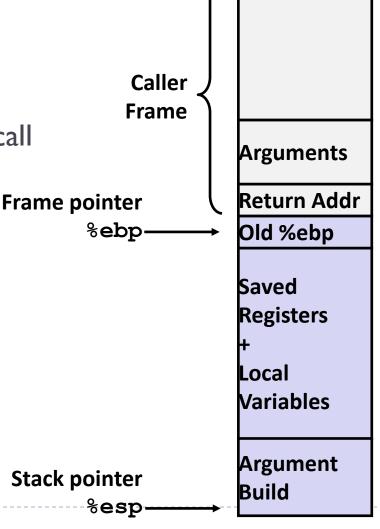




Stack

IA32/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
 - "Argument build:"Parameters for function about to call
 - Local variablesIf can't keep in registers
 - Saved register context
 - Old frame pointer
- Caller Stack Frame
 - Return address
 - Pushed by call instruction
- Arguments for this call



Revisiting swap

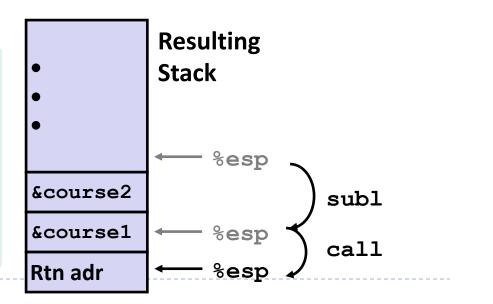
```
int course1 = 15213;
int course2 = 18243;

void call_swap() {
   swap(&course1, &course2);
}
```

Calling swap from call swap

```
call_swap:
    • • •
    subl $8, %esp
    movl $course2, 4(%esp)
    movl $course1, (%esp)
    call swap
    • • •
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Revisiting swap

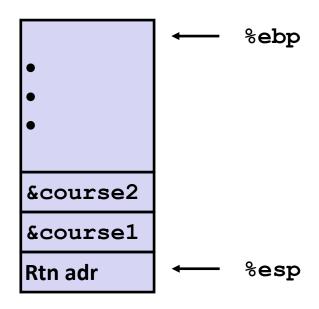
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

swap:

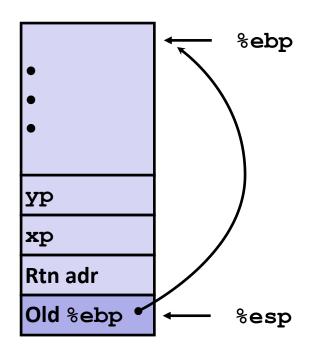
```
pushl %ebp
                      Set
movl %esp, %ebp
pushl %ebx
movl 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
                      Body
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)
popl %ebx
popl %ebp
                      Finish
ret
```

swap Setup #1

Entering Stack



Resulting Stack



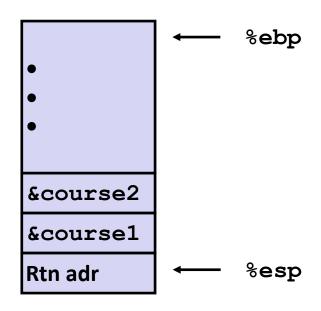
swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx

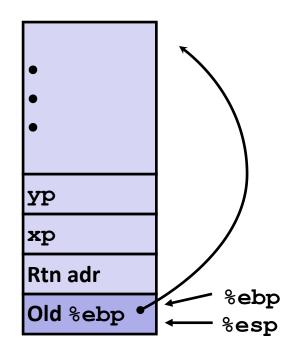


swap Setup #2

Entering Stack



Resulting Stack



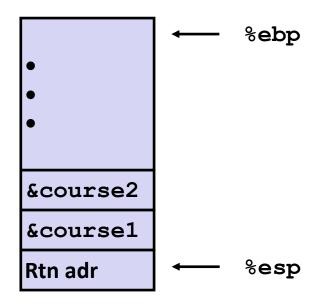
swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx



swap Setup #3

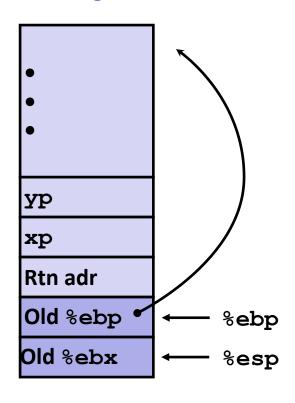
Entering Stack



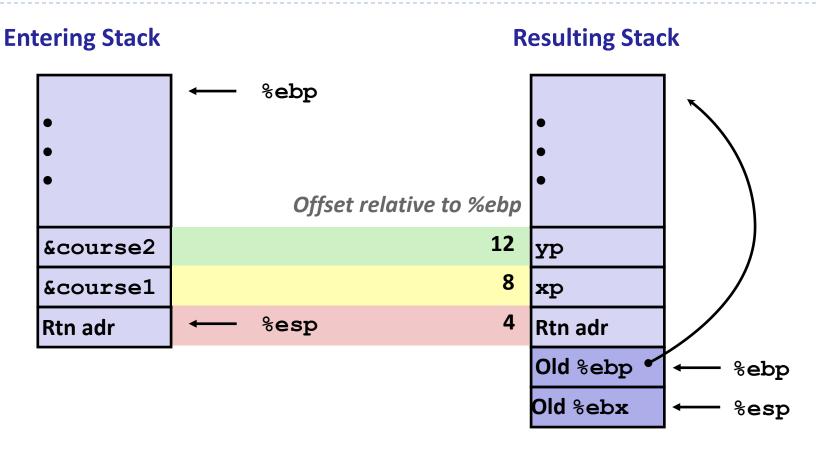
swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx

Resulting Stack



swap Body

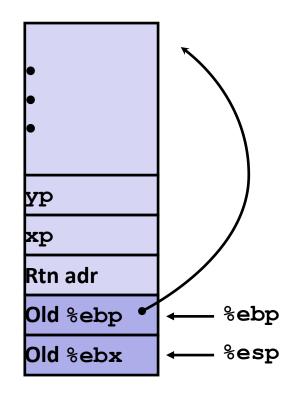


```
movl 8(%ebp),%edx # get xp
movl 12(%ebp),%ecx # get yp
```

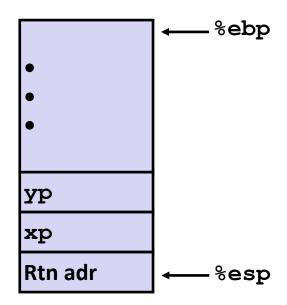
• • •

swap Finish

Stack Before Finish



Resulting Stack



Observation

%ebx

%ebp

popl

popl

- Saved and restored register %ebx
- Not so for %eax, %ecx, %edx

Disassembled swap

▶ 80483c9:

ret

```
08048384 <swap>:
  8048384:
            55
                                    push
                                            %ebp
  8048385: 89 e5
                                            %esp,%ebp
                                    mov
  8048387: 53
                                            %ebx
                                    push
  8048388: 8b 55 08
                                            0x8(%ebp),%edx
                                    mov
  804838b: 8b 4d 0c
                                            0xc(%ebp),%ecx
                                    mov
  804838e:
            8b 1a
                                            (%edx),%ebx
                                    mov
  8048390:
            8b 01
                                            (%ecx),%eax
                                    mov
  8048392:
            89 02
                                            %eax,(%edx)
                                    mov
            89 19
  8048394:
                                            %ebx, (%ecx)
                                    mov
  8048396:
            5b
                                            %ebx
                                    pop
  8048397:
            5d
                                            %ebp
                                    pop
  8048398:
            c3
                                    ret
Calling Code
  80483b4:
            movl
                    $0x8049658,0x4(%esp) # Copy &course2
  80483bc:
            movl
                    $0x8049654, (%esp)
                                         # Copy &course1
  80483c3:
            call
                    8048384 <swap>
                                         # Call swap
  80483c8:
            leave
                                          # Prepare to return
```

Return

Today

- Switch statements
- ► IA 32 Procedures
 - Stack Structure
 - Calling Conventions
 - Illustrations of Recursion & Pointers

Register Saving Conventions

- When procedure yoo calls who:
 - yoo is the *caller*
 - who is the callee
- Can register be used for temporary storage?

```
yoo:

movl $15213, %edx
call who
addl %edx, %eax

ret
```

- This could be trouble → something should be done!
 - Need some coordination



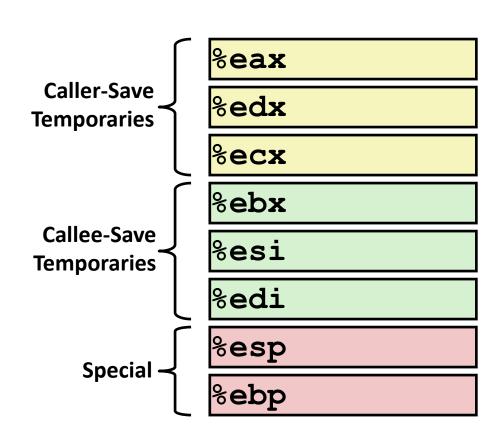
Register Saving Conventions

- When procedure yoo calls who:
 - yoo is the *caller*
 - who is the callee
- Can register be used for temporary storage?
- Conventions
 - "Caller Save"
 - Caller saves temporary values in its frame before the call
 - "Callee Save"
 - Callee saves temporary values in its frame before using



IA32/Linux+Windows Register Usage

- %eax, %edx, %ecx
 - Caller saves prior to call if values are used later
- > %eax
 - also used to return integer value
- %ebx, %esi, %edi
 - Callee saves if wants to use them
- %esp, %ebp
 - special form of callee save
 - Restored to original values upon exit from procedure



Today

- Switch statements
- ▶ IA 32 Procedures
 - Stack Structure
 - Calling Conventions
 - Illustrations of Recursion & Pointers

Recursive Function

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

Registers

- *eax, *edx used without first saving
- %ebx used, but saved at beginning & restored at end

```
pcount r:
   pushl %ebp
   movl %esp, %ebp
   pushl %ebx
    subl $4, %esp
   mov1 8 (%ebp), %ebx
   movl $0, %eax
    testl %ebx, %ebx
    ie .L3
   movl %ebx, %eax
    shrl %eax
   movl %eax, (%esp)
    call pcount r
   movl %ebx, %edx
    andl $1, %edx
    leal (%edx, %eax), %eax
.L3:
   addl $4, %esp
   popl %ebx
   popl %ebp
    ret
```



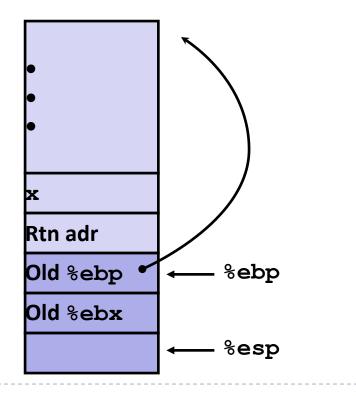
```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

Actions

- Save old value of %ebx on stack
- Allocate space for argument to recursive call
- Store x in %ebx

```
%ebx x
```

```
pcount_r:
    pushl %ebp
    movl%esp, %ebp
    pushl %ebx
    subl$4, %esp
    movl 8(%ebp), %ebx
    • • •
```





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

```
movl $0, %eax
testl %ebx, %ebx
je .L3

• • •
.L3:
```

Actions

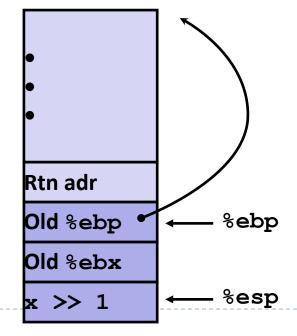
- If x == 0, return
 - ▶ with %eax set to 0

%ebx x

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

```
movl %ebx, %eax
shrl %eax
movl %eax, (%esp)
call pcount_r
```

- Actions
 - Store x >> I on stack
 - Make recursive call
- Effect
 - > %eax set to function result
 - > %ebx still has value of x



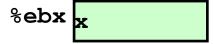




```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
movl %ebx, %edx
andl $1, %edx
leal (%edx,%eax), %eax
• • •
```

- Assume
 - %eax holds value from recursive call
- Actions
 - ▶ Compute (x & I) + computed value
- Effect
 - **%eax** set to function result





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

L3:

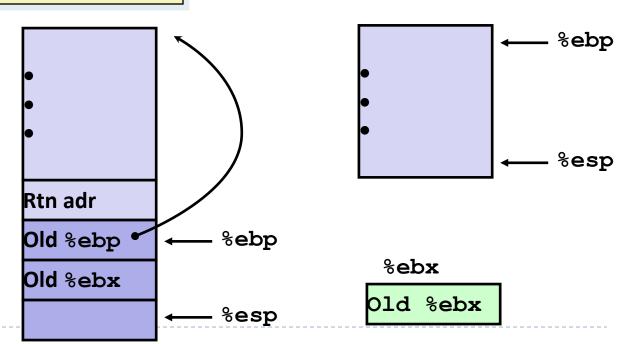
addl\$4, %esp

popl%ebx

popl%ebp

ret

- Actions
 - Restore
 values of
 %ebx and
 %ebp
 - % Restore
 % esp



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
 - 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



Pointer Code

Generating Pointer

```
/* Compute x + 3 */
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

Referencing Pointer

```
/* Increment value by k */
void incrk(int *ip, int k) {
   *ip += k;
}
```

add3 creates pointer and passes it to incrk

Creating and Initializing Local Variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Variable localx must be stored on stack
 - Because: Need to create pointer to it
- Compute pointer as -4(%ebp)

First part of add3

```
add3:
   pushl%ebp
   movl %esp, %ebp
   subl $24, %esp # Alloc. 24 bytes
   movl 8(%ebp), %eax
   movl %eax, -4(%ebp)# Set localx to x
```

```
8 x
4 Rtn adr
0 Old %ebp ← %ebp
-4 localx = x
-8
-12 Unused
-16
-20
-24 ← %esp
```

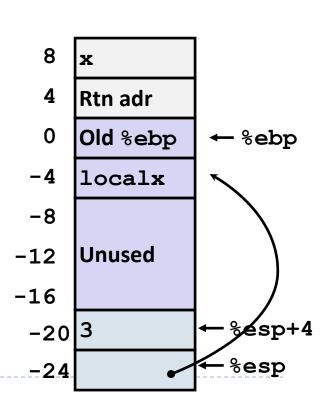
Creating Pointer as Argument

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

 Use leal instruction to compute address of localx

Middle part of add3

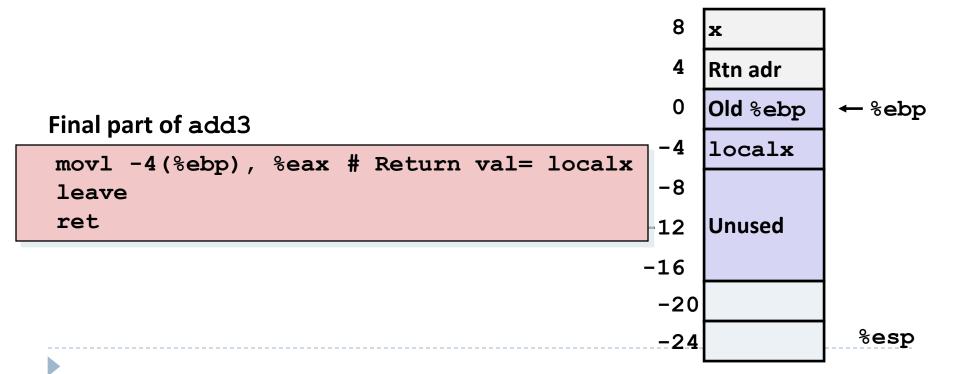
```
movl $3, 4(%esp) # 2<sup>nd</sup> arg = 3
leal -4(%ebp), %eax# &localx
movl %eax, (%esp) # 1<sup>st</sup> arg = &localx
call incrk
```



Retrieving local variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

Retrieve localx from stack as return value



IA 32 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 %eax
- Pointers are addresses of values
 - On stack or global

