Machine-Level Programming III: Procedures

Kai Shen

1

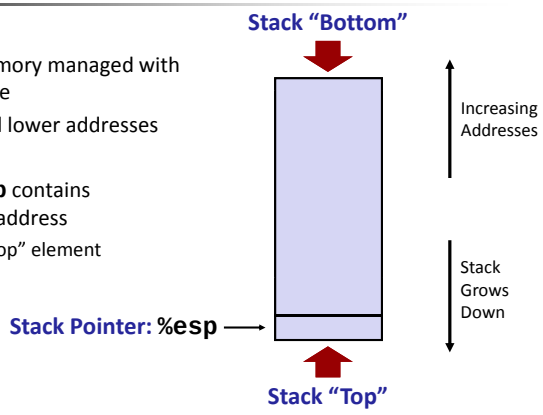
Outline: Procedures

- Stack Structure & Calling Conventions
- Illustrations of Recursion & Pointers
- X86-64 Procedures

2

IA32 Stack

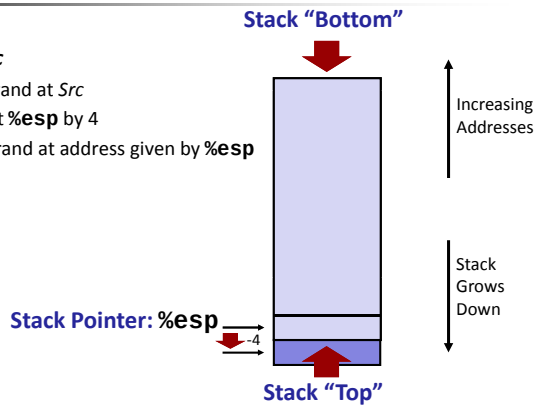
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register **%esp** contains lowest stack address
  - address of "top" element



3

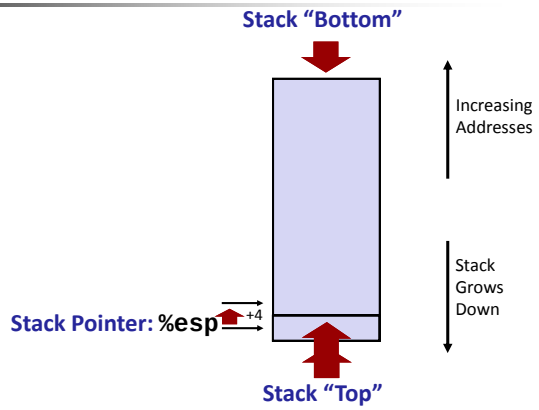
IA32 Stack: Push

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



4

IA32 Stack: Pop



5

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

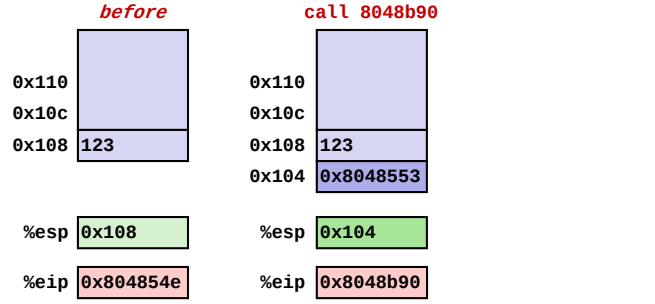
```
804854e:  e8 3d 06 00 00    call   8048b90
  <main>
8048553:  50                pushl  %eax
```

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

6

Procedure Call Example

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



%eip: program counter

7

Procedure Return Example

```
8048591:    c3                  ret
```

%eip: program counter

8

Stack Discipline

- 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
    - Why not registers (not enough registers, addressed variables)
- Stack allocated in ***Frames***
  - state for single procedure instantiation
- Stack discipline
  - One active procedure instantiation at a time (for a thread of execution)
  - Last called returns first
  - State for given procedure needed from its call to its return

9

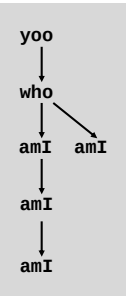
Call Chain Example

```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

Example Call Chain

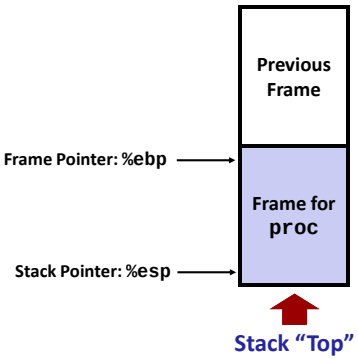


Procedure `amI()` is recursive

10

Stack Frames

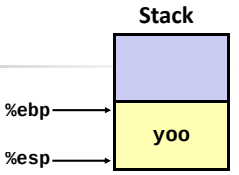
- Contents
  - Arguments
  - Local variables
  - Temporary space
- Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code



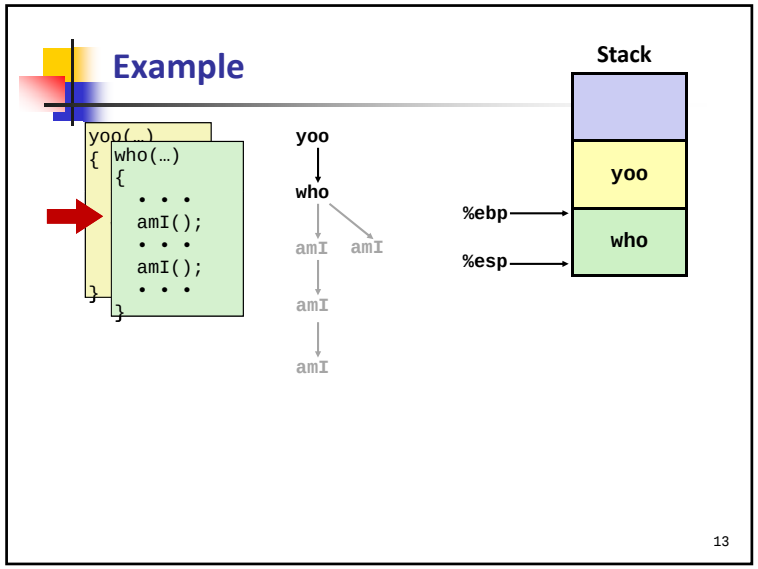
11

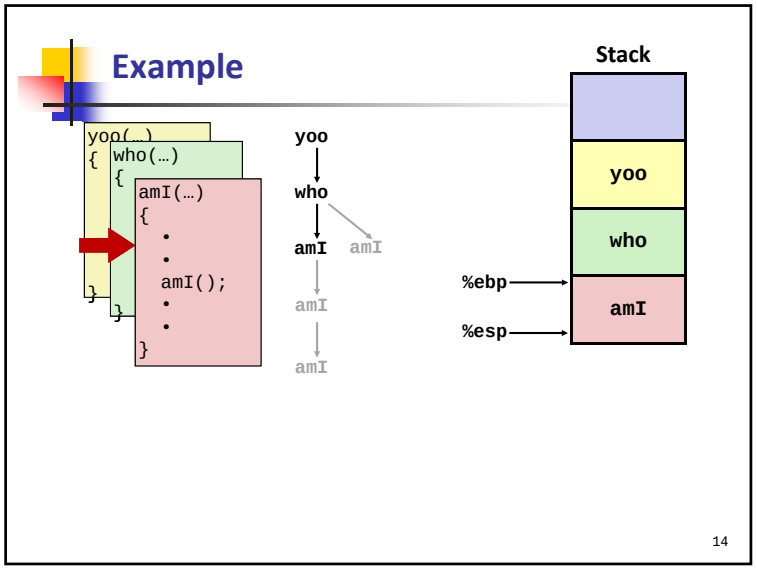
Example

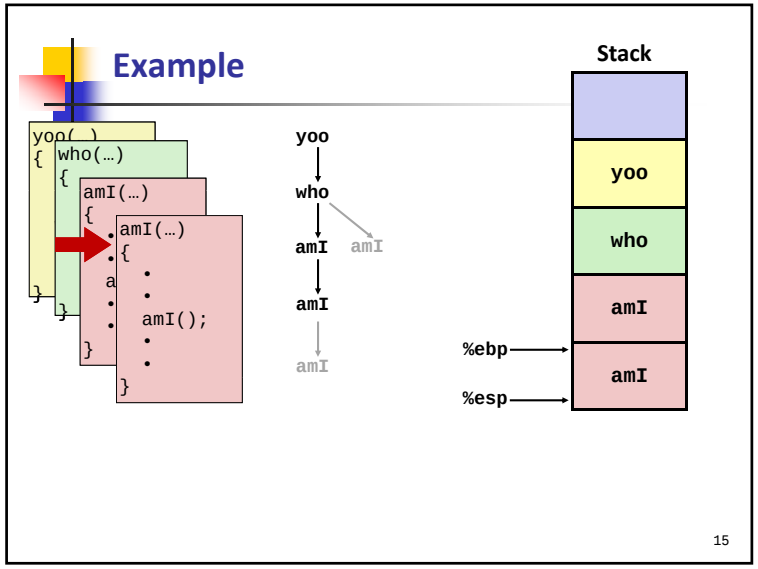
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

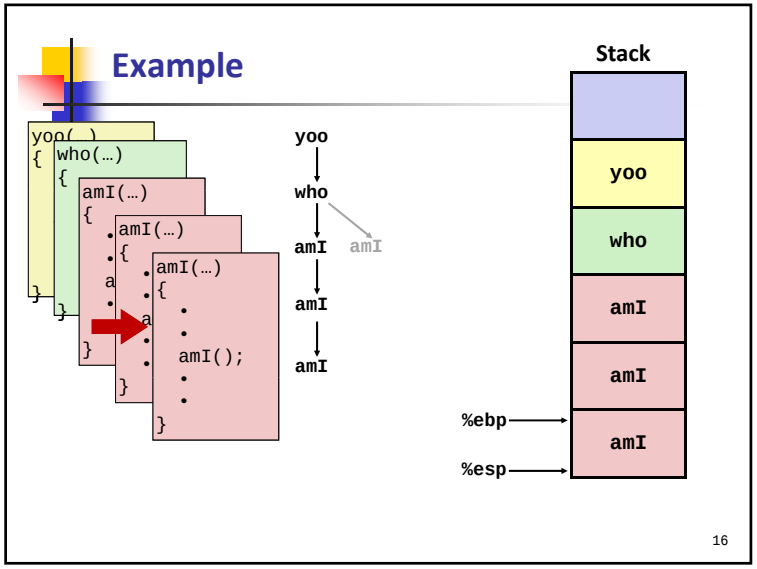


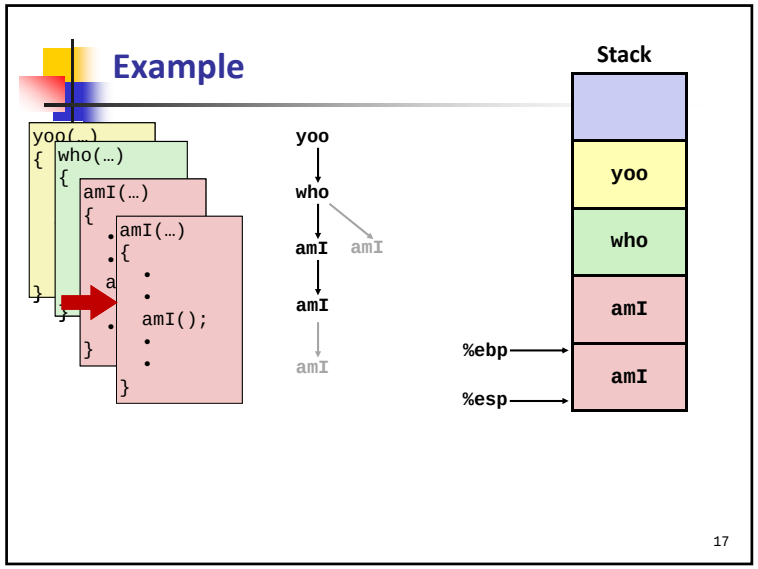
12

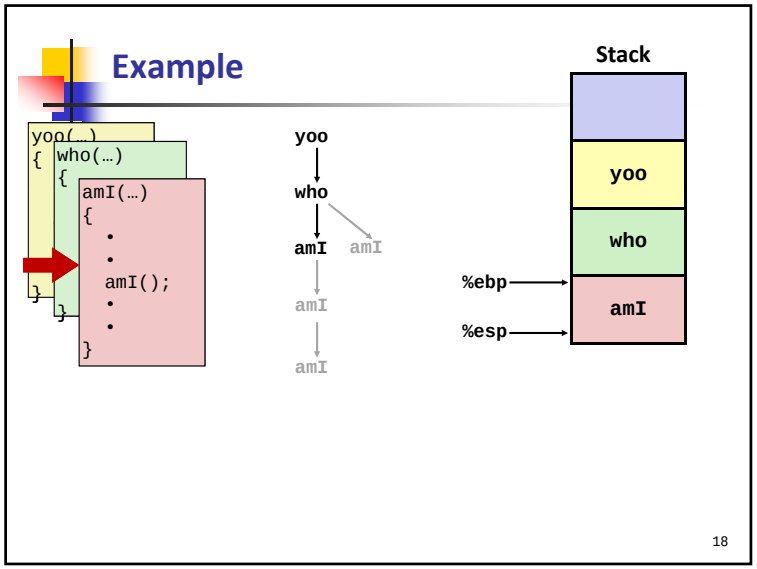

Example
Stack
yoo(…)
{ who(…)
{
• • •
amI();
• • •
amI();
• • •
}
}
yoo
who
amI
amI
amI
amI
yoo
who
%ebp
%esp
13

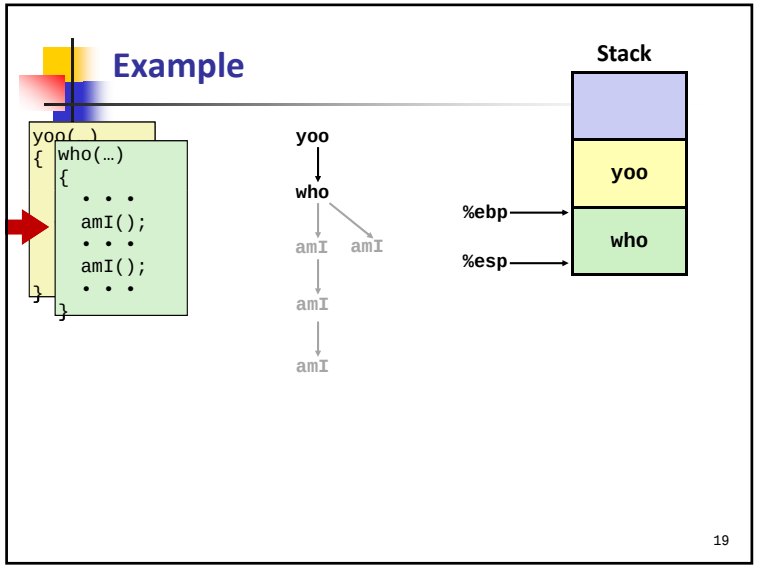

Example
Stack
yoo(…)
{ who(…)
{ amI(…)
{
•
•
amI();
•
•
}
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
%ebp
%esp
14

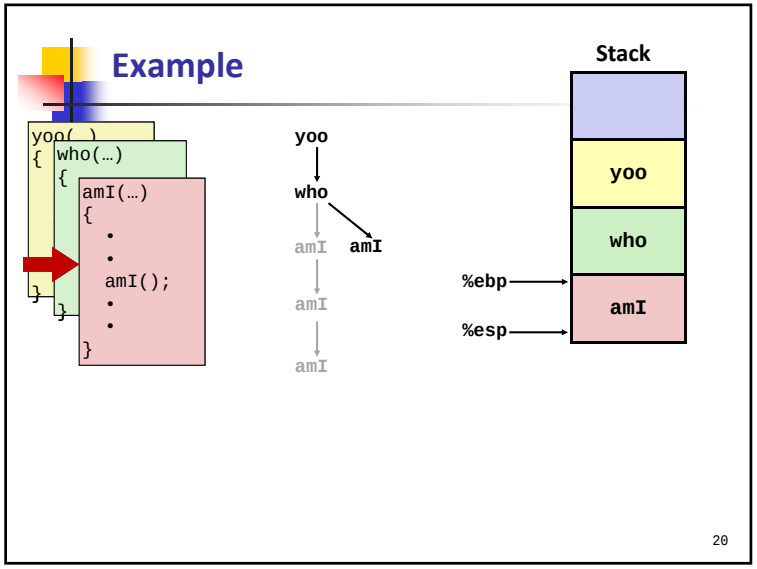

Example
Stack
yoo(…)
{ who(…)
{ amI(…)
{ amI(…)
{
•
•
amI();
•
•
}
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
%ebp
%esp
15


Example
Stack
yoo(…)
{ who(…)
{ amI(…)
{ amI(…)
{ amI(…)
{
•
•
amI();
•
•
}
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
amI
%ebp
%esp
16

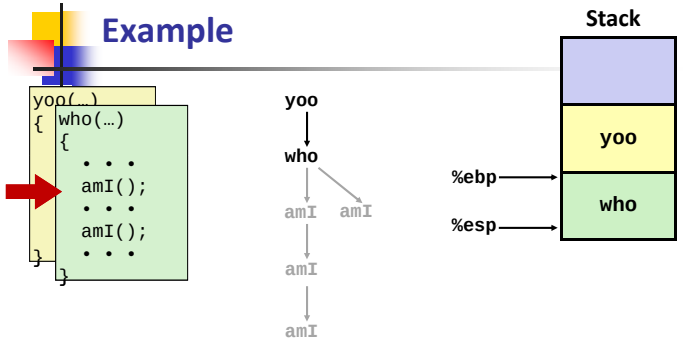

Example
Stack
yoo(…)
{ who(…)
{ amI(…)
{ amI(…)
{
amI();
}
yoo
who
amI
amI
amI
yoo
who
amI
amI
%ebp
%esp
17


Example
Stack
yoo(…)
{ who(…)
{ amI(…)
{
amI();
}
yoo
who
amI
amI
amI
yoo
who
amI
%ebp
%esp
18


Example
Stack
yoo(…)
{ who(…)
{
amI();
amI();
}
}
yoo
who
amI
amI
amI
yoo
who
%ebp
%esp
19

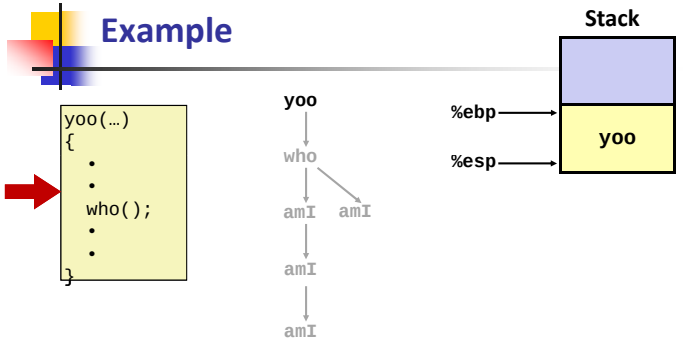

Example
Stack
yoo(…)
{ who(…)
{ amI(…)
{
amI();
}
yoo
who
amI
amI
amI
yoo
who
amI
%ebp
%esp
20

Example



21

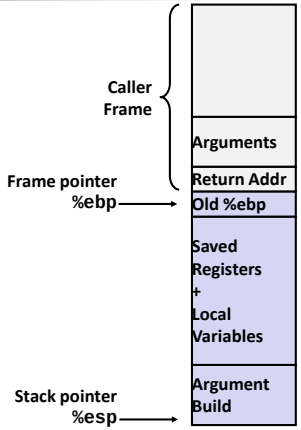
Example



22

IA32/Linux Stack Frame

- Current stack frame (top to bottom)
  - "Argument build:" Parameters for function about to call
  - Local variables If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller stack frame
  - Return address
    - Pushed by **call** instruction
  - Arguments for this call



23

Revisiting swap

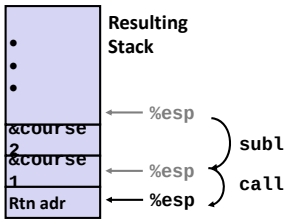
```
int course1 = 15213;
int course2 = 18243;

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;
}
```



24

Revisiting swap

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

```
swap:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx

    movl  8(%ebp), %edx
    movl  12(%ebp), %ecx
    movl  (%edx), %ebx
    movl  (%ecx), %eax
    movl  %eax, (%edx)
    movl  %ebx, (%ecx)

    popl  %ebx
    popl  %ebp
    ret
```

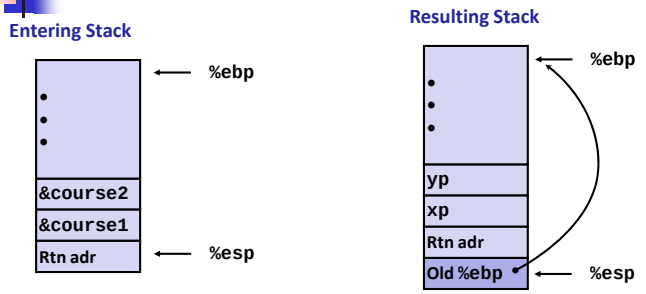
Set Up

Body

Finish

25

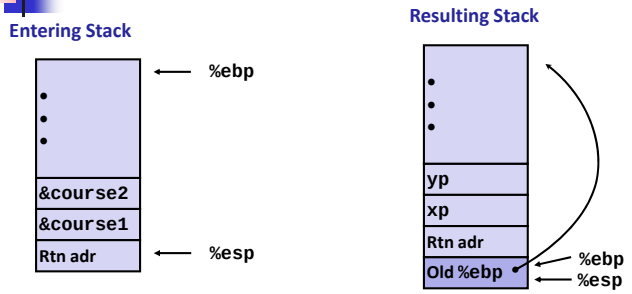
swap Setup #1




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

26

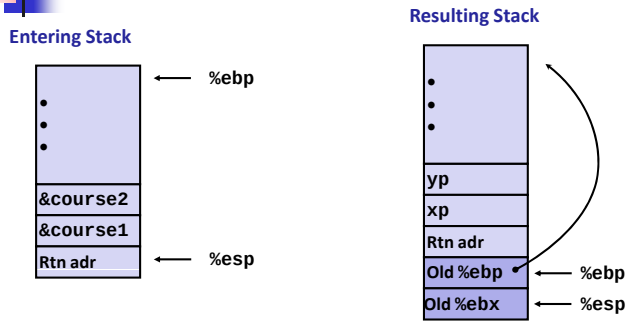
swap Setup #2




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

27

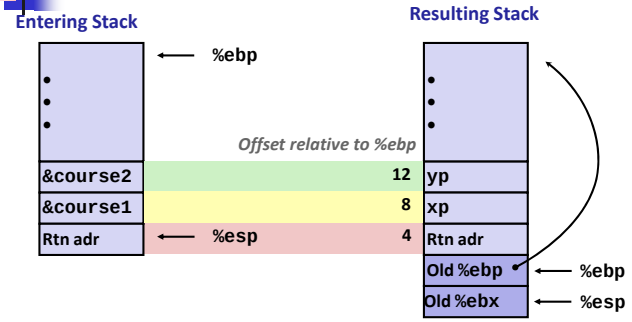
swap Setup #3




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

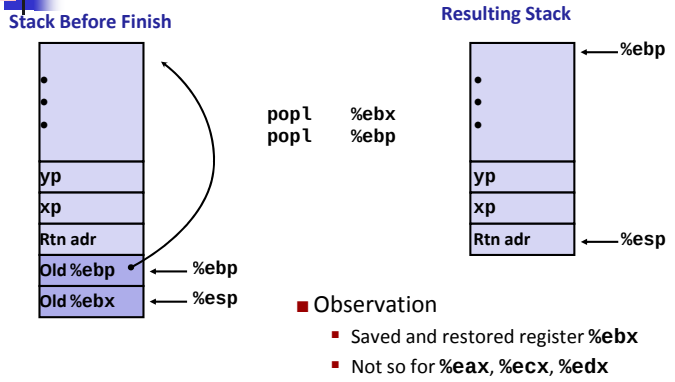
28

swap Body




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


swap Finish



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

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
```

```
who:
    . . .
    movl 8(%ebp), %edx
    addl $18243, %edx
    . . .
    ret
```

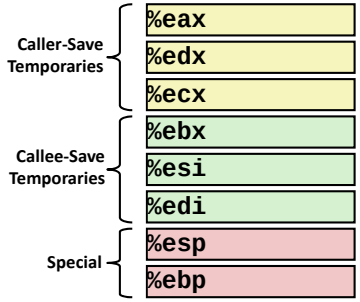
  - Contents of register **%edx** overwritten by **who**
  - This could be trouble → something should be done!
    - Need some coordination

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



33

Outline: Procedures

- Stack Structure & Calling Conventions
- Illustrations of Recursion & Pointers
- X86-64 Procedures

34

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

Nothing special!

```
pcount_r:
        pushl %ebp
        movl  %esp, %ebp
        pushl %ebx
        subl  $4, %esp
        movl  8(%ebp), %ebx
        movl  $0, %eax
        testl %ebx, %ebx
        je  .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
```

35

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**

36

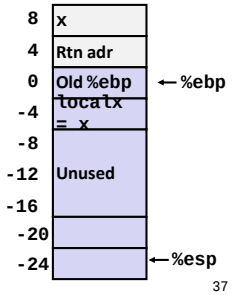
Creating and Initializing Local Variable

```
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
```



37

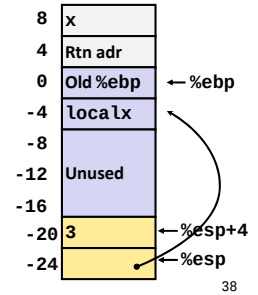
Creating Pointer as Argument

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



38

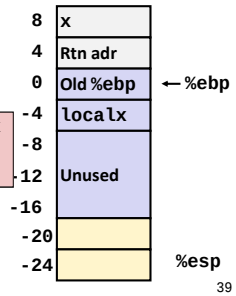
Retrieving local variable

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

- Retrieve localx from stack as return value

Final part of add3

```
  movl -4(%ebp), %eax # Return val= localx
  leave
  ret
```



39

Pointer Passing

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;
}
```

- Can callee return a pointer to its local variable back to caller?

40

Outline: Procedures

- Stack Structure & Calling Conventions
- Illustrations of Recursion & Pointers
- X86-64 Procedures

41

x86-64 Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Twice the number of registers, accessible as 8, 16, 32, 64 bits

42

x86-64 Registers: Usage Conventions

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

43

x86-64 Registers

- Arguments passed to functions via registers
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- All references to stack frame via stack pointer
  - Eliminates need to update **%ebp/%rbp**
  - What is the problem?
- Other Registers
  - 6 callee saved
  - 2 caller saved
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)

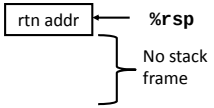
44

x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

- Operands passed in registers
  - First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
  - 64-bit pointers
- No stack operations required (except **ret**)
- Avoiding stack
  - Can hold all local information in registers

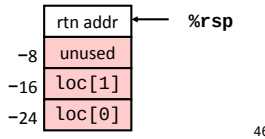


x86-64 Locals in the Red Zone

- 128 bytes beyond the stack top to be usable by current function
  - Save stack pointer change

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  movq  (%rdi), %rax
  movq  %rax, -24(%rsp)
  movq  (%rsi), %rax
  movq  %rax, -16(%rsp)
  movq  -16(%rsp), %rax
  movq  %rax, (%rdi)
  movq  -24(%rsp), %rax
  movq  %rax, (%rsi)
  ret
```



x86-64 NonLeaf without Stack Frame

- No values held while swap being invoked
- No callee save registers needed

```
/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

```
swap_ele:
    movslq %esi,%rsi              # Sign extend i
    leaq   8(%rdi,%rsi,8), %rax   # &a[i+1]
    leaq   (%rdi,%rsi,8), %rdi    # &a[i] (1st arg)   movq
    %rax, %rsi                    # (2nd arg)
    call   swap
    rep                           # No-op
    ret
```

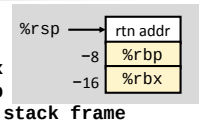

x86-64 Stack Frame Example

- If need callee-save registers, must set up stack frame to save them
- Why not try to use caller-save registers?

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

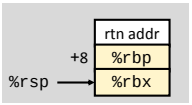
```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

Understanding x86-64 Stack Frame




```
movq    %rbx, -16(%rsp)        # Save %rbx
movq    %rbp, -8(%rsp)         # Save %rbp
subq    $16, %rsp              # Allocate stack frame
```

• • •




```
movq    (%rsp), %rbx           # Restore %rbx
movq    8(%rsp), %rbp          # Restore %rbp
addq    $16, %rsp              # Deallocate frame
```

49

Interesting Features of x86-64 Procedures

- Heavy use of registers
  - Parameter passing, more register-based temporaries
  - Minimal use of stack (sometimes none)
- Got rid of frame pointer
  - All stack accesses can be relative to **`%rsp`**
    - Allocate entire frame at once
  - Simpler deallocation
- Can delay allocation, since safe to temporarily use red zone

50

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

51