Procedures and the Call Stack

Topics
- Procedures
- Call stack
- Procedure/stack instructions
- Calling conventions
- Register-saving conventions

Why Procedures?

Why functions? Why methods?

```
int contains_char(char* haystack, char needle) {
  while (*haystack != '\0') {
    if (*haystack == needle) return 1;
    haystack++;
  }
  return 0;
}
```

Procedural Abstraction

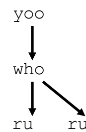
Call Chain

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

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

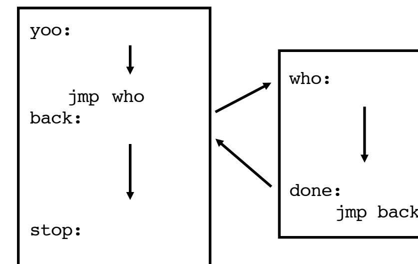
```
ru(…)
{
  •
  •
  •
}
```

Example
Call Chain



4

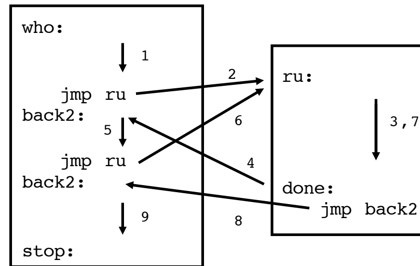
First Try (broken)



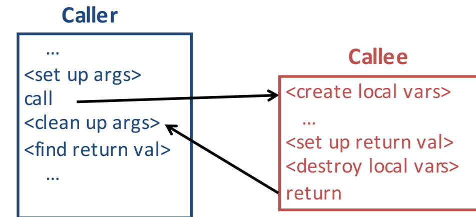
What if I want to call a function multiple times?

First Try (broken)

What if I want to call a function multiple times?



Procedure Calls: Lots to do!



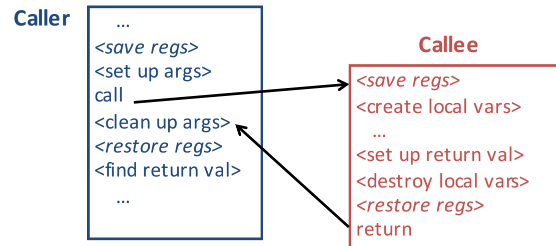
Callee must find **arguments** and **return address,** store local vars

Caller must find **return value**

Caller and **Callee** run on same CPU, use same registers…

7

Procedure *Calling Convention*



Rules about where to save/find things for call/return.

- Will see the convention for **IA32/Linux/GCC** in detail
- Details vary per architecture, OS, maybe compiler
- Why do we need a convention?

8

Memory Layout

Addr	Segment	Perm	Contents	Managed by	Initialized
2^N-1	**Stack**	**RW**	**Procedure context**	**Compiler**	**Run-time**
	Heap	RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run-time
	Statics	RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
	Literals	R	String literals	Compiler/ Assembler/Linker	Startup
0	Text	X	Instructions	Compiler/ Assembler/Linker	Startup

IA32 **Call Stack**

Region of memory
Managed with stack discipline

`%esp` register holds lowest stack address
= address of "top" element.

Stack "Bottom"
higher addresses
stack **grows** toward **lower** addresses
Stack Pointer: `%esp`
(*not* extra-sensory perception)
Stack "Top"

IA32 Call Stack: **Push**

pushl *Src*

Stack "Bottom"
higher addresses
stack **grows** toward **lower** addresses
Stack Pointer: `%esp`
Stack "Top"

IA32 Call Stack: **Push**

pushl *Src*

1. Fetch value from *Src*
2. Decrement **`%esp`** by 4 *(why 4?)*
3. Store value at new address given by **`%esp`**

Stack "Bottom"
higher addresses
stack **grows** toward **lower** addresses
Stack Pointer: `%esp`
-4
Stack "Top"

IA32 Call Stack: **Pop**

popl *Dest*

Stack "Bottom"
higher addresses
stack **grows** toward **lower** addresses
Stack Pointer: `%esp`
Stack "Top"

IA32 Call Stack: **Pop**

`popl` *Dest*

1. Load value from address `%esp`
2. Write value to *Dest*
3. Increment `%esp` by 4

Procedure Control Flow Instructions

Stack supports procedure call and return.

Procedure call: `call` *`label`*

1. Push return address on stack
2. Jump to ***`label`***

Return address: Address of instruction after `call`. Example:

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

Return address = `0x8048553`

Procedure return: `ret`

1. Pop return address from stack
2. Jump to address

Procedure Call Example #1

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


`%eip` = instruction pointer = program counter

Procedure Call Example #2

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


`%eip` = instruction pointer = program counter

Procedure Call Example #3

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

PC-relative address just like jumps…
(chosen by compiler; there's also an absolute call)

call 8048b90

	Before	After call
0x110		
0x10c		
0x108	123	123
0x104		0x8048553
%esp	0x108	0x104
%eip	0x804854e	0x8048553

+ 0x000063d

0x8048b90

%eip: *program counter*

Procedure Return Example #4

```
8048591:  c3               ret
```

ret

	Before	After ret
0x110		
0x10c		
0x108	123	123
0x104	0x8048553	0x8048553
%esp	0x104	0x108
%eip	0x8048591	0x8048553

%eip: *program counter*

Call Chain Example

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

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

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

Procedure amI is recursive (calls itself)

Example Call Chain

Stack Frames

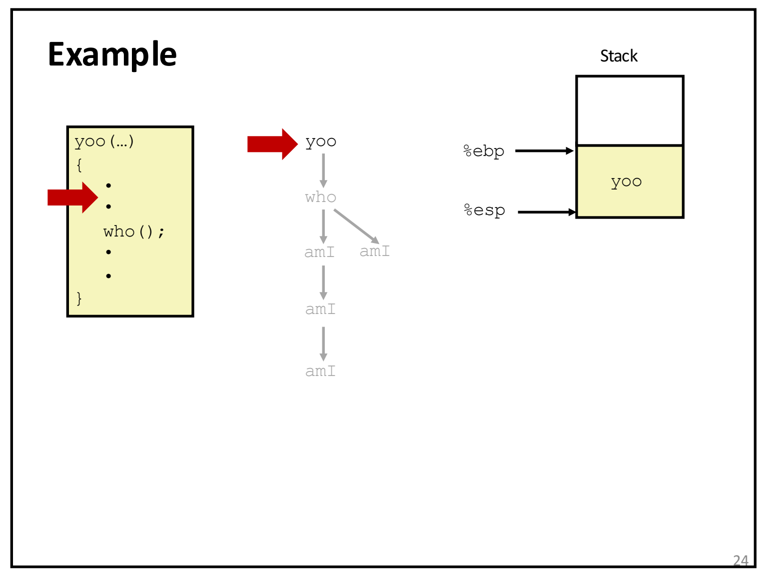
Contents

- Local variables
- Function arguments
- Return information
- Temporary space

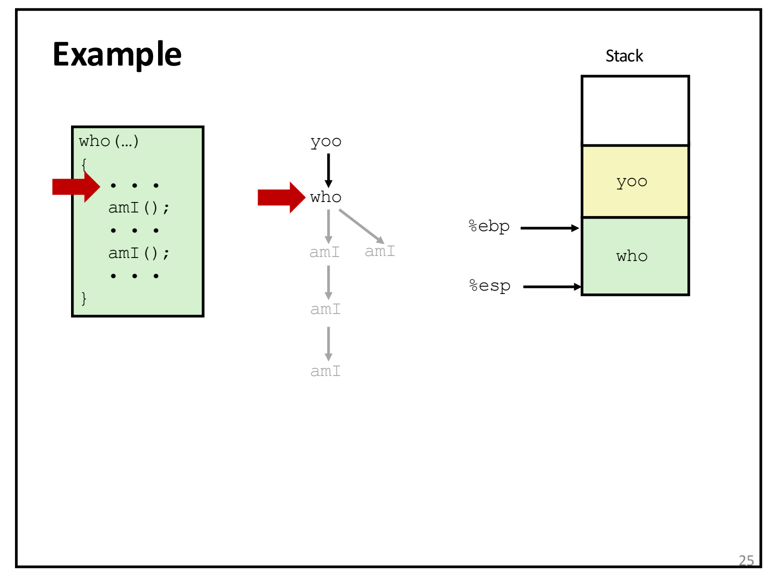
Management

- Space allocated when procedure is entered
  - "Set-up" code
- Space deallocated upon return
  - "Finish" code

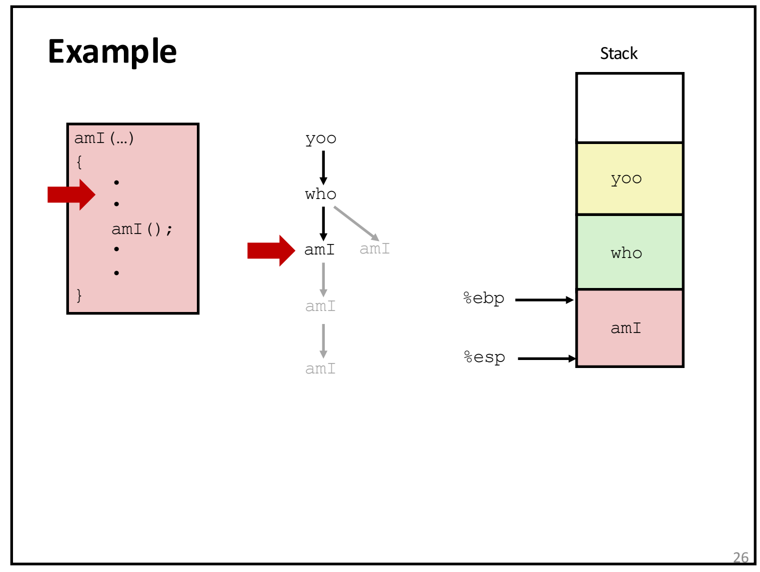
Why can't we just give every *procedure* a permanent chunk of memory to hold its local variables, etc?



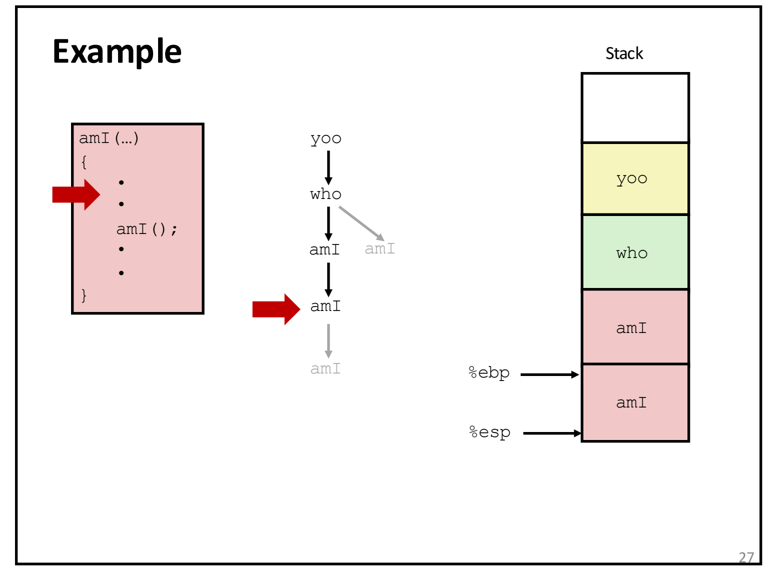
Example
Stack
yoo (…)
{
•
•
who ();
•
•
}
yoo
who
amI
amI
amI
amI
%ebp
%esp
yoo
24



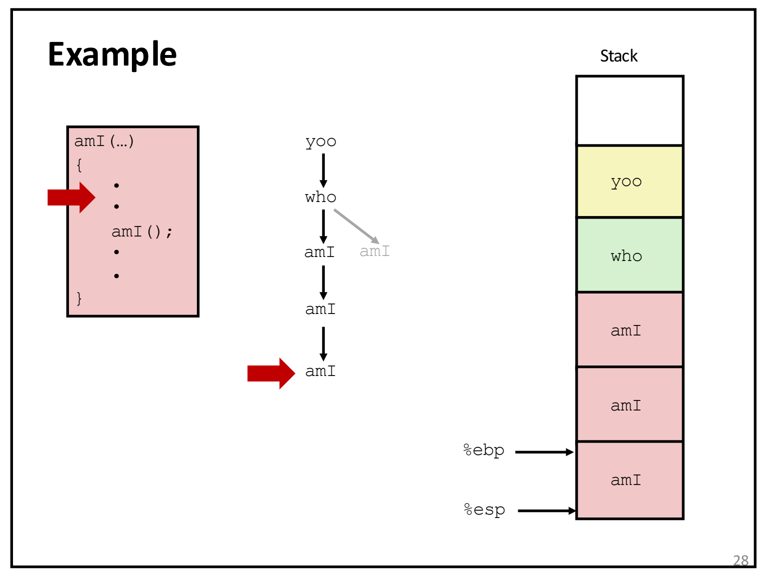
Example
Stack
who (…)
{
• • •
amI ();
• • •
amI ();
• • •
}
yoo
who
amI
amI
amI
amI
%ebp
%esp
yoo
who
25

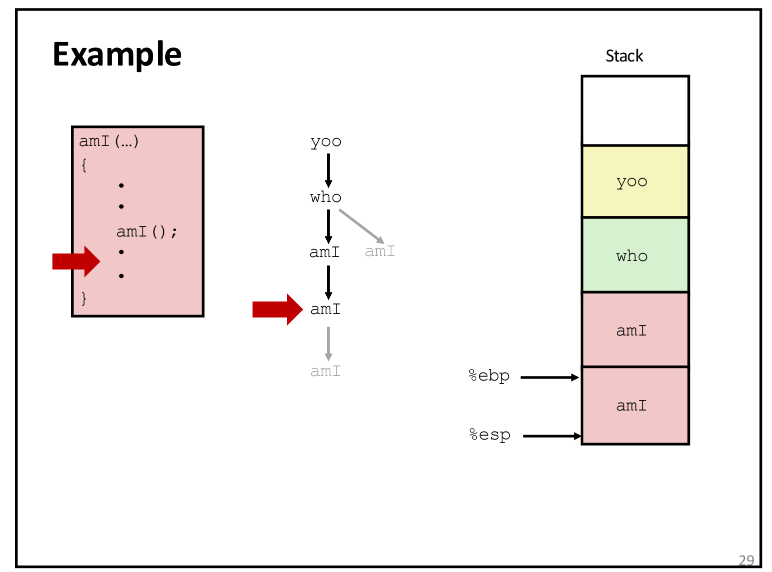


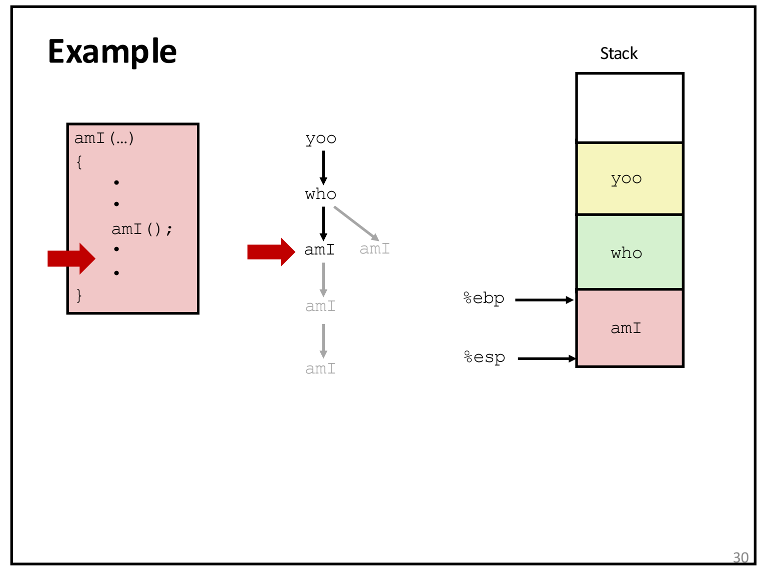
Example
Stack
amI (…)
{
•
•
amI ();
•
•
}
yoo
who
amI
amI
amI
amI
%ebp
%esp
yoo
who
amI
26

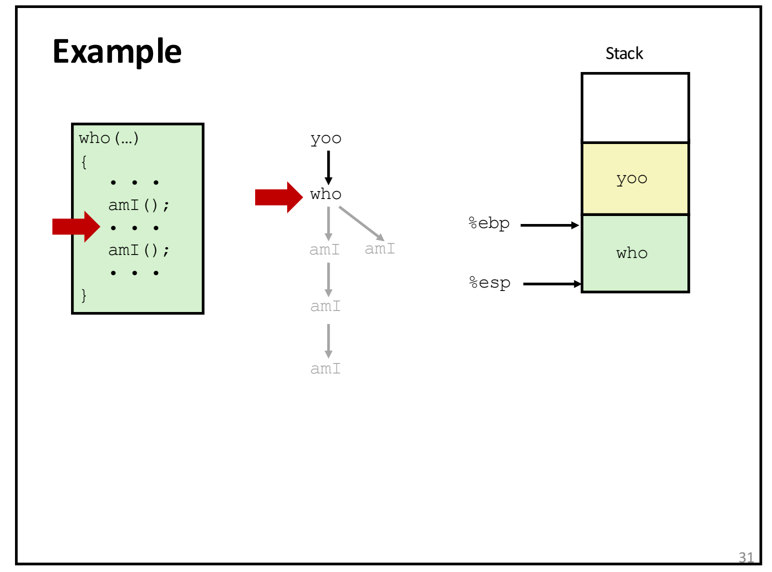


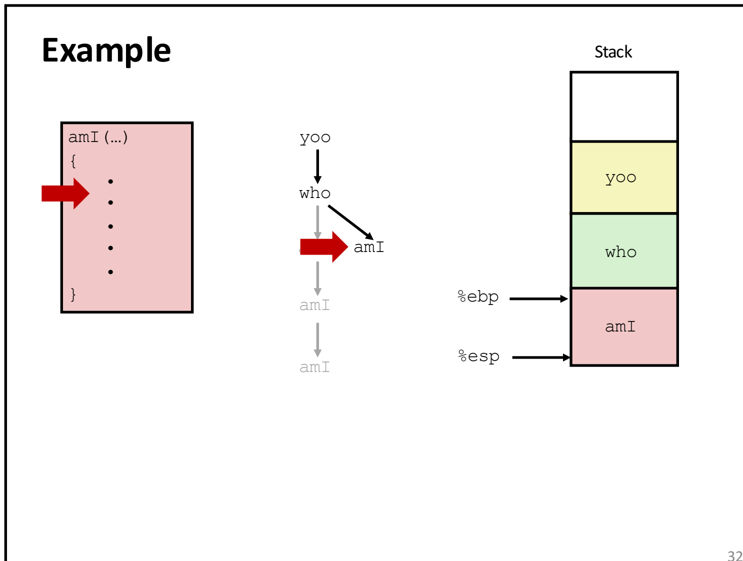
Example
Stack
amI (…)
{
•
•
amI ();
•
•
}
yoo
who
amI
amI
amI
amI
%ebp
%esp
yoo
who
amI
amI
27


Example
Stack
amI (...)
{
amI ();
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
amI
%ebp
%esp
28

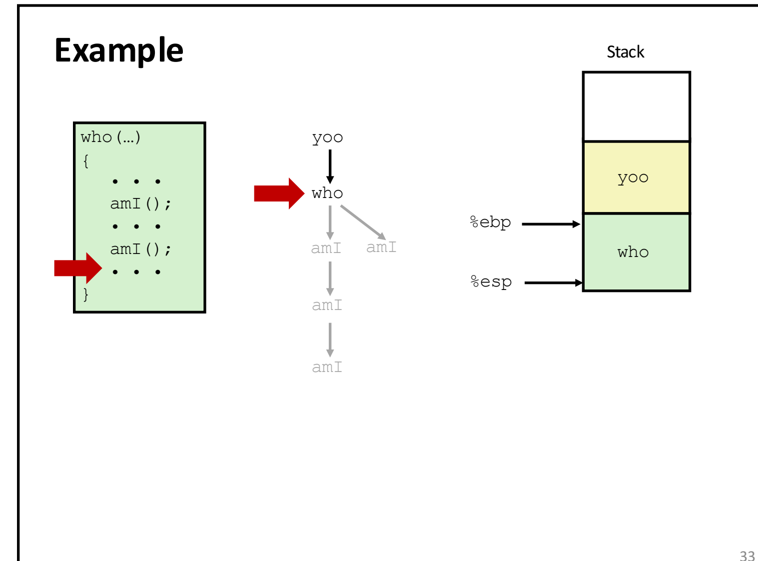

Example
Stack
amI (...)
{
amI ();
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
%ebp
%esp
29


Example
Stack
amI (...)
{
amI ();
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
%ebp
%esp
30

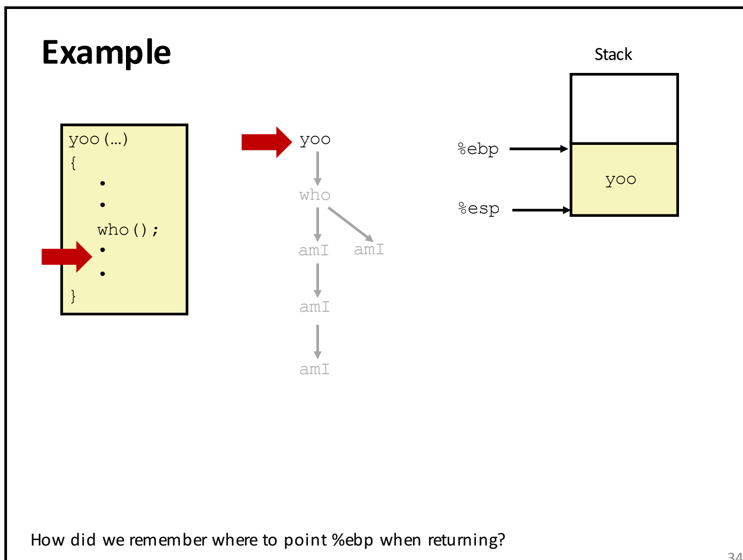

Example
Stack
who (...)
{
amI ();
amI ();
}
yoo
who
amI
amI
amI
amI
yoo
who
%ebp
%esp
31



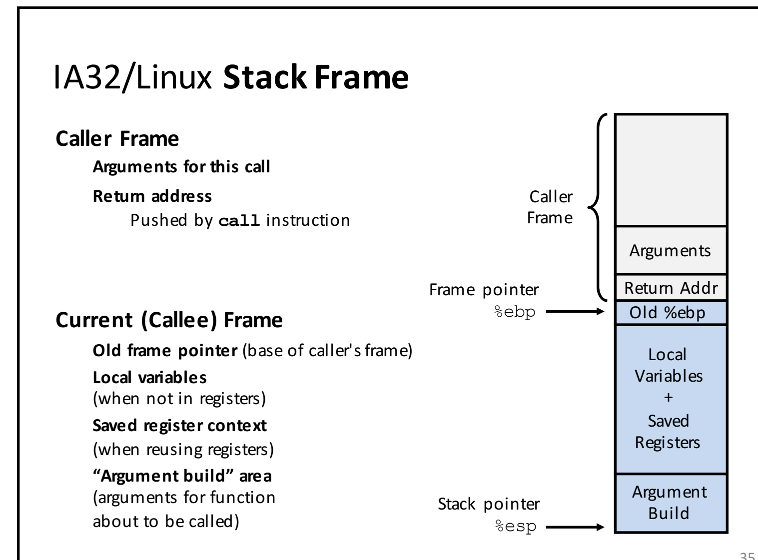
Example
Stack
amI (...)
{
}
yoo
who
amI
amI
amI
yoo
who
%ebp
%esp
amI
32



Example
Stack
who (...)
{
• • •
amI();
• • •
amI();
• • •
}
yoo
who
amI
amI
amI
amI
yoo
%ebp
%esp
who
33



Example
Stack
yoo (...)
{
who();
}
yoo
who
amI
amI
amI
amI
%ebp
%esp
yoo
How did we remember where to point %ebp when returning?
34



IA32/Linux Stack Frame
Caller Frame
Arguments for this call
Return address
Pushed by call instruction
Current (Callee) Frame
Old frame pointer (base of caller's frame)
Local variables
(when not in registers)
Saved register context
(when reusing registers)
"Argument build" area
(arguments for function
about to be called)
Caller
Frame
Frame pointer
%ebp
Stack pointer
%esp
Arguments
Return Addr
Old %ebp
Local
Variables
+
Saved
Registers
Argument
Build
35

Revisiting swap

```
int zip1 = 02481;
int zip2 = 98195;

void call_swap() {
  swap(&zip1, &zip2);
}
```

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

Calling `swap` from `call_swap`

```
call_swap:
   • • •
   pushl $zip2    # Global Var
   pushl $zip1    # Global Var
   call swap
   • • •
```

Resulting Stack

• • •
&zip2
&zip1
Rtn adr ← %esp

Revisiting swap

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

```
swap:
   pushl %ebp
   movl %esp,%ebp       } Set Up
   pushl %ebx

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

   movl -4(%ebp),%ebx
   movl %ebp,%esp       } Finish
   popl %ebp
   ret
```


swap Setup #1

Entering Stack (call_swap frame)

• • • ← %ebp
&zip2
&zip1
Rtn adr ← %esp

Resulting Stack

• • • ← %ebp
yp
xp
Rtn adr
Old %ebp ← **%esp**

```
swap:
   pushl %ebp        (bold)
   movl %esp,%ebp    } Set Up
   pushl %ebx
```


swap Setup #2

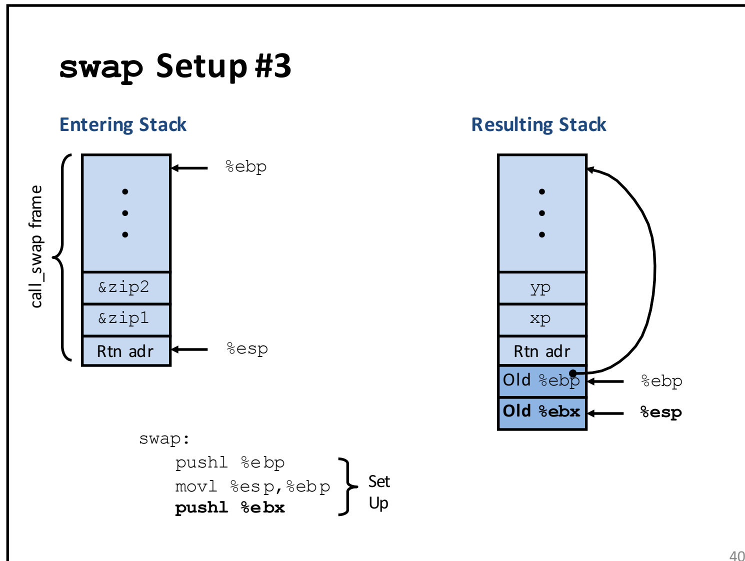
Entering Stack (call_swap frame)

• • • ← %ebp
&zip2
&zip1
Rtn adr ← %esp

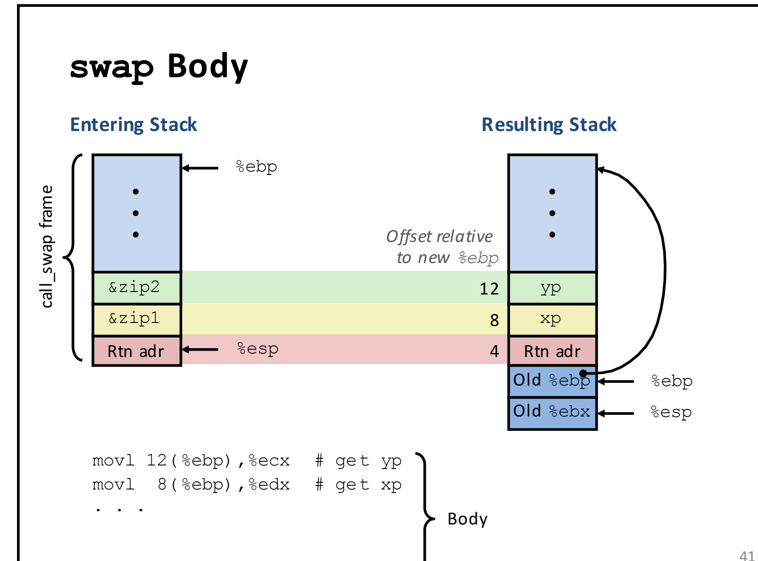
Resulting Stack

• • •
yp
xp
Rtn adr
Old %ebp ← **%ebp**, %esp

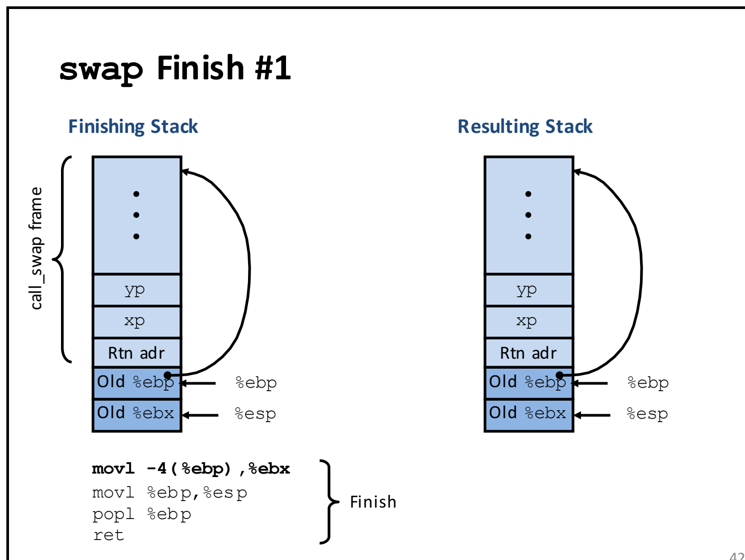
```
swap:
   pushl %ebp
   movl %esp,%ebp    (bold)   } Set Up
   pushl %ebx
```



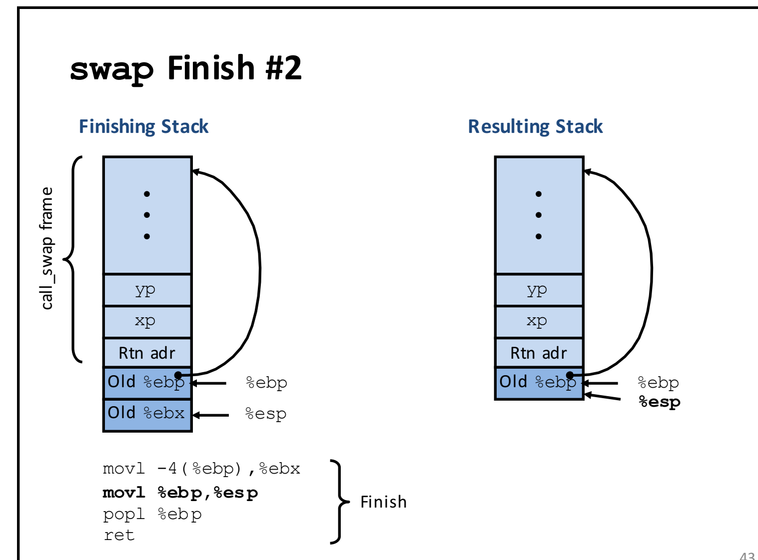
swap Setup #3
Entering Stack
Resulting Stack
call_swap frame
&zip2
&zip1
Rtn adr
%ebp
%esp
yp
xp
Rtn adr
Old %ebp
Old %ebx
%ebp
%esp
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
Set Up
40



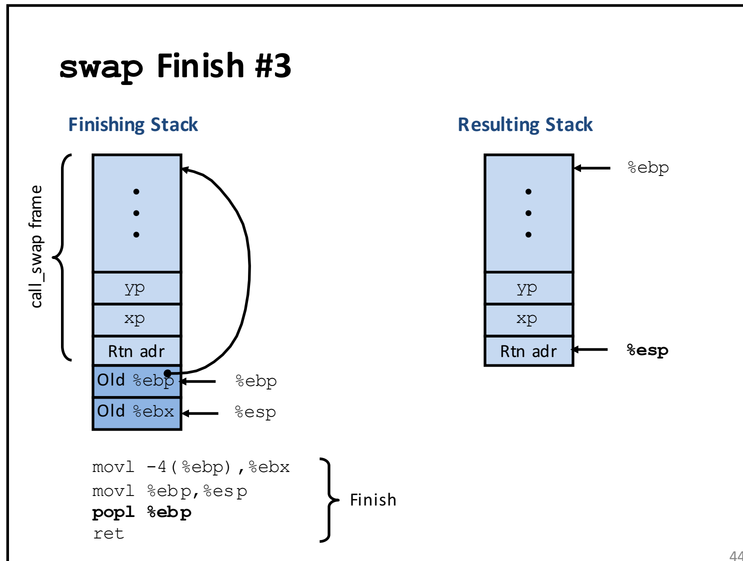
swap Body
Entering Stack
Resulting Stack
call_swap frame
&zip2
&zip1
Rtn adr
%ebp
%esp
Offset relative to new %ebp
12
8
4
yp
xp
Rtn adr
Old %ebp
Old %ebx
%ebp
%esp
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx # get xp
. . .
Body
41

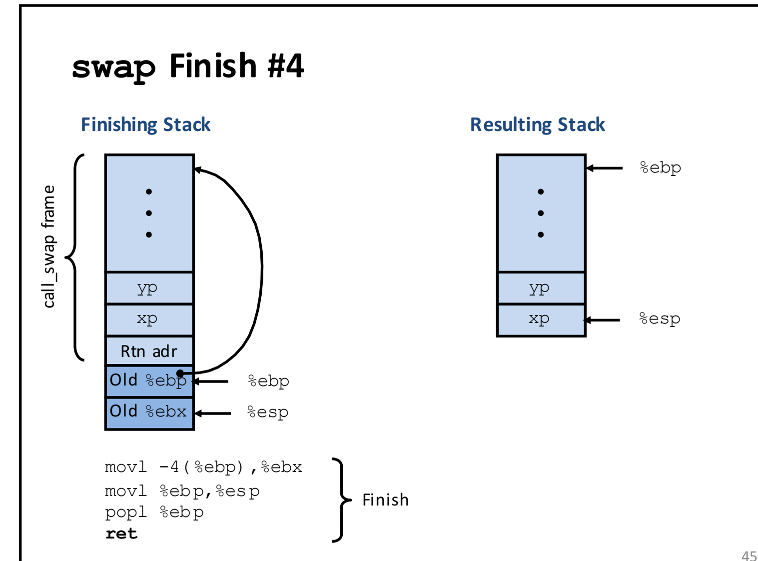


swap Finish #1
Finishing Stack
Resulting Stack
call_swap frame
yp
xp
Rtn adr
Old %ebp
Old %ebx
%ebp
%esp
yp
xp
Rtn adr
Old %ebp
Old %ebx
%ebp
%esp
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
Finish
42



swap Finish #2
Finishing Stack
Resulting Stack
call_swap frame
yp
xp
Rtn adr
Old %ebp
Old %ebx
%ebp
%esp
yp
xp
Rtn adr
Old %ebp
%ebp
%esp
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
Finish
43





Disassembled swap

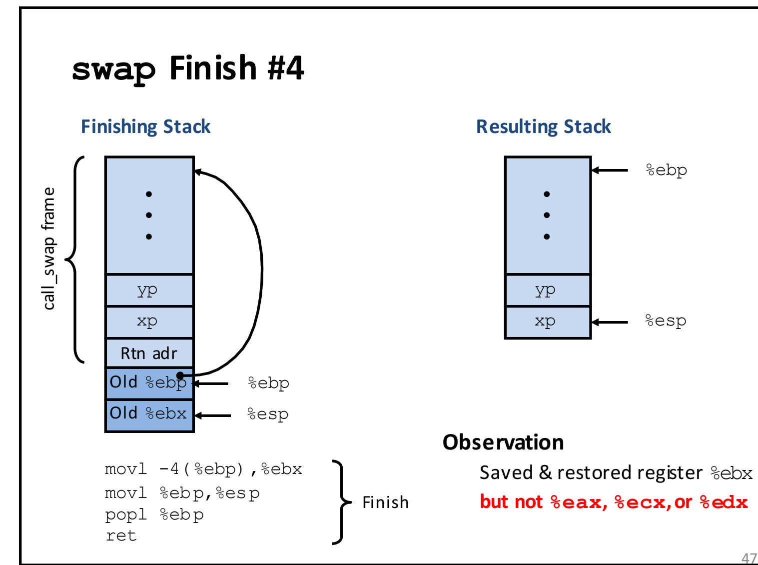
```
080483a4 <swap>:
 80483a4:   55          push   %ebp
 80483a5:   89 e5       mov    %esp,%ebp
 80483a7:   53          push   %ebx
 80483a8:   8b 55 08    mov    0x8(%ebp),%edx
 80483ab:   8b 4d 0c    mov    0xc(%ebp),%ecx
 80483ae:   8b 1a       mov    (%edx),%ebx
 80483b0:   8b 01       mov    (%ecx),%eax
 80483b2:   89 02       mov    %eax,(%edx)
 80483b4:   89 19       mov    %ebx,(%ecx)
 80483b6:   5b          pop    %ebx
 80483b7:   c9          leave          mov   %ebp,%esp
                                       pop   %ebp
 80483b8:   c3          ret
```

Calling Code

```
 8048409:   e8 96 ff ff ff    call 80483a4 <swap>
 804840e:   8b 45 f8          mov  0xfffffff8(%ebp),%eax
```

relative address (little endian)

46



A Puzzle

C function body:

```
*p = d;
return x - c;
```

assembly:

```
movsbl 12(%ebp),%edx
movl   16(%ebp),%eax
movl   %edx,(%eax)
movswl 8(%ebp),%eax
movl   20(%ebp),%edx
subl   %eax,%edx
movl   %edx,%eax
```

Write the C function header, types, and order of parameters.

movsbl = move sign-extending a byte to a long (4-byte)
movswl = move sign-extending a word (2-byte) to a long (4-byte)

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the ***caller***
- `who` is the ***callee***

Will register contents still be there after a procedure call?

```
yoo:
   • • •
   movl $12345, %edx
   call who
   addl %edx, %eax
   • • •
   ret
```

```
who:
   • • •
   movl 8(%ebp), %edx
   addl $98195, %edx
   • • •
   ret
```


Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the ***caller***
- `who` is the ***callee***

Will register contents still be there after a procedure call?

Conventions

- ***Caller Save***
  - **Caller** saves temporary values in its frame **before calling**
- ***Callee Save***
  - **Callee** saves temporary values in its frame **before using**

IA32/Linux Register Saving Conventions

`%eax, %edx, %ecx`

- **Caller saves** prior to call if needs values after call

`%eax`

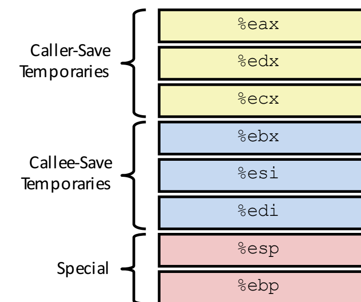
- also used to **return** integer value

`%ebx, %esi, %edi`

- **Callee saves** if will use

`%esp, %ebp`

- special form of callee save
- restored to original values before returning



Example: **Pointers to Local Variables**

Top-Level Call

```
int sfact(int x) {
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Recursive Procedure

```
void s_helper
  (int x, int *accum) {
  if (x <= 1) {
    return;
  } else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

sfact(3)	val = 1
s_helper(3, &val)	val = 3
s_helper(2, &val)	val = 6
s_helper(1, &val)	val = 6.

Pass pointer to update location

53

Creating & Initializing Pointer

```
int sfact(int x) {
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Variable `val` must be stored on stack

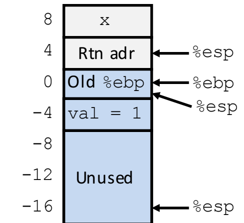
Because: Need to create pointer to it

Compute pointer as `-4(%ebp)`

Push on stack as second argument

Initial part of `sfact`

```
_sfact:
   pushl %ebp              # Save %ebp
   movl %esp,%ebp          # Set %ebp
   subl $16,%esp           # Add 16 bytes
   movl 8(%ebp),%edx       # edx = x
   movl $1,-4(%ebp)        # val = 1
```



54

Passing Pointer

```
int sfact(int x) {
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Variable `val` must be stored on stack

Because: Need to create pointer to it

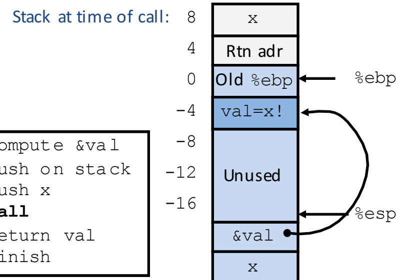
Compute pointer as `-4(%ebp)`

Push on stack as second argument

Stack at time of call:

Calling `s_helper` from `sfact`

```
   leal -4(%ebp),%eax   # Compute &val
   pushl %eax           # Push on stack
   pushl %edx           # Push x
   call s_helper        # call
   movl -4(%ebp),%eax   # Return val
   • • •                # Finish
```



55

IA32/Linux **Procedure Summary**

call, ret, push, pop

Stack discipline fits procedure call / return.*

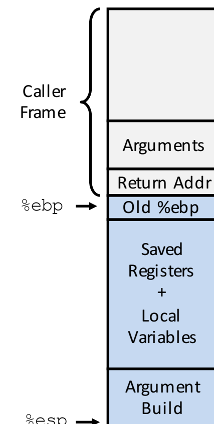
If P calls Q, Q returns before P, including recursion.

Conventions support arbitrary function calls.

Safely store per-call values in local stack frame and callee-saved registers

Push function arguments at top of stack before call

Result returned in **`%eax`**



*take 251 to learn about languages where it doesn't quite.

56