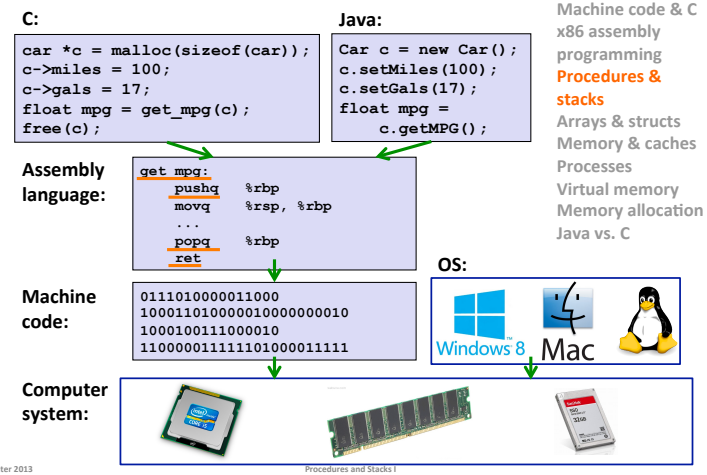
The Hardware/Software Interface

CSE351 Winter 2013

Procedures and Stacks I

Roadmap



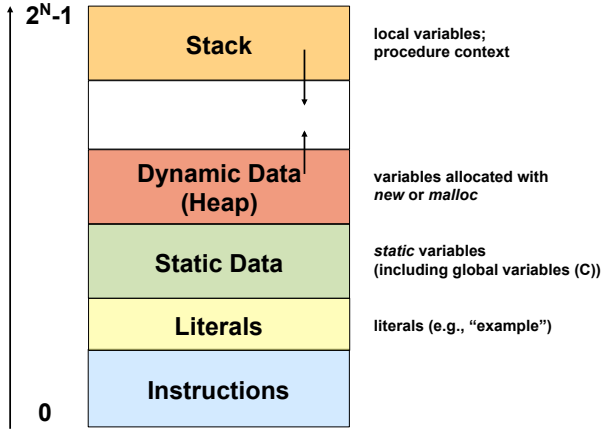
Data & addressing
Integers & floats
Machine code & C
x86 assembly programming
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Procedures and Call Stacks

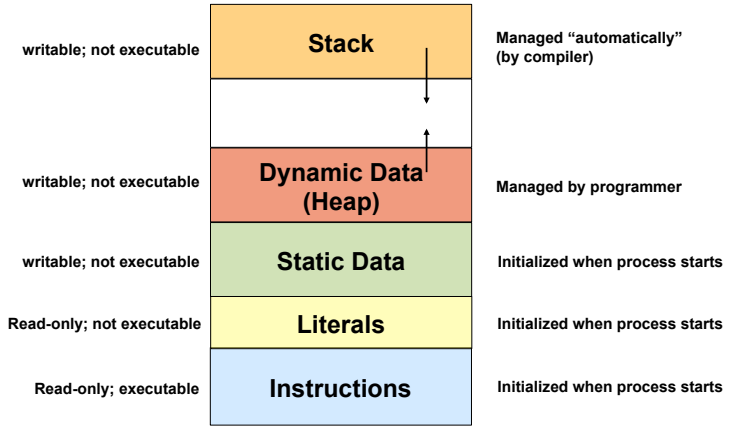
- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return to?

- To answer these questions, we need a *call stack* …

Memory Layout

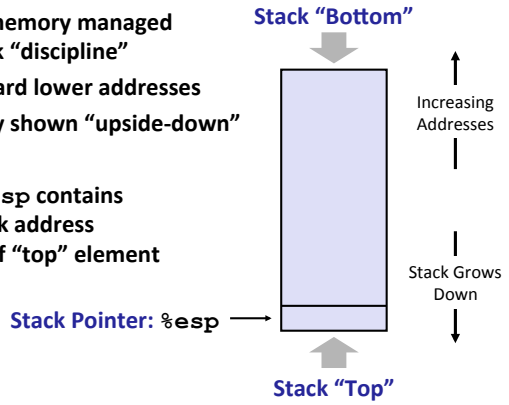


Memory Layout



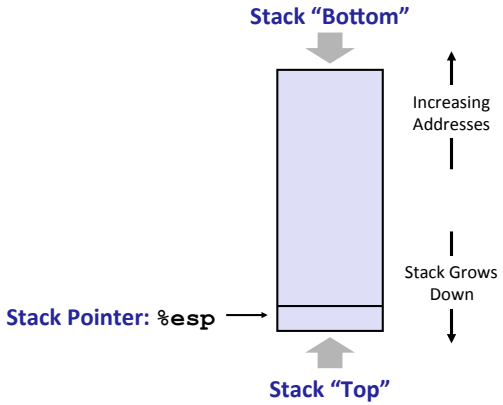
IA32 Call Stack

- Region of memory managed with a stack "discipline"
- Grows toward lower addresses
- Customarily shown "upside-down"
- Register `%esp` contains lowest stack address = address of "top" element



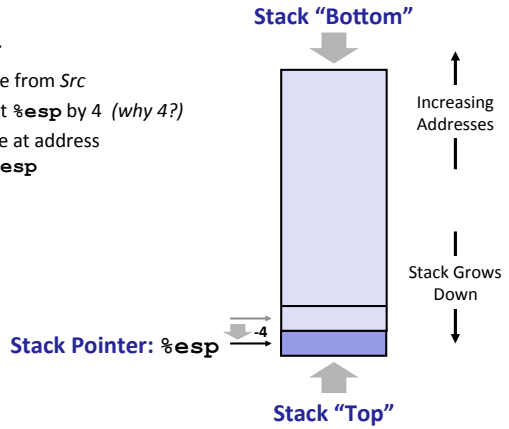
IA32 Call Stack: Push

- `pushl` *Src*



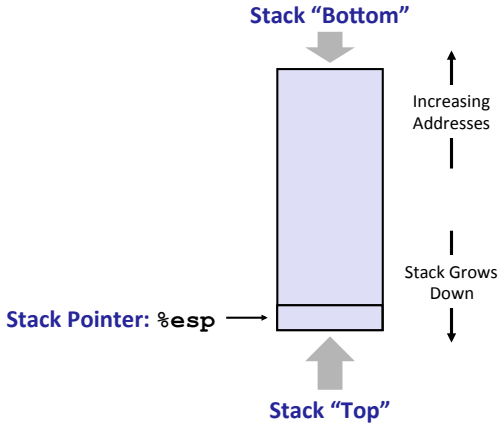
IA32 Call Stack: Push

- `pushl` *Src*
  - Fetch value from *Src*
  - Decrement `%esp` by 4 *(why 4?)*
  - Store value at address given by `%esp`



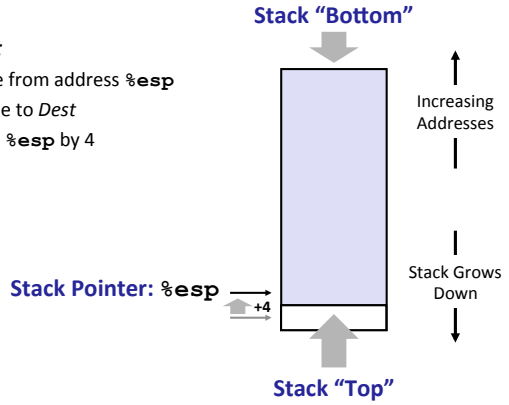
IA32 Call Stack: Pop

- `popl` *Dest*

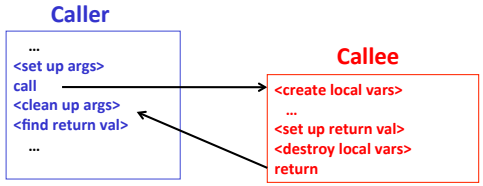


IA32 Call Stack: Pop

- `popl` *Dest*
  - Load value from address `%esp`
  - Write value to *Dest*
  - Increment `%esp` by 4

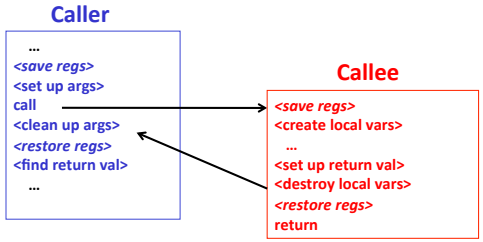


Procedure Call Overview



- **Callee must know where to find args**
- **Callee must know where to find "return address"**
- **Caller must know where to find return val**
- **Caller and Callee run on same CPU → use the same registers**
  - Caller might need to save registers that Callee might use
  - Callee might need to save registers that Caller has used

Procedure Call Overview



- **The convention of where to leave/find things is called the procedure call linkage**
  - Details vary between systems
  - We will see the convention for IA32/Linux in detail
  - What could happen if our program didn't follow these conventions?

Procedure Control Flow

- **Use stack to support procedure call and return**
- **Procedure call: `call` *`label`***
  - Push return address on stack
  - Jump to *`label`*

Procedure Control Flow

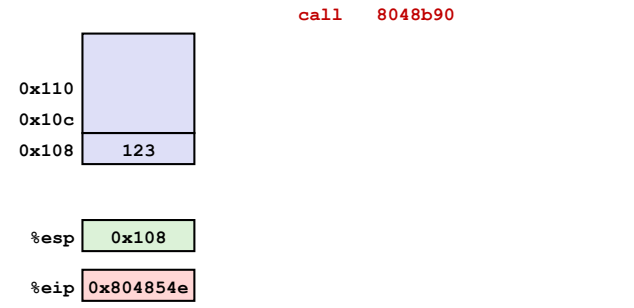
- **Use stack to support procedure call and return**
- **Procedure call: `call` *`label`***
  - Push return address on stack
  - Jump to *`label`*
- **Return address:**
  - Address of instruction after `call`
  - Example from disassembly:

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

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

Procedure Call Example

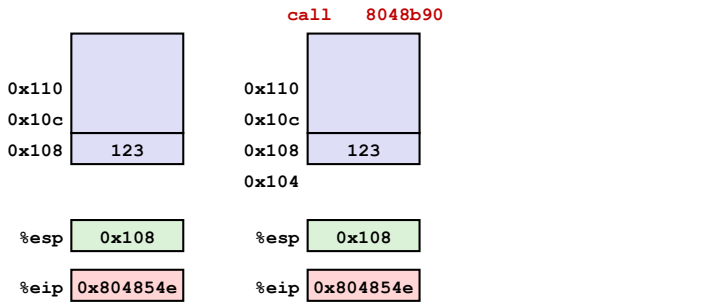
```
804854e:  e8 3d 06 00 00    call   8048b90 <main>
8048553:  50                pushl  %eax
```



`%eip`: program counter

Procedure Call Example

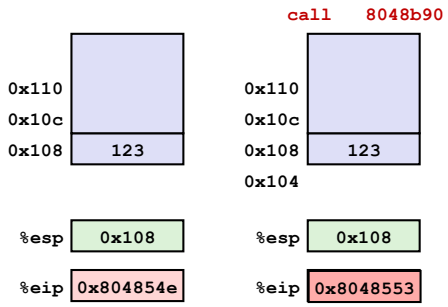
```
804854e:  e8 3d 06 00 00    call   8048b90 <main>
8048553:  50                pushl  %eax
```



`%eip`: program counter

Procedure Call Example

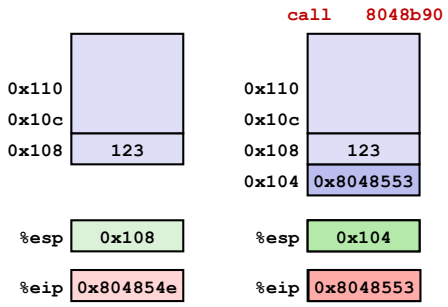
```
804854e:    e8 3d 06 00 00      call   8048b90 <main>
8048553:    50                  pushl  %eax
```



%eip: program counter

Procedure Call Example

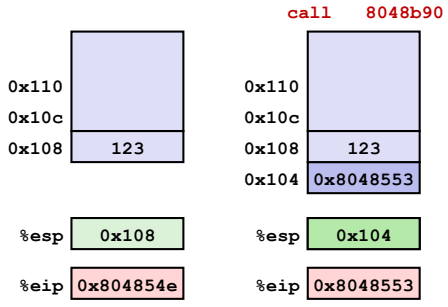
```
804854e:    e8 3d 06 00 00      call   8048b90 <main>
8048553:    50                  pushl  %eax
```



%eip: program counter

Procedure Call Example

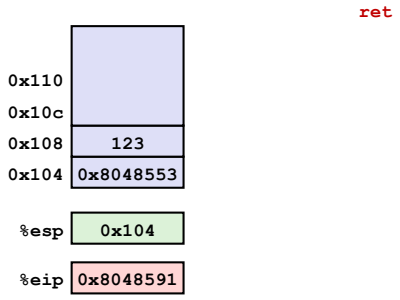
```
804854e:    e8 3d 06 00 00      call   8048b90 <main>
8048553:    50                  pushl  %eax
```



%eip: program counter

Procedure Return Example

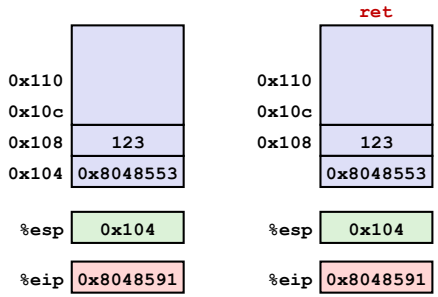
```
8048591:    c3                  ret
```



%eip: program counter

Procedure Return Example

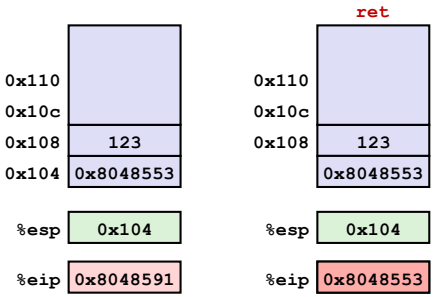
```
8048591:    c3                  ret
```



%eip: program counter

Procedure Return Example

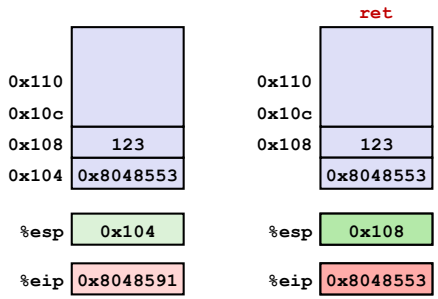
```
8048591:    c3                  ret
```



%eip: program counter

Procedure Return Example

```
8048591:    c3                  ret
```



%eip: program counter

Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- **Stack discipline**
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does
- **Stack allocated in *frames***
  - State for a single procedure instantiation

Call Chain Example

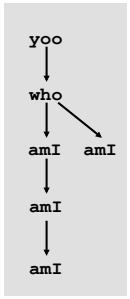
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

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

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

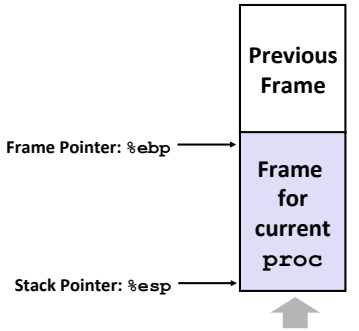
Procedure `amI` is recursive (calls itself)

Example Call Chain



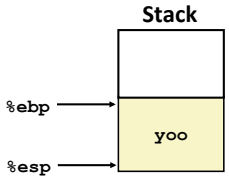
Stack Frames

- **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



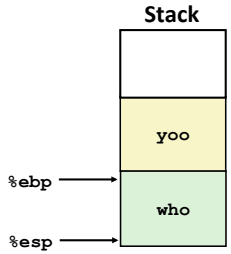
Example

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

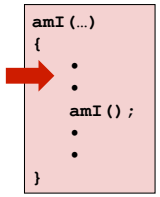


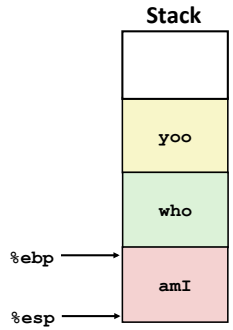
Example

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

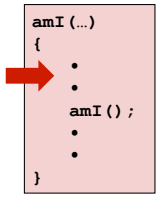


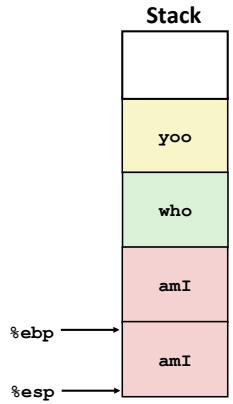
Example



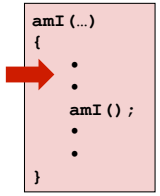


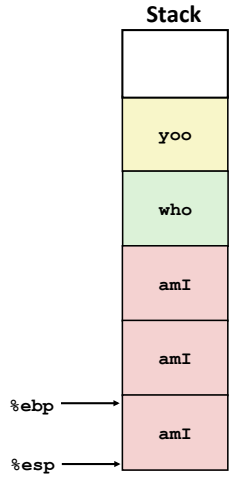
Example



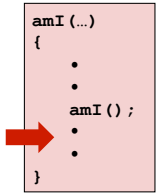


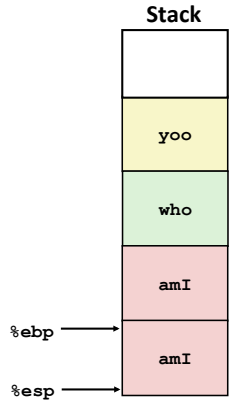
Example





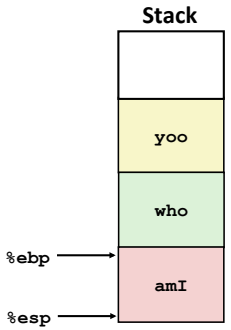
Example





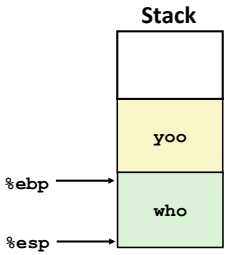
Example

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



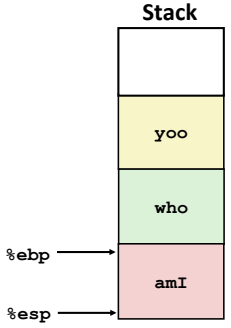
Example

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



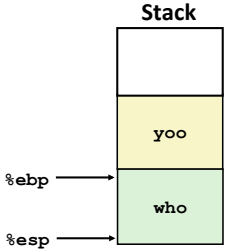
Example

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



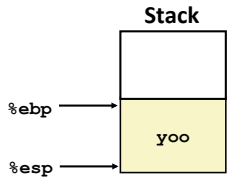
Example

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



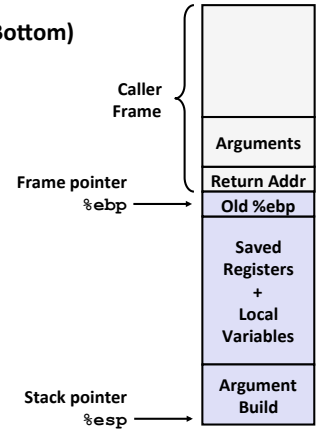
Example

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



IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build" area (parameters for function about to be called)
  - Local variables (if can't be kept in registers)
  - Saved register context (when reusing registers)
  - Old frame pointer (for caller)
- **Caller's Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



Revisiting swap

```
int zip1 = 15213;
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;
}
```


Revisiting swap

```
int zip1 = 15213;
int zip2 = 98195;

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

Calling swap from `call_swap`

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

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

Revisiting swap

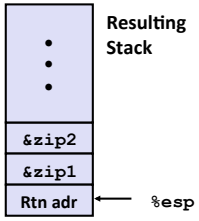
```
int zip1 = 15213;
int zip2 = 98195;

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

Calling swap from call_swap

```
call_swap:
    • • •
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    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
    movl %esp,%ebp
    pushl %ebx

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

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

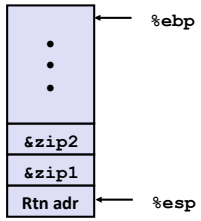
Set Up

Body

Finish

swap Setup #1

Entering Stack

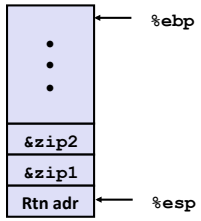


Resulting Stack?

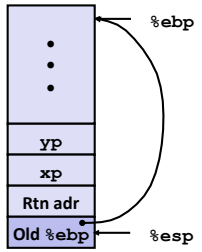
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```


swap Setup #1

Entering Stack



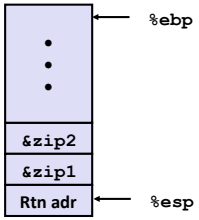
Resulting Stack



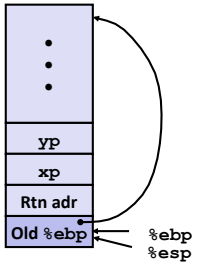

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



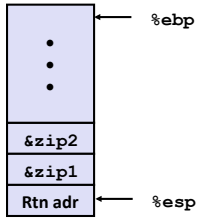
Resulting Stack



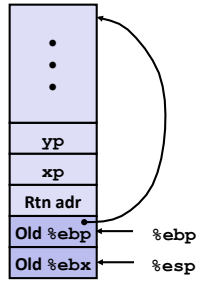

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```


swap Setup #3

Entering Stack



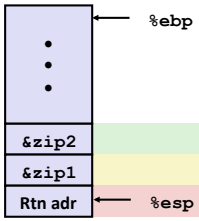
Resulting Stack




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


swap Body

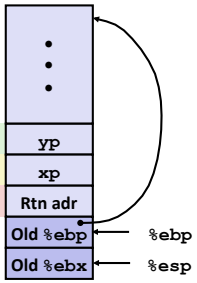
Entering Stack



Offset relative to new %ebp

12
8
4

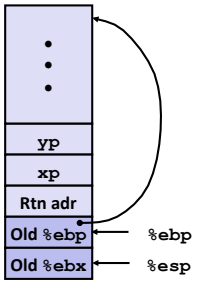
Resulting Stack




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


swap Finish #1

swap's Stack

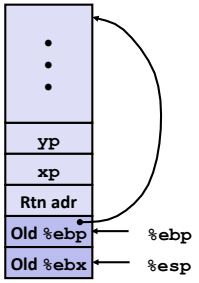


Resulting Stack?

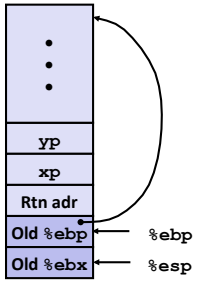
```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #1

swap's Stack



Resulting Stack

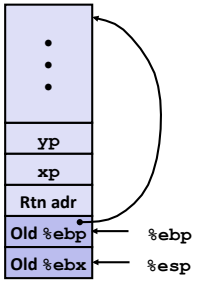



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

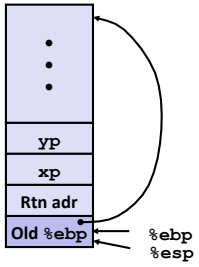
Observation: Saved and restored register %ebx

swap Finish #2

swap's Stack



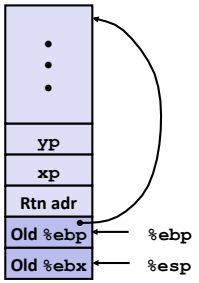
Resulting Stack



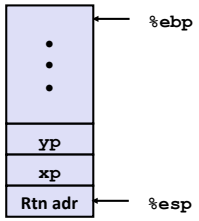

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```


swap Finish #3

swap's Stack



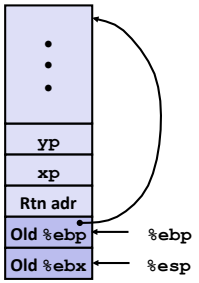
Resulting Stack



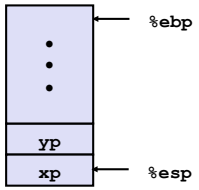

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```


swap Finish #4

swap's Stack



Resulting Stack




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

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
 80483b8:   c3          ret
```

```
mov   %ebp,%esp
pop   %ebp
```

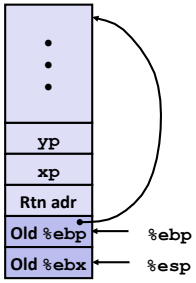
Calling Code

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

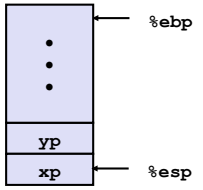
0x0804840e + 0xffffff96 = 0x080483a4

swap Finish #4

swap's Stack



Resulting Stack




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

- **Observation**
  - Saved & restored register **%ebx**
  - Didn't do so for **%eax**, **%ecx**, or **%edx**

Register Saving Conventions

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

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

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

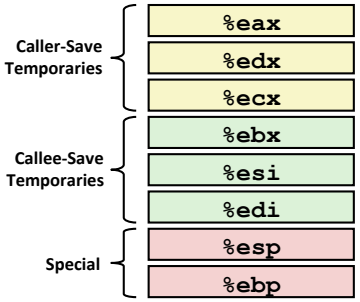
  - Contents of register **%edx** overwritten by **who**

Register Saving Conventions

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



Example: Pointers to Local Variables

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);
  }
}
```

Top-Level Call

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

- **Pass pointer to update location**

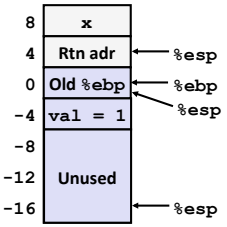
Creating & Initializing Pointer

```
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
```



Passing Pointer

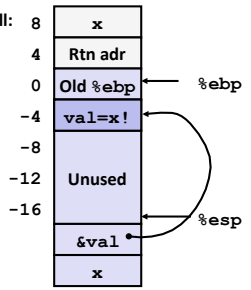
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

- **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**

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
```

Stack at time of call:



IA 32 Procedure Summary

- **Important points:**
  - IA32 procedures are a combination of *instructions* and *conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **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 returned in `%eax`