Procedures/Stacks

- Stacks
- Procedures
- Parameter passing
What is the stack for?

- Why a stack?
Memory Layout

- **Instructions**
- **Literals** (e.g., “example”)
- **Static Data**
  - static variables (including global variables (C))
- **Dynamic Data (Heap)**
  - new'ed variables
- **Stack**
  - local variables
Memory Layout

- **Instructions**
  - Read-only; executable
  - Read-only; not executable
  - Writable; not executable

- **Literals**
  - Read-only; not executable
  - Writable; not executable

- **Static Data**
  - Writable; not executable

- **Dynamic Data (Heap)**
  - Writable; not executable

- **Stack**
  - Writable; not executable

- **Execution lifetime**
  - Immutable
  - Mutable

- **“Automatic” lifetime**
  - Mutable
IA32 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

Stack Pointer: `%esp`

Stack “Bottom”

Stack Grows Down

Increasing Addresses
IA32 Stack: Push

- `pushl Src`

Stack Pointer: `%esp` \(-4\)

Stack “Bottom”

Stack Grows Down

Increasing Addresses

Stack “Top”
IA32 Stack: Push

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

Stack Pointer: `%esp`

Stack "Bottom"

Stack Grows Down

Increasing Addresses

Stack "Top"
IA32 Stack: Pop

Stack Pointer: %esp

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”
IA32 Stack: Pop

- `popl Dest`

Stack Pointer: `%esp`

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”
IA32 Stack: Pop

Stack Pointer: `%esp`

Stack “Bottom”

Stack Grows Down

Increasing Addresses

Stack “Top”

+4
IA32 Stack: Pop

- popl Dest
  - Read operand at address %esp
  - Increment %esp by 4
  - Write operand to Dest

Stack Pointer: %esp

Stack “Top”

Stack Grows Down

Increasing Addresses

Stack “Bottom”
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 beyond `call`
- Example from disassembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e</td>
<td>e8 3d 06 00 00 00</td>
</tr>
<tr>
<td>8048553</td>
<td>50</td>
</tr>
</tbody>
</table>

- 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

0x108       %esp
0x10c       %eip
0x110

%esp  0x108
%eip  0x804854e
```

%esp: stack pointer
%eip: program counter
Procedure Call Example

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

\%esp:
0x108

\%eip:
0x804854e

\%esp:
0x108

\%eip:
0x804854e

\%esp:
0x104

\%eip:
0x8048553
Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>8048553</td>
<td>50</td>
<td>pushl %eax</td>
</tr>
</tbody>
</table>

---

%esp: 0x108
%eip: 0x804854e

%esp: 0x104
%eip: 0x8048553

program counter

123
Procedure Call Example

8048454e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax

Program counter

%esp 0x108
%eip 0x804854e

%esp 0x104
%eip 0x8048553

call 8048b90
Procedure Call Example

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

Procedure Call Example
```

```
call  8048b90

%esp  0x108
%esp  0x108
%esp  0x104
%esp  0x104

%eip  0x804854e
%eip  0x8048553
+ 0x000063d
```

%eip: program counter
# Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Binary Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804854e</td>
<td>e8 3d 06 00 00</td>
<td>0x108</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>0x8048553</td>
<td>50</td>
<td>0x10c</td>
<td>pushl %eax</td>
</tr>
</tbody>
</table>

**Diagram: Call 8048b90**

- `%esp` 0x108
- `%esp` 0x104
- `%esp` 0x10c
- `%esp` 0x110
- `%eip` 0x804854e
- `%eip` 0x8048553
- `%eip` 0x8048b90

+ 0x000063d

**%eip:** program counter
Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e:</td>
<td>e8 3d 06 00 00</td>
<td>8048b90</td>
<td>call</td>
</tr>
<tr>
<td>8048553:</td>
<td>50</td>
<td>&lt;main&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pushl %eax</td>
<td></td>
</tr>
</tbody>
</table>
Procedure Return Example

8048591: c3 ret

%esp 0x104
0x108 0x8048553
0x10c 123
%eip 0x8048591

%esp 0x104
0x108 0x8048553
0x10c 123
%eip 0x8048591

%eip: program counter
Procedure Return Example

8048591:  c3  ret

%esp  0x104
%eip  0x8048591

ret

8048553:  ret

%esp  0x104
%eip  0x8048591

%esp  0x104
%eip  0x8048591

%eip:  program counter
Procedure Return Example

8048591:   c3  ret

%esp  0x104
%esp  0x108
%esp  0x10c
%esp  0x110
%eip  0x8048553
%eip  0x8048591

ret
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
    - What would happen if code could not be reentrant?
- 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

Procedure \texttt{amI} is recursive (calls itself)
Stack Frames

- **Contents**
  - Local variables
  - Return information
  - Temporary space

- **Management**
  - Space allocated when procedure is entered
    - “Set-up” code
  - Space deallocated upon return
    - “Finish” code
Example

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

Stack

```
%ebp
%esp
```

Diagram:
- `yoo` is at the top of the stack.
- `who()` is called, with `%ebp` and `%esp`指向 `yoo`.
Example

```c
who(...) {
    ... ...
    ami(); ...
    ami(); ...
}
```

Stack

```
Stack

<table>
<thead>
<tr>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>who</td>
</tr>
</tbody>
</table>

```

```plaintext
Stack

<table>
<thead>
<tr>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>yoo</td>
</tr>
</tbody>
</table>
```
Example

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

Stack

- yoo
- who
- amI
- %ebp
- %esp

Example yoo who amI amI amI amI
Example

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

Stack:

- yoo
- who
- amI
- amI
- amI
- %ebp
- %esp
Example

```
void amI(...) {
    ::
    amI();
    ::
}
```

Stack

```
%ebp
%esp
```

```
yoo
who
amI
amI
amI
```
Example

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

Stack

```
%ebp
%esp
```

```
yoo
who
amI
amI
amI
amI
```
Example

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

Stack

```plaintext
Stack:

- yoo
- who
- amI

Variables:
- %ebp
- %esp
```
Example

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

Stack

```
Stack

who

yoo

%ebp

%esp
```

Example yoo who amI amI amI amI %ebp %esp
Example

```c
amI(...) {
    ...
    ...
    ...
}
```

Stack

- yoo
- who
- amI
- %ebp
- %esp

Example:

- `amI(...)`: A C function call.
- `yoo`: A variable or function output.
- `who`: A variable or function output.
- `amI`: A variable or function output.
- `%ebp`: A memory address.
- `%esp`: A memory address.
who(...)  
{  
  ...  
  amI();  
  ...  
}  

Example  

Stack  

Stack diagram:  
- Stack frame with variables named `yoo`, `who`, `%ebp`, and `%esp`.  
- Function call stack showing the invocation of `who(...)`.  
- Variable `amI` is used multiple times in the stack frame.
Example

```cpp
yoo (...) {
  ...
  who();
  ...
}
```

Stack

```
%ebp
yoo
%esp
```

Diagram of stack frame with function `yoo` and variables `who`, `amI`, `amI`, `amI`.
IA32/Linux Stack Frame

- Caller Frame
  - Arguments
  - Return Addr
  - Old %ebp
  - Saved Registers + Local Variables
  - Argument Build

- Frame pointer: %ebp
- Stack pointer: %esp
IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - Old frame pointer
  - Local variables
    - If can’t be just kept in registers
  - Saved register context
    - When reusing registers
  - “Argument build area”
    - Parameters for function about to be called

```
Frame pointer %ebp
Stack pointer %esp
```

```
Caller Frame
Arguments
Return Addr
Old %ebp
Saved Registers + Local Variables
Argument Build
```
IA32/Linux Stack Frame

- Current Stack Frame (“Top” to Bottom)
  - Old frame pointer
  - Local variables
    If can’t be just kept in registers
  - Saved register context
    When reusing registers
  - “Argument build area”
    Parameters for function about to be called
IA32/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - Old frame pointer
  - Local variables
    If can’t be just kept in registers
  - Saved register context
    When reusing registers
  - "Argument build area"
    Parameters for function about to be called

- Caller Stack Frame
  - Return address
    Pushed by `call` instruction
  - Arguments for this call

![Diagram of IA32/Linux Stack Frame](image)
Revisiting swap

```c
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

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

int zip1 = 15213;
int zip2 = 98195;

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

Calling **swap** from **call_swap**

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

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

```c
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;
}
```

Calling swap from call_swap

call_swap:
    ...
    pushl $zip2     # Global Var
    pushl $zip1     # Global Var
    call swap
    ...

Resulting Stack

- %esp
- &zip1
- &zip2
- Rtn adr
Revisiting `swap`

```c
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?

\[
\begin{align*}
\text{pushl } \%ebp \\
\text{movl } \%esp, \%ebp \\
\text{pushl } \%ebx
\end{align*}
\]
swap Setup #1

Entering Stack

Resulting Stack

swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
swap Setup #1

Entering Stack

\[
\text{swap:} \quad \begin{align*}
\text{pushl } &\%ebp \\
\text{movl } &\%esp,\%ebp \\
\text{pushl } &\%ebx
\end{align*}
\]
swap Setup #1

Entering Stack

Resulting Stack

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

%ebp
%esp

%ebp
%esp

&zip2
&zip1
Rtn adr

yp
xp

Rtn adr
Old %ebp
swap Setup #1

Entering Stack

\[
\begin{align*}
\text{swap:} & \\
pushl \ %ebp & \\
movl \ %esp,\ %ebp & \\
pushl \ %ebx & \\
\text{\&}zip2 & \\
\text{\&}zip1 & \\
\text{Rtn adr} & \\
\%esp & \\
\%ebp & \\
\end{align*}
\]
swap Setup #1

Entering Stack

Resulting Stack

Offset relative to new %ebp

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

...
swap Finish #1

swap’s 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
swap Finish #1

swap's 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
swap Finish #1

swap’s Stack

Resulting Stack

Rtn adr

Old %ebp

Old %ebx

%esp

%ebp

%ebp

%esp

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

Observation: Saved and restored register %ebx
swap Finish #2

swap’s Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish #2

swap’s Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish #2

swap’s Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish #3

swap’s 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

movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
swap Finish #4

**swap's Stack**

- yp
- xp
- Rtn adr
- Old %ebp
- Old %ebx

**Resulting Stack**

- %ebp
- %esp

- yp
- xp

---

**Observation**

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

```assembly
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

Calling Code

8048409:  e8 96 ff ff ff ff   call 80483a4 <swap>
804840e:  8b 45 f8         mov    0xfffffffff8(%ebp),%eax
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
```

**Calling Code**

```
8048409:  e8 96 ff ff ff ff   call 80483a4 <swap>
804840e:  8b 45 f8          mov    0xfffffffff8(%ebp),%eax
```
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

Calling Code

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

mov   %ebp,%esp
pop   %ebp

0x0804840e + 0xffffffff = 0x080483a4
swap Finish #4

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

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

- When procedure \texttt{yoo} calls \texttt{who}:
  - \texttt{yoo} is the \texttt{caller}
  - \texttt{who} is the \texttt{callee}

- Can Register be used for temporary storage?
Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the caller
  - `who` is the callee

- Can Register be used for temporary storage?
Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the **caller**
  - `who` is the **callee**

- Can Register be used for temporary storage?
Register Saving Conventions

- When procedure \texttt{yoo} calls \texttt{who}:
  - \texttt{yoo} is the \texttt{caller}
  - \texttt{who} is the \texttt{callee}

- Can Register be used for temporary storage?
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 $98195, %edx
  ... 
  ret
  ```

  - Contents of register `%edx` overwritten by `who`
Saving registers

- When should you save them?
- When should you not save them?
  - Why not save all of them?
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 in its frame before calling
  - “**Callee Save**”
    - Callee saves temporary in its frame before using

- Why do we have these conventions?
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
Recursive Factorial

int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
Recursive Factorial

```c
int rfact(int x) {
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

### Registers

- **%ebx** used, but saved at beginning & restored at end
- **%eax** used without first saving
  - expect caller to save
  - pushed onto stack as parameter for next call

```assembly
rfact:
pushl %ebp
    movl %esp,%ebp
pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
jle .L78
leal -1(%ebx),%eax
pushl %eax
call rfact
imull %ebx,%eax
jmp .L79
.align 4
.L78:
movl $1,%eax
    .L79:
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
**Pointer Code**

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

- **Pass pointer to update location**
Creating & Initializing Pointer

```c
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**

```asm
 pushl %ebp  # Save %ebp
 movl %esp,%ebp  # Set %ebp
 subl $16,%esp  # Add 16 bytes
 movl 8(%ebp),%edx
 movl $1,-4(%ebp)
```

```
<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>4</th>
<th>0</th>
<th>-4</th>
<th>-8</th>
<th>-12</th>
<th>-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>_sfact:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- x
- Rtn adr

---
Creating & Initializing Pointer

```c
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
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```

- 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
Creating & Initializing Pointer

```c
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
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```

```
<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>4</th>
<th>0</th>
<th>-4</th>
<th>-8</th>
<th>-12</th>
<th>-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtn adr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```
Creating & Initializing Pointer

```c
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**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>_sfact:</td>
<td>pushl %ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl %esp,%ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>subl $16,%esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl 8(%ebp),%edx</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>movl $1,-4(%ebp)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rtn adr
Creating & Initializing Pointer

```c
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
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>-16</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Creating & Initializing Pointer

```c
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
   movl %esp,%ebp
   subl $16,%esp
   movl 8(%ebp),%edx
   movl $1,-4(%ebp)
```

```
x
Rtn adr
Old %ebp
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
</tbody>
</table>
```
Creating & Initializing Pointer

```c
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
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>%ebp</td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
</tr>
</tbody>
</table>
Creating & Initializing Pointer

```c
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
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```

<table>
<thead>
<tr>
<th>Rtn adr</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
</tr>
</tbody>
</table>
Creating & Initializing Pointer

```c
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
    movl %esp,%ebp
    subl $16,%esp
    movl 8(%ebp),%edx
    movl $1,-4(%ebp)
```

```
<table>
<thead>
<tr>
<th>Rtn adr</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Old</td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
</tr>
</tbody>
</table>

%ebp

%esp

Temp. Space

x
```
Creating & Initializing Pointer

```c
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
    movl $1,-4(%ebp)
```

```
x
 Rtn adr
+--------+
 | %ebp   |
 +--------+
 0
4
8

| Old %ebp |
+-----------+
| %ebp      |

| Temp. Space |
+-------------+
| %esp        |
```
Creating & Initializing Pointer

```c
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
    movl $1,-4(%ebp)
```

---

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
Creating & Initializing Pointer

```c
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

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

Calling `s_helper` from `sfact`

```
leal -4(%ebp), %eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp), %eax
...  
```

Stack at time of call
Passing Pointer

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

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
```
Passing Pointer

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

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:

```
<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rtn adr</td>
</tr>
<tr>
<td>val = 1</td>
</tr>
<tr>
<td>Unused</td>
</tr>
</tbody>
</table>
```

- Rtn adr
- val = 1
- Unused
- Old ebp
- %ebp

Stack diagram:

- `x`
- `Rtn adr`
- `val = 1`
- `Unused`
- `Old %ebp`
- `%ebp`
Passing Pointer

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

Calling `s_helper` from `sfact`

```asm
leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax
```

Stack at time of call:

- `x`
- `Rtn adr`
- `Old %ebp`
- `val = 1`
- `Unused`
- `%ebp`
Passing Pointer

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

Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax
...  
```

Stack at time of call
Passing Pointer

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

Calling `s_helper` from `sfact`

```assembly
leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax
...```

Stack at time of call
Passing Pointer

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

Calling `s_helper` from `sfact`

```assembly
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
Passing Pointer

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

Calling s_helper from sfact

leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax
...
Passing Pointer

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

Calling `s_helper` from `sfact`

```assembly
leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax
...`

Stack at time of call

```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>-4</td>
<td>val = 1</td>
</tr>
<tr>
<td>-8</td>
<td>Unused</td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
</tr>
</tbody>
</table>
```
Passing Pointer

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

Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax  # Compute &val
pushl %eax          # Push on stack
pushl %edx
 call s_helper      # call
 movl -4(%ebp),%eax  # Return val
 . . .
```
Passing Pointer

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

Calling `s_helper` from `sfact`

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

Calling s_helper from sfact

leal -4(%ebp),%eax
pushl %eax
pushl %edx
    call s_helper
movl -4(%ebp),%eax
...
Passing Pointer

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

Calling `s_helper` from `sfact`

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

Calling s_helper from sfact

leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax

Stack at time of call
Passing Pointer

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

Calling `s_helper` from `sfact`

```assembly
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
```
IA 32 Procedure Summary

### Caller Frame
- Arguments
- Return Addr
- Old ebp
- Saved Registers + Local Variables
- Argument Build

### Registers
- %ebp
- %esp
IA 32 Procedure Summary

- Stack makes recursion work
  - Private storage for each instance of procedure call
    - Instantiations don’t clobber each other
    - Addressing of locals + arguments can be relative to stack positions
  - Managed by stack discipline
    - Procedures return in inverse order of calls
IA 32 Procedure Summary

- **Stack makes recursion work**
  - Private storage for each instance of procedure call
    - Instantiations don’t clobber each other
    - Addressing of locals + arguments can be relative to stack positions
  - Managed by stack discipline
    - Procedures return in inverse order of calls

- **IA32 procedures**
  **Combination of Instructions + Conventions**
  - call / ret instructions
  - Register usage conventions
    - caller / callee save
    - %ebp and %esp