Procedural Memory Usage

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

- Where is the memory that holds t0 and t1 (or local variables in general)?
- What happens if we run out of registers (x86 only has 8)?
- Where are parameters passed from callers to callee? Registers? Memory? What memory?
Stack is a LIFO (Last In, First Out) structure.

It’s allocated somewhere in memory; where doesn’t really matter as long as we store the stack pointer.

By convention, the stack grows toward smaller addresses, but it could be the other way.

Values within the stack are referenced relative to the stack pointer.
- Region of memory managed with stack discipline.
- Grows toward lower addresses.
- Register %esp is the stack pointer, and always points to lowest stack address, which is the top element on the stack.
Pushing

- `pushl Src`
- Decrement `%esp` by 4
- Write operand at address given by `%esp`
IA32 Stack Popping

Popping

- `popl Dest`
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to Dest
Stack Operation Examples

**pushl %eax**

<table>
<thead>
<tr>
<th>%eax</th>
<th>213</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>555</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%eax</th>
<th>213</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>555</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

**popl %edx**

<table>
<thead>
<tr>
<th>%eax</th>
<th>213</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>555</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%eax</th>
<th>213</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>555</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

0x10c 0x108 0x110
0x10c 0x108 0x110
0x104 213
0x104 213
555555
0x108 0x108
213
0x10c 0x108 0x110
0x104 0x104
123 123 123
213 213
pushl %eax popl %edx
Elements for Procedures

- Need to compute address of first instruction of called procedure.
- Place to store passed parameters.
  - Call by value
  - Call by reference
- Need to compute and store return address (first instruction after point of procedure call).
- Need to pass returned value(s) back to the caller.
Procedure Control Flow

We use the stack to support procedure call and return.

**Procedure call:**
Push return address on stack and jump to label

```
call label
```

**Return address value:**
- Address of instruction beyond call site.
- Example from disassembly:

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

**Procedure return:**
Pop address from stack and jump to address.

```
ret
```
Procedure Call Example

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

%eip is the program counter.
Procedure Return Example

8048591: c3 \textit{ret}

\texttt{0x10c}
\texttt{0x108}
\texttt{0x110}
\texttt{123}
\texttt{0x10c}
\texttt{0x108}
\texttt{0x110}
\texttt{123}
\texttt{0x104} \texttt{0x8048553}
\texttt{%eip}
\texttt{%esp} \texttt{0x104}
\texttt{0x8048591}
\texttt{0x104} \texttt{0x8048553}
\texttt{%eip}
\texttt{%esp} \texttt{0x108}
\texttt{0x8048553}

\texttt{0x10c}
\texttt{0x108}
\texttt{0x110}
\texttt{123}
\texttt{0x10c}
\texttt{0x108}
\texttt{0x110}
\texttt{123}
\texttt{0x104} \texttt{0x8048553}
\texttt{%eip}
\texttt{%esp} \texttt{0x104}
\texttt{0x8048591}
\texttt{0x104} \texttt{0x8048553}
\texttt{%eip}
\texttt{%esp} \texttt{0x108}
\texttt{0x8048553}

\%eip is the program counter.
Languages that Support Recursion

- e.g., C, Pascal, Java
- Code must be “reentrant”: Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - arguments
  - local variables
  - return address

Stack Discipline

- State for given procedure needed for a limited time (from call to return)
- Callee always returns before caller does.

Stack is allocated in Frames: state for a single procedure invocation.
Code Structure

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

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

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

Procedure amI is recursive.

```
  yoo
  
  who
  
  amI
  
  amI
  
  amI
```

Procedure amI is recursive.
Stack Frames

Contents
- Local variables
- Return information
- Temporary space

Management
- Space is allocated when you enter the procedure ("set-up" code).
- Space is deallocated at return ("finish" code).

Pointers
- Stack pointer %esp indicates stack top.
- Frame pointer %ebp indicates start of current frame.
Stack Snapshot

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

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

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

Call Chain

```
Frame Pointer
%ebp

yan

who

amI

amI

amI

Stack Pointer
%esp
```

CS429 Slideset 9: 15  Instruction Set Architecture IV
**Current Stack Frame (Top to Bottom)**

- Parameters for function about to call.
- “Argument build”
- 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.

---

**Diagram:**

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

- **Frame Pointer** %ebp
- **Stack Pointer** %esp
Revisiting Swap

```c
int zip1 = 15213;
int zip2 = 91125;

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`

```c
... pushl $zip2 # Global var
pushl $zip1 # Global var
call swap
...
```

Resulting Stack

```
%esp
&zip1
&zip2
Rtn addr
```

CS429 Slideset 9: 17 Instruction Set Architecture IV
Revisiting Swap

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

```
swap:
    # Set up
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    # Body
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    # Finish
    movl −4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
swap:
pushl %ebp  # <-- save old %ebp
movl %esp, %ebp
pushl %ebx

Entering Stack

Resulting Stack

&zip2
&zip1
Rtn addr

Rtn addr
Old %ebp
swap:

pushl %ebp  # save old %ebp
movl %esp, %ebp  # <-- establish new frame
pushl %ebx

Resulting Stack

%esp
&zip2
&zip1
Rtn addr
Old %ebp

%esp
%ebp

Entering Stack

%ebp
&zip2
&zip1
Rtn addr

CS429 Slideset 9: 20  Instruction Set Architecture IV
Swap Setup 3

**swap:**

- `pushl %ebp`  # save old %ebp
- `movl %esp, %ebp`  # establish new frame
- `pushl %ebx`  # <-- save reg. %ebx

![](image)

---

**Entering Stack**

- `%ebp`
  - &zip2
  - &zip1
  - Rtn addr

**Resulting Stack**

- `%ebp`
  - &zip2
  - &zip1
  - Rtn addr
  - Old %ebp
  - Old %ebx

---

Instruction Set Architecture IV
Effect of Swap Setup

Executing the body

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

![Stack diagram]

- Entering Stack:
  - %ebp
  - &zip2
  - &zip1
  - Rtn addr

- Resulting Stack:
  - Offset (relative to %ebp)
  - 12: &zip2
  - 8: &zip1
  - 4: Rtn addr
  - 0: Old %ebp
  - Old %ebx

---

Instruction Set Architecture IV

CS429 Slideset 9: 22
Observation: We’ve saved and restored register `%ebx.

\[
\begin{align*}
\text{movl} & \ -4(%ebp),%ebx \quad \# \leftarrow \text{restore saved } %ebx \\
\text{movl} & \ %ebp,%esp \quad \# \\
\text{popl} & \ %ebp \\
\text{ret}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Offset (relative to <code>%ebp</code>)</th>
<th>12</th>
<th>8</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;zip2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;zip1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtn addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old %ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old %ebx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resulting Stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset (relative to <code>%ebp</code>)</td>
<td>12</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>&amp;zip2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;zip1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtn addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old %ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old %ebx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
movl -4(%ebp),%ebx  # restore saved %ebx
movl %ebp,%esp    # <-- discard callee’s frame
popl %ebp
ret

### Diagram

![Diagram showing stack and memory layout before and after a swap operation.](image-url)
movl -4(%ebp),%ebx  # restore saved %ebx
movl %ebp,%esp    # discard callee’s frame
popl %ebp        # <-- restore old %ebp
ret
Swap Finish 4

movl -4(%ebp),%ebx  # restore saved %ebx
movl %ebp,%esp     # discard callee’s frame
popl %ebp          # restore old %ebp
ret                 # <-- return to the caller

Note: We saved and restored %ebx, but not %eax, %ecx, or %edx
When procedure `yoo` calls `who`: `yoo` is the caller, `who` is the callee.

Can some register be used for temporary storage?

```
yoo:
  ...  
  movl $15213, %edx
  call who
  addl %edx, %eax
  ...
  ret
```

```
who:
  ... 
  movl 8(%ebp), %edx
  addl $91125, %edx
  ...
  ret
```

Contents of register `%edx` are overwritten by `who`. 
When procedure *yoo* calls *who*: *yoo* is the caller, *who* is the callee.

Can some register be used for temporary storage?

**Conventions:**

- “Caller Save” means caller saves temporary in its frame before calling.
- “Callee Save” means callee saves temporary in its frame before using.
## Integer Registers

- Two (%ebp and %esp) have special uses.
- Three (%ebx, %esi, %edi) managed as callee-save.
- Old values saved on stack prior to using.
- Three (%eax, %edx, %ecx) managed as caller-save.
- Do what you please, but expect callee to do so also.
- Register %eax also stores returned value.

<table>
<thead>
<tr>
<th>Reg.</th>
<th>saved by</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>caller</td>
</tr>
<tr>
<td>%edx</td>
<td>caller</td>
</tr>
<tr>
<td>%ecx</td>
<td>caller</td>
</tr>
<tr>
<td>%ebx</td>
<td>callee</td>
</tr>
<tr>
<td>%esi</td>
<td>callee</td>
</tr>
<tr>
<td>%edi</td>
<td>callee</td>
</tr>
<tr>
<td>%esp</td>
<td>special</td>
</tr>
<tr>
<td>%ebp</td>
<td>special</td>
</tr>
</tbody>
</table>
int rfact(int x) {
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}

Registers

- %eax is used without first saving.
- %ebx used, but save at beginning and restore at end.

.globl rfact
.type rfact, @function
rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%ebx
cmpl $1,%ebx
jle .L78
lea −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
Rfact Stack Setup

```assembly
r fact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

![Diagram showing stack setup process](image)
Rfact Body

```assembly
movl  8(%ebp),%ebx  # ebx = x
cmpl $1,%ebx       # compare x:1
jle .L78          # if <= goto Term
leal -1(%ebx),%eax # eax = x-1
pushl %eax        # push x-1
call rfact        # rfact(x-1)
imull %ebx,%eax    # rval * x
jmp .L79          # goto Done

.L78:               # Term:
    movl $1,%eax   # return val = 1
.L79:               # Done:
```

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

Registers:
- `%ebx`: Stored value of x
- `%eax`: Temp value of x-1, Returned value from rfact(x-1), Returned value from this call.
Rfact Recursion

leal -1(%ebx),%eax

<table>
<thead>
<tr>
<th>pre %ebp</th>
<th>pre %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Rtn addr</td>
</tr>
<tr>
<td>Old %ebp</td>
<td></td>
</tr>
<tr>
<td>Old %ebx</td>
<td></td>
</tr>
</tbody>
</table>

%eax  x−1
%ebx  x

pushl %eax

<table>
<thead>
<tr>
<th>pre %ebp</th>
<th>pre %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Rtn addr</td>
</tr>
<tr>
<td>Old %ebp</td>
<td></td>
</tr>
<tr>
<td>Old %ebx</td>
<td></td>
</tr>
</tbody>
</table>

%eax  x−1
%ebx  x

call rfact

<table>
<thead>
<tr>
<th>pre %ebp</th>
<th>pre %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old %ebp</td>
<td></td>
</tr>
<tr>
<td>Old %ebx</td>
<td></td>
</tr>
</tbody>
</table>

%eax  x−1
%ebx  x
We assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`.

```
<table>
<thead>
<tr>
<th>Rtn addr</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old %ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old %ebx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%eax</th>
<th>(x-1)!</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebx</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rtn addr</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old %ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old %ebx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%eax</th>
<th>x!</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebx</td>
<td>x</td>
</tr>
</tbody>
</table>
```

**Return from Call**

```
imull %ebx, %eax

Return from Call
```
movl $-4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
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);
    }
}
```

We pass a pointer to the update location.

Top Level Call

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

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

Use the stack for local variables.

- Variable `val` must be stored on the stack.
- Need to create a pointer for it.
- Compute the pointer as `−4(%ebp)`.
- Push on the stack as the second argument.

Use the stack for local variables.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn addr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>−4</td>
<td>val = 1</td>
</tr>
<tr>
<td>−8</td>
<td></td>
</tr>
<tr>
<td>−12</td>
<td>Unused</td>
</tr>
<tr>
<td>−16</td>
<td></td>
</tr>
</tbody>
</table>
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
Using Pointer

```c
void helper(int x, int *accum)
{
    ... int z = *accum * x;
    *accum = z;
    ...
}
```

- Register `%ecx` holds `x`.
- Register `%edx` holds a pointer to `accum`.
- Use access `%edx`) to reference the memory.
Summary

The stack makes recursion work.

- Private storage for each instance of procedure call.
- Instantiations don’t clobber one another.
- Addressing of locals and arguments can be relative to stack positions.
- This all can be managed by stack discipline.
- Procedures always return in reverse order of the calls.

IA32 procedures are combinations of instructions and conventions.

- Call / Ret instructions.
- Register usage conventions.
- Caller / Callee save conventions.
- Stack frame organization conventions.
- Special registers %esp and %ebp.