Procedures and the Call Stack

Topics

• Procedures
• Call stack
• Procedure/stack instructions
• Calling conventions
• Register-saving conventions
Why Procedures?

Why functions? Why methods?

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

*Procedural Abstraction*
Implementing Procedures

How does a caller pass arguments to a procedure?

How does a caller get a return value from a procedure?

Where does a procedure store local variables?

How does a procedure know where to return (what code to execute next when done)?

How do procedures share limited registers and memory?
Call Chain

Example Call Chain

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

who(...)
{
    
    ru();
    
    ru();
    
}

ru(...)
{
    
    
}
```
What if I want to call a function multiple times?
First Try (broken)

What if I want to call a function multiple times?
Implementing Procedures

How does a caller pass arguments to a procedure?

How does a caller get a return value from a procedure?

Where does a procedure store local variables?

How does a procedure know where to return (what code to execute next when done)?

How do procedures share limited registers and memory?

All these need separate storage per call! (not just per procedure)
Memory Layout

<table>
<thead>
<tr>
<th>Addr</th>
<th>Perm</th>
<th>Contents</th>
<th>Managed by</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^{N-1})</td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Global variables/static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>

- **Addr**: Address range
- **Perm**: Permissions (RW = Read/Write, R = Read, X = Execute only)
- **Contents**: Memory contents
- **Managed by**: Entity responsible for managing the content
- **Initialized**: Timing of initialization
Call Stack

We see x86 organization.
Details differ across architectures, but big ideas are shared.

Region of memory
Managed with stack discipline

%esp holds lowest stack address
(address of "top" element)
IA32 Call Stack: **Push**

```plaintext
pushl Source
```

- **Stack Pointer**: `%esp`
- **Stack “Top”**: Located at lower addresses
- **Stack “Bottom”**: Located at higher addresses
- Stack grows toward lower addresses
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`
IA32 Call Stack: Pop

```.
popl Dest
```

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

Stack Pointer:
IA32 Call Stack: **Pop**

**popl** \( \text{Dest} \)

1. Load value from address \( \%\text{esp} \)
2. Write value to \( \text{Dest} \)
3. Increment \( \%\text{esp} \) by 4

Stack Pointer: \( \%\text{esp} \)

Stack “Top”

Stack “Bottom”

Those bits are still there; we’re just not using them.
Procedure amI is recursive (calls itself)
**Stack frames** support procedure calls.

**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 not just give every *procedure* a permanent chunk of memory to hold its local variables, etc?

---

**Diagram Notes:**
- **Base/Frame Pointer:** %ebp
- **Stack Pointer:** %esp
- **Caller Frame**
- **Frame for current procedure**
  - Stack “Top”
Example

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

Stack

```
%ebp
%esp
yoo
```
Example

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

Stack

yoo

%ebp

%esp

who

amI

amI

amI

amI

amI
Example

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

Stack

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

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

Stack

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

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

Stack

```
%ebp
%esp
amI
amI
amI
yoo
who
```

Diagram showing function calls and stack structure.

"Example"
Example

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

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

Stack

![Diagram of a stack with `amI`, `who`, `yoo`, `%ebp`, and `%esp`]
Example

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

Stack

```plaintext
%ebp
%esp
```

```plaintext
yoo
who
```

Diagram:

- Function `who` calls `amI`.
- `amI` calls itself recursively.
- The stack contains `yoo` and `who` frames.
Example

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

yoo

who

amI

amI

amI

Stack

yoo

who

%ebp

%esp
Example

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

![Diagram of stack and function calls](image-url)
Example

yoo (...) {
  •
  •
  who () ;
  •
}

How did we remember where to point %ebp when returning?
Procedure Control Flow Instructions

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
```

Procedure return: `ret`

1. Pop return address from stack
2. Jump to address
Procedure Call/Return: 1

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</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>

```assembly
804854e:
  call 8048b90 <main>
8048553:
  pushl %eax

%esp = 0x108
%eip = 0x804854e

%eip = instruction pointer = program counter
```
Procedure Call/Return: 2

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

call 8048b90

%esp \(\text{0x110}\) %eip \(\text{0x804854e}\)

0x108 123

%esp 0x108 %esp 0x104

0x10c

0x10c

0x108 123

0x104 0x8048553

%eip 0x804854e %eip 0x8048553

%eip = instruction pointer = program counter
Procedure Call/Return: 3

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

```
call 8048b90
50
```

PC-relative address

```
%esp 0x108
%eip 0x804854e
+ 0x000063d
0x8048b90
```
Procedure Call/Return: 4

8048591: c3 ret

%esp: program counter

%eip: program counter
IA32/Linux Stack Frame

- **Caller Frame**
  - Arguments to callee
  - Return Address
  - Caller's base pointer
  - Saved Registers + Local Variables
  - Arguments for next call

- **Stack Registers**
  - Base/Frame pointer `%ebp`
  - Stack pointer `%esp`
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

Rtn adr

%esp

&zip1

&zip2
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
```
swap Setup #1

**Entering Stack**

- call_swap frame
- &zip2
- &zipl
- Rtn adr

**Resulting Stack**

- yp
- xp
- Rtn adr
- Old %ebp

swap:

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Setup #2

Entering Stack

Resulting Stack

\[
\text{call_swap frame}
\]

\[
\begin{align*}
\text{old ebp} & \quad \text{esp} \\
& \quad \text{zip1} \\
& \quad \text{zip2} \\
& \quad \text{Rtn adr}
\end{align*}
\]

\[
\begin{align*}
\text{yp} & \\
\text{xp} & \\
\text{Rtn adr} & \\
\text{Old ebp}
\end{align*}
\]

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

\[
\text{Set Up}
\]
swap Setup #3

Entering Stack

Resulting Stack

\text{call\_swap\ frame}

\begin{align*}
\text{\%ebp} & \\
\text{\%esp} & \\
\text{\&zip2} & \\
\text{\&zip1} & \\
\text{Rtn\ adr} & \\
\end{align*}

\begin{align*}
\text{\%ebp} & \\
\text{\%esp} & \\
\text{yp} & \\
\text{xp} & \\
\text{Rtn\ adr} & \\
\text{Old\ \%ebp} & \\
\text{Old\ \%ebx} & \\
\end{align*}

\text{swap:}
\begin{align*}
\text{pushl\ \%ebp} \\
\text{movl\ \%esp,\%ebp} \\
\text{pushl\ \%ebx}
\end{align*}

\text{Set Up}
**swap Body**

- **Entering Stack**
  - call_swap frame
  - &zip2
  - &zipl
  - Rtn adr

- **Resulting Stack**
  - Offset relative to new %ebp
  - 12 yp
  - 8 xp
  - 4 Rtn adr

- **Body**
  - movl 12(%ebp),%ecx  # get yp
  - movl  8(%ebp),%edx  # get xp
  - . . .
swap Finish #1

Finishing Stack

Resulting Stack

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

Finishing Stack

```
call_swap frame

•
•
•
yp
xp
Rtn adr
Old %ebp
Old %ebx
```

Resulting Stack

```
•
•
•
yp
xp
Rtn adr
Old %ebp

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

Finish
swap Finish #3

Finishing Stack

call_swap frame

...%ebp
xp
Rtn adr
Old%ebp
Old%ebx

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

Resulting Stack

...%ebp
xp
Rtn adr
%ebp
%esp

Finish
swap Finish #4

Finishing 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
80483b8:  c3          ret

mov    %ebp,%esp
pop    %ebp
leave

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

relative address (little endian)
**swap Finish #4**

**Finishing Stack**

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

**Resulting Stack**

Observation

- Saved & restored register `%ebx`
- but not `%eax`, `%ecx`, or `%edx`
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 \texttt{yoo} calls \texttt{who}:

- \texttt{yoo} is the \textit{caller}
- \texttt{who} is the \textit{callee}

Will register contents still be there after a procedure call?

Conventions

\textit{Caller Save}

- \texttt{Caller} saves temporary values in its frame \textit{before calling}

\textit{Callee Save}

- \texttt{Callee} saves temporary values in its frame \textit{before using}
**IA32/Linux Register Saving Conventions**

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax, %edx, %ecx</td>
<td><strong>Caller saves</strong> prior to call if needs values after call</td>
</tr>
<tr>
<td>%eax</td>
<td>also used to <strong>return</strong> integer value</td>
</tr>
<tr>
<td>%ebx, %esi, %edi</td>
<td><strong>Callee saves</strong> if will use</td>
</tr>
<tr>
<td>%esp, %ebp</td>
<td>special form of callee save restored to original values before returning</td>
</tr>
</tbody>
</table>

**Diagram**

- **Caller-Save Temporaries**:
  - %eax
  - %edx
  - %ecx

- **Callee-Save Temporaries**:
  - %ebx
  - %esi
  - %edi

- **Special**:
  - %esp
  - %ebp
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

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

Write the C function header, types, and order of parameters.
Example: **Pointers to Local Variables**

**Top-Level Call**

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

**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);
    }
}
```

sfact(3) \( \text{val} = 1 \)

s_helper(3, &val) \( \text{val} = 3 \)

s_helper(2, &val) \( \text{val} = 6 \)

s_helper(1, &val) \( \text{val} = 6. \)

Pass pointer to update location
Creating & Initializing Pointer

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

Must store `val` in memory (stack); registers do not have addresses.

**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;
}
```

Must store `val` in memory (stack); registers do not have addresses.

**Stack at time of call:**

```
<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>val=x!</td>
</tr>
<tr>
<td>-8</td>
<td>Unused</td>
</tr>
<tr>
<td>-12</td>
<td>&amp;val</td>
</tr>
<tr>
<td>-16</td>
<td>x</td>
</tr>
</tbody>
</table>
```

**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
```
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.