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 answer these questions, we need a call stack...
**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

**Stack Pointer: `%esp`**

- Stack “Bottom”
- Stack Grows Down
- Increasing Addresses

**IA32 Call Stack: Push**

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

**Stack Pointer: `%esp`**

- Stack “Bottom”
- Stack Grows Down
- Increasing Addresses

**IA32 Call Stack: Pop**

- `popl Dest`

**Stack Pointer: `%esp`**

- Stack “Bottom”
- Stack Grows Down
- Increasing Addresses
IA32 Call Stack: Pop

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

Procedures and Stacks

Procedure Call Overview

- **Caller**
  - `<save regs>`
  - `<set up args>`
  - `call`
  - `<clean up args>`
  - `<find return val>`
  - ...%

- **Callee**
  - `<create local vars>`
  - `<set up return val>`
  - `<destroy local vars>`
  - `return`

- **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
  - So how do we deal with register reuse?

Procedure Call Overview

- **Callee**
  - `<save regs>`
  - `<set up args>`
  - `call`
  - `<clean up args>`
  - `<restore regs>`
  - `<find return val>`
  - ...

- **Callee**
  - `<create local vars>`
  - `<set up return val>`
  - `<destroy local vars>`
  - `return`

- The convention of where to leave/find things is called the calling convention (or 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 *(why?, and which exact address?)*
    - Jump to `label`

  **Procedure Call Example**

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

  ![Call Stack Diagram]

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

  ![Call Stack Diagram with Variables]

  ![Program Counter Diagram]

  **Return address:**
  - Address of instruction after `call`
  - Example from disassembly:

    ```asm
    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

  ![Call Stack Diagram with Return Address]

  ![Program Counter Diagram with Return Address]
Procedure Call Example

```
0x804854e: e8 3d 06 00 00  call   8048b90 <main>
0x8048553: 50                  pushl %eax
```

Relative address just like jumps... (chosen by compiler; there's also an absolute call)

Procedure Return Example

```
8048591: c3        ret
```

Procedure Return Example

```
8048591: c3        ret
```
Procedure Return Example

```
8048591: c3 ret
```

```
ret

0x110 0x110
0x10c 0x10c
0x108 123 0x108 123
0x104 0x8048553 0x8048553
%esp 0x104 %esp 0x108
%eip 0x8048591 %eip 0x8048553
```

%eip: program counter

Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java, most modern languages
  - 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();
    ...
}
```

Procedure amI is recursive (calls itself)

Example Call Chain

```
yoo
  who
    amI
      amI
```

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

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

Example

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

Example

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

Example

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

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

Example

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

How did we remember where to point `%ebp` when returning?

### 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

```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

```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

```assembly
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
```

we know the address
not Global Var!
**swap Setup #1**

**Entering Stack**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`

**Resulting Stack?**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`

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

**Set Up**

**ResulJng Stack**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`

- `&zip2`
- `&zip1`

- `Rtn addr`  

- `Set`  

- `adr`  

- `%esp`

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

**Set Up**

**swap Setup #2**

**Entering Stack**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`

**Resulting Stack**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`

- `&zip2`
- `&zip1`

- `Rtn addr`

- `Old %ebp`

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

**Set Up**

**swap Setup #3**

**Entering Stack**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`

**Resulting Stack**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`

- `&zip2`
- `&zip1`

- `Rtn addr`

- `Old %ebp`

- `Old %ebx`

- `Old %esp`

- `Old %ebp`

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

**Set Up**
**swap Body**

**Entering Stack**

- %ebp
- zip2
- zip1
- Rtn adr
- %esp

**Resulting Stack**

- Offset relative to new %ebp
- 12
- yp
- 8
- xp
- Rtn adr
- %ebp
- %esp
- Old %ebp
- Old %ebx

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

**Body**

**swap Finish #1**

**Finishing Stack**

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

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

**Resulting Stack**

**Observation:** Saved and restored register %ebx

**swap Finish #2**

**Finishing Stack**

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

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

**Resulting Stack**
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 19          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  call 80483a4 <swap>
804840a:   8b 45 f8          mov   0xfffffffff8(%ebp),%eax

\[\text{movl } -4(%ebp),%ebx\]
\[\text{movl } %ebp,\%esp\]
\[\text{popl } \%ebp\]
\[\text{ret}\]
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`

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

```
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)

Pass pointer to update location
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);
    }
}
```

Pass pointer to update location

sfact(3) val = 1
s_helper(3, &val)

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);
    }
}
```

Pass pointer to update location

sfact(3) val = 1
s_helper(3, &val)

Creating & Initializing Pointer

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

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
```

- 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

```
8  x
4  Rtn adr
0  %esp
%esp
%ebp
%ebp
%ebp

-4 val = 1
-8
-12 Unused
-16
```

Pass pointer to update location
### Passing 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

#### Calling `s_helper` from `sfact`

```c
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:

```
+---+---+---+---+---+---+---+
| x | Rtn adr | 4 | Old %ebp | %ebp | −4 | −8 |
|---+---+---+---+---+---+---+
| −12 | Unused | %esp | −16 | &val | x |
```

### 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`

### x86-64 Procedure Calling Convention

- **Doubling of registers makes us less dependent on stack**
  - Store argument in registers
  - Store temporary variables in registers

- What do we do if we have too many arguments or too many temporary variables?

### x86-64 64-bit Registers: Usage Conventions

- **%rax**: Return value
- **%r8** : Argument #5
- **%rbx** : Callee saved
- **%r9** : Argument #6
- **%rcx** : Argument #4
- **%r10**: Caller saved
- **%rdx** : Argument #3
- **%r11**: Caller Saved
- **%rsi** : Argument #2
- **%r12**: Callee saved
- **%rdi** : Argument #1
- **%r13**: Callee saved
- **%r8p**: Stack pointer
- **%r14**: Callee saved
- **%r15**: Callee saved
Revisiting swap, IA32 vs. x86-64 versions

**Set Up**

\[
\text{swap:} \\
\begin{align*}
pushl &\%ebp \\
movl &\%esp, &\%ebp \\
pushl &\%ebx \\
movl 12(\%ebp), &\%ecx \\
movl 8(\%ebp), &\%edx \\
movl (\%ecx), &\%eax \\
movl (\%edx), &\%ebx \\
movl &\%eax, (\%edx) \\
movl &\%ebx, (\%ecx) \\
movl -4(\%ebp), &\%ebx \\
movl &\%ebp, &\%esp \\
popl &\%ebp \\
ret
\end{align*}
\]

**Body**

**Arguments passed in registers**
- First \((x)p\) in \(%rdi\), second \((y)p\) in \(%rsi\)
- 64-bit pointers

**No stack operations required (except `ret`)**

**Avoiding stack**
- Can hold all local information in registers

**Finish**

\[
\text{swap (64-bit long ints):} \\
\begin{align*}
\text{movq} &\%rdi, &\%rdx \\
\text{movq} &\%rsi, &\%rax \\
\text{movq} &\%rax, (\%rdi) \\
\text{movq} &\%rdx, (\%rsi) \\
\text{ret}
\end{align*}
\]

X86-64 procedure call highlights

- **Arguments (up to first 6) in registers**
  - Faster to get these values from registers than from stack in memory
- **Local variables also in registers (if there is room)**
- **Registers still designated “caller-saved” or “callee-saved”**
- **`callq` instruction stores 64-bit return address on stack**
  - Address pushed onto stack, decrementing %rsp by 8
- **No frame pointer**
  - All references to stack frame made relative to %rsp; eliminates need to update %ebp/%rbp, which is now available for general-purpose use
- **Functions can access memory up to 128 bytes beyond %rsp: the “red zone”**
  - Can store some temps on stack without altering %rsp

x86-64 Memory Layout

- Stack
- Dynamic Data (Heap)
- Static Data
- Literals
- Instructions

- 128-byte **red zone** space lower than the stack pointer that procedures can use for data not needed across procedure calls

- Optimization to avoid extra %rsp updates

x86-64 Stack Frames

- **Often (ideally), x86-64 functions need no stack frame at all**
  - Just a return address is pushed onto the stack when a function call is made
- **A function does need a stack frame when it:**
  - Has too many local variables to hold in registers
  - Has local variables that are arrays or structs
  - Uses the address-of operator (&) to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Needs to save the state of caller-save registers before calling a procedure
  - Needs to save the state of callee-save registers before modifying them
Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

NB: Details may vary depending on compiler.

Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

Note: Sizes
### Example 1

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, x1, x2, x2, x3, x3, x4, x4);
    return (x1+x2)*(x3-x4);
}
```

**Return address to caller of call_proc**

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arg 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arg 7</td>
<td></td>
</tr>
</tbody>
</table>

**Return address to line after call to proc**

%rsp

### Example 2

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

**Return address to caller of call_proc**

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arg 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arg 7</td>
<td></td>
</tr>
</tbody>
</table>

%rsp

### Example 3

```c
sign extension!
```

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

**Return address to caller of call_proc**

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arg 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arg 7</td>
<td></td>
</tr>
</tbody>
</table>

%rsp

### x86-64 Procedure Summary

- **Heavy use of registers (faster than using stack in memory)**
  - Parameter passing
  - More temporaries since more registers

- **Minimal use of stack**
  - Sometimes none
  - When needed, allocate/deallocate entire frame at once
  - No more frame pointer: address relative to stack pointer

- **More room for compiler optimizations**
  - Prefer to store data in registers rather than memory
  - Minimize modifications to stack pointer