The Hardware/Software Interface
CSE351 Winter 2013

Procedures and Stacks I

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 ...

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

Assembly language:

Machine code:

Computer system:

OS:

Memory Layout

2^{N-1}

Stack

Dynamic Data (Heap)

Static Data

Literals

Instructions

local variables; procedure context

variables allocated with new or malloc

static variables (including global variables (C))
literals (e.g., "example")
### Memory Layout

<table>
<thead>
<tr>
<th>Stack</th>
<th>Dynamic Data (Heap)</th>
<th>Static Data</th>
<th>Literals</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>writable; not executable</td>
<td>Managed by programmer</td>
<td>Initialized when process starts</td>
<td>Initialized when process starts</td>
<td>Initialized when process starts</td>
</tr>
</tbody>
</table>

**Heap**
- Managed “automatically” (by compiler)
- Writable; not executable

**Stack**
- Managed by programmer
- Initialized when process starts
- Writable; not executable

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

**Instructions**
- Read-only; executable

### 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

**Register `%esp`**
- Contains lowest stack address
- Address of “top” element

**Stack Pointer**
- `%esp`

**Stack “Top”**
- Stack grows downward

**Stack “Bottom”**
- Stack grows downward

**IA32 Call Stack: Push**

- `pushl Src`

- **Stack “Bottom”**

- **Stack “Top”**

  - Stack grows downward

- **Increasing Addresses**

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

**Stack Pointer**
- `%esp`

**Stack Grows Down**

**Increasing Addresses**

**Stack “Bottom”**

**Stack “Top”**

**Stack Grows Down**

**Stack “Bottom”**

**Stack “Top”**
IA32 Call Stack: Pop

- **popl Dest**

![Stack Diagram]

Stack Pointer: %esp

Stack Grows Down

Increasing Addresses

Stack “Bottom”

Stack “Top”

Procedure Call Overview

**Caller**
-...<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
  - **Caller** might need to save registers that **Callee** might use
  - **Callee** might need to save registers that **Caller** has used

- 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 call:**
- `call label`
  - Push return address on stack
  - Jump to `label`

**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

**Procedure Call Example**

```
0x108 0x10c 0x110 0x10e
%esp  %eip
%esp  %eip
```

```
0x104 0x10c 0x110 0x10e
%esp  %eip
%esp  %eip
```
Procedure Call Example

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

0x110 0x110
0x10c 0x10c
0x108 123  0x108 123
0x104

%esp 0x108 %esp 0x108
%eip 0x804854e %eip 0x8048553
```

Procedure Return Example

```
8048591:   c3      ret

0x110 0x110
0x10c 0x10c
0x108 123
0x104 0x8048553

%esp 0x108 %esp 0x104
%eip 0x804854e %eip 0x8048591
```
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
    - Caller always returns before caller does
- Stack allocated in frames
  - State for a single procedure instantiation
**Call Chain Example**

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

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

**Example Call Chain**

```
yoo
  who
  amI
  amI
  amI
```

Procedure `amI` is recursive (calls itself)

**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();
  ...
}
```

```
yoo
  who
  amI
  amI
  amI
```

Stack Pointer: `%esp`

Frame Pointer: `%ebp`

Frame for current proc

Previous Frame

Stack “Top”
Example:
```
amI(…)
{...
  amI();
  ...
}
```

Stack:
```
who

yoo

amI

%ebp
%esp
```

Example:
```
who(…)
{...
  amI();
  ...
}
```

Stack:
```
who

yoo

%ebp
%esp
```

Example:
```
amI(…)
{...
  ...
  amI()
  ...
}
```

Stack:
```
who

yoo

%ebp
%esp
```

Example:
```
who(…)
{...
  amI();
  ...
}
```

Stack:
```
who

yoo

%ebp
%esp
```
Example

```c
void 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

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

```c
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);
}

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

```asm
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,%ecx
    movl %edx,%ebx
    movl %ecx,%eax
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
### swap Setup #2

**Entering Stack**

- \%ebp
- \%esp
- \&zip2
- \&zip1
- Rtn addr

**Resulting Stack**

- \%ebp
- \%esp
- yp
- xp
- Rtn addr
- Old \%ebp

**swap**:

- pushl \%ebp
- movl \%esp, \%ebp
- pushl \%ebx

### swap Setup #3

**Entering Stack**

- \%ebp
- \%esp
- yp
- xp
- \&zip2
- \&zip1
- Rtn addr

**Resulting Stack**

- \%ebp
- \%esp
- yp
- xp
- \&zip2
- \&zip1
- Rtn addr
- Old \%ebp
- Old \%ebx

**swap**:

- pushl \%ebp
- movl \%esp, \%ebp
- pushl \%ebx

### swap Body

**Entering Stack**

- \%ebp
- \%esp
- \&zip2
- \&zip1
- Rtn addr

**Resulting Stack**

- Offset relative to new \%ebp

- 12
- 8
- yp
- xp
- Rtn addr
- Old \%ebp
- Old \%ebx

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

### swap Finish #1

**swap’s Stack**

- \%ebp
- \%esp
- yp
- xp
- Rtn addr
- Old \%ebp
- Old \%ebx

**Resulting Stack?**

- \%ebp
- \%esp
- yp
- xp
- \&zip2
- \&zip1
- Rtn addr
- Old \%ebp
- Old \%ebx

movl -4(\%ebp), %ebx
movl \%ebp, %esp
popl \%ebp
ret
Observation: Saved and restored register %ebx
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
```
8048e09:  e8 00 ff ff ff  call  80483a4 <swap>
8048e0e:  8b 45 f8      mov    0xffffffff(%ebp),%eax
```

\[ \text{0x08048e0e} + 0xffffffff = 0x080483a4 \]

### Register Saving Conventions

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

- Can a register be used for temporary storage?

  - Contents of register `%edx` overwritten by `who`
IA32/Linux Register Usage

- **%eax, %edx, %ecx**
  - Caller-Save Temporaries
    - %eax
    - %edx
    - %ecx
- **%eax**
  - also used to return integer value
- **%ebx, %esi, %edi**
  - Caller saves if wants to use them
- **%esp, %ebp**
  - special form of callee save – restored to original values upon exit from procedure

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 0(%ebp), %edx # edx = x
    movl $1,-4(%ebp) # val = 1
```

Example: Pointers to Local Variables

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

Top-Level Call

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

Passing Pointer

- 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

### Stack at time of call:

```
     8       4       0
     %esp    %ebp   %esp
```

### 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
```
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

<table>
<thead>
<tr>
<th>Caller Frame</th>
<th>Arguments</th>
<th>Return Addr</th>
<th>Old %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>Saved Registers + Local Variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>Argument Build</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>