Machine-Level Programming III: Procedures

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/CSCE230J

Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
- I have modified them and added new slides.
Topics

- IA32 stack discipline
- Register saving conventions
- Creating pointers to local variables

IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp indicates lowest stack address
  - address of top element

![Stack Diagram](image-url)
IA32 Stack Pushing

Pushing
- `pushl Src`
- Fetch operand at `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**

```
<table>
<thead>
<tr>
<th></th>
<th>pushl %eax</th>
<th></th>
<th>popl %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td></td>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x108</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eax</td>
<td>213</td>
<td>%eax</td>
<td>213</td>
</tr>
<tr>
<td>%edx</td>
<td>555</td>
<td>%edx</td>
<td>213</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
<td>%esp</td>
<td>0x104</td>
</tr>
</tbody>
</table>
```

**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 value**

- Address of instruction beyond call
- Example from disassembly
  - `804854e: e8 3d 06 00 00  call  8048b90 <main>`
  - `8048553: 50  pushl %eax`
  - 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

call  8048b90

- `%esp 0x108`
- `%esp 0x104`
- `%eip 0x804854e`
- `%eip 0x8048b90`

%eip is program counter

**Procedure Return Example**

8048591: c3  ret

- `%esp 0x108`
- `%esp 0x104`
- `%eip 0x8048591`
- `%eip 0x8048553`

%eip is program counter
Stack-Based Languages

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 pointer

Stack Discipline
- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

Stack Allocated in Frames
- state for single procedure instantiation

Call Chain Example

Code Structure

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

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

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

- Procedure amI recursive

Call Chain

```
yoo
  ↓
  who
  ↓
  amI
  ↓
  amI
  ↓
  amI
```

Page 6
Stack Frames

Contents
- Local variables
- Return information
- Temporary space

Management
- Space allocated when enter procedure
  - “Set-up” code
- Deallocated when return
  - “Finish” code

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

Stack Operation

Call Chain
Stack Operation

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

Call Chain

```
Frame Pointer %ebp
Stack Pointer %esp
```

Stack Operation

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

Call Chain

```
Frame Pointer %ebp
Stack Pointer %esp
```

Page 8
Stack Operation

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

Call Chain

```
yoo
  └→ who
    └→ amI
      └→ amI
```

Frame Pointer
%ebp
Stack Pointer
%esp

---

Stack Operation

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

Call Chain

```
yoo
  └→ who
    └→ amI
      └→ amI
```

Frame Pointer
%ebp
Stack Pointer
%esp

---

Page 9
Stack Operation

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

Call Chain

```
Yoo -> Who -> AmI
```

Stack Pointer %esp
Frame Pointer %ebp

Page 10
Stack Operation

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

Call Chain

yoo
who
amI
amI
amI

Stack Pointer %esp
Frame Pointer %ebp

Stack Operation

amI (...) {
    
}

Call Chain

yoo
who
amI
amI
amI
amI

Stack Pointer %esp
Frame Pointer %ebp

Page 11
**Stack Operation**

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

amI

amI

amI

amI

amI

amI

amI

Frames

Call Chain

yoo

who

amI

amI

amI

amI

amI

amI

amI

amI

amI

Frames
Revisiting \texttt{swap}

\begin{verbatim}
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;
}
\end{verbatim}

Calling \texttt{swap} from \texttt{call\_swap}

\begin{verbatim}
call_swap:
    * * *
push $zip2  # Global Var
push $zip1  # Global Var
call swap
    * * *
\end{verbatim}
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`

**Set Up**
- `movl 12(%ebp),%ecx`
- `movl 8(%ebp),%edx`
- `movl (%ecx),%eax`
- `movl (%edx),%ebx`
- `movl %eax,(%edx)`
- `movl %ebx,(%ecx)`

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

**Finish**

---

**swap Setup #1**

**Entering Stack**
- `%ebp`
- `%esp`
- `&zip2`
- `&zip1`
- `Rtn adr`

**Resulting Stack**
- `%ebp`
- `%esp`
- `yp`
- `xp`
- `Rtn addr`
- `Old %ebp`
### swap Setup #2

**Entering Stack**

- \( %ebp \)
- \( \&\text{zip2} \)
- \( \&\text{zip1} \)
- \( \text{Rtn adr} \)

**Resulting Stack**

- \( %ebp \)
- \( \text{yp} \)
- \( \text{xp} \)
- \( \text{Rtn adr} \)
- \( \text{Old %ebp} \)

**swap**:

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

### swap Setup #3

**Entering Stack**

- \( %ebp \)
- \( \&\text{zip2} \)
- \( \&\text{zip1} \)
- \( \text{Rtn adr} \)

**Resulting Stack**

- \( %ebp \)
- \( \text{yp} \)
- \( \text{xp} \)
- \( \text{Rtn adr} \)
- \( \text{Old %ebp} \)
- \( \text{Old %ebx} \)

**swap**:

- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
Effect of swap Setup

- **Entering Stack**
  - %ebp
  - Offsets (relative to %ebp): 12, 8, 4, 0
  - Variables: %ebp0, %esp

- **Resulting Stack**
  - %ebp
  - Variables: %ebp0, %esp

Body:

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

**Observation**

- Saved & restored register %ebx

swap Finish #1

- **swap's Stack**
  - Offsets: 12, 8, 4, 0
  - Variables: %ebp0, %esp

- **Offset**
  - 12: yp
  - 8: xp
  - 4: Rtn adr
  - 0: Old %ebp
  - 0: Old %ebx

- **Observation**

  - Saved & restored register %ebx

Body:

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

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

**swap Finish #3**

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

#### Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Exiting Stack

<table>
<thead>
<tr>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;zip1</td>
</tr>
<tr>
<td>&amp;zip2</td>
</tr>
</tbody>
</table>

#### Observation

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

#### Code Snippet

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

---

### 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 $91125, %edx
    . . .
    ret
```

- Contents of register %edx overwritten by who
Register Saving Conventions

When procedure you calls who:
- you 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

IA32/Linux Register Usage

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

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

Registers
- `%eax` used without first saving
- `%ebx` used, but save at beginning & 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
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
```

Rfact Stack Setup

---

**Entering Stack**

**Pre:**
- `%ebp` to `%ebp`
- `%ebx` to `%ebx`
- `x` to `%eax`
- `Rtn adr` to `%esp`

---

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

- Temporary value of x-1
- Returned value from rfact(x-1)
- Returned value from this call
### Rfact Result

**Return from Call**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>Rtn adr</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Old ebp</td>
<td>%ebp</td>
</tr>
<tr>
<td></td>
<td>Old ebx</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x-1$</td>
<td>%esp</td>
</tr>
<tr>
<td>%eax</td>
<td>$(x-1)!$</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>$x$</td>
<td></td>
</tr>
</tbody>
</table>

Assume that $rfact(x-1)$ returns $(x-1)!$ in register %eax

**Rfact Completion**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>Rtn adr</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Old ebp</td>
<td>%ebp</td>
</tr>
<tr>
<td></td>
<td>Old ebx</td>
<td>%esp</td>
</tr>
<tr>
<td></td>
<td>$x-1$</td>
<td>%esp</td>
</tr>
<tr>
<td>%eax</td>
<td>$x!$</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>Old ebx</td>
<td></td>
</tr>
</tbody>
</table>

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

43
**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);
    }
}
```

*Pass pointer to update location*

---

**Creating & Initializing Pointer**

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
```

Using Stack for Local Variable

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

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

### Calling s_helper from sfact

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

Stack at time of call

- **8** x
- **4** Rtn adr
- **0** Old %ebp
- **-4** val = x!
- **-8** Unused
- **-12** &val
- **-16** x
- **-20** %esp

```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
```

## Using Pointer

### void s_helper

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

- **%edx** holds x
- **%ecx** holds pointer to *accum
  - Use access (*%edx) to reference memory
Summary

The 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
- Can be 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
- Stack frame organization conventions