Machine-Level Programming III: Procedures

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 $\text{esp}$ indicates lowest stack address
  - address of top element
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

**Pushl %eax**

- 0x108: 123
- 0x10c: 0
- 0x110: 0
- 0x104: 213

**Popl %edx**

- 0x108: 123
- 0x10c: 0
- 0x110: 0
- 0x104: 213

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

- Stack memory layout before push:
  - 0x108: 123
  - 0x10c: 0
  - 0x110: 0
  - 0x104: 213

- Stack memory layout after push:
  - 0x108: 123
  - 0x10c: 0
  - 0x110: 0
  - 0x104: 213

- Stack memory layout after pop:
  - 0x108: 123
  - 0x10c: 0
  - 0x110: 0
  - 0x104: 213

- Stack memory layout after pop:
  - 0x108: 123
  - 0x10c: 0
  - 0x110: 0
  - 0x104: 213
Procedure Control Flow

- Use stack to support procedure call and return

Procedure call:

```plaintext
call label                      Push return address on stack;
    Jump to label
```

Return address value

- Address of instruction beyond call
- Example from disassembly
  ```plaintext
  804854e: e8 3d 06 00 00 call 8048b90 <main>
  8048553: 50 pushl %eax
  Return address = 0x8048553
  ```

Procedure return:

- ```plaintext
  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 %eip 0x804854e
%esp 0x104
%esp 0x8048553
%eip is program counter
Procedure Return Example

8048591: c3

%esp 0x104
%eip 0x8048591

0x104 0x108 0x10c 0x110
123
0x8048553

%esp 0x104
%eip 0x8048591

0x110 0x10c 0x108 0x110
123
0x8048553

%esp 0x108
%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

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

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

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

- **Procedure** `amI`
  - recursive
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

```c
yoo (...) {
    •
    •
    who();
    •
    •
}
```
Stack Operation

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

Call Chain

- Frame Pointer: `%ebp`
- Stack Pointer: `%esp`

Diagram:
- Stack operation with call chain:
  - `who(…)`
  - `amI();`
  - `amI();`
  - Pointer relationships:
    - `%ebp` to `yoo`
    - `%esp` to `who`
Stack Operation

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

Call Chain

- `amI` (Frame Pointer)
- `%ebp`
- `who` (Stack Pointer)
- `%esp`
- `yoo`
Stack Operation

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

Call Chain

![Call Chain Diagram]

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

- `yoo`
- `who`
- `amI`
- `amI`
Stack Operation

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

Call Chain

- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
- `amI`

Frame Pointer
- `%ebp`

Stack Pointer
- `%esp`
Stack Operation

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

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

Call Chain

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

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

Call Chain

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

Diagram:
- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
- `amI`
Stack Operation

```
amI(...) {
  •
  •
  •
}
```

Call Chain

Frame Pointer `%ebp`

Stack Pointer `%esp`
Stack Operation

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

who

amI

yoo

- 21 -
Stack Operation

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

Call Chain

```
```

Frame Pointer
%ebp

Stack Pointer
%esp

- 22 -
IA32/Linux Stack Frame

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
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

call_swap:

```
    pushl $zip2       # Global Var
    pushl $zip1       # Global Var
    call swap
```

Resulting Stack

High

- - -

Low

- - -

Rtn adr

%esp

&zip2

&zip1
Revisiting swap

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
swap Setup #1

**Entering Stack**

- %ebp
- %esp

- &zip2
- &zip1
- Rtn adr

**Resulting Stack**

- %ebp
- %esp

- YP
- xp
- Rtn adr
- Old %ebp

**swap:**

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

Entering Stack

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

Resulting Stack

\[
\begin{align*}
\text{\%ebp} & \quad \text{\%esp} \\
\text{YP} & \quad \text{xp} \\
\text{Rtn adr} & \quad \text{Old \%ebp}
\end{align*}
\]

\[
\text{swap:}
\begin{align*}
\text{pushl \%ebp} \\
\text{movl \%esp,\%ebp} \\
\text{pushl \%ebx}
\end{align*}
\]
swap Setup #3

Entering Stack

Resulting Stack

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

Entering Stack

Offset (relative to %ebp)

Resulting Stack

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

Body
swap Finish #1

Observation
- Saved & restored register %ebx

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

swap’s Stack

Offset
12
8
4
0

Old %ebp

%ebp

%esp

swap’s Stack

Offset
12
8
4

%ebp

%esp

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

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

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?

- 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

```assembly
.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
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
Rfact Stack Setup

Entering Stack

rfact:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  pre %ebp
  pre %ebx
  x
  Rtn adr
  %ebp
  %esp

Caller

Callee

0
-4
4
8

Old %ebp
Old %ebx
Rtn adr
x
pre %ebp
pre %ebx

%ebp
%esp
int rfact(int x) {
    int rval;
    if (x <= 1) {
        return 1;
    }
    rval = rfact(x-1);
    return rval * x;
}

movl 8(%ebp),%ebx  # ebx = x
cmp %eax,45678h  # 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 %eax,%eax  # rval * x
jmp .L79  # Goto done
.L78:  # Term:
movl $1,%eax  # return val = 1
.L79:  # Done:

int rfact(int 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 Recursion

leal -1(%ebx),%eax

- pushl %eax
- call rfact

%eax  x-1
%ebx  x
%eax  x-1
%ebx  x
%eax  x-1
%ebx  x
%eax  x-1
%ebx  x

Assume that \( \text{rfact}(x-1) \) returns \((x-1)!\) in register \( \%eax \)
Rfact Completion

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

The Stack Makes Recursion Work

- Private storage for each \textit{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