Machine-Level Programming III - Procedures

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

Next time
- Structured data
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register $\%esp$ 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**
  - Before: %eax 555, %edx 213, %esp 0x108
  - After: %eax 123, %edx 0x104, %esp 0x110

- **popl %edx**
  - Before: %eax 123, %edx 213, %esp 0x110
  - After: %eax 555, %edx 0x108, %esp 0x104
Procedure control flow

- Use stack to support procedure call and return
- Procedure call
  ```
  call label     Push return address on stack; jump to label
  call *Operand Similar, but indirect
  ```
- Return address value
  - Address of instruction immediately following `call`
  - Example from disassembly
    ```
    | Address       | Instruction                  |
    |---------------|-----------------------------|
    | 804854e:      | e8 3d 06 00 00 call 8048b90 <main> |
    | 8048553:      | 50 pushl %eax               |
    ```
    - Return address = 0x8048553
- Procedure return
  - `leave` Prepare stack for return
  - `ret` Pop address from stack; jump to address
    (stack should be ready)
Procedure call example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804854e</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>0x8048553</td>
<td>50</td>
<td>pushl %eax</td>
</tr>
</tbody>
</table>

- `call 8048b90`: Call instruction to 8048b90
- `%esp`: Stack pointer
- `%eip`: Program counter

**Diagram:***
- `%esp`: 0x108
- `%eip`: 0x804854e
- `%esp`: 0x104
- `%eip`: 0x8048b90

**Push Return Address on Stack:**
- `pushl %eax` pushes the return address onto the stack.
- `jump to 0x8048b90` jumps to the address specified in the return address on the stack.
Procedure return example

8048591: c3 ret

ret

ret Pop address from stack; jump to address

%esp 0x104 %esp 0x108
%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

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

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

}

amI(...)
{
  
  
  amI();

}
```

Procedure `amI` recursive

Call Chain

```
yoo

who

amI  amI

amI

amI
```

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

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

Call Chain

```
Frame Pointer
%ebp

Stack Pointer
%esp

yoo

yoo
```
Stack operation

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo
who
Stack operation

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

Call Chain

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

Frame Pointer: `%ebp`

Stack Pointer: `%esp`
Stack operation

Call Chain

```
yoo(…)
{
  
  who(…)
  {
    
    amI(…)
    {
      
        amI(…)
        {
          
            amI()
          
        }
        
        amI();
        
        amI()
      
    }
    
  }
  
}
```

Frame Pointer %ebp

Stack Pointer %esp
Stack operation

Call Chain

Stack Pointer %esp

Frame Pointer %ebp

16
Stack operation

yoo(…)
{
  ...
  who (…)
  {
    ...
    amI (…)
    {
      amI (…)
      {
        ...
        amI ()
      }
    }
  }
}

Call Chain

who(…)
{
  ...
  call amI()
  ...
}

amI(…)
{
  ...
  call amI()
  ...
}

amI(…)
{
  ...
  call amI()
  ...
}
Stack operation

Call Chain

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

Frame Pointer  
%ebp

Stack Pointer  
%esp
Stack operation

```
yoo(...) {
    .
    who(...) {
      • • •
      amI();
      .
    }
}
```
Stack operation

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

amI

who

yoo

amI

amI

amI
```
Stack operation

```c
void yoo(...)
{
    ...
}

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

Call Chain

- Frame Pointer %ebp
- Stack Pointer %esp
Stack operation

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp
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

![Stack Frame Diagram]

Frame Pointer (%ebp)

Caller Frame

Arguments

Return Addr

Old %ebp

Saved Registers + Local Variables

Argument Build

Stack Pointer (%esp)
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  %ebp
    movl  %esp, %ebp
    subl  $16, %esp
    pushl  $zip2  # Global Var
    pushl  $zip1  # Global Var
    call  swap

Resulting Stack

- &zip2
- &zip1
- Rtn adr
- %esp
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 8(%ebp),%edx
    movl 12(%ebp),%ecx
    movl (%edx),%ebx
    movl (%ecx),%eax
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    pop %ebx
    leave
    ret

Set Up

Body

Finish
swap Setup #1

Entering Stack

Resulting Stack

\begin{verbatim}
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
\end{verbatim}
swap Setup #2

**Entering Stack**

- %ebp
- &zip2
- &zip1
- Rtn adr

**Resulting Stack**

- yp
- xp
- Rtn adr
- Old %ebp

swap:

- pushl %ebp
- movl %esp,%ebp
- pushl %ebx
swap Setup #3

Entering Stack

Resulting Stack

swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx
Effect of swap setup

Entering Stack

Resulting Stack

<table>
<thead>
<tr>
<th>&amp;zip2</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;zip1</td>
<td>8</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>4</td>
</tr>
</tbody>
</table>

Offset (relative to %ebp)

<table>
<thead>
<tr>
<th>&amp;zip2</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;zip1</td>
<td>8</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>4</td>
</tr>
</tbody>
</table>

Old %ebp

Old %ebx

Body

\[
\text{movl} \ 12(\%ebp),\%ecx \ # \ \text{get} \ \text{yp}
\text{movl} \ 8(\%ebp),\%edx \ # \ \text{get} \ \text{xp}
\ldots
\]
swap Finish #1

- Observation
  - Saved & restored register %ebx
swap Finish #2

- Save stack pointer to beginning of frame

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
swap Finish #3

- Restore saved `%ebp` and set stack ptr to end of caller’s frame

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

Pop address from stack & jump there
Observation
- Saved & restored register %ebx
- Didn’t do so for %eax, %ecx, or %edx
Register saving conventions

- When procedure \texttt{yoo} calls \texttt{who}:
  - \texttt{yoo} is the \textit{caller}, \texttt{who} is the \textit{callee}

- Can register be used for temporary storage?

\texttt{yoo:}
\begin{verbatim}
  . . .
  movl $15213, %edx
  call who
  addl %edx, %eax
  . . .
  ret
\end{verbatim}

\texttt{who:}
\begin{verbatim}
  . . .
  movl 8(%ebp), %edx
  addl $91125, %edx
  . . .
  ret
\end{verbatim}

- Contents of register \texttt{%edx} overwritten by \texttt{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
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
Rfact stack setup

Entering Stack

rfact:
\begin{verbatim}
pushl %ebp
movl %esp, %ebp
pushl %ebx
\end{verbatim}
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 recursion

leal -1(%ebx),%eax

pushl %eax

call rfact

leal -1(%ebx),%eax

pushl %eax

call rfact

leal -1(%ebx),%eax

pushl %eax

call rfact

leal -1(%ebx),%eax

pushl %eax

call rfact

leal -1(%ebx),%eax

pushl %eax

call rfact
Return from Call

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

```
Rtn adr
Old %ebp
Old %ebx
x-1

%eax x!
%ebx Old %ebx

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

pre %ebp
pre %ebx

8
4
0
-4
-8

%ebp
%esp
```
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