CS165 – Computer Security

Understanding low-level program execution
Oct 6th, 2015
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
Assembly is “Spaghetti Code”

Nice C Abstractsions
• if-then-else
• while
• for loops
• do-while

Assembly
• Jump
  – Direct: jmp addr
  – Indirect: jmp reg
• Branch
  – Test EFLAG
  – if(EFLAG SET) goto line
"For" → "While" → "Do-While"

For Version

\[
\text{for } (\text{Init}; \text{ Test}; \text{ Update }) \\
\quad \text{Body}
\]

While Version

\[
\text{Init;} \\
\quad \text{while } (\text{Test}) \{ \\
\quad \quad \text{Body} \\
\quad \quad \text{Update} ;
\}
\]

Do-While Version

\[
\text{Init;} \\
\quad \text{if } (!\text{Test}) \\
\quad \quad \text{goto done;} \\
\quad \text{do } \{ \\
\quad \quad \quad \text{Body} \\
\quad \quad \quad \text{Update} ;
\quad \} \quad \text{while } (\text{Test})
\]

Goto Version

\[
\text{Init;} \\
\quad \text{if } (!\text{Test}) \\
\quad \quad \text{goto done;} \\
\quad \text{loop:} \\
\quad \quad \text{Body} \\
\quad \quad \text{Update} ; \\
\quad \text{if } (\text{Test}) \\
\quad \quad \text{goto loop;} \\
\text{done:}
\]
### Jump Table

#### Table Contents

```
.section .rodata
    .align 4
.L62:
    .long  .L61  # x = 0
    .long  .L56  # x = 1
    .long  .L57  # x = 2
    .long  .L58  # x = 3
    .long  .L61  # x = 4
    .long  .L60  # x = 5
    .long  .L60  # x = 6
```

```c
switch(x) {
    case 1:  // .L56
        w = y*z;
        break;
    case 2:  // .L57
        w = y/z;
        /* Fall Through */
    case 3:  // .L58
        w += z;
        break;
    case 5:  
    case 6:  // .L60
        w -= z;
        break;
    default: // .L61
        w = 2;
}
```
Jumps
- jmp 0x45, called a **direct jump**
- jmp *eax, called an **indirect jump**

Branches
- if (EFLAG) jmp x
  Use one of the 32 EFLAG bits to determine if jump taken
Jumps
• jmp 0x45, called a direct jump
• jmp *eax, called an indirect jump

Branches
• if (EFLAG) jmp x
  Use one of the 32 EFLAG bits to determine if jump taken

Note:
No direct way to get or set EIP
Implementing “if”

C

1. if(x <= y)
2.   z = x;
3. else
4.   z = y;

Assembly is 2 instrs
1. Set eflag to conditional
2. Test eflag and branch
Implementing “if”

C
1. if(x <= y)
2.   z = x;
3. else
4.   z = y;

Psuedo-Assembly
1. Computing x – y. Set eflags:
   1. CF =1 if x < y
   2. ZF =1 if x==y

Assembly is 2 instrs
1. Set eflag to conditional
2. Test eflag and branch
Implementing “if”

C
1. if(x <= y)
2.   z = x;
3. else
4.   z = y;

Psuedo-Assembly
1. Computing x – y. Set eflags:
   1. CF =1 if x < y
   2. ZF =1 if x==y
2. Test EFLAGS. If both CF and ZF not set, branch to 5
3. mov x, z
4. Jump to 6
5. mov y, z
6. <end of if-then-else>

Assembly is 2 instrs
1. Set eflag to conditional
2. Test eflag and branch
If \((x > y)\)

\(\%eax\) holds \(x\) and \(0xc(\%ebp)\) holds \(y\)

cmp \(0xc(\%ebp)\), \(\%eax\)

ja addr
If \((x > y)\)

\%eax holds \(x\) and \(0xc(\%ebp)\) holds \(y\)

cmp \(0xc(\%ebp)\), \%eax

ja \(addr\)

Same as “sub” instruction

\(r = \%eax - M[\%ebp+0xc]\), i.e., \(x - y\)
If (x > y)

%eax holds x and 0xc(%ebp) holds y

cmp 0xc(%ebp), %eax

ja addr

Same as “sub” instruction
r = %eax - M[%ebp+0xc], i.e., x - y

Jump if CF=0 and ZF=0
If \((x > y)\)

\(\%eax\) holds \(x\) and \(0xc(\%ebp)\) holds \(y\)

cmp \(0xc(\%ebp), \%eax\)

ja addr

Same as “sub” instruction

\(r = \%eax - M[\%ebp+0xc]\), i.e., \(x - y\)

Jump if \(\text{CF}=0\) and \(\text{ZF}=0\)

\((x >= y) \land (x != y) \Rightarrow x > y\)
Setting EFLAGS

• Instructions may set an eflag, e.g.,
• “cmp” and arithmetic instructions most common
  – Was there a carry (CF Flag set)
  – Was the result zero (ZF Flag set)
  – What was the parity of the result (PF flag)
  – Did overflow occur (OF Flag)
  – Is the result signed (SF Flag)
Aside: Although the x86 processor knows every time integer overflow occurs, C does not make this result visible.

From the Intel x86 manual
See the x86 manuals available on Intel’s website for more information

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
<td>OF == 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td>OF == 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign</td>
<td>SF == 1</td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if equal</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less than</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less than or equal</td>
<td>ZF == 1 or SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below</td>
<td>CF == 1</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if parity</td>
<td>PF == 1</td>
</tr>
</tbody>
</table>
Agenda

- Compilation Workflow
- x86 Execution Model
  - Basic Execution
  - Memory Operation
  - Control Flow
  - Memory Organization
The Stack grows down towards lower addresses.
Variables

• On the stack
  – Local variables
  – Lifetime: stack frame

• On the heap
  – Dynamically allocated via new/malloc/etc.
  – Lifetime: until freed
Procedures and Stacks

• Procedures are not native to assembly
• Compilers *implement* procedures
  – On the stack
  – Following the call/return stack discipline
Procedures/Functions

• We need to address several issues:
  1. How to allocate space for local variables
  2. How to pass parameters
  3. How to pass return values
  4. How to share 8 registers with an infinite number of local variables
Procedures/Functions

• We need to address several issues:
  1. How to allocate space for local variables
  2. How to pass parameters
  3. How to pass return values
  4. How to share 8 registers with an infinite number of local variables

• A stack frame provides space for these values
  – Each procedure invocation has its own stack frame
  – Stack discipline is LIFO
    • If procedure A calls B, B’s frame must exit before A’s
orange(...) 
{
    ...
    red()
    ...
}

red(...) 
{
    ...
    green()
    ...
    green()
}
green(...) 
{
    ...
    green()
    ...
}
orange(...)  
{
    ...
    red()
    ...
}

red(...)  
{
    ...
    green()
    ...
    green()
}

green(...)  
{
    ...
    green()
    ...
}

Function Call Chain

orange
   ↓
  red
       ↓
green
           ↓
green
               ↓
           ...

26
Frame for
• locals
• pushing parameters
• temporary space

Function Call Chain

orange
↓
red
↓
green
↓
green
↓
...

•
•
•
Call to red “pushes” new frame

Function Call Chain

orange
   ↓
  red
   ↓
green
   ↓
green
   ↓
...
Function Call Chain

orange
↓
red
↓
green
↓
green
↓
...

...
When green returns it “pops” its frame
Function Call Chain

orange
↓
red
↓
green
↓
green
↓
...

orange
↓
green
↓
green
↓
...
Function Call Chain

- orange
  - red
    - green
      - green
        - green
          - ...

Function Call Chain

orange
→
red

→
green

→
green

...
Function Call Chain

orange
  ↓
  red
  ↓
green
  ↓
green
  ↓
  ...
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
On the stack

```c
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```
int orange(int a, int b) {
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
int orange(int a, int b) {
    char buf[16];
    int c, d;
    if(a > b) {
        c = a;
    } else {
        c = b;
    }
    d = red(c, buf);
    return d;
}
On the stack

```c
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```

Need to access arguments

Need space to store local vars (buf, c, and d)

Need space to put arguments for callee

Need a way for callee to return values
```c
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```

- **Need to access arguments**
- **Need space to store local vars (buf, c, and d)**
- **Need space to put arguments for callee**
- **Need a way for callee to return values**

**Calling convention determines the above features**
cdecl – the default for Linux & gcc

```c
int orange(int a, int b) {
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```
cdecl – the default for Linux & gcc

int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}

Don’t worry! We will walk through these one by one.

-parameter area (caller)

-return addr

-caller’s ebp

callee-save

-locals (buf, c, d ≥ 24 bytes if stored on stack)

caller-save

buf

c

-return addr

-orange’s ebp

...
Register Saving Conventions

• When procedure foo calls bar:
  – foo is the caller
  – bar is the callee

• Can register be used for temporary storage?

  foo:
  • • •
  movl $15213, %edx
  call bar
  addl %edx, %eax
  • • •
  ret

  bar:
  • • •
  movl 8(%ebp), %edx
  addl $18243, %edx
  • • •
  ret

  – Contents of register %edx overwritten by who
  – This could be trouble ➔ something should be done!
    • Need some coordination
Register Saving Conventions

• When procedure **foo** calls **bar**:  
  – **foo** is the **caller**  
  – **bar** is the **callee**

• Can register be used for temporary storage?  

• Conventions  
  – “**Caller Save**”  
    • Caller saves temporary values in its frame before the call  
  – “**Callee Save**”  
    • Callee saves temporary values in its frame before using
IA32/Linux+Windows 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
When orange attains control,
1. return address has already been pushed onto stack by caller
When *orange* attains control,

1. return address has already been pushed onto stack by caller

2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
When orange attains control,

1. return address has already been pushed onto stack by caller

2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8

3. save values of other callee-save registers if used
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers if used
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
4. allocate space for locals
   - subtracting from esp
   - “live” variables in registers, which on contention, can be “spilled” to stack space
For *caller orange* to call *callee red*,

- return addr
- caller’s ebp
- callee-save
- locals (buf, c, d ≥ 24 bytes if stored on stack)
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx

```
+---------------------+  +---------------------+  +---------------------+
|   ...               |  |        b            |  |        a            |
|                      |  | return addr        |  | caller's ebp        |
| caller's save       |  | callee-save        |  | locals              |
| (buf, c, d ≥ 24     |  | (buf, c, d ≥ 24     |  | (buf, c, d ≥ 24     |
| bytes if stored     |  | bytes if stored     |  | bytes if stored     |
| on stack)           |  | on stack)           |  | on stack)           |
| caller-save         |  | caller-save         |  | %ebp                |
|                     |  |                     |  | %esp                |
```
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after *red* returns
   - eax, edx, ecx

2. push arguments to *red* from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after *red* returns
   - eax, edx, ecx

2. push arguments to *red* from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack

3. push return address, i.e., the next instruction to execute in *orange* after *red* returns
For *caller orange* to call *callee red*,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx

2. push arguments to red from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack

3. push return address, i.e., the *next* instruction to execute in orange after red returns

4. transfer control to red
   - usually happens together with step 3 using call
When **red** attains control,

1. Return address has already been pushed onto stack by **orange**

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>return addr</td>
</tr>
<tr>
<td>caller’s ebp</td>
</tr>
<tr>
<td>callee-save</td>
</tr>
<tr>
<td>locals (buf, c, d ( \geq 24 ) bytes if stored on stack)</td>
</tr>
<tr>
<td>caller-save</td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>
| return addr | %esp

%ebp
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...

... %ebp %esp

b
a
return addr
caller’s ebp
callee-save
locals (buf, c, d ≥ 24 bytes if stored on stack)
caller-save
buf
c
return addr
orange’s ebp
...

%ebp %esp
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ... 
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - pop %ebp
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - pop %ebp
8. return control to orange
   - ret
   - pops return address from stack and jumps there (EIP changed)
When orange regains control,
When *orange* regains control,

1. clean up arguments to *red*
   - adding to esp
2. restore any caller-save registers
   - pops
3. ...

<table>
<thead>
<tr>
<th>...</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>caller’s ebp</td>
<td></td>
</tr>
<tr>
<td>callee-save</td>
<td></td>
</tr>
<tr>
<td>locals (buf, c, d ≥ 24 bytes if stored on stack)</td>
<td></td>
</tr>
</tbody>
</table>
Terminology

• **Function Prologue** – instructions to set up stack space and save callee saved registers
  – Typical sequence:
    ```
    push ebp
    ebp = esp
    esp = esp - <frame space>
    ```

• **Function Epilogue**– instructions to clean up stack space and restore callee saved registers
  – Typical Sequence:
    ```
    leave    // esp = ebp, pop ebp
    ret      // pop and jump to ret addr
    ```
## cdecl – One Convention

<table>
<thead>
<tr>
<th>Action</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller saves: eax, edx, ecx</td>
<td>push (old), or mov if esp already adjusted</td>
</tr>
<tr>
<td>arguments pushed right-to-left</td>
<td></td>
</tr>
<tr>
<td>linkage data starts new frame</td>
<td>“call” pushes return addr</td>
</tr>
<tr>
<td>callee saves: ebx, esi, edi, ebp, esp</td>
<td>ebp often used to deref args and local vars</td>
</tr>
<tr>
<td>return value</td>
<td>pass back using eax</td>
</tr>
<tr>
<td>argument cleanup</td>
<td>caller’s responsibility</td>
</tr>
</tbody>
</table>
Quiz

- `printf("%s, %d", aString, anInteger);`
- How are the arguments pushed onto stack?
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

int course1 = 15213;
int course2 = 18243;

void call_swap() {
    swap(&course1, &course2);
}

Calling swap from call_swap

subl $8, %esp
movl $course2, 4(%esp)
movl $course1, (%esp)
call swap

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

Resulting Stack

subl %esp
Rtn adr

%esp
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

movl 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)

popl %ebx
popl %ebp
ret
```
swap Setup #1

Entering Stack

Resulting Stack

\[
\begin{align*}
\text{swap:} & \\
& \text{pushl} \%ebp \\
& \text{movl} \%esp,\%ebp \\
& \text{pushl} \%ebx
\end{align*}
\]
swap Setup #2

Entering Stack

- \&course2
- \&course1
- Rtn adr

Resulting Stack

- YP
- xp
- Rtn adr
- Old \%ebp

swap:

- pushl \%ebp
- movl \%esp,\%ebp
- pushl \%ebx
**Carnegie Mellon**

**swap Setup #3**

**Entering Stack**

- &course2
- &course1
- Rtn adr

**Resulting Stack**

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

**swap:**

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Body

Entering Stack

Resulting Stack

Offset relative to %ebp

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

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

...
Swap Finish

Stack Before Finish

Observation
- Saved and restored register %ebx
- Not so for %eax, %ecx, %edx
Disassembled swap

08048384 <swap>:

8048384: 55  push %ebp
8048385: 89 e5  mov %esp,%ebp
8048387: 53  push %ebx
8048388: 8b 55 08  mov 0x8(%ebp),%edx
804838b: 8b 4d 0c  mov 0xc(%ebp),%ecx
804838e: 8b 1a  mov (%edx),%ebx
8048390: 8b 01  mov (%ecx),%eax
8048392: 89 02  mov %eax,(%edx)
8048394: 89 19  mov %ebx,(%ecx)
8048396: 5b  pop %ebx
8048397: 5d  pop %ebp
8048398: c3  ret

Calling Code

80483b4:  movl $0x8049658,0x4(%esp)  # Copy &course2
80483bc:  movl $0x8049654,(%esp)  # Copy &course1
80483c3:  call 8048384 <swap>  # Call swap
80483c8:  leave  # Prepare to return
80483c9:  ret  # Return
Quiz - Control Flow: Function Calls

- What must assembly/machine language do?

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Save function arguments</td>
<td>3. Execute body</td>
</tr>
<tr>
<td>2. Branch to function body</td>
<td>4. May allocate memory</td>
</tr>
<tr>
<td></td>
<td>5. May call functions</td>
</tr>
<tr>
<td></td>
<td>6. Save function result</td>
</tr>
<tr>
<td></td>
<td>7. Branch to where called</td>
</tr>
</tbody>
</table>

- May allocate memory
- May call functions
## Control Flow: Function Calls

- What must assembly/machine language do?

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Save function arguments</td>
<td>3. Execute body</td>
</tr>
<tr>
<td>2. Branch to function body</td>
<td>• May allocate memory</td>
</tr>
<tr>
<td></td>
<td>• May call functions</td>
</tr>
<tr>
<td></td>
<td>4. Save function result</td>
</tr>
<tr>
<td></td>
<td>5. Branch to where called</td>
</tr>
<tr>
<td>1. Push to stack</td>
<td></td>
</tr>
</tbody>
</table>

1. Push to stack
Control Flow: Function Calls

- What must assembly/machine language do?

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Save function arguments</td>
<td>3. Execute body</td>
</tr>
<tr>
<td>2. Branch to function body</td>
<td>• May allocate memory</td>
</tr>
<tr>
<td></td>
<td>• May call functions</td>
</tr>
<tr>
<td></td>
<td>4. Save function result</td>
</tr>
<tr>
<td></td>
<td>5. Branch to where called</td>
</tr>
</tbody>
</table>

2. Use call (jump to procedure, save return location on stack)
### Control Flow: Function Calls

- What must assembly/machine language do?

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Save function arguments</td>
<td>3. Execute body</td>
</tr>
<tr>
<td>2. Branch to function body</td>
<td>- May allocate memory</td>
</tr>
<tr>
<td></td>
<td>- May call functions</td>
</tr>
<tr>
<td>4. Save function result</td>
<td>5. Branch to where called</td>
</tr>
<tr>
<td>3. Use <code>sub %esp</code> to create new stack frame</td>
<td></td>
</tr>
</tbody>
</table>

3. Use `sub %esp` to create new stack frame
Control Flow: Function Calls

- What must assembly/machine language do?

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
</table>
| 1. Save function arguments | 3. Execute body  
| 2. Branch to function body | • May allocate memory  
|       | • May call functions |

4. **Use %eax**
Control Flow: Function Calls

• What must assembly/machine language do?

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Save function arguments</td>
<td>3. Execute body</td>
</tr>
<tr>
<td>2. Branch to function body</td>
<td>• May allocate memory</td>
</tr>
<tr>
<td></td>
<td>• May call functions</td>
</tr>
<tr>
<td></td>
<td>4. Save function result</td>
</tr>
<tr>
<td></td>
<td>5. Branch to where called</td>
</tr>
</tbody>
</table>

5. Use `ret`
/* Compute x + 3 */
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}

/* Increment value by k */
void incrk(int *ip, int k) {
    *ip += k;
}

• **add3** creates pointer and passes it to **incrk**
Creating and Initializing Local Variable

int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}

- Variable localx must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as -4(%ebp)

First part of add3

```asm
add3:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp  # Alloc. 24 bytes
    movl 8(%ebp), %eax
    movl %eax, -4(%ebp)  # Set localx to x
```

```
8     x
4     Rtn adr
0     Old %ebp
-4    localx = x
-8    Unused
-12
-16
-20
-24  ← %esp
```
Creating Pointer as Argument

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

- Use `leal` instruction to compute address of `localx`

Middle part of `add3`

```assembly
movl $3, 4(%esp)  # 2^{nd} arg = 3
leal -4(%ebp), %eax# &localx
movl %eax, (%esp)  # 1^{st} arg = &localx
call incrk
```
Retrieving local variable

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

- Retrieve localx from stack as return value

**Final part of add3**

```
movl -4(%ebp), %eax  # Return val = localx
leave
ret
```
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
Summary

• Compiler workflow
• The machine execution model
  – Register to register moves
    • Register mnemonics
  – Register/memory
    • mov and addressing modes for common codes
  – Control flow
    • EFLAGS
  – Program Memory Organization
    • Stack grows down
• Functions
  • Pass arguments, callee and caller saved, stack frame
For more information

• Overall machine model: *Computer Systems, a Programmer’s Perspective* by Bryant and O’Hallaron

• Calling Conventions: