Machine-Level Programming III: Procedures
Sept 19, 2000

Topics
• IA32 stack discipline
• Register saving conventions
• Creating pointers to local variables
• Stack buffer overflow exploits
  – finger
  – AIM (AOL Instant Messenger)

IA32 Stack
• Region of memory managed with stack discipline
• Register %esp indicates lowest allocated position in stack
  – i.e., address of top element

Pushing
• pushl Src
• Fetch operand at Src
• Decrement %esp by 4
• Write operand at address given by %esp

Popping
• popl Dest
• Read operand at address given by %esp
• Increment %esp by 4
• Write to Dest

Stack Operation Examples

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 / Return Example

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

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

```c
yoo(…)
{
  ...
  who();
  ...
}

who(…)
{
  ...
  amI();
  ...
}

amI(…)
{
  ...
  amI();
  ...
}
```

- Procedure amI recursive

Call Chain

```c
... yoo
    ↓ who
    ↓ amI
    ↓ amI
    ↓ amI
```

IA32 Stack Structure

Stack Growth
- Toward lower addresses

Stack Pointer
- Address of next available location in stack
- Use register %esp

Frame Pointer
- Start of current stack frame
- Use register %ebp

Stack Pointer
- %esp
- %ebp

Frame Pointer
- amI
- amI
- amI

Stack “Top”
IA32/Linux Stack Frame

Callee Stack Frame (“Top” to Bottom)
- Parameters for called functions
- 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
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Revisiting swap

```c
call_swap:
    pushl $zip2
    pushl $zip1
    call swap
    ...
```

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

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
swaps Stack Finish

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

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

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
  call who
  addl $91125, %edx
• • •
ret
  - Contents of register %edx overwritten by who

Conventions
- “Caller Save”
  - Caller saves temporary in its frame before calling
- “Callee Save”
  - Callee saves temporary in its frame before using

Recursive Factorial

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

Complete Assembly
- Assembler directives
  - Not of concern to us
- Labels
  - .Lxx
- Actual instructions

Recursive Factorial
Rfact Stack Setup

Entering Stack

<table>
<thead>
<tr>
<th>Caller</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>%esp</td>
</tr>
</tbody>
</table>

Rfact: pushl %ebp
       movl %esp, %ebp
       pushl %ebx

Rfact Body

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

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

Rfact Recursion

leal -1(%ebx), %eax

Rfact Result

Return from Call

imull %ebx, %eax

Registers

$ebx Stored value of x
$eax

- Temporary value of x-1
- Returned value from rfact(x-1)
- Returned value from this call
**Rfact Completion**

<table>
<thead>
<tr>
<th>8</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
</tr>
<tr>
<td>-8</td>
<td>x-1</td>
</tr>
</tbody>
</table>

```
%eax x!
%ebx x
```

**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);
}
}
```

**Top-Level Call**

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

- Pass pointer to update location
- Uses tail recursion
  - But GCC only partially optimizes it

**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
```

**Passing Pointer**

**Calling s_helper from sfact**

```
leal -4(%ebp),%eax  # Compute &val
pushl %ebp  # Push on stack
pushl %edx  # Push x
call _s_helper  # call
movl -4(%ebp),%eax  # Return val
      
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

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

**Stack at time of call**

- Val = 1
### Using Pointer

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

- Register %ecx holds `x`
- Register %edx holds `accum`
  - Use access (%edx) to reference memory

### Internet Worm and IM War

**November, 1988**
- Internet Worm attacks thousands of Internet hosts.
- How did it happen?

**July, 1999**
- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers

### Internet Worm and IM War (cont)

**August 1999**
- Mysteriously, Messenger clients can no longer access AIM servers.
- Even though the AIM protocol is an open, published standard.
- Microsoft and AOL begin the IM war:
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes.
  - At least 13 such skirmishes.
- How did it happen?

The Internet Worm and AOL/Microsoft War were both based on **stack buffer overflow** exploits!
- many Unix functions, such as gets() and strcpy(), do not check argument sizes.
- allows target buffers to overflow.

### Stack buffer overflows

```c
void bar()
{
  char buf[64];
  gets(buf);
  • • •
}
```
Stack buffer overflows (cont)

```c
void bar() {
    char buf[64];
    gets(buf);
    ...
}
```

```c
void foo () {
    bar();
    ...
}
```

When bar() returns, control passes silently to B instead of A!!

Exploits based on buffer overflows

**Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.**

**Internet worm**
- Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
  ```
  finger droh@cs.cmu.edu
  ```
- Worm attacked fingerd client by sending phony argument:
  ```
  finger "exploit code padding new return address"
  ```
- Exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

**IM War**
- AOL exploited existing buffer overflow bug in AIM clients
- Exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
- When Microsoft changed code to match signature, AOL changed signature location.

Main Ideas

**Stack Provides Storage for Procedure Instantiation**
- Save state
- Local variables
- Any variable for which must create pointer

**Assembly Code Must Manage Stack**
- Allocate / deallocate by decrementing / incrementing stack pointer
- Saving / restoring register state

**Stack Adequate for All Forms of Recursion**
- Including multi-way and mutual recursion examples in the bonus slides.

Good programmers know the stack discipline and are aware of the dangers of stack buffer overflows.

Free Bonus Slides!

(not covered in lecture)

Topics
- how the stack supports multi-way recursion.
- how the stack supports mutual recursion.
Multi-Way Recursion

```c
int bfact (int x)
{
    return r_prod(1, x);
}

int r_prod(int from, int to)
{
    int middle;
    int prodA, prodB;
    if (from >= to)
        return from;
    middle = (from + to) >> 1;
    prodA = r_prod(from, middle);
    prodB = r_prod(middle+1, to);
    return prodA * prodB;
}
```

- Compute product \( x \cdot (x+1) \cdot \ldots \cdot (y-1) \cdot y \)
- Split into two ranges:
  - Left: \( x \cdot (x+1) \cdot \ldots \cdot (m-1) \cdot m \)
  - Right: \((m+1) \cdot \ldots \cdot (y-1) \cdot y \)
  \[ m = \lceil (x+y)/2 \rceil \]
- No real advantage algorithmically

Multi-Way Recursive Code

Stack Frame

<table>
<thead>
<tr>
<th>Arg 2</th>
<th>Arg 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-16</td>
<td>-20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Old $ebp</th>
<th>Old $edi</th>
<th>Old $esi</th>
<th>Old $ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>-8</td>
<td>-12</td>
<td>0</td>
</tr>
</tbody>
</table>

Callee Save Regs.

- $eax: from
- $ebx: return values

Binary Splitting Example

Top-Level Call
```c
int bfact(int x)
{
    return r_prod(1, x);
}
```

Multi-Way Recursive Code Finish

Stack

- After making recursive calls, still has two arguments on top

Finishing Code

- Moves stack pointer to start of saved register area
- Pops registers
Mutual Recursion

Top-Level Call

```c
int lrfact(int x)
{
    int left = 1;
    return left_prod(&left, &x);
}
```

```c
int lrfact(int x)
{
    int left = 1;
    return left_prod(&left, &x);
}
```

```c
int left_prod
(int *leftp, int *rightp)
{
    int left = *leftp;
    return left * right_prod(&left+1, rightp);
}
```

```c
int left_prod
(int *leftp, int *rightp)
{
    int left = *leftp;
    return left * right_prod(&left+1, rightp);
}
```

```c
int right_prod
(int *leftp, int *rightp)
{
    int right = *rightp;
    if (*leftp == right)
        return right;
    else {
        int minus1 = right-1;
        return right * left_prod(leftp, &minus1);
    }
}
```

```c
int right_prod
(int *leftp, int *rightp)
{
    int right = *rightp;
    if (*leftp == right)
        return right;
    else {
        int minus1 = right-1;
        return right * left_prod(leftp, &minus1);
    }
}
```

Mutually Recursive Execution Example

Calling

- Recursive routines pass two arguments
  - Pointer to own local variable
  - Pointer to caller's local variable

Implementation of lrfact

Call to Recursive Routine

```c
int left = 1;
return left_prod(&left, &x);
```

Code for Call

<table>
<thead>
<tr>
<th>Code for Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>leal 8(%ebp),%edx # edx = &amp;x</td>
</tr>
<tr>
<td>pushl %edx # push 6x</td>
</tr>
<tr>
<td>leal -4(%ebp),%eax # eax = &amp;left</td>
</tr>
<tr>
<td>pushl %eax # push 8(left</td>
</tr>
<tr>
<td>call _left_prod # Call</td>
</tr>
</tbody>
</table>

Implementation of left_prod

Call to Recursive Routine

```c
int plus1 = left+1;
return left * right_prod(&plus1, rightp);
```

Stack at time of call

<table>
<thead>
<tr>
<th>Stack at time of call</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>8 Rtn adr</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>0 Old %ebp</td>
</tr>
<tr>
<td>-4 left= 1</td>
</tr>
<tr>
<td>-8</td>
</tr>
<tr>
<td>-12</td>
</tr>
<tr>
<td>-16 Unused</td>
</tr>
<tr>
<td>&amp;x</td>
</tr>
<tr>
<td>&amp;left</td>
</tr>
</tbody>
</table>

Call to Recursive Routine

```c
int plus1 = left+1;
return left * right_prod(&plus1, rightp);
```

Stack at time of call

<table>
<thead>
<tr>
<th>Stack at time of call</th>
</tr>
</thead>
<tbody>
<tr>
<td>rightp</td>
</tr>
<tr>
<td>12 Rtn adr</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>4 leftp</td>
</tr>
<tr>
<td>0 Old %ebp</td>
</tr>
<tr>
<td>-4 plus1</td>
</tr>
<tr>
<td>-8</td>
</tr>
<tr>
<td>-12</td>
</tr>
<tr>
<td>-16 Unused</td>
</tr>
<tr>
<td>&amp;right</td>
</tr>
<tr>
<td>&amp;plus1</td>
</tr>
</tbody>
</table>

Implementation of right_prod

Call to Recursive Routine

```c
int right = *rightp;
if (*leftp == right)
    return right;
else {
    int minus1 = right-1;
    return right * left_prod(leftp, &minus1);
}
```

Call to Recursive Routine

```c
int right = *rightp;
if (*leftp == right)
    return right;
else {
    int minus1 = right-1;
    return right * left_prod(leftp, &minus1);
}
```
Tail Recursion

Tail Recursive Procedure

```c
int tfact (int x)
{
    return t_helper (x, 1);
}
```

t_helper(x, val)

• • •

```c
return
t_helper(Xexpr, Vexpr)
```

Top-Level Call

```c
int tfact(int x)
{
    return t_helper(x, 1);
}
```

General Form

Form

• Directly return value returned by recursive call

Consequence

• Can convert into loop

Removing Tail Recursion

Optimized General Form

```c
int t_helper
{
    t_helper(x, val)
    {
        • • •
        return
        t_helper(Xexpr, Vexpr)
    }
}
```

Effect of Optimization

• Turn recursive chain into single procedure
• No stack frame needed
• Constant space requirement
  — Vs. linear for recursive version

Generated Code for Tail Recursive Proc.

Optimized Form

```c
int t_helper
{
    start:
    if (x <= 1)
        return val;
    val = val *x;
    x = x-1;
    goto start;
}
```

Code for Loop

```c
# %edx = x
# %ecx = val
L53: # start:
cmpl $1,%edx # x : 1
    jle L52 # if <= goto done
    movl %edx,%eax # eax = x
    imull %ecx,%eax # eax = val * x
    decl %edx # x--
    movl %eax,%ecx # val = val * x
    jmp L53 # goto start
L52: # done:
```

Registers

$edx x
$ecx val