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

```
pushl %eax

%esp | %eax | %edx
0x108 | 123 | 213
0x10c | 0x110 | 0x108
0x110 | 213 | 123

popl %edx

%esp | %eax | %edx
0x104 | 0x108 | 213
0x108 | 123 | 555
0x10c | 0x110 | 0x108
0x110 | 213 | 555
```
Procedure Control Flow

Use stack to support procedure call and return

Procedure call:

\texttt{call label} \quad \text{Push return address on stack; Jump to \textit{label}}

Return address value

- Address of instruction beyond \texttt{call}
- Example from disassembly

\begin{align*}
804854e: & \quad \text{e8 3d 06 00 00 call 8048b90 <main>} \\
8048553: & \quad 50 \quad \text{pushl %eax}
\end{align*}

- Return address = 0x8048553

Procedure return:

- \texttt{ret} \quad \text{Pop address from stack; Jump to address}
Procedure Call / Return Example

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

%esp  0x108  %esp  0x104
%eip  0x804854e  %eip  0x8048b90  %eip  0x8048553

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

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

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

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

- Procedure `amI` recursive

Call Chain

```
\[ \begin{align*}
yoo & \rightarrow \text{who} \\
  & \rightarrow \text{amI} \\
  & \rightarrow \text{amI} \\
  & \rightarrow \text{amI}
\end{align*} \]
```
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
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
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;
}
```

call_swap:

```assembly
    pushl $zip2
    pushl $zip1
    call swap
    ...  // stack frames
```

Resulting Stack

- `&zip2`
- `&zip1`
- `Rtn adr`
- `%esp`
Revisiting swap

\begin{align*}
\text{void swap(int } *xp, \text{ int } *yp) \{ \\
& \quad \text{int } t0 = *xp; \\
& \quad \text{int } t1 = *yp; \\
& \quad *xp = t1; \\
& \quad *yp = t0; \\
\} \\
\end{align*}

\begin{align*}
\text{swap:} \\
& \quad \text{pushl } \%ebp \\
& \quad \text{movl } \%esp,\%ebp \\
& \quad \text{pushl } \%ebx \\
& \quad \text{movl } 12(\%ebp),\%ecx \\
& \quad \text{movl } 8(\%ebp),\%edx \\
& \quad \text{movl } (\%ecx),\%eax \\
& \quad \text{movl } (\%edx),\%ebx \\
& \quad \text{movl } \%eax, (\%edx) \\
& \quad \text{movl } \%ebx, (\%ecx) \\
& \quad \text{movl } -4(\%ebp),\%ebx \\
& \quad \text{movl } \%ebp,\%esp \\
& \quad \text{popl } \%esp \\
& \quad \text{ret} \\
\end{align*}

\begin{enumerate}
\item \textbf{Set Up}
\item \textbf{Body}
\item \textbf{Finish}
\end{enumerate}
swap Setup

Entering Stack

Resulting Stack

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

Offset

0 4 8 12

%esp

&zip1

&zip2

Rtn adr

%ebp

yp
xp
Rtn adr

Old %ebp
Old %ebx

class07.ppt
swap Finish

swap’s Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>Old %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exiting Stack

<table>
<thead>
<tr>
<th>&amp;zip2</th>
<th>&amp;zip1</th>
</tr>
</thead>
</table>

Observation

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

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

```
*who*:
  • • •
  movl 8(%ebp), %edx
  addl $91125, %edx
  • • •
  ret

*yoo*:
  • • •
  movl $15213, %edx
  call *who*
  addl %edx, %eax
  • • •
  ret
```
IA32/Linux Register Usage

- Surmised by looking at code examples

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;
}
```

Complete Assembly

- Assembler directives
  - Lines beginning with “.”
  - Not of concern to us
- Labels
  - .Lxx
- Actual instructions

```
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

```
rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

<table>
<thead>
<tr>
<th>Caller</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calllee</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

Rtn adr  %esp

x

Old %ebp

Old %ebx

%ebp

%esp
Rfact Body

```
movl 8(%ebp),%ebx  # ebx = x
cmpl $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;
}
```

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

 Old %ebp
 Rtn adr
  %ebp

 Old %ebx
 %esp

 %eax x-1
 %ebx x

 pushl %eax

 Old %ebp
 Rtn adr
  %ebp

 Old %ebx
 %esp

 %eax x-1
 %ebx x

 call rfact

 Old %ebp
 Rtn adr
  %ebp

 Old %ebx
 %esp

 %eax x-1
 %ebx x
```
Return from Call

imull %ebx, %eax

|x|
Rtn adr
|---|
Old %ebp ← %ebp
Old %ebx
x-1 ← %esp

%eax (x-1)!
%ebx x

|x|
Rtn adr
|---|
Old %ebp ← %ebp
Old %ebx
x-1 ← %esp

%eax x!
%ebx x
Rfact Completion

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

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

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

```
Rtn adr
8 %esp
4 x
0 %ebp
-4 %ebx
-8 x-1
```

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

```
x
%esp
```

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

```
```
void s_helper (int x, int *accum) {
    if (x <= 1) return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1, accum);
    }
}

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
```

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 \texttt{s\_helper} from \texttt{sfact}

\begin{verbatim}
int sfact(int x)
{
    int val = 1;
    s\_helper(x, &val);
    return val;
}
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}
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

• • •
  `movl %ecx,%eax`  # z = x
  `imull (%edx),%eax`  # z *= *accum
  `movl %eax,(%edx)`  # *accum = z
  • • •
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

![Diagram showing the interaction between MSN server, MSN client, AIM server, and AIM clients](image)
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);
    ...
}
```

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

**Stack before call to `gets()`**

- **foo stack frame**
  - Old `%ebp`
  - `buf`

- **bar stack frame**
  - `buf`
Stack buffer overflows (cont)

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

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

Stack
after call to gets()

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 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 $x \times (x+1) \times \ldots \times (y-1) \times y$
- Split into two ranges:
  - Left: $x \times (x+1) \times \ldots \times (m-1) \times m$
  - Right: $(m+1) \times \ldots \times (y-1) \times y$
    \[
    m = \lfloor (x+y)/2 \rfloor
    \]
- No real advantage algorithmically

```c
int bfact(int x)
{
    return r_prod(1, x);
}
```
Binary Splitting Example
## Multi-Way Recursive Code

### Stack Frame

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>from</td>
</tr>
<tr>
<td>8</td>
<td>to</td>
</tr>
<tr>
<td>4</td>
<td>Rtn Adr</td>
</tr>
<tr>
<td>0</td>
<td>Old $ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old $edi</td>
</tr>
<tr>
<td>-8</td>
<td>Old $esi</td>
</tr>
<tr>
<td>-12</td>
<td>Old $ebx</td>
</tr>
</tbody>
</table>

### _r_prod:

```
movl  8(%ebp),%eax     # eax = from
movl  12(%ebp),%edi    # edi = to
cmpl %edi,%eax         # from : to
jge  L8                 # if >= goto done
leal (%edi,%eax),%ebx  # from + to
sarl $1,%ebx           # middle
pushl %ebx              # 2nd arg: middle
pushl %eax              # 1st arg: from
call _r_prod            # 1st call
pushl %edi              # 2nd arg: to
movl %eax,%esi          # esi = ProdA
incl %ebx               # middle + 1
pushl %ebx               # ... 1st arg
call _r_prod            # 2nd call
imull %eax,%esi         # ProdA * ProdB
movl %esi,%eax          # Return value
```

### L8: # done:

```
...                     # Finish
```

### Callee Save Regs.

- $ebx: middle
- $edi: to
- $esi: ProdA

### Return Values

- $eax: from

### Stack Frame Legend

- $eax: from
- $ebx: middle
- $edi: to
- $esi: ProdA
- $ebp: Old

---

class07.ppt
Multi-Way Recursive Code Finish

<table>
<thead>
<tr>
<th>12</th>
<th>from</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>to</td>
</tr>
<tr>
<td>4</td>
<td>Rtn Adr</td>
</tr>
<tr>
<td>0</td>
<td>Old $ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old $edi</td>
</tr>
<tr>
<td>-8</td>
<td>Old $esi</td>
</tr>
<tr>
<td>-12</td>
<td>Old $ebx</td>
</tr>
<tr>
<td>-16</td>
<td>Arg 2</td>
</tr>
<tr>
<td>-20</td>
<td>Arg 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L8:</th>
<th># done:</th>
</tr>
</thead>
<tbody>
<tr>
<td>leal -12(%ebp),%esp # Set Stack Ptr</td>
<td></td>
</tr>
<tr>
<td>popl %ebx # Restore %ebx</td>
<td></td>
</tr>
<tr>
<td>popl %esi # Restore %esi</td>
<td></td>
</tr>
<tr>
<td>popl %edi # Restore %edi</td>
<td></td>
</tr>
<tr>
<td>movl %ebp,%esp # Restore %esp</td>
<td></td>
</tr>
<tr>
<td>popl %ebp # Restore %ebp</td>
<td></td>
</tr>
<tr>
<td>ret # Return</td>
<td></td>
</tr>
</tbody>
</table>

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 left_prod
(int *leftp, int *rightp)
{
    int left = *leftp;
    if (left >= *rightp)
        return left;
    else {
        int plus1 = left+1;
        return left *
        right_prod(&plus1, 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);
    }
}
```
Mutually Recursive Execution Example

Calling

• Recursive routines pass two arguments
  – Pointer to own local variable
  – Pointer to caller’s local variable

\[
\begin{align*}
&\text{lrfact (4)} \\
&\quad \text{left: } 1 & \quad \text{x: } 4 & \quad \rightarrow 24 \\
&\quad \text{left_prod ( , )} \\
&\quad \quad \text{plus1: } 2 & \quad \rightarrow 24 \\
&\quad \text{right_prod ( , )} \\
&\quad \quad \text{minus1: } 3 & \quad \rightarrow 24 \\
&\quad \text{lrfact (3)} \\
&\quad \quad \text{left: } 1 & \quad x: 3 & \quad \rightarrow 6 \\
&\quad \text{left_prod ( , )} \\
&\quad \quad \text{plus1: } 3 & \quad \rightarrow 3 \\
&\quad \text{right_prod ( , )}
\end{align*}
\]
Implementation of \texttt{lrfact}

Call to Recursive Routine

\begin{verbatim}
int left = 1;
return left_prodp(&left, &x);
\end{verbatim}

Code for Call

\begin{verbatim}
leal 8(%ebp),%edx # edx = &x
pushl %edx # push &x
gleal -4(%ebp),%eax# eax = &left
pushl %eax # push &left
call _left_prod # Call
\end{verbatim}
Implementation of **left_prod**

Call to Recursive Routine

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

Stack at time of call

```plaintext
# %ebx holds left
# %edx holds rightp
leal 1(%ebx),%ecx       # left+1
movl %ecx,-4(%ebp)      # Store in plus1
pushl %edx              # Push rightp
leal -4(%ebp),%eax      # &plus1
pushl %eax              # Push &plus1
call _right_prod        # Call
```
Tail Recursive Procedure

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

General Form

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

Top-Level Call

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

Form

- Directly return value returned by recursive call

Consequence

- Can convert into loop
Removing Tail Recursion

**Optimized General Form**

```c
int t_helper(int x, int val) {
    start:
        • • •
        val = Vexpr;
        x = Xexpr;
        goto start;
}
```

**Resulting Code**

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

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

#### Code for Loop

```c
int t_helper
    (int x, int val)
{
    start:
        # %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