EECS08

Instructor: Aleksandar Kuzmanovic

TA: Ionut Trestian

Recitation 4
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 $%esp$ indicates lowest stack address
  - address of top element

![Stack Diagram]

Stack Pointer
$%esp$

Stack “Top”

Stack Grows Down

Increasing Addresses

Stack “Bottom”
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`
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

0x108 0x804854e
0x10c 0x8048553
0x110 123

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

%eip is program counter

call  8048b90

0x110 0x10c 0x108 123
0x104 0x8048553
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

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

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

amI(...)
{
  
}
```

- **Procedure amI recursive**

Call Chain

```
yoo

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

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

Call Chain

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

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

Call Chain

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

- yoo
- who
- ....

Diagram:
- Stack Pointer %esp
- Frame Pointer %ebp
- yoo
- who

EECS-213, S’08
Stack Operation

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

Call Chain

yoo

who

amI

Frame Pointer
%ebp

Stack Pointer
%esp

EECS-213, S'08
Stack Operation

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp

amI

who

yoo
Stack Operation

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

Call Chain

```
amI
  •
  •
  amI
  •
  •
```

Frame Pointer

```
%ebp
```

Stack Pointer

```
%esp
```
Stack Operation

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

```
Yoo

Who

AmI
```

Stack Operation

amI(...)
{
  •
  •
  •
amI();
  •
}

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

who

yoo

amI

amI

amI

amI
Stack Operation

Call Chain

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

Stack Pointer
%esp

Frame Pointer
%ebp

who

yoo

yoo

amI

amI

amI

amI
Stack Operation

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

Call Chain

- Frame Pointer: `%ebp`
- Stack Pointer: `%esp`
- `amI`
- `who`
- `yoo`

Diagram:

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

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

Call Chain

yoo

amI

amI

amI

Frame Pointer
%ebp

Stack Pointer
%esp

who

yoo

• • •
Stack Operation

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

Call Chain

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

```
\[
\begin{array}{c}
\text{\%esp} \\
\text{Rtn adr} \\
\text{\&zip1} \\
\text{\&zip2}
\end{array}
\]`
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 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
swap Setup #1

Entering Stack

Resulting Stack

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

**Entering Stack**

- `&zip2`
- `&zip1`
- `Rtn adr`  

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

**Resulting Stack**

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

- `%ebp`
- `%esp`
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

<table>
<thead>
<tr>
<th>&amp;zip2</th>
<th>&amp;zip1</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Resulting Stack

<table>
<thead>
<tr>
<th>Offset (relative to %ebp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
</tr>
<tr>
<td>xp</td>
</tr>
</tbody>
</table>

### Body

```assembly
movl 12(%ebp),%ecx  # get yp
movl 8(%ebp),%edx   # get xp
...                
```
swap Finish #1

Observation

- Saved & restored register %ebx

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish #2

swap's Stack

Offset
12
8
4
0
-4

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

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

swap's Stack

Offset

12  yp
8  xp
4  Rtn adr
0  Old %ebp

swap's Stack

Offset

12  yp
8  xp
4  Rtn adr

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

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

```assembly
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?

**yoo:****

```assembly
... movl $15213, %edx call who addl %edx, %eax ...
ret```

**who:****

```assembly
... movl 8(%ebp), %edx addl $91125, %edx ... ret```

- Contents of register %edx overwritten by **who**
Register Saving Conventions

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