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
- push1 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

- 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 Example

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

Procedure Return Example

- Example from disassembly:
  8048591: c3 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

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
**Stack Operation**

**Call Chain**

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

```
yoo
```

```
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

```
Current Stack Frame ("Top"
```

```
Frame Pointer (%ebp)
```

```
Saved Registers +
Local Variables
```

```
Argument Variables
```

```
Return Addr Old %ebp
```

```
Caller Frame
```

```
Arguments
```

```
Stack Pointer (%esp)
```

```
Argument Build
```

**Revisiting swap**

**Calling swap from call_swap**

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

```
int zipl = 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;
}
```

```
Resulting Stack
```

```
%esp
```

```
&zip2
```

```
&zip1
```

```
Rtn addr
```
Revisiting swap

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

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

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

Body
```
movl -4(%ebp),%ebx
movl %ebp,%esp
```

Finish
```
popl %ebp
ret
```

swap Setup #1

Entering Stack
```
| \%ebp |
```

Resulting Stack
```
| \%ebp |
```

Enter Stack
```
Old \%ebp
```

Resulting Stack
```
| %ebp |
```

swap Setup #2

Entering Stack
```
| \%ebp |
```

Resulting Stack
```
| \%ebp |
```

Enter Stack
```
Old \%ebp
```

Resulting Stack
```
| \%ebp |
```

swap Setup #3

Entering Stack
```
| \%ebp |
```

Resulting Stack
```
| \%ebp |
```

Enter Stack
```
Old \%ebp
```

Resulting Stack
```
| \%ebp |
```

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

Effect of swap Setup

Entering Stack

Resulting Stack

Observation

- Saved & restored register %ebx
**Register Saving Conventions**

**swap Finish #4**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
</tbody>
</table>

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

```
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?**

- Contents of register %edx overwritten by who

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

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

**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

Rfact Body

```assembly
.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
    jmp .L79
.L79:
    movl $-4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

### Rfact Stack Setup

- **Entering Stack**
  - Pre- %ebp
  - x
  - Rtn adr

- **Rfact Recursion**
  - %eax
  - x-1
  - x

- **Rfact Body**
  - Recursion

```assembly
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:
```

- **Registers**
  - `%ebx` Stored value of x
  - `%eax` Temporary value of x-1
  - Returned value from rfact(x-1)
  - Returned value from this call
Rfact Result

Return from Call

\[
\begin{array}{c|c|c|c}
\hline
\text{Rtn adr} & x & \text{ebp} & \%ebp \\
\text{Old \%ebp} & \%ebp & \%ebp & \%ebp \\
\text{Old \%ebx} & \%ebx & \%ebx & \%ebx \\
\hline
\end{array}
\]

\%eax \ (x-1)!

\%ebx \ x

Assume that \( rfact(x-1) \) returns \( (x-1)! \) in register \%eax

Rfact Completion

\[
\begin{array}{c|c|c|c}
\hline
\text{Rtn adr} & x & \text{ebp} & \%ebp \\
\text{Old \%ebp} & \%ebp & \%ebp & \%ebp \\
\text{Old \%ebx} & \%ebx & \%ebx & \%ebx \\
\hline
\end{array}
\]

\%eax \ x!

\%ebx \ Old \%ebx

\%ebx \ Old \%ebx

Creating & Initializing Pointer

Initial part of \( sfact \)

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

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;
}

int __sfact
{
    pushl %ebp
    # Save %ebp
    movl %esp, %ebp
    # Set %ebp
    subl $16, %esp
    # Add 16 bytes
    movl 8(%ebp), %edx
    # edx = x
    movl %edx, %eax
    # %eax = x!
    addl $4, %esp
    # %esp += 4
    popl %ebp
    ret
}

Pass pointer to update location
**Passing Pointer**

Calling `s_helper` from `sfact`

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

**Using Pointer**

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

**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