Systems I

Machine-Level Programming V: Procedures

Topics
- Stack abstraction and implementation
- IA32 stack discipline
Procedural Memory Usage

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

Where is the memory that holds t0 and t1 (or local variables in general)?

What happens if we run out of registers (x86 only has 8!)?

Where are parameters passed from callers to callee?

- Registers? Memory? What memory?
Stack Data Structure

- **LIFO data structure**
  - Last In, First Out

- **Allocated somewhere in memory**
  - Where doesn’t really matter as long as we store the stack pointer

- **By convention, stack grows toward smaller addresses**
  - Could do it either way

- **Values within the stack are referenced relative to the stack pointer**
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

- pushl $Src$
- Fetch operand at $Src$
- Decrement $\%esp$ by 4
- Write operand at address given by $\%esp$

Stack Grows Down

Increasing Addresses

Stack “Bottom”

Stack “Top”

Stack Pointer $\%esp$
IA32 Stack Popping

**Popping**
- `popl Dest`
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to `Dest`
### Stack Operation Examples

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x10c</td>
<td>213</td>
</tr>
<tr>
<td>0x110</td>
<td>555</td>
</tr>
</tbody>
</table>

#### pushl %eax

Initial State:
- %eax: 213
- %edx: 555
- %esp: 0x108

Push 213:
- %eax pushed to 0x108
- %edx: 555
- %esp: 0x104

Post-push State:
- %eax: 213
- %edx: 555
- %esp: 0x104

#### popl %edx

Initial State:
- %eax: 213
- %edx: 555
- %esp: 0x108

Pop 213:
- %edx popped from 0x108
- %eax: 213
- %esp: 0x104

Post-pop State:
- %eax: 213
- %edx: 213
- %esp: 0x104
What Elements for Procedures?

Method of computing address of first instruction of called procedure.

Place to store passed parameters.
- Call by value or by reference

Method of computing return address
- Need to come back to first instruction after point of procedure call

Method of passing return value(s) back
Procedure Control Flow

- Use stack to support procedure call and return

Procedure call:

```markdown
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:

```markdown
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       0x10c       0x104
%esp  0x108  %esp  0x104  0x104  0x8048553
%eip  0x804854e  %eip  0x8048b90

%eip  is program counter
Procedure Return Example

8048591: c3 ret

%esp 0x104
%eip 0x8048591
%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(...)
{
  •
  •
  •
  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

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo
Stack Operation

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

Call Chain

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

Stack Operation

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

Call Chain

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

```
•
•
•
```

```
yoo
who
amI
```

```
•
•
•
```
Stack Operation

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

Call Chain

```
ymoo
   who
   amI
   amI
```

Frame Pointer
%ebp
Stack Pointer
%esp

yoo
who
amI
amI
Stack Operation

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

Call Chain

```
Frame Pointer
%ebp

Stack Pointer
%esp
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
yoo
who
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```

```
•
•
•
```
Stack Operation

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

Call Chain

![Diagram showing call chain with arrows pointing from `amI` to `who` and `yoo` on the stack.](image-url)
Stack Operation

```
ami (...) {
    ...
    ami();
    ...
}
```

Call Chain

Frame Pointer `%ebp`

Stack Pointer `%esp`

 ami

 ami

 ami

 ami

 ami

 ami

 yoo

 who
Stack Operation

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

Call Chain

```
frame  
  
  yoo  
  who  
  
  amI  
  amI  
  amI
```

Frame Pointer
%ebp

Stack Pointer
%esp
Stack Operation

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

Call Chain

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

```
  •
  •
  •
  •
  yoo
  who
  amI
  amI
  amI
  amI
```
Stack Operation

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

Call Chain

yoo

Frame Pointer
%ebp

Stack Pointer
%esp

•

who

•

amI

•

amI

•

amI
Stack Operation

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

Call Chain

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

Diagram showing function calls and stack operations.
Summary

Today
- Basic stack organization and access
- Activation records (stack frames)
- Call chains

Next time
- Detailed example of calls and stack state
- Register saving conventions
- Recursion