Stacks

15-213: Introduction to Computer Systems
Recitation 5: February 11, 2012

Ian Hartwig
Section F

Thanks to Joon-sup for this week’s slides.
Today: Stacks on stacks on stacks

- News
- Stack discipline review
  - Quick review of registers and assembly
  - Stack frames
  - Function calls (IA32)
  - x86 (IA32) and x86-64
- Example
News

- bomblab is due tomorrow night
  - Don’t use your late days yet
  - But “if you wait till the last minute, it only takes a minute!”
- buflab is coming out tomorrow night
  - All about stacks
- Pro-tip: we love stack questions on exams
  - Students find stack questions difficult
  - Make sure you know stacks well
- Early TA Feedback is still open until Feb. 17th:
  - [https://www.ugrad.cs.cmu.edu/ta/S13/feedback/course.cgi?course=15213](https://www.ugrad.cs.cmu.edu/ta/S13/feedback/course.cgi?course=15213)
Quick review of registers (IA32)

- **Caller saved: %eax, %ecx, %edx**
  - You must save these before a function call if you need them

- **Callee saved: %ebx, %edi, %esi**
  - You must save these before any work if you need them

- **Base pointer: %ebp**
  - Points to the “bottom” of a stack frame

- **Stack pointer: %esp**
  - Points to the “top” of a stack frame

- **Instruction pointer: %eip**
  - Generally don’t need to worry about this one
IA32 stack

- This is a memory region that grows down
- Confusingly, refer to the bottom of the stack as the “top”
- `%esp` refers to the lowest stack address
pushing and popping

- It may be helpful to remember this correspondence (IA32)
  - Note: This is probably not how it actually works

```
pushl src   → subl $4,%esp
            movl src, (%esp)

popl dest  → movl (%esp), dest
            addl $4, %esp
```

- %esp “points” to the top value on the stack
Quick example

```
pushl %eax
popl %edx
```

**Before:**
- %eax: 0x213
- %edx: 0x555
- %esp: 0x108

**After:**
- %eax: 0x213
- %edx: 0x213
- %esp: 0x108
Stack frames

- Every function call is given a stack frame
- What does a C function need?
  - Local variables (scalars, arrays, structs)
    - Space for scalars if the compiler couldn’t allocate enough registers
  - Space to save callee saved registers
  - Space to put computations/temporaries
  - A way to give arguments and call other functions
  - A way to grab arguments
- We can do all these things with the stack!
Function calls

- Use the stack for function calls

- Function call
  - `call label`  Push “return address” on stack, jump to label

- Return address?
  - Address of the instruction immediately after the call
  - Example from disassembly:
    - `804854e: e8 3d 06 00 00  call 8048b90 <main>`
    - `8048553: 50  pushl %eax`
  - Return address is 0x8048553

- Returning from a function call
  - `ret`  Pop return address [(%esp)] into %eip, keep running
  - Remember that the function’s actual return value must be in %eax
What does this look like?

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

call <main>
Returning

- 0x8048591: c3
  - ret

```
0x8048591:
  c3
  ret
```

```
0x110
0x10c
0x108
0x123
0x104
0x8048553
```

```
0x8048591
0x10c
0x108
0x123
0x104
0x8048553
```

%eip 0x8048591
%esp 0x104

%eip 0x8048553
%esp 0x108

%esp ←
Function calls and stack frames

- Suppose you have

```c
int scalar_saxpy(int a, int x, int y)
{
    return a * x + y;
}
int main(void)
{
    int x = 3, y = 4, a = 5;
    return scalar_saxpy(a, x, y);
}
```

- In IA32 we pass arguments using the stack

- `scalar_saxpy` grabs arguments by reaching up the caller’s stack frame!
What exactly does this look like?

- **Hint:** You have probably seen this in bomblab

```c
int scalar_saxpy(int a, int x, int y) {
    return a * x + y;
}
```

08048374 <scalar_saxpy>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Machine Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>08048374</td>
<td>55</td>
<td>push %ebp</td>
<td></td>
</tr>
<tr>
<td>08048375</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
<td></td>
</tr>
<tr>
<td>08048377</td>
<td>8b 45 08</td>
<td>mov 0x8(%ebp),%eax</td>
<td># Move a to %eax</td>
</tr>
<tr>
<td>0804837a</td>
<td>0f af 45 0c</td>
<td>imul 0xc(%ebp),%eax</td>
<td># Multiply x with a</td>
</tr>
<tr>
<td>0804837e</td>
<td>03 45 10</td>
<td>add 0x10(%ebp),%eax</td>
<td># Add y to a*x</td>
</tr>
<tr>
<td>08048381</td>
<td>5d</td>
<td>pop %ebp</td>
<td></td>
</tr>
<tr>
<td>08048382</td>
<td>c3</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Example (from a 213 exam I took)

08048374 <power>:
 # On entry to power(2,4):
 # %esp = 0xffffffff1c,
 # %ebp = 0xffffffff838
8048374: push %ebp
8048375: mov %esp,%ebp
8048377: sub $0x8,%esp
804837a: mov 0xc(%ebp),%edx
804837d: mov $0x1,%eax
8048382: test %edx,%edx
8048384: je 804839c <power+0x28>
8048386: lea 0xffffffff(%edx),%eax
8048389: mov %eax,0x4(%esp)
804838d: mov 0x8(%ebp),%eax
8048390: mov %eax,(%esp)
8048393: call 8048374 <power>
8048398: imul 0x8(%ebp),%eax
804839c: leave
804839d: ret

- Consider the following bit of disassembled IA32 machine code (notice the header comment):
- Suppose that the main routine calls power(2,4) from some unspecified location.
- Fill in the values on the stack immediately after the subsequent call to power(2,3). If a value is unknown, please write UNKNOWN in the blank:
- (Bonus: How does foo access its arguments after the sub?)
Consider the following bit of dissassembled IA32 machine code (notice the header comment):

Suppose that the main routine calls `power(2,4)` from some unspecified location.

Fill in the values on the stack immediately after the subsequent call to `power(2,3)`. If a value is unknown, please write `UNKNOWN` in the blank:

(Bonus: How does `foo` access its arguments after the sub?)

---

**Example (from a 213 exam I took)**

Solution:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>C Equiv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfffffd824:</td>
<td>0x4</td>
<td>int n</td>
</tr>
<tr>
<td>0xfffffd820:</td>
<td>0x2</td>
<td>int x</td>
</tr>
<tr>
<td>0xfffffd81c:</td>
<td>UNKNOWN</td>
<td>return addr</td>
</tr>
<tr>
<td>0xfffffd818:</td>
<td>0xfffffd838</td>
<td>old ebp</td>
</tr>
<tr>
<td>0xfffffd814:</td>
<td>0x3</td>
<td>arg2</td>
</tr>
<tr>
<td>0xfffffd810:</td>
<td>0x2</td>
<td>arg1</td>
</tr>
<tr>
<td>0xfffffd80c:</td>
<td>0x8048398</td>
<td>return addr</td>
</tr>
</tbody>
</table>
But what about floating point arguments?

- It’s complicated
- You don’t need to worry about it
- In some chips there is a separate floating point stack (!!)
- Example of complication
  - In x86-64 the stack on function entry needs to be 16-byte aligned if a function will need to use floating point (??)
  - Many, many more tricky things to know
Stacks/Functions on x86-64

- Everything is even easier
- Arguments ($\leq 6$) are passed in registers
  - %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Extra arguments are still passed on stack – IA32 knowledge is still useful!
- We have nicer compilers now – don’t need %rbp
- Overall less stack usage - potentially better performance
  - See memory hierarchy
- You are expected to know how 64-bit stacks work
  - Even if there are no labs on it
# x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
x86-64 Locals in the Red Zone

/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}

- **Avoiding Stack Pointer Change**
  - Can hold all information within small window beyond stack pointer
  - 128 bytes in x86-64 (don’t need to know)
  - Compiler will choose to use red zone if optimal

```
swap_a:
    movq (%rdi), %rax
    movq %rax, -24(%rsp)
    movq (%rsi), %rax
    movq %rax, -16(%rsp)
    movq -16(%rsp), %rax
    movq %rax, (%rdi)
    movq -24(%rsp), %rax
    movq %rax, (%rsi)
    ret
```
Red Zone vs. Allocated Stack Frame

```
  movq  %rbx, -16(%rsp)  # Save %rbx
  movq  %rbp, -8(%rsp)   # Save %rbp

  subq  $16, %rsp        # Allocate stack frame

  movq  (%rsp), %rbx     # Restore %rbx
  movq  8(%rsp), %rbp    # Restore %rbp

  addq  $16, %rsp        # Deallocate frame
```

```c

```
Generating Machine Code (Buflab)

- We don’t expect you to write x86/x86-64 machine code.
  - CISC architectures have very complicated machine code
- Write assembly code (*.S file), compile, disassemble
  - gcc [codeFilename].S -m32 -c -o [outputFilename]
  - objdump -d [outputFilename]
- Sample .S File:

```
movl $0x1, %eax    # your code here
ret                # and here
```
Questions?
(stacks, bomblab, what is buflab)

(come to office hours if you need help)