15-213 Recitation 8: Midterm Review

Your TAs
Monday, March 2nd, 2020 (15-213, 18-213)
Wednesday, March 4th, 2020 (18-613)
Midterm Exam This Week

- 3 hours + 30 minutes for regrade requests
- Bring your ID!
- 1 double-sided page of notes (in English)
  - No preworked problems from prior exams
- 7 questions

Report to the room
- Log in to the exam server using your andrew id
- Present your CMU ID and cheat sheet to TA, who will then give you access to the server
- Bring your notes sheet and some writing utensil!
Midterm Topics

- Arrays
- Cache
- Bit Operations
- Floating Point
- Stack
- Structs
- Assembly
Floating Point

- Given a floating point representation $S$ $EEE$ $FFFF$ where $S =$ significant bit, $E =$ exponent bits, $F =$ fraction bits, convert these to their proper decimal values

- **1 000 0000**
  
  For normalized numbers:
  
  $M = 1.xxx$  
  $E = \text{exp} - \text{bias}$

- **0 000 1111**
  
  For denormalized numbers:
  
  $M = 0.xxx$  
  $E = 1 - \text{bias}$

- **0 101 0110**

- **1 111 1111**

  $\text{Bias} = 2^{(k-1)} - 1$

  $$v = (-1)^s \ M \ 2^E$$

  $$E = \text{exp} - \text{Bias}$$
Floating Point

- Given a floating point representation $S$ EEEE FFFF where $S =$ significant bit, $E =$ exponent bits, $F =$ fraction bits, convert these to their proper decimal values

- $1\ 000\ 0000$: -$0$ (all zeroes, but sig bit = 1)

- $0\ 000\ 1111$: $\frac{15}{64}$

- $0\ 101\ 0110$: $\frac{11}{2}$

- $1\ 111\ 1111$: NaN
Stack Manipulation

- We execute:

  ```
  mov $0x15213, %rax
  pushq %rax
  ```

- For each of the following instructions, determine if they will result in the value 0x15213 being placed in %rcx?

  1) `mov (%rsp), %rcx`
  2) `mov 0x8(%rsp), %rcx`
  3) `mov %rsp, %rcx`
  4) `popq %rcx`
Stack Manipulation

- We execute:

  ```
  mov 0x15213, %rax
  pushq %rax
  ```

- For each of the following instructions, determine if they will result in the value 0x15213 being placed in %rcx?

  1) `mov (%rsp), %rcx`
  2) `mov 0x8(%rsp), %rcx`
  3) `mov %rsp, %rcx`
  4) `popq %rcx`
Stack is memory

- We execute:

```
mov $0x15213, %rax
pushq %rax
popq %rax
```

- If we now execute:  
  
  ```
mov -0x8(%rsp), %rcx
```
  what value is in %rcx?
  
  1) 0x0 / NULL
  2) Seg fault
  3) Unknown
  4) 0x15213
Stack is memory

- We execute:
  
  \[
  \begin{align*}
  \text{mov} & \ 0x15213, \ %rax \\
  \text{pushq} & \ %rax \\
  \text{popq} & \ %rax
  \end{align*}
  \]

- If we now execute: \( \text{mov} \ -0x8(%rsp), \ %rcx \)
  
  what value is in \%rcx?
  
  1) 0x0 / NULL
  2) Seg fault
  3) Unknown
  4) 0x15213
x86-64 Calling Convention

What does the calling convention govern?

1) How large each type is.
2) How to pass arguments to a function.
3) The alignment of fields in a struct.
4) When registers can be used by a function.
5) Whether a function can call itself.
x86-64 Calling Convention

What does the calling convention govern?

1) How large each type is.
2) How to pass arguments to a function.
3) The alignment of fields in a struct.
4) When registers can be used by a function.
5) Whether a function can call itself.
Register Usage

- The calling convention gives meaning to every register, describe the following 9 registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Function Argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>Return Value</td>
</tr>
<tr>
<td>%rcx</td>
<td>Callee Save</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
</tr>
</tbody>
</table>
Register Usage

- The calling convention gives meaning to every register, describe the following 9 registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Function Argument</th>
<th>Return Value</th>
<th>Callee Save</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Register Usage

Which line is the first violation of the calling convention?

mov $0x15213, %rax
push %rax
mov 0x10(%rsp), %rcx
mov %rbx, %rax
pop %rdx
push %rax
push %rax
pop %rbx
pop %rbx
mov %rcx, %rbx
Register Usage

Which line is the first violation of the calling convention?
Note: this is a function that was called (callee function)

mov $0x15213, %rax
push %rax
mov 0x10(%rsp), %rcx
mov %rbx, %rax
pop %rdx
push %rax
pop %rbx
mov %rcx, %rbx

Until this point, the callee has preserved the callee-save value.
Sometimes arguments are implicit

How many arguments does “rsr” take?
What do you think this function is doing? (Hint: its recursive)

(Note, %sil is the low 8 bits of %rsi)

```
0x0400596 <+0>:     cmp    %sil,(%rdi,%rdx,1)
0x040059a <+4>:     je     0x4005ae <rsr+24>
0x040059c <+6>:     sub    $0x8,%rsp
0x04005a0 <+10>:    sub    $0x1,%rdx
0x04005a4 <+14>:    callq  0x400596 <rsr>
0x04005a9 <+19>:    add    $0x8,%rsp
0x04005ad <+23>:    retq
0x04005ae <+24>:    mov    %edx,%eax
0x04005b0 <+26>:    retq
```
Arguments can already be “correct”

- rsr does not modify s and t, so the arguments in those registers are always correct

```c
int rsr(char* s, char t, size_t pos)
{
    if (s[pos] == t) return pos;
    return rsr(s, t, pos - 1);
}
```
Recursive calls

- Describe the stack after doThis(4) returns.

```c
void doThis(int count)
{
    char buf[8];
    strncpy(buf, "Hi 15213", sizeof(buf));
    if (count > 0) doThis(count - 1);
}
```

```
push %rbx
sub $0x10, %rsp
mov %edi,%ebx
movabs $0x3331323531206948,%rax
mov %rax,(%rsp)
...```
## Recursive Calls

| ret addr (main) | saved rbx | “Hi 15213” | ret addr (doThis 4) | saved rbx | “Hi 15213” | ret addr (doThis 3) | saved rbx | “Hi 15213” | ret addr (doThis 2) | saved rbx | “Hi 15213” |
Struct Alignment

Char: 1 byte
Short: 2 byte
Int, Float: 4 bytes
Long, Double, Pointer: 8 bytes

struct foo {
    int *a;
    char b;
    char c;
    int d;
    short e;
    char buf[4];
};

How would this be represented? Discuss!
Struct Alignment

struct foo {
    int *a;
    char b;
    char c;
    int d;
    short e;
    char buf[4];
};

<table>
<thead>
<tr>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>buf</td>
<td>buf</td>
<td>buf</td>
<td>buf</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Struct Alignment

Char: 1 byte
Short: 2 byte
Int, Float: 4 bytes
Long, Double, Pointer: 8 bytes

struct foo {
    int *a;
    char b;
    char c;
    int d;
    short e;
    char buf[4];
};

struct bar {
    char g;
    int h;
    struct foo f;
};

Now how do we represent bar?
Struct Alignment

```c
struct foo {
    int *a;
    char b;
    char c;
    int d;
    short e;
    char buf[4];
};

struct bar {
    char g;
    int h;
    struct foo f;
};
```

<table>
<thead>
<tr>
<th></th>
<th>g</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>h</th>
<th>h</th>
<th>h</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>f.a</td>
<td>f.a</td>
<td>f.a</td>
<td>f.a</td>
<td>f.a</td>
<td>f.a</td>
<td>f.a</td>
<td>f.a</td>
<td>f.a</td>
</tr>
<tr>
<td>f.b</td>
<td>f.c</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>f.d</td>
<td>f.d</td>
<td>f.d</td>
<td>f.d</td>
</tr>
<tr>
<td>f.e</td>
<td>f.e</td>
<td>f.buf</td>
<td>f.buf</td>
<td>f.buf</td>
<td>f.buf</td>
<td>f.buf</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>