Computer Systems

CS107

Cynthia Lee
Today’s Topics

- Some miscellaneous x86 instructions that gcc likes
  - These may come up in your assign6 bombs!
- Function call and return in x86-64
  - Registers
  - Call stack

Next time:
  - NEW topic: the build process
    - Taking a look at each step of the process
    - Preprocessor, compiler, assembler, linker, loader
Some instructions you might see in your bomb
lea instruction

- “Load effective address”
  - This instruction does some math, usually piecing together a memory address in preparation to do a move
  ```
  lea 0x20(%rsp), %rdi # register %rdi = %rsp + 0x20
  ```
  - Unlike what we may expect from mov with indirect addressing mode, this does NOT do any memory access

- Use for simple addition
  - Because it just does math, not a dereference, sometimes you’ll see gcc use it for addition that has nothing to do with memory addresses
  - Why wouldn’t gcc just use add?  "(╯°□°)╯"
  - Actually, there is a reason having to do with hardware
movbz/movbs instructions

- “Move byte **zero**-extend” and “Move byte **sign**-extend”

**movzbl %al, %edx**
- Copy low (least-significant) byte from register %eax, zero-extend to 4 bytes wide in %edx

**movsbl %al, %edx**
- Copy low (least-significant) byte from register %eax, sign-extend to 4 bytes wide in %edx
  - Sometimes you’ll see this as a way to zero out the top bytes of a register

**movzbl %al, %eax**  # notice src, dst are the same
Another strangely used instruction: xor

- Sometimes you’ll see this puzzlement in your code:
  \[
  \text{xor } \%\text{ebx}, \%\text{ebx}
  \]
  › What is that doing? XOR of a value with itself is always 0.
  › So it’s setting ebx to zero? Why not just do this:
    • \text{mov } 0, \%\text{ebx}

- For hardware reasons, this may be faster (similar reasons as to why gcc would choose \textit{lea} instead of \textit{add})
nop/nop1 instruction

- This instruction is pronounced with two syllables like “no-op”
  - Short for “no operation”
- Literally means to do nothing except increment %rip
- gcc sometimes inserts them because \(\_\(\_\)~/\)
  - Actually the reason is for padding to make functions align on nice multiple-of-8 memory address boundaries

- Also gives rise to a derogatory slang usage you may have heard from computer scientists (e.g., “That person/thing is kind of a nop to me.”) meaning someone or something that doesn’t necessarily do harm, but is useless or unhelpful
Nuance of mov instruction

- Sometimes you’ll see this puzzlement in your code:
  
  \[
  \text{mov } \%ebx, \%ebx
  \]

  What is that doing? Looks like a \textit{nop}!

- \texttt{gcc} is likely using it to zero out the top 32 bits of the register
- When \texttt{mov} instruction is performed on a register whose name starts with “e” (the 32-bit portion), the rest of the 64 bits (the part of the corresponding “r”-named register beyond the “e” part) are cleared out to all zeros
- Same as \texttt{movbz1}
Function calls in assembly

Tools for implementing function call and return
Terminology: “caller” and “callee”

- When talking about function call and return:
  - the function that calls another is known as the “caller”
  - the function that is being called is known as the “callee”

- Of course, a function can simultaneously be a callee and a caller!
  - In using these terms, we just try to be clear for the context which particular caller-callee exchange we are speaking about.
Register state

TOOLS FOR IMPLEMENTING FUNCTION CALL AND RETURN
## Register state associated with function call and return

### Registers (on CPU)

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return value</td>
<td>%rax</td>
</tr>
<tr>
<td>1st argument</td>
<td>%rdi</td>
</tr>
<tr>
<td>2nd argument</td>
<td>%rsi</td>
</tr>
<tr>
<td>3rd argument</td>
<td>%rdx</td>
</tr>
<tr>
<td>4th argument</td>
<td>%rcx</td>
</tr>
<tr>
<td>5th argument</td>
<td>%r8</td>
</tr>
<tr>
<td>6th argument</td>
<td>%r9</td>
</tr>
<tr>
<td>Stack pointer</td>
<td>%rsp</td>
</tr>
<tr>
<td>Instruction ptr</td>
<td>%rip</td>
</tr>
</tbody>
</table>

If the function takes more than 6 arguments, the extras are stored on the stack (in memory not registers)
Memory state

Tools for implementing function call and return
Reminder: what is a stack frame?

MEMORY

- main()
- myfunction()
- foo()
- foo2()
- Heap
- Data
- Text (code)

0x0
Putting it together: registers and memory

Tools for implementing function call and return
Your turn: RSP and RIP roles

<table>
<thead>
<tr>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
</tr>
<tr>
<td>%rip</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Text (code)</td>
</tr>
<tr>
<td></td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>Heap</td>
</tr>
<tr>
<td></td>
<td>foo2()</td>
</tr>
<tr>
<td></td>
<td>foo()</td>
</tr>
<tr>
<td></td>
<td>myfunction()</td>
</tr>
<tr>
<td></td>
<td>main()</td>
</tr>
</tbody>
</table>

Where, generally, do rsp and rip point?

A. rsp and rip both point to the stack
B. rsp points to the stack and rip points to the heap
C. Something else
Typical stack frame layout and functions’ register use

Main memory

Increasing memory addresses

Return address

Local if need

Local if need

Caller’s stack frame

Callee’s stack frame

%rax

%rdi

%rsi

%rdx

%rcx

%r8

%r9

%rsp

%rip
Activity: fill in values

Main memory

parm8
parm7
Return address
Local if need
Local if need

Increasing memory addresses

Caller's stack frame

%rax
%rdi
%rsi
%rdx
%rcx
%r8
%r9
%rsp

Callee's stack frame

int caller() {
    int x = callee(3, 5, 7, 9, 11, 13, 15, 17);
    x++;
    return x;
}

int callee(int parm1, int parm2, int parm3, int parm4, int parm5, int parm6, int parm7, int parm8) {
    int local1 = parm1 + parm2 + parm3 + parm4;
    int local2 = parm5 + parm6 + parm7 + parm8;
    return func(local1, local2);
}
How we address typical stack frame layout

```c
int callee(int parm1, int parm2, int parm3, int parm4, int parm5, int parm6, int parm7, int parm8) {
    int local1 = parm1 + parm2 + parm3 + parm4;
    int local2 = parm5 + parm6 + parm7 + parm8;
    return func(local1, local2);
}
```

```c
<callee>:
add %esi,%edi
add %edx,%edi
add %ecx,%edi
add %r9d,%r8d
mov %r8d,%esi
```

```
callq 4006d0 <func>
repz retq
```
How we address typical stack frame layout

Main memory

parm8
parm7
Return address
Callee local if needed
Callee local if needed

Increasing memory addresses

0x10(%rsp)
0x8(%rsp)

<int callee>(int parm1, int parm2, int parm3, int parm4, int parm5, int parm6, int parm7, int parm8) {
    int local1 = parm1 + parm2 + parm3 + parm4;
    int local2 = parm5 + parm6 + parm7 + parm8;
    return func(local1, local2);
}
Caller-saved registers

Tools for implementing function call and return
Register usage: caller-saved and callee-saved

- There is only one copy of each register on the hardware
  - **Not** the case that each function call or stack frame has their own copy!

- So if you write something to %rax, you write to the %rax that EVERYONE (in particular all other functions on the stack) sees

- If you write something to %rdi, you write to the %rdi that EVERYONE (in particular all other functions on the stack) sees

- To prevent functions from trashing each others’ registers, we have **caller-saved and callee-saved register usage conventions**
  - A sort of etiquette for how to use registers in functions
Register usage: caller-saved and callee-saved

- **Caller-saved**: if you are the caller about to call another function, and you care about keeping the value of a register that is designated as “caller-saved” intact, you’d better copy that value elsewhere before making the function call.
  - It is **not** guaranteed that the value will be preserved by the callee!
  - Your caller-saved register could be ruined by the callee!
  - (If you are the callee, feel free to trash this register.)

- **Callee-saved**: if you are the callee about to change the value of a register that is designated as “callee-saved,” you’d better copy that value elsewhere before changing the register value, and then restore the value from your saved copy before you return.
  - Callee **must** guarantee that the value is preserved (either unchanged, or at least restored to original state before returning).
  - (If you are the caller, feel free to **not** save a copy of the register before calling a function, it’s guaranteed to be there for you safe and sound when the callee function returns!)
Saving backup copies of registers to the stack (memory) using push and pop

- To save caller-saved registers, we often use the stack (in memory, not registers)
- Two instructions help with this:
  - `push op1`
    - Take the value `op1` and store it to the next free slot on the stack (push onto the stack); adjust the `%rsp` to show that the stack now extends lower than before because it has one more item
  - `pop op1`
    - Take the topmost (most recent) element on the stack and pop it off the stack, storing it into `op1`; adjust the `%rsp` to show that the stack now has one fewer item
Saving caller-saved values using push/pop

```c
int recur(int x)
{
    if (x <= 0) return 0;
    return x + recur(x-1);
}
```

Main memory

Return address

Increasing memory addresses

%rip

%rsp

<recur>:
push %rbx
mov %edi,%ebx
test %edi,%edi
jle <recur+0x13>
lea -0x1(%rdi),%edi
callq 40060d <recur>
add %ebx,%eax
jmp <recur+0x18>
mov $0x0,%eax
pop %rbx
retq
Saving caller-saved values using push/pop

Main memory

Return address
%rbx value

Increasing memory addresses

<recur>:
push %rbx
mov %edi,%ebx
test %edi,%edi
jle <recur+0x13>
lea -0x1(%rdi),%edi
callq 40060d <recur>
add %ebx,%eax
jmp <recur+0x18>
mov $0x0,%eax
pop %rbx
retq

int recur(int x)
{
    if (x <= 0) return 0;
    return x + recur(x-1);
}
Saving caller-saved values using push/pop

```c
int recur(int x)
{
    if (x <= 0) return 0;
    return x + recur(x-1);
}
```
How we address typical stack frame layout

```c
int recur(int x)
{
    if (x <= 0) return 0;
    return x + recur(x-1);
}
```
Your turn: the role of $rsp

Main memory

Return address
%rbx value

You saw on myth that we typically print the return address using “p *(void**)rsp” in gdb. Would that work here? If not, how can we print the return address?

A. p *(void**)$rsp (same thing works here)
B. p *(void**)(%rsp + 0x8)
C. p *(void**)(%rsp - 0x8)
D. Something else
(optional study)
More complex stack frame management

This is a less-common way of managing the stack under the new x86-64, but you’ll sometimes see it in GCC output
More complex stack frame layout (with rbp)
For use with complex non-leaf functions

Main memory

(Earlier stack frames)

Caller local variable
Arguments to callee (if the callee function takes more than 6 args)
Caller’s return address
Saved %rbp
Callee local variable
Callee local variable
Callee local variable
“Red zone” 128 bytes

Caller’s stack frame

Callee’s stack frame
More complex stack frame layout (with rbp)

Main memory

- param8
- param7
- Return address
- Saved %rbp
- local1
- local2
- “Red zone” 128 bytes

Caller’s stack frame

- %rax
- %rdi
- %rsi
- %rdx
- %rcx
- %r8
- %r9
- %rbp
- %rsp

Callee’s stack frame

Increasing memory addresses
How we address the more complex stack frame layout (with rbp)

<table>
<thead>
<tr>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increasing memory addresses</td>
</tr>
</tbody>
</table>

- **param8**: $0x18(%rbp)$  # parameters are aligned on 8-byte
- **param7**: $0x10(%rbp)$
- **Return address**: $0x8(%rbp)$
- **Saved %rbp**: [current %rbp points here to saved rbp]
- **local1**: $-0x4(%rbp)$
- **local2**: $-0x8(%rbp)$  # [%rsp points here]