CSE 351: The Hardware/Software Interface

Section 4

Procedure calls

Procedure calls

* In x86 assembly, values are passed to function calls on the stack
  * Perks: Concise, easy to remember
  * Drawbacks: Always requires memory accesses
* In x86-64 assembly, values are passed to function calls in registers
  * Perks: Less wasted space, faster
  * Drawbacks: Potentially requires a lot of register manipulation

x86 calling conventions

* Simply push arguments onto the stack in order, then “call” the function!
* Suppose we define the following function:

```
int sum(int a, int b) {
  return a + b;
}
```

* (See also sum.c from the provided code)

x86 calling conventions

```
int sum(int a, int b) {
  return a + b;
}
```

* In assembly, we have something like this:

```
sum:
  pushl %ebp   # Save base pointer
  movl  %esp, %ebp   # Save stack pointer
  movl  12(%ebp), %eax   # Load b
  movl  8(%ebp), %edx   # Load a
  addl  %edx, %eax   # Compute a + b
  popl  %ebp   # Restore base pointer
  ret   # Return
```

x86 calling conventions

- What is happening with %ebp and %esp?

```
pushl %ebp
```

  - The base pointer %ebp is the address of the caller, which is the location to which “ret” returns. The function pushes it into the stack so that it won’t be overwritten

```
movl %esp, %ebp
```

  - Functions often shift the stack pointer to allocate temporary stack space, so this instruction makes a backup of the original location. In the body of the function, %ebp is now the original start of the stack

```
ret
```

  - When sum() returns, execution picks up at the stored base pointer address. The return value is passed back through %eax

x86 calling conventions

- Now let's look at the caller's side of things

```
int a = 3, b = 2;
int c = sum(a, b);
```

- In assembly code, we have something like this:

```
movl $3, 20(%esp)   # Store a = 3
movl $2, 24(%esp)   # Store b = 3
movl 24(%esp), %eax   # Load b
movl %eax, 4(%esp)   # Store b for call
movl 20(%esp), %eax   # Load a
movl %eax, (%esp)   # Store a for call
call sum   # Call the sum() function
```

x86 calling conventions

* Note that the given assembly code is terribly inefficient, but it's what GCC will emit without any optimization
* The value to which the stack pointer %esp points is the first parameter (a in this case) while the second (b) is stored just above at 4(%esp)

x86-64 calling conventions

* %rdi, %rsi, %rdx, %rcx, %r8, and %r9 act as the first through sixth arguments to functions
* The return value from a function is stored in %rax
* All of these registers are caller-saved (more on this later)

x86-64 calling conventions

- The sum example from earlier in x86-64:

```
sum:
  pushq %rbp   # Save base pointer
  movq  %rsp, %rbp   # Save stack pointer
  movl  %edi, -4(%rbp)   # Store a
  movl  %esi, -8(%rbp)   # Store b
  movl  -8(%rbp), %eax   # Load b
  movl  -4(%rbp), %edx   # Load a
  addl  %edx, %eax   # Compute a + b
  popq  %rbp   # Restore a + b
  ret   # Return
```

- Again, this is unoptimized GCC output

x86-64 calling conventions

- What changed compared to the x86 example?
  - a and b passed through %rdi (actually %edi, since it’s an int) and %rsi (%esi)
  - Manipulation of %rbp and %rsp is just like that of %ebp and %esp in the x86 version

x86-64 calling conventions

- From the caller's side:

```
movl $3, -12(%rbp)   # Store a
movl $2, -8(%rbp)   # Store b
movl -8(%rbp), %edx   # Load b
movl -12(%rbp), %eax   # Load a
movl %edx, %esi   # Move b to %esi
movl %eax, %edi   # Move a to %edi
call  sum   # Call the sum() function
```

- Lots of wasteful register and stack manipulation, but 3 and 2 end up as first and second parameters to call to sum()

Caller- versus callee-saved

- Some registers are caller-saved, whereas some are callee-saved
- Caller-saved: If the contents of the register need to be preserved, the caller should save them on the stack prior to invoking a function
- Callee-saved: If the callee of a function wants to use a register, it must save the value and restore it to the register before returning

Caller- versus callee-saved

- In x86, the callee-saved registers are %edx, %esi, %edi, and %ebp; all others are caller-saved
- In x86-64, the callee-saved registers are %rbx, %rbp, and %r12-%r15; all others are caller-saved
- Why use a callee-saved register versus a caller-saved register and vice versa?

Calling convention examples

- Next we'll take a look at some examples to go over usage of these conventions
- The code is available under today's section on the course website
- After running "make", you'll have a some binary files, some assembly files (.s), and some listing files (.lst), the latter of which contains a mix of assembly and the original C code that was used to generate it