CSO: Recitation 10

Goktug Saatcioglu

10.31.2019

1 Continuing Assembly

2 Function Calls

When a function `P` calls `Q` we have to deal with the following issues

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return

2.1 x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address (address of "top" element)

2.2 x86-64 Stack Instructions

- `pushq Src`
  - Fetch operand at `Src`
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`
  - It is equivalent to the following instructions:

```
subq $8, %rsp
movq Src, (%rsp)
```

- `popq Dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at `Dest` (must be register)
  - It is equivalent to the following instructions:

```
movq (%rsp), Dest
addq $8, %rsp
```

2.3 Callee vs Caller

When `P` calls `Q`

- `P` is suspended and control moves to `Q`.
- A stack frame is setup on top of the stack for `Q`
- That stack frame contains:
  - saved registers
  - local variables
  - arguments if `Q` is calling another function
- Some procedures may not need a stack frame (why?).

2.4 Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label` [or `call *op`]
  - Push return address on stack
  - Jump to label
- Return address:
  - Address of the next instruction right after call
- Procedure return: `ret`
  - Pop address from stack
  - Jump to address

2.5 Calling Conventions

Answers questions such as:

- Which arguments are passed in which registers?
- Which register holds the return value?
- Where can auxiliary arguments that don't fit into registers be found on the stack?
- Who is responsible for restoring which registers?
- caller vs. callee-saved registers

Calling conventions are part of the Application binary interface (ABI), which are typically OS-specific. We focus on the ABI for x86-64 adhered to by most Unix-like operating systems. See slides for the table of calling conventions.

2.6 Procedure Data Flow

- Registers, first 6 arguments: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
- Stack:
  - Only allocate stack space when needed
  - When passing parameters on the stack, all data sizes are rounded up to be multiple of eight.
- Return value: `%rax`

2.7 When is local storage needed?

- Not enough registers
- A variable in high-level language is passed by reference (`&` in C) so it needs to have an address!
- Arrays, structures, ...

3 Exercise

So far we have been looking at C code and figuring out what the relevant assembly is. Let's try something different today. Consider the simplified assembly code given below.

```
_fun:                                   ## @fun
  pushq %rbp
  movq  %rsp, %rbp
  imulq %rdx, %rsi
  addq  (%rdi), %rsi
  movq  %rsi, (%rdi)
  movq  %rsi, %rax
  popq  %rbp
  retq

_main:                                  ## @main
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16, %rsp
  movq  $5, -8(%rbp)
  leaq  -8(%rbp), %rdi
  movl  $7, %esi
  movl  $11, %edx
  callq _fun
  xorl  %eax, %eax
  addq  $16, %rsp
  popq  %rbp
  retq
```

So, what is this code doing? We should approach this problem modulalarly. In my opinion, it is easier to start with the caller rather than the callee. So we will follow this approach here (however, both approaches should work just fine). Let's consider the function `_main`. It starts with the instructions

```
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16, %rsp
```

which as we now know first starts off by creating a new stack frame on the stack by doing `pushq %rbp`, then it moves the current value of `%rsp` into `%rbp` so that we have a frame pointer for this stack and finally it decrements the stack pointer by `$16`. This last instruction should gives us a clue about what is possbily going on as we grow the stack by 16 bytes. The first 8 bytes are for the temporary frame pointer value we saved and the next 8 bytes will hold our data. Next up, we have the instructions

```
  movq  $5, -8(%rbp)
  leaq  -8(%rbp), %rdi
```

which stores the value `$5` into 8 bytes below the frame pointer with the first instruction. This is most likely a local variable that holds the value 5. We also do `leaq  -8(%rbp), %rdi` and we know that `%rdi` is the

first register used for an argument of a function call. Also, it seems that the type of this local variable is `long` as we use 8 bytes to store it. So, recalling what `leaq` does, we load the address of this variable onto the first argument register. Next, we have the instructions

```
movl  $7, %esi
movl  $11, %edx
```

and recall that `%esi` is the second argument register and `%edx` is the third argument register. Since a register that begins with `e` is only 4 bytes we can conclude that the types of these values are most likely int (I say most likely and we will see why later). There is a call to the function `_fun` and then we have this strange instruction `xorl  %eax, %eax`. Can you tell what it does? Well, xor of anything with itself is always zero so it basically zeros out the first 32-bits of the return reigster `%rax`. Note that the value is never stored anywhere so it seems that whatever the function computes, we do not store as a local variable. Next, we have the instructions

```
addq  $16, %rsp
popq  %rbp
retq
```

which basically pops off the stack frame for `_main`. Bringing everything together, `_main` appears to look like this

```
int main() {
  long x = 5;
  fun(&x, 7, 11);
  return 0;
}
```

So now we turn our attention to the function call. We have the usual stack frame set-up with the instructions

```
pushq %rbp
movq  %rsp, %rbp
```

which is followed by

```
imulq %rdx, %rsi
addq  (%rdi), %rsi
movq  %rsi, (%rdi)
```

which basically multiplies the second and third argument of the function, adds whatever was stored in the first argument of the function and then stores the result in the address given by the first argument of the function. Notice that the arguments are now in `%rxx` compared to what we had in main which was `%exx`. It seems that the arguments to the function are 64-bits. Next, we have

```
movq  %rsi, %rax
popq  %rbp
retq
```

which basically moves whatever we just calculated into the return register and pops off the stack frame which ends the function call. Notice how the assembly uses `%rbp` which is a caller-saved register to calculate intermediate values. Bringing everything together, we see that the function `_fun` in C could look something like this

```
long f_ma(long* a, long b, long c) {
  *a = *a + (b * c);
  return *a;
}
```

This completes our translation of the assembly back into C.

I would like to point out one more thing here. The original code used to get this assembly was

```
long f_ma(long* a, long b, long c) {
  *a = *a + (b * c);
  return *a;
}

int main() {
  long a = 5;
  long b = 7;
  long c = 11;
  long res = f_ma(&a, b, c);

  return 0;
}
```

which suggests to use that the compiler did some optimizations as it realized it was not necessary to store the values of `b`, `c` and `res` on the stack. Furthermore, for the function `f_ma` the compiler does not store any local variables as it realizes it can only use registers. What's even more interesting is that the compiler allocates 16 bytes on the stack but only uses 8 of it. This suggests to us that even more optimization can be done. To generate this code I used the following command

```
gcc -Og -S -fno-asynchronous-unwind-tables -fno-stack-protector
```

and for more even more aggresive optimization we can use the command

```
gcc -O2 -S -fno-asynchronous-unwind-tables -fno-stack-protector
```

which gets us the following assembly

```
_f_ma:                                  ## @f_ma
  pushq %rbp
  movq  %rsp, %rbp
  imulq %rdx, %rsi
  addq  (%rdi), %rsi
  movq  %rsi, (%rdi)
  movq  %rsi, %rax
  popq  %rbp
  retq
_main:                                  ## @main
  pushq %rbp
  movq  %rsp, %rbp
  xorl  %eax, %eax
  popq  %rbp
  retq
```

which is basically the compiler realizing that the local variables in `main()` have no use so it never allocates them and then it never does the function call `f_ma()` since its return value will also not be used. Notice how the compiler will still zero out the return register. Finally, it still translates the code for `f_ma()` fully as even though the function is not called here the compiler cannot just eliminate the code entirely. The idea here is that the compiler may change the code you write to make it more performant or space-optimized. Thus, you are not gauranteed to always get back the same code if you translate assembly back into C but you will get back code that are functionally (semantically) equivalent.