CS107, Lecture 20

Assembly: Function Call, Take II

Reading: B&O 3.7

Ed Discussion: https://edstem.org/us/courses/46162/discussion/3817979

Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan, Nick Troccoli, and others.

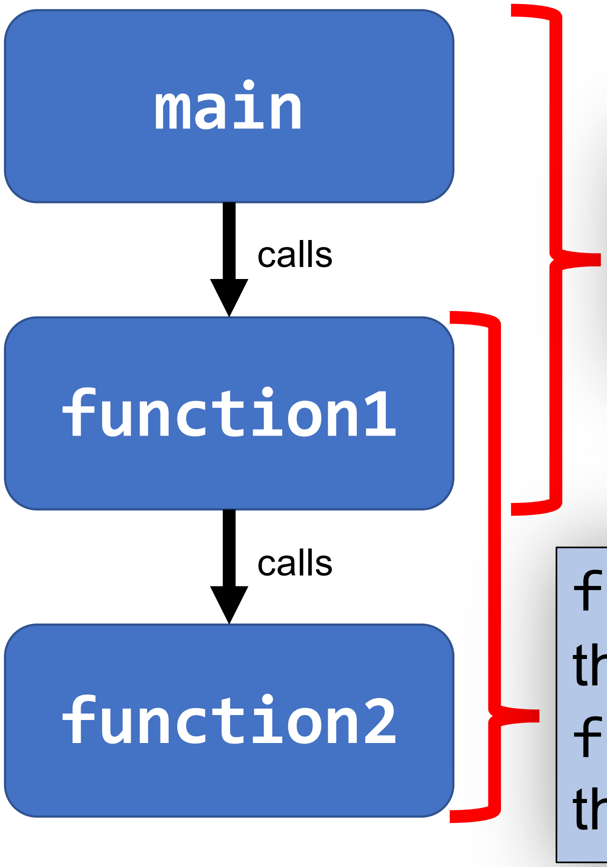
Register Restrictions

There is only one copy of registers for all programs and functions.

- **Problem:** what if ***funcA*** is building up a value in register %r10, and calls ***funcB*** in the middle, which also has instructions that modify %r10? ***funcA***'s value will be destroyed!
- **Solution:** lay down some "rules of the road" that callers and callees must follow when using registers so they do not interfere with one another.
- These rules define two types of registers: **caller-owned** and **callee-owned**

Caller/Callee

Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. `function1` at right).



`main` is the caller, and `function1` is the callee.

`function1` is the caller, and `function2` is the callee.

Register Restrictions

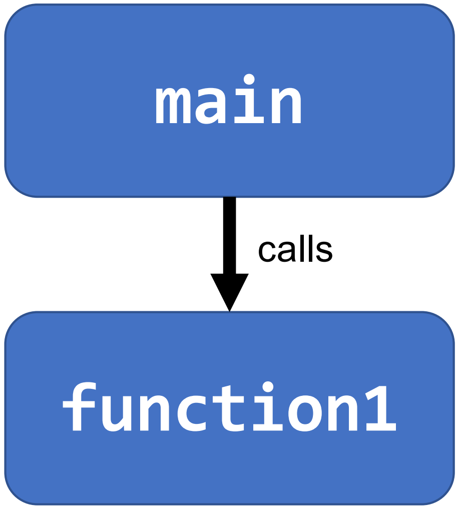
Caller-Owned

- Callee must *save* the existing value and *restore* it when done.
- Caller can store values in them and assume they'll be preserved across function calls.

Callee-Owned

- Callee does not need to save the existing value.
- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.

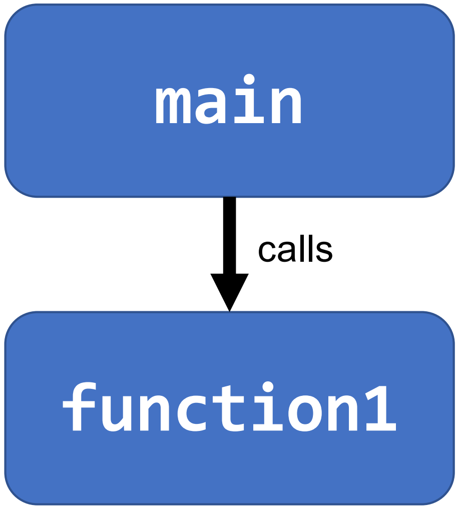
Caller-Owned Registers



`main` can use caller-owned registers and know that `function1` will not permanently modify their values.

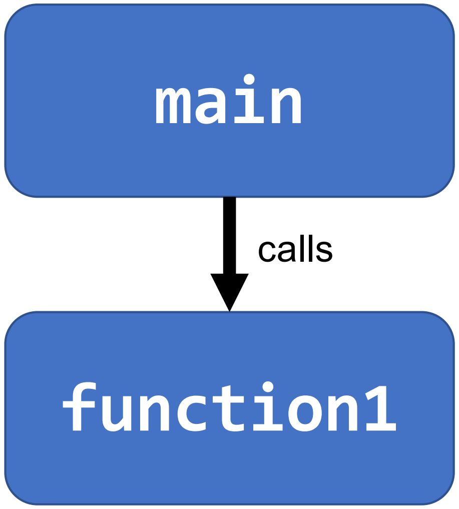
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers



```
function1:
  push %rbp
  push %rbx
  ...
  pop %rbx
  pop %rbp
  retq
```

Callee-Owned Registers



main

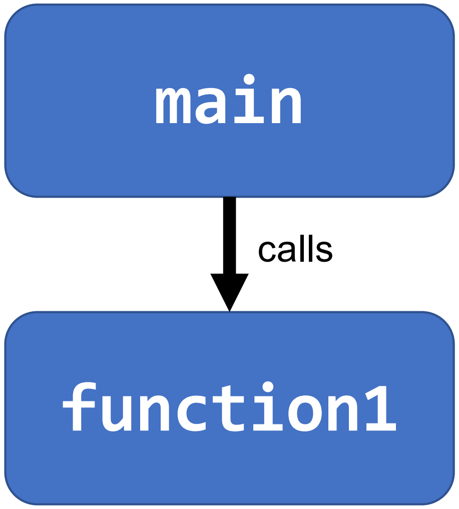
calls

function1

`main` can use callee-owned registers but calling `function1` may permanently modify their values.

If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

Callee-Owned Registers



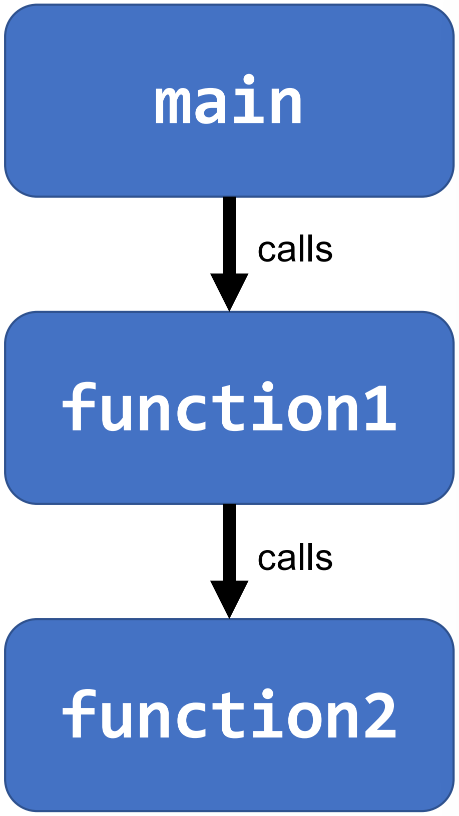
main

calls

function1

```
main:
   ...
   push %r10
   push %r11
   callq function1
   pop %r11
   pop %r10
   ...
```

A Day In the Life of `function1`



Caller-owned registers:

- **`function1`** must save/restore existing values of any it wants to use.
- **`function1`** can assume that calling **`function2`** will not permanently change their values.

Callee-owned registers:

- **`function1`** does not need to save/restore existing values of any it wants to use.
- calling **`function2`** may permanently change their values.

Example: Recursion

- Let’s look at an example of recursion at the assembly level.
- We’ll use everything we’ve learned about registers, the stack, function calls, parameters, and assembly instructions!
- We’ll also see how helpful GDB can be when tracing through assembly.

`factorial.c and factorial`

gdb tips ⭐⭐⭐

`layout split`	(ctrl-x a: exit, ctrl-l: resize, refresh: refresh, layout reg/asm, focus next)	View C, assembly, and gdb (lab5)
`info reg`		Print all registers
`p $eax`		Print register value
`p $eflags`		Print all condition codes currently set
`b *0x400546`		Set breakpoint at assembly instruction
`b *0x400550 if $eax > 98`		Set **conditional breakpoint**
`ni`		Next assembly instruction
`si`		Step into assembly instruction (will step into function calls)

gdb tips ⭐⭐⭐

`p/x $rdi`	Print register value in hex
`p/t $rsi`	Print register value in binary
`x $rdi`	Examine the byte stored at this address
`x/4bx $rdi`	Examine 4 bytes starting at this address
`x/4wx $rdi`	Examine 4 ints starting at this address
`finish`	Finish function, return to caller

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We’re done with all our assembly lectures! Now we can fully understand what’s going on in the assembly below, including how someone would call sum_array in assembly and what the **ret** instruction does.

```
0000000000401136 <sum_array>:
  401136 <+0>:  mov    $0x0,%eax
  40113b <+5>:  mov    $0x0,%edx
  401140 <+10>: cmp    %esi,%eax
  401142 <+12>: jge    0x40114f <sum_array+25>
  401144 <+14>: movslq %eax,%rcx
  401147 <+17>: add    (%rdi,%rcx,4),%edx
  40114a <+20>: add    $0x1,%eax
  40114d <+23>: jmp    0x401140 <sum_array+10>
  40114f <+25>: mov    %edx,%eax
  401151 <+27>: retq
```