Recitation 07

More Assembly

Today’s agenda

- We will discuss in recitation
  - More assembly programming
    - Function calls and the stack
    - Linking
- What you will do tonight
  - R07
    - Do something idk

Procedure Calls

Calling functions

How do you call functions?

- How do you actually start executing the code of a function?
  - Well, we know about jmp, does that help us? Why not?
- Do you need to do something before calling a function?
  - What?

How do you call functions?

```
mystrlen:
movl $0, %eax
jmp .condition
.loop:
addl $1, %eax
.condition:
movb (%rdi,%rax), %bl
cmp $0, %bl
jne .loop
```

```
main:
jmp mystrlen
```

How do you call functions?

```
trle
ovl $0, %
jmp .conditio
.loop:
addl $1, %eax
.condition:
movb (%rdi,%rax), %bl
cmp $0, %bl
ne .loop
```

```
main:
jmp mystrlen
```

How do you call functions?

```
mystrlen:
movl $0, %eax
jmp .condition
.loop:
addl $1, %eax
.condition:
movb (%rdi,%rax), %bl
cmp $0, %bl
jne .loop
//How do we get back?
```

```
main:
jmp mystrlen
```

How do you call functions?

```
mystrlen:
movl $0, %eax
jmp .condition
.loop:
addl $1, %eax
.condition:
movb (%rdi,%rax), %bl
cmp $0, %bl
jne .loop
//How do we get back?
```

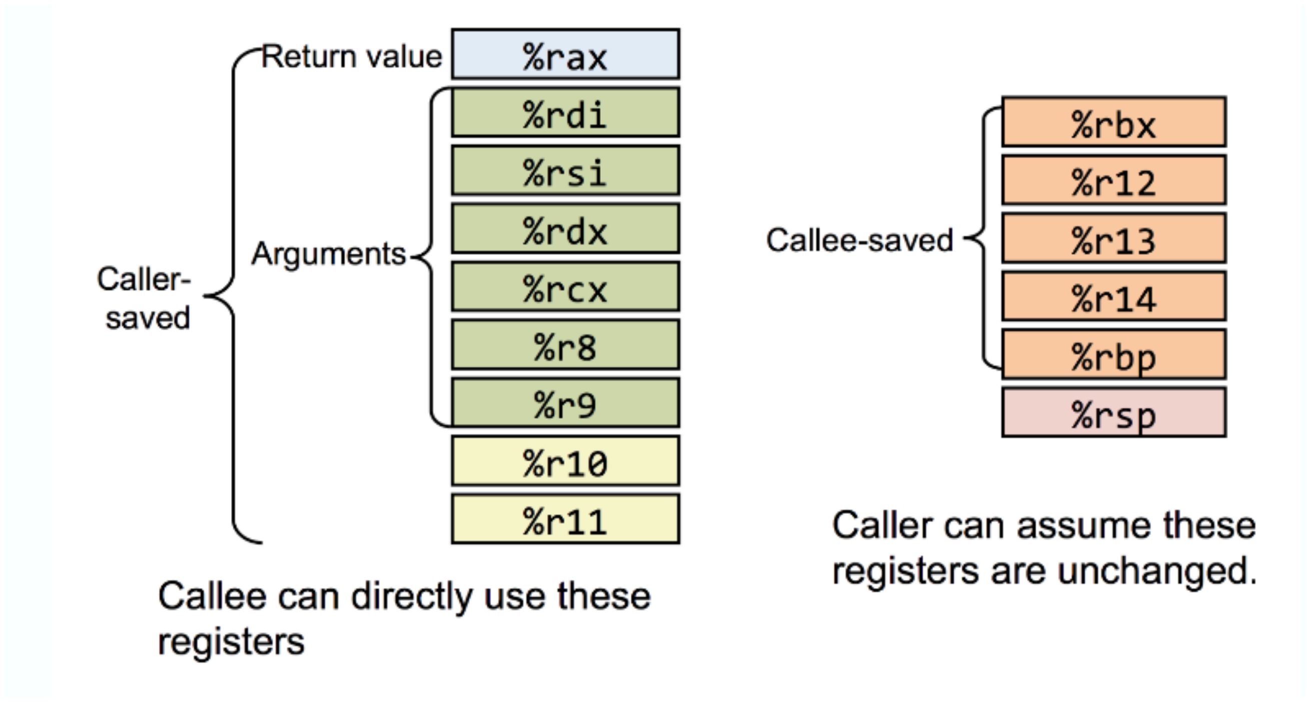
```
main:
//Where are the arguments?
jmp mystrlen
```

Remember where we came from

- A function that calls another (a caller) knows what it is calling
- A function that is called (a callee) does not know who its caller is
  - But it needs to know where to resume execution when it is done
  - It is the responsibility of the caller to tell the callee where to resume execution
  - We want to resume execution on the instruction after we called the function
  - We store this return address on the stack
    - callq handles this for us

Set up registers

- As mentioned last time, arguments are stored in %rdi, %rsi, %rdx, %rcx, %r8, and %r9
  - So when calling a function, you must set those registers to the correct value for that argument
- If the caller was using the argument registers for something, it must save them first, as the callee may use those registers for any purpose
  - It can save them to the stack, as with the return values
  - This is also true of the registers %r10, %r11, and %rax
- The callee must save certain registers if it plans on using them
  - They are %rbx, %r12, %13, %r14, %rbp, and %rsp



Callee can directly use these registers

Caller can assume these registers are unchanged.

Set up registers

The Stack

- The register %rsp points to the top of the stack
- The stack grows downwards
- We use it to store return addresses as well as registers whose values we don’t want to lose
- We also use it to store local variables
- You can use pushq and popq to add and remove things from the stack

The Stack

pushq

- Takes one operand
- **DECREASES** %rsp by 8
- THEN stores the operand at the memory location given by the new %rsp

The Stack

popq

- Takes one operand
- Takes the value in memory located at %rsp and stores it in the operand
- THEN **INCREASES** %rsp by 8

The Stack

callq

- Takes one operand
- **DECREASES** %rsp by 8
- THEN stores the %rip at the memory location given by the new %rsp
- THEN jumps to the operand

The Stack

retq

- Takes no operands
- Jumps to the location given by the value in memory located at %rsp
- THEN **INCREASES** %rsp by 8

Arrays

And assembly

That crazy complete addressing mode

- Remember the crazy (%rsi, %rdi, 4) address notation?
- This is super useful for accessing arrays
  - Why?
  - If I wanted to copy an array element into a register, how would I do that for an array of int?
  - For an array of chars?

2D Arrays

- In C, a 2D array is stored in row major order
  - That means elements within a row are contiguous in memory
  - Consider this array:

```
int myFavorite[3][6] = {{2, 9, 7, 3, 5, 6},
                        {1, 1, 3, 4, 5, 6},
                        {9, 3, 7, 0, 1, 2}};
myFavorite[0][0] is 2
myFavorite[2][5] is the other 2
&myFavorite[i][j] is myFavorite + (i*6 + j)*4
```

2D Arrays

```
int myFavorite[3][6] = {{2, 9, 7, 3, 5, 6},
                        {1, 1, 3, 4, 5, 6},
                        {9, 3, 7, 0, 1, 2}};
```

- How do we address this in assembly?
  - We want `myFavorite + (i*6 + j)*4`
  - Say `%rax` contains myFavorite, `%rsi` contains i, `%rdi` contains j
  - Move result into `%ebx`

```
leaq (%rsi, %rsi, 2), %rsi //%rsi = 3*%rsi
addq %rsi, %rsi            //%rsi = 2*%rsi
addq %rdi, %rsi            //%rsi = %rsi + %rdi
movl (%rax, %rsi, 4), %ebx //finally get the value
```

Arrays of arrays

- argv is stored as an array of pointers, where each pointer points to an array of characters
- `argv = {0x7f00, 0x7d00, 0x7e00}`
- `0x7f00: ”hello world”`
- `0x7d00: ”it is thurs”`
- `0x7e00: ”pizza time!”`

Arrays of arrays

- How do we address argv in assembly?
  - We want the jth character of the ith string
  - `We want *(*(argv+i)+j)`
  - `Say %rax contains argv, %rsi contains i, %rdi contains j`
  - `Move result into %bl`

```
movq (%rax, %rsi, 8), %rax  //%rax = %rax + 8*%rsi
movb (%rax, %rdi), %bl      //get the character
```