Systems Programming

A6

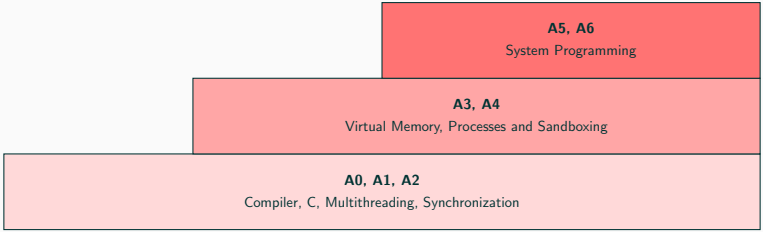
Florian Kargl

November 22, 2019

IAIK – Graz University of Technology

A6 - Inline Assembly and Calling Conventions

Course Overview



A6 - Inline Assembly and Calling Conventions

Have you ever wondered what happens in your CPU when you call a function?

Caller

```
int main()
{
  // ...
  foo();
  // ...
}
```

Callee

```
void foo()
{
  // do stuff...
}
```

Function Calls

Let's take a look at the compiler output

```
objdump -d <executable>
```

Caller (ASM)

```
main:
  # ...
  call foo
  # ...
```

Callee (ASM)

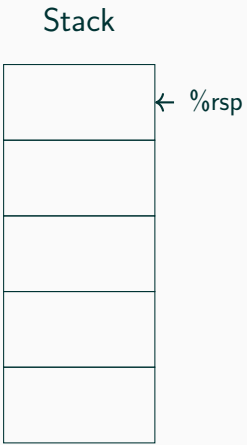
```
foo:
  # do stuff...
  ret
```

Caller (ASM)

```
main:
  # ...
  call foo
  # ...
```

Callee (ASM)

```
foo:
  # do stuff...
  ret
```



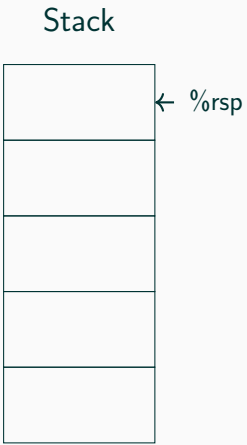
Caller (ASM)

```
main:
  # ...
  call foo
  # ...
```

Call instruction pushes return address onto stack and jumps to target

Callee (ASM)

```
foo:
  # do stuff...
  ret
```



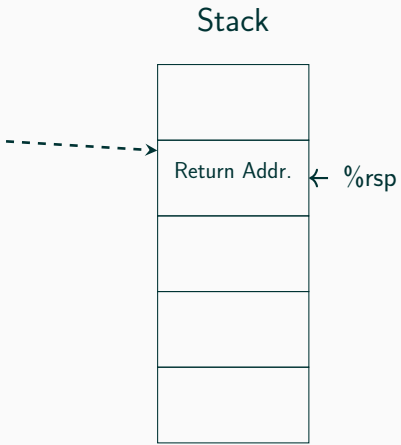
Caller (ASM)

```
main:
  # ...
  call foo
  # ...
```

Call instruction pushes return address onto stack and jumps to target

Callee (ASM)

```
foo:
  # do stuff...
  ret
```



Function Calls

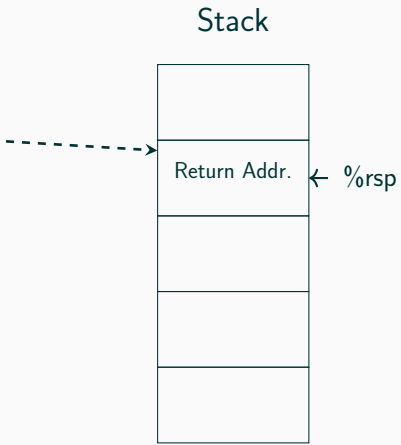
Caller (ASM)

```
main:
  # ...
  call foo
  # ...
```

Call instruction pushes return address onto stack and jumps to target

Callee (ASM)

```
foo:
  # do stuff...
  ret
```



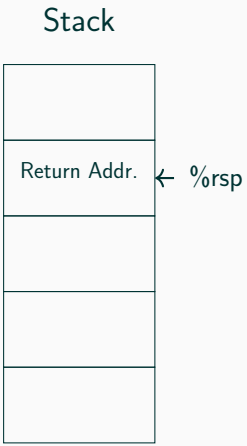
Caller (ASM)

```
main:
  # ...
  call foo
  # ...
```

Callee (ASM)

```
foo:
  # do stuff...
  ret
```

Ret instruction pops return address from stack and jumps back



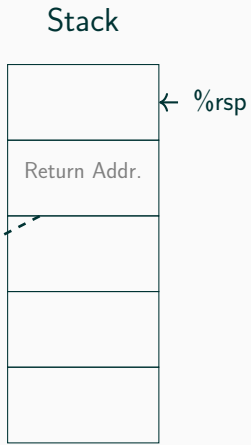
Caller (ASM)

```
main:
  # ...
  call foo
  # ...
```

Callee (ASM)

```
foo:
  # do stuff...
  ret
```

Ret instruction pops return address from stack and jumps back



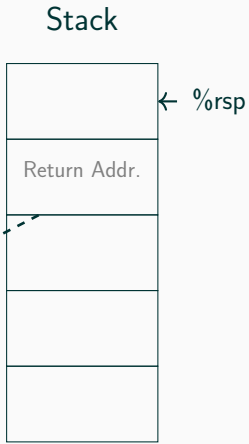
Caller (ASM)

```
main:
  # ...
  call foo
  # ...
```

Callee (ASM)

```
foo:
  # do stuff...
  ret
```

Ret instruction pops return address from stack and jumps back



Easy enough, but what about function arguments and return values?

Caller

```
int main()
{
  char arg1 = 5;
  char arg2 = 7;

  int retval = foo(arg1, arg2);
}
```

How does this...

Callee

```
int foo(char a, char b)
{
  return a > b;
}
```

Easy enough, but what about function arguments and return values?

Caller

```
int main()
{
  char arg1 = 5;
  char arg2 = 7;

  int retval = foo(arg1, arg2);
}
```

Callee

```
int foo(char a, char b)
{
  return a > b;
}
```

...get here?

How does this...

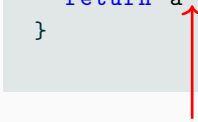
Easy enough, but what about function arguments and return values?

Caller

```
int main()
{
  char arg1 = 5;
  char arg2 = 7;

  int retval = foo(arg1, arg2);
}
```

Callee

```
int foo(char a, char b)
{
  return a > b;
}
```



And this...

Easy enough, but what about function arguments and return values?

Caller

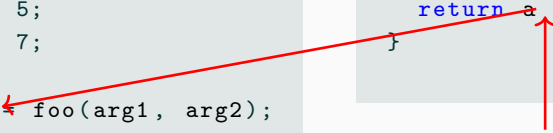
```
int main()
{
  char arg1 = 5;
  char arg2 = 7;

  int retval = foo(arg1, arg2);
}
```

...back here?

Callee

```
int foo(char a, char b)
{
  return a > b;
}
```

And this...



Where do we put the function arguments?

Function Arguments

Where do we put the function arguments?

- Registers

Function Arguments

Where do we put the function arguments?

- Registers
  - Which ones?

Where do we put the function arguments?

- Registers
  - Which ones?
  - What if we don't have enough registers?

Function Arguments

Where do we put the function arguments?

- Registers
  - Which ones?
  - What if we don't have enough registers?
- Memory (i.e. on the stack)

Function Arguments

Where do we put the function arguments?

- Registers
  - Which ones?
  - What if we don't have enough registers?
- Memory (i.e. on the stack)
  - In which order?

A calling convention defines the interaction between functions on the level of CPU-instructions

- Function parameters
- Return values
- Registers that need to be saved/restored across function calls

Calling conventions are not only relevant within a single binary. All interfaces between binary modules need to conform to a common interface to be compatible.

Calling conventions are not only relevant within a single binary. All interfaces between binary modules need to conform to a common interface to be compatible.

- Object files that are linked together at compile time
- Dynamically loaded libraries (e.g. libc)

Calling conventions are not only relevant within a single binary. All interfaces between binary modules need to conform to a common interface to be compatible.

- Object files that are linked together at compile time
- Dynamically loaded libraries (e.g. libc)

$\Rightarrow$ Defined as part of an ABI (Application Binary Interface)

- A complete ABI also defines the executable format (e.g. ELF), instruction set, ...

Calling Conventions

The used ABI/calling convention depends on

- CPU architecture
- Operating system
- Compiler

The used ABI/calling convention depends on

- CPU architecture
- Operating system
- Compiler

Mostly standardized

Commonly used calling conventions

	Linux	Windows
i386	cdecl	cdecl, stdcall, fastcall, ... *
x86_64	System V amd64 ABI	Microsoft x64

System calls usually use a different calling convention than the rest of the userspace

* 32-bit Windows is a bit of a mess

Calling Conventions

Calling conventions relevant for the assignment

	Linux	Windows
i386	cdecl	cdecl, stdcall, fastcall, ...
x86_64	System V amd64 ABI	Microsoft x64

Main difference: Function arguments on stack vs. in registers

In this assignment you will need to write (inline) assembly.

No C code allowed!

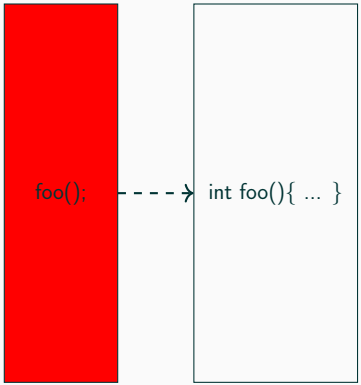
GCC allows you to write assembly code inside C functions

GCC Inline Assembly

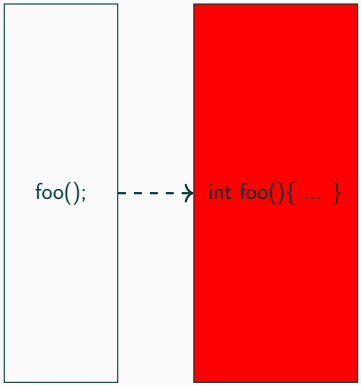
```
int foobar(uint64_t* result) {
  uint64_t a = 3;
  uint64_t b = 4;

  asm("movq %[op1], %%rax\n"
      "addq %[op2], %%rax\n"
      "movq %%rax, %[res]\n"
  :[res]"=m"(*result) // output (memory location, not value)
  :[op1]"m"(a),       // input (op1 in memory)
   [op2]"r"(b)        //       (op2 in register)
  :"rax", "cc");      // clobbers the rax register and status flags
 ("m" output constraint -> no need to explicity list "memory")
}
```

Your Tasks

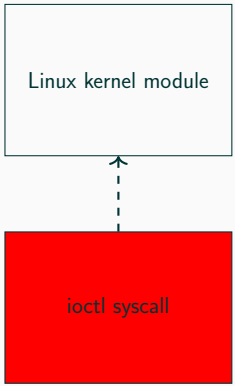
- A - Implement the caller side of a function call using gcc inline assembly
  - cdecl (32-bit)
  - System V amd64 ABI (64-bit)
- B - Implement a function in assembly for different calling conventions
  - cdecl (32-bit)
  - System V amd64 ABI (64-bit)
- C - Call Linux system calls using gcc inline assembly
  - int 0x80 (32-bit)
  - syscall (64-bit)



- A - Implement the caller side of a function call using gcc inline assembly
  - cdecl (32-bit)
  - System V amd64 ABI (64-bit)
- B - Implement a function in assembly for different calling conventions
  - cdecl (32-bit)
  - System V amd64 ABI (64-bit)
- C - Call Linux system calls using gcc inline assembly
  - int 0x80 (32-bit)
  - syscall (64-bit)



- A - Implement the caller side of a function call using gcc inline assembly
  - cdecl (32-bit)
  - System V amd64 ABI (64-bit)
- B - Implement a function in assembly for different calling conventions
  - cdecl (32-bit)
  - System V amd64 ABI (64-bit)
- C - Call Linux system calls using gcc inline assembly
  - int 0x80 (32-bit)
  - syscall (64-bit)



Questions?