System Level Programming

Calling Conventions, Inline Assembly, Kernel Modules

Claudio Canella

November 22, 2021

IAIK – Graz University of Technology

Outline

ABI - 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...
}
```

Let's take a look at the compiler output

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

Caller (ASM)

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

Callee (ASM)

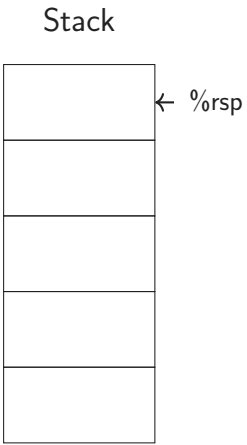
```
foo:
  # do stuff...
  ret
```

Caller (ASM)

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

Callee (ASM)

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



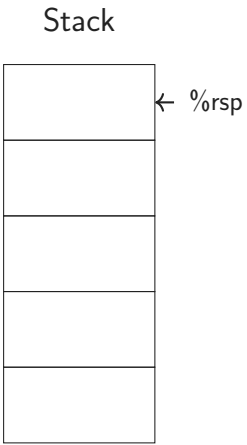
Caller (ASM)

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

Call instruction pushes return address onto stack and jumps to target

Callee (ASM)

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



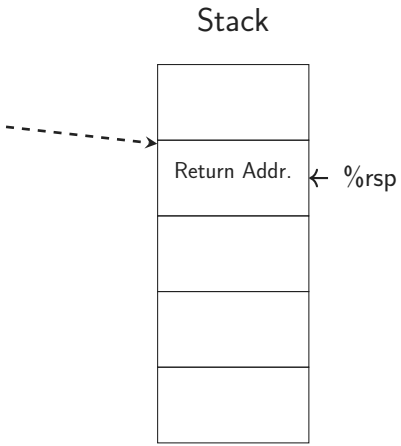
Caller (ASM)

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

Call instruction pushes return address onto stack and jumps to target

Callee (ASM)

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



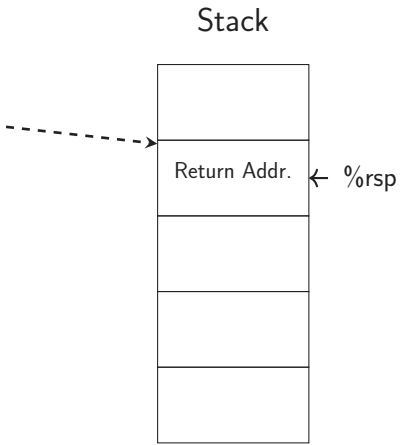
Caller (ASM)

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

Call instruction pushes return address onto stack and jumps to target

Callee (ASM)

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



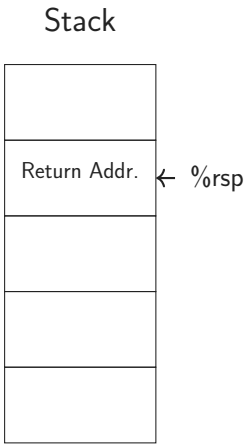
Caller (ASM)

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

Callee (ASM)

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

Ret instruction pops return address from stack and jumps back



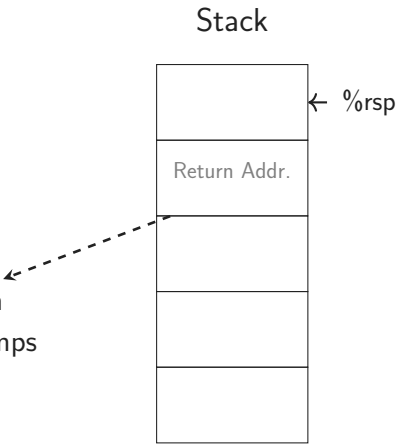
Caller (ASM)

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

Callee (ASM)

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

Ret instruction pops return address from stack and jumps back



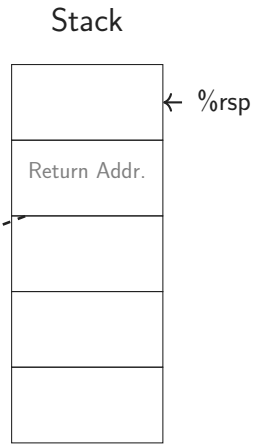
Caller (ASM)

```
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?

Where do we put the function arguments?

- Registers

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?

Where do we put the function arguments?

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

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, ...

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

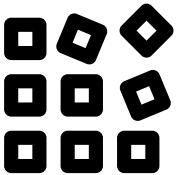
	Linux	Windows
i386	cdecl	cdecl, stdcall, fastcall, ...
x86_64	System V amd64 ABI	Microsoft x64

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

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

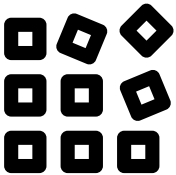
Main difference: Function arguments on stack vs. in registers

Inline Assembler



Sometimes it is required to use assembler within a C program

- e.g., for optimization of certain computational tasks
- or tasks, which require low level instructions



Sometimes it is required to use assembler within a C program

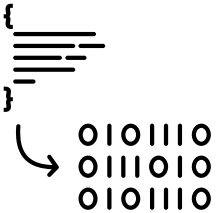
- e.g., for optimization of certain computational tasks
- or tasks, which require low level instructions

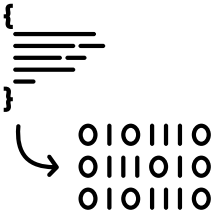
We use the GNU inline assembler

- Allows to inline assembler code in C programs
- Uses AT&T syntax

Assembler is a low level language,

- easily convertable to machine code



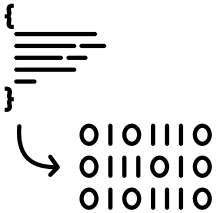


Assembler is a low level language,

- easily convertable to machine code

Relies on a small set of instructions

- Arithmetic/logical: `add, sub, and,...`
- Load/store/move: `ld, st, mov`
- Comparisons: `cmp`
- Jumps/conditional jumps: `jmp, je, jne,...`



Assembler is a low level language,

- easily convertable to machine code

Relies on a small set of instructions

- Arithmetic/logical: `add, sub, and,...`
- Load/store/move: `ld, st, mov`
- Comparisons: `cmp`
- Jumps/conditional jumps: `jmp, je, jne,...`

In contrast to high-level languages

- Very exact specification of what the CPU should do

```
instruction source, destination
```

- opposite to Intel syntax

`instruction source, destination`

- opposite to Intel syntax

Instructions are suffixed with

- `b`... byte, `w`... word, `l`... long word, or `q`... quad word
- Indicate the operand size

`instruction source, destination`

- opposite to Intel syntax

Instructions are suffixed with

- `b`... byte, `w`... word, `l`... long word, or `q`... quad word
- Indicate the operand size

Registers prefixed with `%`

- Thus, no confusion with C symbols

`instruction source, destination`

- opposite to Intel syntax

Instructions are suffixed with

- `b`... byte, `w`... word, `l`... long word, or `q`... quad word
- Indicate the operand size

Registers prefixed with %

- Thus, no confusion with C symbols

```
movl %ebx, %eax
```

Immediate operands prefixed with `$`

- `movl $5, %eax`

Immediate operands prefixed with `$`

- `movl $5, %eax`

Accessing in-memory content

- `movl foo, %eax` → contents of variable `foo` put into `eax`
- `movl $foo, %eax` → address of variable `foo` put into `eax`
- `movl (%rax), %eax` → dereference pointer in `rax` and put into `eax`

`SECTION:DISP(BASE, INDEX, SCALE)`

- Addresses calculated as $\texttt{BASE} + (\texttt{INDEX} \cdot \texttt{SCALE}) + \texttt{DISP}$

`movl $5, %eax:` Move the constant with value 5 to the EAX register.

`movl $5, (%eax):` Move the constant 5 to the address EAX points to.

`movl $5, 4(%eax):` Add 4 to the address in EAX and move the constant 5 to this location.

Including assembler snippets in a C program:

```
asm volatile("movl %ebx, %eax");
```

Including assembler snippets in a C program:

```
asm volatile("movl %ebx, %eax");
```

Accessing global C variables:

```
int cVar = 7;
asm volatile("movl cVar, %eax");
```

Including assembler snippets in a C program:

```
asm volatile("movl %ebx, %eax");
```

Accessing global C variables:

```
int cVar = 7;
asm volatile("movl cVar, %eax");
```

What happens if `eax` is already used for something else in your program?

- Define interface information for your assembler snippets!

Example - add two operands:

```
int sum, op1 = 5, op2 = 3;      // C variables
asm volatile("addl %2, %1\n\t"
             "movl %1, %0\n\t"
             : "=r" (sum)              // output ops
             : "r" (op1), "r" (op2) // input ops
             :                         // clobbered ops
); printf("result %d\n", sum);
```

Example - add two operands:

```
int sum, op1 = 5, op2 = 3;      // C variables
asm volatile("addl %2, %1\n\t"
             "movl %1, %0\n\t"
             : "=r" (sum)              // output ops
             : "r" (op1), "r" (op2)    // input ops
             :                         // clobbered ops
); printf("result %d\n", sum);
```

- "`"=r" (sum)`": the operand is written to `sum`

Example - add two operands:

```
int sum, op1 = 5, op2 = 3;      // C variables
asm volatile("addl %2, %1\n\t"
             "movl %1, %0\n\t"
             : "=r" (sum)              // output ops
             : "r" (op1), "r" (op2)    // input ops
             :                         // clobbered ops
); printf("result %d\n", sum);
```

- "`"=r" (sum)`": the operand is written to `sum`
- "`"r" (op1)`": a general register is used to store `op1`

Example - add two operands:

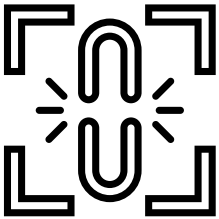
```
int sum, op1 = 5, op2 = 3;      // C variables
asm volatile("addl %2, %1\n\t"
             "movl %1, %0\n\t"
             : "=r" (sum)              // output ops
             : "r" (op1), "r" (op2)    // input ops
             :                         // clobbered ops
); printf("result %d\n", sum);
```

- "`"=r" (sum)`": the operand is written to `sum`
- "`"r" (op1)`": a general register is used to store `op1`
- Inputs and outputs accessed via `%0,%1,%2,...`

Example - add two operands:

```
int sum, op1 = 5, op2 = 3;      // C variables
asm volatile("addl %2, %1\n\t"
             "movl %1, %0\n\t"
             : "=r" (sum)              // output ops
             : "r" (op1), "r" (op2)    // input ops
             :                         // clobbered ops
); printf("result %d\n", sum);
```

- "`"=r" (sum)`": the operand is written to `sum`
- "`"r" (op1)`": a general register is used to store `op1`
- Inputs and outputs accessed via `%0,%1,%2,...`
- If registers are directly modified: prefix register with `%%` and add to clobber list (e.g., `%%eax`)

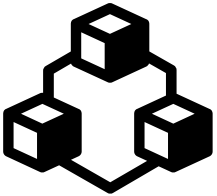


Constraints define whether an operand is:

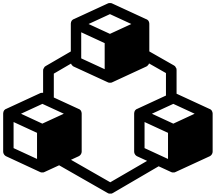
- in a register
- in which kind of register
- a memory reference
- which kind of address
- an immediate constant
- ...

$\rightarrow$ allow to write highly compact inline assembly

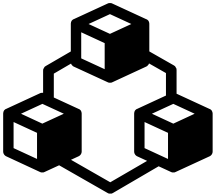
Kernel Modules



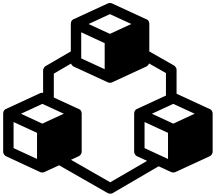
- Kernel module is a “normal” application



- Kernel module is a “normal” application
- Executed with higher privileges → access to everything

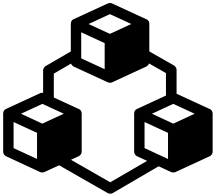


- Kernel module is a “normal” application
- Executed with higher privileges → access to everything
- Loaded into the operating system kernel

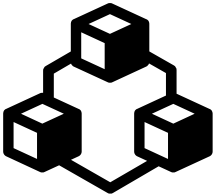


- Kernel module is a “normal” application
- Executed with higher privileges → access to everything
- Loaded into the operating system kernel
- Provides interface to user applications

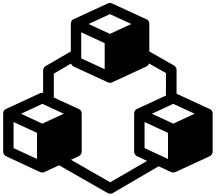

GOOD, GOOD
LET THE ROOT
PRIVILEGES FLOW THROUGH YOU
imgflip.com



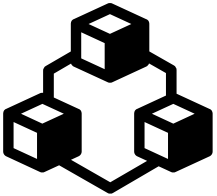
- On Unix, everything is a file



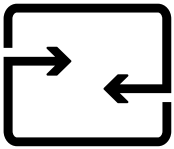
- On Unix, everything is a file
- Communication via pseudo files



- On Unix, everything is a file
- Communication via pseudo files
- E.g., files in `/proc`

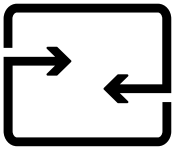


- On Unix, everything is a file
- Communication via pseudo files
- E.g., files in `/proc`
- Can be used like normal files

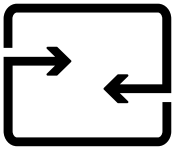


- Alternative to file operations: `ioctl`

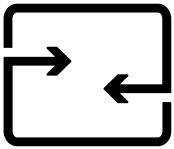
Communication using ioctl



- Alternative to file operations: `ioctl`
- Device-specific syscall



- Alternative to file operations: `ioctl`
- Device-specific syscall
- Consists of command identifier and arguments

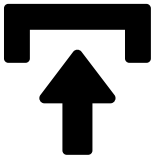


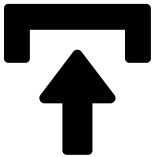
- Alternative to file operations: `ioctl`
- Device-specific syscall
- Consists of command identifier and arguments

ioctl

```
ioctl(device_fd, IOCTL_CMD_FOO, (size_t)&params);
```

- Modern CPUs: Kernel cannot directly access user data

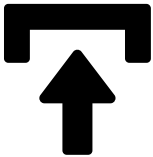




- Modern CPUs: Kernel cannot directly access user data
- Dedicated functions

Copy from user space

```
copy_from_user(void* to, const void __user* from, unsigned long n);
```



- Modern CPUs: Kernel cannot directly access user data
- Dedicated functions

Copy from user space

```
copy_from_user(void* to, const void __user* from, unsigned long n);
```

Copy to user space

```
copy_to_user(void __user* to, const void* from, unsigned long n);
```

- Kernel modules have the highest privileges

- Kernel modules have the highest privileges
- Can access everything

- Kernel modules have the highest privileges
- Can access everything
- No process isolation

- Kernel modules have the highest privileges
- Can access everything
- No process isolation
- Easy to corrupt data or crash computer

PROBLEM #36

"WITH GREAT POWER COMES GREAT RESPONSIBILITY"

Your Tasks

- Task 1 - Use the cpuid instruction to read information about the CPU

- Task 1 - Use the cpuid instruction to read information about the CPU
  - Processor brand string (`Intel(R) Core(TM) i7-4760HQ CPU @ 2.10GHz`)

- Task 1 - Use the cpuid instruction to read information about the CPU
  - Processor brand string (`Intel(R) Core(TM) i7-4760HQ CPU @ 2.10GHz`)
  - Feature support (FPU, TSC, PAE, VMX, FMA, TSC-Deadline, SGX, SHA)

Tasks

- Task 1 - Use the cpuid instruction to read information about the CPU
  - Processor brand string (`Intel(R) Core(TM) i7-4760HQ CPU @ 2.10GHz`)
  - Feature support (FPU, TSC, PAE, VMX, FMA, TSC-Deadline, SGX, SHA)
- Task 2a - Call a function in inline assembly - System V amd64 ABI (64-bit)

Tasks

- Task 1 - Use the cpuid instruction to read information about the CPU
  - Processor brand string (`Intel(R) Core(TM) i7-4760HQ CPU @ 2.10GHz`)
  - Feature support (FPU, TSC, PAE, VMX, FMA, TSC-Deadline, SGX, SHA)
- Task 2a - Call a function in inline assembly - System V amd64 ABI (64-bit)
- Task 2b - Implement a function executing a syscall in assembly (x86 64-bit)

- Task 1 - Use the cpuid instruction to read information about the CPU
  - Processor brand string (`Intel(R) Core(TM) i7-4760HQ CPU @ 2.10GHz`)
  - Feature support (FPU, TSC, PAE, VMX, FMA, TSC-Deadline, SGX, SHA)
- Task 2a - Call a function in inline assembly - System V amd64 ABI (64-bit)
- Task 2b - Implement a function executing a syscall in assembly (x86 64-bit)

→ find more details on the assignment page


Sandeep S.
GCC-Inline-Assembly-HOWTO.
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html


Miyagi R.
Introduction to GCC Inline ASM.
http://asm.sourceforge.net/articles/rmiyagi-inline-asm.txt


GCC
How to Use Inline Assembly Language in C Code.
https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html

Constraints

Constraints for asm Operands.

https://gcc.gnu.org/onlinedocs/gcc/Constraints.html

Ruan de Bruyn

How to write your first Linux Kernel Module.

https://medium.com/dvt-engineering/how-to-write-your-first-linux-kernel-module-cf284408beeb

Questions?

Appendix

- Several general purpose registers: RAX, RBX, RCX, RDX
- Stack registers: RSP, RBP
- String index registers: RSI, RDI
- Instruction pointer: RIP
- Flags register: EFLAGS
- Segment registers: CS, DS, SS, ES, FS, GS
- Control registers: CR0, CR1, CR2, CR3, CR4
- Floating point and MMX registers: ST0,...