ECE 471 – Embedded Systems
Lecture 7

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

18 September 2026

Announcements

- HW#1 grades will be sent out soon
- HW#2 was due
- HW#3 will be posted soon
  - Got sidetracked trying to make the HW better for people with 64-bit pi3/4/5 (which I think is everyone)
  - Turns out the armhf cross compiler on arm64/debian is broken
  - Also turns out pi5 due to its 16k page size has trouble running 32-bit programs

Debugging – when things go wrong

- Use a debugger like gdb
  - Compile your code with `-g` for debug symbols
  - Run `gdb ./hello`
  - `bt` backtrace, `info regis` gives register, `disassem` disassembles, etc.
- Sprinkle printf calls

Coding Style

- How should you format your code?
- Does C have rules? Not really.
- International Obfuscated C Code Competition (IOCCC)
  `https://www.ioccc.org/`
- Your company or open-source project might have strict rules
- In this class as long as your code is relatively easy to follow I am fine with it

Coding Style – Tabs vs Spaces

- Indent code with a tab character?
  Or 8 spaces (traditional size of a tab)? Or some other amount of spaces?
- How long should lines be? Traditionally was 80 columns (historical size of screens)
- Other spacing, like `if ( x == 5 )` how many of those spaces should be there?

Coding Style – Curly Braces

- `int function() {`?
- Or should it be next line?
- Should int be on its own line too?

Coding Style – Variable Names

- Function Naming Styles
  - `count_active_users()`
  - `CountActiveUsers()` (camel-case)
- Variable Naming Styles
  - `int i;`
  - `int IndexForTheFirstForLoop;`
  - `u32iLoopIndex` (Hungarian notation, include type info in name)

indent tool

- The `indent` program can reformat your code to match the "proper" style for a project

Coding Style – Linux kernel stuff

- Use of typedefs to make types shorter? `vpt_a` vs `struct virtual_pointer *a`
- Having only one exit to a function (using goto)
- Restricting the size functions can get

How Executables are Made

- Compiler generates ASM (Cross-compiler)
- Assembler generates machine language objects
- Linker creates Executable (out of objects)

Tools – Compiler

- takes code, usually (but not always) generates assembly
- Compiler can have front-end which generates intermediate language, which is then optimized, and back-end generates assembly
- Can be quite complex
- Examples: gcc, clang
- What language is a compiler written in? Who wrote the first one?
  Thompson's *Reflections on Trusting Trust*

Tools – Assembler

- Takes assembly language and generates machine language
- creates object files
- Relatively easy to write (mostly string parsing and bit-manipulation)
- Examples: GNU Assembler (gas), tasm, nasm, masm, etc.

Tools – Linker

- Creates executable files from object files
- Resolves addresses of symbols.
- Links to symbols in libraries.
- Examples: ld, gold (hard to write)
- “Linker script” describes how to lay out executable

ELF Executable Format

- Binary contains your code
- Also contains initialized data
- Also a bunch of headers to tell the OS how to run things
- We'll discuss this more later

Application Binary Interface (ABI)

- The rules an executable needs to follow in order to talk to other code/libraries on the system
- A software agreement, this is not enforced at all by hardware

ARM32 Linux C/userspace ABI

- r0-r3 are first 4 arguments/scratch (extra go on stack) (caller saved)
- r0-r1 are return value
- r4-r11 are general purpose, callee saved
- r12-r15 are special (stack, LR, PC)
- Things are more complex than this. Passing arrays and structs? 64-bit values? Floating point values? etc.

Kernel Programming ABIs

- OABI – "old" original ABI (arm). Being phased out. slightly different syscall mechanism, different alignment restrictions
- EABI – new "embedded" ABI (armel)
- hard float – EABI compiled with ARMv7 and VFP (vector floating point) support (armhf). Raspberry Pi (raspbian) is compiled for ARMv6 armhf.

Linux System Calls (EABI/armhf)

- System call number in r7
- Arguments in r0 - r6
- Return value in r0 (-1 if error, errno in -4096 - 0)
- Call `swi 0x0`
- System call numbers can be found in `/usr/include/arm-linux-gnueabihf/asm/unistd.h`
  They are similar to the 32-bit x86 ones.

How was OABI different

- The previous implementation had the same system call numbers, but instead of r7 the number was the argument to `swi`.
- This was very slow, as there is no way to determine that value without having the kernel backtrace the callstack and disassemble the instruction.

Manpage

The easiest place to get system call documentation.

`man open 2`

Finds the documentation for "open". The 2 means look for system call documentation (which is type 2).

ARM ISAs

- ARM32
- Thumb
- Thumb2 (as seen on ECE271 ARM Cortex-M)
- AARCH64

A first ARM32 assembly program: `hello_exit`

```
.equ SYSCALL_EXIT,      1

        .globl _start
_start:

        #=================================
        # Exit
        #=================================
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT        @ put exit
        swi     0x0                     @ and exit
```

Some GNU assembler notes

- Code comments
  - @ is the traditional comment character
  - # can be used on line by itself but will confuse assembler if on line with code.
  - Can also use /* */ and //
  - *Cannot* use ;
- Instruction opcode operand order is destination, source
- Constant value indicated by # or $
- .equ is equivalent to a C #define

`hello_exit` example

Assembling/Linking using `make`, running, and checking the output.

```
lecture8$ make hello_exit_arm
as -o hello_exit_arm.o hello_exit_arm.s
ld -o hello_exit_arm hello_exit_arm.o
lecture8$ ./hello_exit_arm
lecture8$ echo $?
5
```

Let's look at our executable

- `ls -la ./hello_exit_arm`
  Check the size
- `strip ./hello_exit_arm`
  Strip off debugging information (makes smaller)
- `hexdump -C ./hello_exit_arm`
  See the raw binary (well, hex) values
- `readelf -a ./hello_exit_arm`
  Look at the ELF executable layout
- `objdump --disassemble-all ./hello_exit_arm`
  See the machine code we generated
- `strace ./hello_exit_arm`
  Trace the system calls as they happen.

64-bit `hello_exit` example

```
.equ SYSCALL_EXIT,      93

        .globl _start
exit:
        mov     x0,#5                   // put retun va
        mov     x8,#SYSCALL_EXIT        // put syscall
        svc     0x0                     // and exit
```

hello_world example

```
.equ SYSCALL_EXIT,      1
.equ SYSCALL_WRITE,     4
.equ STDOUT,            1

        .globl _start
_start:
        mov     r0,#STDOUT              /* stdout *
        ldr     r1,=hello
        mov     r2,#13                  @ length
        mov     r7,#SYSCALL_WRITE
        swi     0x0

        # Exit
exit:
        mov     r0,#5
```

New things to note in `hello_world`

- The fixed-length 32-bit ARM cannot hold a full 32-bit immediate
- Therefore a 32-bit address cannot be loaded in a single instruction
- In this case the "=" is used to request the address be stored in a "literal" pool which can be reached by PC-offset, with an extra layer of indirection.
- Data can be declared with .ascii, .word, .byte
- BSS can be declared with .lcomm

Using gdb with `hello_world`

- Run `gdb ./hello_world`
- Type *run* to run program, will exit normally
- Can set breakpoint `break exit`
- Can single-step
- Can *info regis* to see registers
- Cam *disassem* to see disassembly

simple loop example

```
        # for(i=0;i<10;i++) do_something();

        mov         r0,#0                   # set loop c
loop:
        push         {r0}                   # save r0 or
        bl         do_something           # branch to su
                                        # return address in
        pop          {r0}                  # restore r0

        add         r0,r0,#1            # increment loop
        cmp         r0,#10                  # have we r
        bne         loop                  # if not, loo
```

string count example

Count the number of chars in a string until we hit a space.

```
        ldr         r1,=hello        # load pointer t
        mov         r2,#0               # initialize
loop:
        ldrb         r0,[r1]                # load by
        cmp         r0,#' '               # compare
                                  # this updates the s
        beq         done                 # if it was e
        add         r2,r2,#1          # increment our c
        add         r1,r1,#1          # increment our p
        b         loop                  # branch (uncon
done:
```