# ECE 471 – Embedded Systems Lecture 8

Vince Weaver

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

vincent.weaver@maine.edu

21 September 2018

## Announcements

- HW#2 was due
- HW#3 will be posted today. Work in groups?
- Note the sample code for this lecture will be posted to the website.

# Low-Level ARM Linux Assembly

## Linux C (ABI)

- Application Binary Interface
- 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
- 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
- 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.

## 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.

## System Calls (OABI)

- 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).

## A first ARM assembly program: `hello_exit`

```
.equ SYSCALL_EXIT,     1

        .globl _start
_start:

        #================================
        # Exit
        #================================
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT          @ put exit syscall number (1) in r7
        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 ;
- Order is source, destination
- Constant value indicated by # or $
- .equ is equivalent to a C #define

## `hello_exit` example

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

```
lecture6$ make hello_exit_arm
as -o hello_exit_arm.o hello_exit_arm.s
ld -o hello_exit_arm hello_exit_arm.o
lecture6$ ./hello_exit_arm
lecture6$ echo $?
5
```

## Let's look at our executable

- `ls -la ./hello_exit_arm`
  Check the size
- `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.

## 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
        mov     r7,#SYSCALL_EXIT        @ put exit syscall number in r7
        swi     0x0                     @ and exit

.data
hello:          .ascii "Hello␣World!\n"
```

## 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

## string count example

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

```
        mov     r1,=hello       # load pointer to hello string into r1
        mov     r2,#0           # initialize count to zero
loop:
        ldrb    r0,[r1]         # load byte pointed by r1 into r0
        cmp     r0,#'␣'         # compare r0 to space character
                                # this updates the status flags
        beq     done            # if it was equal, we are done
        add     r2,r2,#1        # increment our count
        add     r1,r1,#1        # increment our pointer
        b       loop            # branch (unconditionally) to loop
done:
```

## simple loop example

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

        mov     r0,#0           # set loop index to zero
loop:
        push    {r0}            # save r0 on stack
        bl      do_something    # branch to subroutine, saving
                                # return address in link register
        pop     {r0}            # restore r0 from stack

        add     r0,r0,#1        # increment loop counter
        cmp     r0,#10          # have we reached 10 yet?
        bne     loop            # if not, loop
```