# ECE 471 – Embedded Systems
# Lecture 7

Vince Weaver

http://www.eece.maine.edu/∼vweaver

vincent.weaver@maine.edu

23 September 2014

## Announcements

- Does everyone have access to a breadboard and jumper wires?
- How about LED, switch, and suitable resistors?
- Just taking stock of what's needed for HW#4.

## HW2 Review

- Everyone seems to be accessing the Pi OK.
- More people using network to copy than I thought.
- Be sure to follow directions! Was lax, but if it says to print ECE471 and a number, be sure you do it!
- more info on ls. Looking for man. “info” or `ls --help`
- ls -a shows hidden files. Hidden files on UNIX
- BSS. Uninitialized or zeroed. Saves space in executable.

## HW3 Notes

- Mention `strace` to see the syscalls
- Can disassemble code with `objdump --disassemble-all`
- gdb debugger
  - `gdb ./hello_world.c`
  - `run` – to run program
  - `bt` – show backtrace
  - `disassem` – disassemble
  - `info regis` – show register values
  - More advanced features like single-step, breakpoint,

etc. also available.

## 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)

- System call number in r7
- Arguments in r0 - r6
- 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.

## Low-level System Calls

```
int fd,result;
char buffer[1024];

fd=open("hello_world.c",O_RDONLY);
if (fd<0) printf("Error␣%s!\n",strerror(errno));

result=read(fd,buffer,1024);
if (result<0) printf("Error␣%s!\n",strerror(errno));
printf("Read␣%d␣bytes\n",result);

result=write(fd,buffer,1024);
if (result<0) printf("Error␣%s!\n",strerror(errno));
printf("Wrote␣%d␣bytes\n",result);

result=close(fd);
```

## File Descriptors

Your process starts with 3 already configured

- 0 – stdin – Standard Input
- 1 – stdout – Standard Output
- 2 – stderr – Standard Error

## Redirecting I/O

You can redirect these at the shell:

- `ls > ls_output` – pipe to file
- `sort < ls_output` – pipe from file
- `hello_world 2> errors` – stderr
- `ls | sort | wc` – pipelines

## 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).

## Assembly

- @ is the comment character. # can be used on line by itself but will confuse assembler if on line with code. Can also use /* */
- Order is source, destination
- Constant value indicated by # or $
- Optionally put % in front of register name

## 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.

## Coding Directly for the Hardware

One way of developing embedded systems is coding to the raw hardware, as you did with the STM Discovery Boards in ECE271.

- Compile code
- Prepare for upload (hexbin?)
- Upload into FLASH
- Boots to offset

- Setup, flat memory (usually), stack at top, code near bottom, IRQ vectors

- Handle Interrupts

- Must do I/O directly (no drivers)
  Although if lucky, can find existing code.

- **Code is specific to the hardware you are on**

## Instead, one can use an Operating System

## Why Use an Operating System?

- Provides Layers of Abstraction
  - Abstract hardware: hide hardware differences. same hardware interface for classes of hardware (things like video cameras, disks, keyboards, etc) despite differing implementation details
  - Abstract software: with VM get linear address space, same system calls on all systems
  - Abstraction comes at a cost. Higher overhead, unknown timing

- Multi-tasking / Multi-user
- Security, permissions (Linus dial out onto /dev/hda)
- Common code in kernel and libraries, no need to re-invent

## What's included with an OS

- kernel / drivers – Linux definition
- also system libraries – Solaris definition
- low-level utils / software / GUI – Windows definition
  Web Browser included?
- Linux usually makes distinction between the OS Kernel and distribution. OSX/Windows usually doesn't.