ECE 531 – Advanced Operating Systems Lecture 9

Vince Weaver

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

vincent.weaver@maine.edu

22 September 2025

Announcements

- Homework #3 was assigned, due Friday

Serial Port Programming Aside

- Programming serial port at hardware level not so bad
- Programming from inside of Linux a huge pain
  - Linux inherited old UNIX code for dealing with ttys
  - ioctl/syscall interface to this is a pain, old legacy
  - Designed to be overly flexible
  - Just doing simple input/output to console runs into this because it assumes every text console is a tty
  - `scanf()`, canonical mode (line-at-time until enter)

```
/* example code to access serial port from Linux */
struct termios tty;
```

```
int serial_fd;
serial_fd=open("/dev/ttyUSB0",O_RDWR);
tcgetattr(serial_fd,&tty);  // get current settings
tty.c_cflag &= ~PARENB; // disable partiy
tty.c_cflag &= ~CSTOPB; // one stop bit
tty.c_cflag &= ~CSIZE;  // clear data size bits
tty.c_cflag |= CS8;     // set 8 bits per byte
tty.c_cflag &= ~CRTSCTS;// Disable hardware flow control
tty.c_cflag |= CREAD|CLOCAL; // Turn on READ and ignore control

tty.c_lflag &= ~ICANON;  // disable canonical mode
tty.c_lflag &= ~ECHO;    // disable echo
tty.c_lflag &= ~ECHOE;   // disable erasure
tty.c_lflag &= ~ECHONL;  // disable new-line echo
tty.c_lflag &= ~ISIG;    // disable signals INTR, QUIT and SUSP
tty.c_iflag &= ~(IXON | IXOFF | IXANY); // turn off s/w flow ctrl
tty.c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP|INLCR|IGNCR|ICRNL);
                         // disable special handling of received bytes

tty.c_oflag &= ~OPOST; // prevent special handling of recvd bytes
tty.c_oflag &= ~ONLCR; // Prevent newline to CR/LF conversion

tty.c_cc[VTIME] = 10;    // wait up to 1s for data, return when any comes in
tty.c_cc[VMIN] = 0;
```

```
cfsetispeed(&tty, B9600); // set input speed to 9600
cfsetospeed(&tty, B9600); // set output speed to 9600

tcsetattr(serial_port, TCSANOW, &tty); // save settings

// left out all error checking

// at this point should be able to read()/write()
```

Terminal Programs

- To communicate over a serial port you need a program that talks to the port and send/receives bytes
- `learn.adafruit.com/adafruits-raspberry-pi-lesson-5-using-a-console-cable/test-and-configure`
- putty is a decent one for Windows
- I use minicom for Linux. A bit of a pain. Not installed by default. Control-A Control-Z for help. Has similar keybindings to old DOS program Telix
- "screen" on MacOS and also Linux

  ```
  sudo screen /dev/ttyUSB0 115200
  ```

File Transfer

- Need special protocols to send binary data, especially if using software flow control and need to escape some characters
- On remote connection (inside terminal) start a receiving program that listens, gets the filename and contents, and writes out
- On local end you tell your terminal program to send the data
- Common methods

- Plain ASCII
- Kermit
- Zmodem/Xmodem/Ymodem/etc
- On Linux use sz / rz to send things via Zmodem

USB Serial Converters

- Modern machines often don't have serial ports
- Instead you can use USB to Serial converters
- PL2303 / FTDI chips used in these
- Often counterfeited, in the news for that recently (and how the companies tried to kill the counterfeits)
- Takes serial in, presents as a serial port to the OS. ttyUSB0 on Linux, COM something really high on windows, /dev/cu.usbserial on MacOS
- I've tried to give everyone the same type of adapter this

year, the “Silicon Labs” type (not prolific)

HW#3: Writing header files

- Including with “ ” versus $<>$

HW#3: Writing printk

```c
int printk(char *string,...) {

        va_list ap;
        va_start(ap, string);

        while(1) {
                if (*string==0) break;

                if (*string=='%') {
                        string++;
                        if (*string=='d') {
                                string++;
                                x=va_arg(ap, int);
```

Integer to String Conversion

This it the algorithm I use, there are other ways to do it that don't involve the backwards step (starting off by dividing by 1 billion and dividing the divisor by 10 each time).

- Repeatedly divide by 10.
- Digit is the remainder. Repeat until quotient 0.
- Make sure handle 0 case.
- Convert each digit to ASCII by adding 48 ('0')
- Why does the number end up backwards?

HW#3 Division by 10

- ARM1176 in Pi has no divide routine! (ARMv7/v8 does)
- Generic x=y/z division is not possible without fancy work (iterative subtraction? Newton approximation?)
- Dividing by a constant is easier

HW#3 Multiply by Reciprocal

- C compiler cheats, for /10 it effectively multiplies by 1/10.
- Look at generated assembly, you'll see it multiply by `0x66666667` (signed) or `0xcccccccd` (unsigned)

HW#3 How Does Compiler Handle Division on ARM1176

- The C compiler will generate a call to the C-library or gcc-library divide routine
- This is a problem on our own OS as we have no libraries
- If on ARM 1176 you try to use division, C compiler will try to call something like `__aeabi_uidiv()` which you have to provide.
- We can write it, either some fancy assembly, or just iterative subtraction

HW#3 Printing Hexademical Instead

- Each digit is power of 16, not 10.
  Why is it not a problem when dividing by 16?
- Need to handle case where digits above 9, make A-F

Knowing when Hardware has new Data

- There are two ways to do this
  - Polling – periodically check the hardware
    difficult as you have to manually check all the time in your code and things might get lost if you are busy
  - Interrupts – the hardware sends a signal to the CPU saying it needs attention

Are interrupts good or bad?

- Can reduce latency... or make it worse (real-time, slow handler)
- Can add overhead. On OoO need to flush entire pipeline, then enter kernel. Slow slow slow.

Can You Avoid Interrupt Overhead?

- Some HPC or virtual turn off interrupts if possible.
- Linux NAPI Interrupt Mitigation will turn off IRQs and use polling instead when system under heavy load
- Linux tickless system will turn off regular timers to save power/overhead by setting a timer to wake later (allowing deeper sleep) [this is particularly useful on systems with lots of VMs]
- Linux top-half/bottom-half: top-half receives interrupt, handles it, does bare minimum work. Bottom-half runs

later when there's more time.

Other Interrupt Topics

- Precise interrupts – on modern out-of-order processors it can be extremely difficult to know exactly what instruction was running when an IRQ came in.
  Does this matter? Mostly just for perf optimization

What generates interrupts?

- What types of hardware generate interrupts? Keyboard, timers, Network, Disk I/O, serial etc.
- Some can be critical. Not empty UART FIFO fast enough can drop data on floor.
- What is most frequent interrupt on typical OS? Timer interrupt. regular timer. What is used for?
  - Context switching
  - Timekeeping, time accounting

Typical Interrupts

- Tell pointless 6502/Mockingboard example
- Set up interrupt source (Timer at 50Hz?)
- Install interrupt handler (usually vector at address that jumps to your code to handle things)
  - Handler should be fast, do whatever it needs to do (my case, load up 14 registers with data) or even schedule more work than later
  - Disable interrupts if HW didn't for us. Save/restore any registers we're going to change so when we return

no one notices

  - Handler should ACK the interrupt (let hardware know we handled things so it doesn't retrigger as soon as we exit)
- Enable interrupts (often need to do this two ways)
  - On device (often a flag to set)
  - Enable (unmask) interrupts on your CPU. Often a processor flag.

Exceptions and Interrupts

- All architectures are different
- ARM does it a little differently from others.
- Note ARM32 on Cortex-A (this class) can be different than Cortex-M (like the STM32 boards in 271)
- Possibly also different in ARM64

How to find out?

- ARM ARM for ARMv7 (2700+ pages)
- Look at Linux source code
- Look at Raspberry Pi Forums
- Note Pi4 has extra gic-400 interrupt controller you need to enable