ECE 531 – Advanced Operating Systems
Lecture 12

Vince Weaver

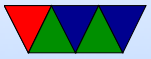
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

29 September 2025

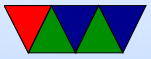
Announcements

- Homework #4 will be posted (sorry for the delay)



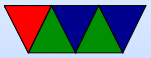
Interrupt Roundup

Any questions on interrupts?



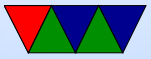
Timer IRQs Critical

- I think I mentioned previously, but it's extremely difficult to make a good multi-tasking OS on hardware without a timer interrupt



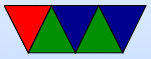
Interrupts On Pi-1B

- TODO: verify this info and make diagrams
- On Original Pi1, one interrupt controller. Peripheral interrupts fed into interrupt controller, which generated the IRQ/FIQ lines to the single CPU



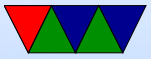
Interrupts On Pi-2/3

- TODO: digram
- On these machines multicore (4 cores)
- Which CPU gets an interrupt? Only core0? All of them? Rotate between them? There are reasons to do any of those.
- Each core has own "local" interrupt controller?
- Each core has own source of interrupts (PMU, timer) that get fed into controller along with peripheral interrupts, which then generate IRQ/FIQ?



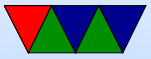
Interrupts On Pi-4

- On Pi4 same as Pi2/Pi3 but a GIC-400 global interrupt controller is added.
- The peripheral interrupts feed into both the legacy and GIC-400.
- Things like ARM timer can be handled either by legacy (have to enable that input on GIC-400) or also handled directly by the IRQ feeding into GIC
- Extremely complex to get working right



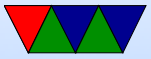
Interrupts On Pi-5

- I think it has all changed again, I have not had much time to investigate



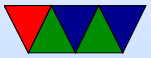
Timers On Raspberry Pis

- ARM sp084 timer
  - Available all models BCM2835 Chapter 14, BCM2711 Chapter 12
  - Based on 250MHz bus? Different on some models?
  - Simple countdown with auto-reload timer, and interrupts
  - The timer's location in the interrupt pending map changed in the Pi4
- System Timer

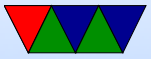


- Available all models? BCM2835 Chapter 12, BCM2711 Chapter 10
- Single free-running 64-bit counter
- 4 32-bit compare registers that can generate interrupt when matches the bottom 64 bits
- Based on which clock source?

- Local Timer
  - Only pi2 and newer?
    BCM2711 Chapter 6, BCM2836
  - per-core?
  - which clock?

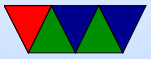


- Scaled on Cortex A7?
- Note, access to these is not via the peripheral area but in the special local cpu area



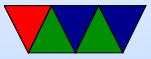
Interrupts Schemes

- More info on nested interrupts
- More info on interrupt priority
- Non-Maskable Interrupts



Interrupts on Linux

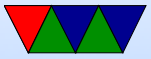
- Can look in `/proc/interrupts`
- Latency matters. Old days had problems where you'd lose serial interrupts (small FIFOs) if your disk drive took too long, etc.
- Cannot do anything that might block in an interrupt. Can you do I/O? Can you do a printk? (re-entrancy)
- Top Half / Bottom Half
  Have interrupt routine be bare minimum short. ACK



interrupt, handle super pressing thing (copy data out of FIFO) Then tell the kernel to handle the rest later.

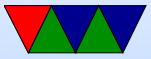
So you might have a tasklet/kernel thread that runs occasionally (and is fully interruptible) that will do the rest.

For example, network packet comes in, important to read the packet and ACK interrupt. Put it in queue, then later the code that does longer latency stuff (decodes packet, does ethernet or TCP/IP stuff, then finally copies the data to the code waiting)

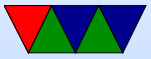


User / Kernel Separation

- Why use userspace (why not all in kernel like DOS?)
  Provides stability/protection/security
  Can be slower to access hardware, but more protection from crashing
- Can't access all of CPSR register
  Can't turn off interrupts (why?)
  Can't switch to privileged modes via CPSR writes
- If virtual memory enabled, can't access protected/kernel memory



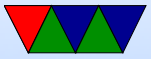
- Can you still access MMIO?



Entering User Mode

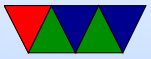
Generally done once at boot

```
mov r0, #0x10    // set up user bits for CPSR
msr SPSR, r0     // put in the saved status register
ldr lr, =first   // point link register to entry point of our user code
movs pc, lr // switch modes
```



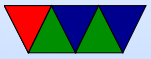
System Calls

- If we are running in user mode, how can we get back into the kernel?
- Interrupts!
  - Timer interrupt is often used to periodically switch to the kernel and it can then do any accumulated tasks.
- System calls! (software interrupt)
  - How can we manually call into the kernel when we need to?



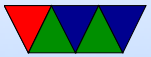
ARM32 System Calls

- On ARM a SWI instruction (sometimes is shown as a SVC instruction) causes a software interrupt.
- This calls into the kernel SWI Interrupt handler (which we will have to set up)
- Based on the state of the registers at the time of the SWI, the kernel will do something useful.



Debugging Linux System Calls

- Can watch system calls with `strace` command on Linux



Linux ARM32 System Call Interface

- EABI: Arguments in r0 through r6. System call number in r7.

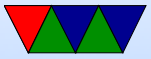
  `swi 0`

  Return value in r0

- OABI: Arguments in r0 through r6.

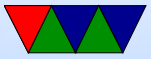
  `swi SYSBASE+SYSCALLNUM`

  Why bad? No way to get swi value except parsing back in instruction stream.



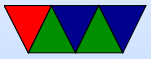
SWI Interrupt Handler

```
uint32_t __attribute__((interrupt("SVC"))) swi_handler(
        uint32_t r0, uint32_t r1, uint32_t r2, uint32_t r3) {
        register long r7 asm ("r7");

        printk("Syscall %d\n",r7);

    /* Copy result into place of r0 on return stack */
    asm volatile("str %[result],[sp,#0]\n"
                 :       /* output */
                 :       [result] "r" (result) /* input */
                 :);     /* clobber */

    return result;
}
```



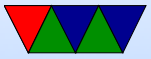
Linux System Call Results

- Result is a single value (plus contents of structures pointed to)
- How can you indicate error?
- On Linux, values between -4096 and -1 are treated as errors. Usually -1 is returned and the negative value is made positive and stuck in `errno`.
- What are the limitations of this? (what if -4000 is a valid return?)



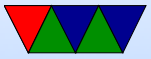
syscalls on non-ARM systems

- It's up to the OS and architecture
- x86 it's `int 0x80` on 32-bit and `syscall` on 64-bit
- Some OSes pass parameters on stack, Linux it's usually in registers for speed.



Advanced Syscall methods

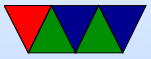
- Linux: vsyscalls, vdso, io_uring
- We'll discuss these later



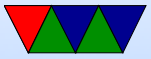
Calling a Syscall from Userspace

Generally the C library does this and hides the assembly from you.

```
static inline uint32_t syscall3(int arg0, int arg1, int arg2, int which) {

        uint32_t result;

        asm volatile (  "mov␣r0,␣%[arg0]\n"
                        "mov␣r1,␣%[arg1]\n"
                        "mov␣r2,␣%[arg2]\n"
                        "mov␣r7,␣%[which]\n"
                        "swi␣0\n"
                        "mov␣%[result],␣r0\n"
                :       [result] "=r" (result)
                :       [arg0] "r" (arg0),
                        [arg1] "r" (arg1),
                        [arg2] "r" (arg2),
                        [which] "r" (which)
                :       "r0", "r1", "r2", "r7" );
```



```
        return result;
}
```



Syscall Correctness/Security

- If a syscall asks the kernel to write a result to the pointer, should the kernel trust userspace to pass in a valid pointer?
- On a real system the kernel will check and only write if the address belongs to the program (otherwise you could trick the kernel into writing over arbitrary memory)
- It might make more sense to have a copy_to_user() and copy_from_user() that handles this rather than just de-referencing pointers passed in via syscalls