

ECE 531/598 – Advanced Operating Systems Lecture 9

Vince Weaver

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

`vincent.weaver@maine.edu`

26 September 2023

Announcements

- Homework #4 was posted
- Issues getting homework posted, mostly due to getting Raspberry Pi4 Interrupt support working
- Do we really need timer interrupts?
Actually hard to make a single-core multitasking OS without a regular timer interrupt. Why it's hard to make an OS on the Apple II (some people made cards for just this purpose)



More notes for HW#4



Accessing MMIO registers

- For the BCM2835 peripherals, use `bcm2835_read()` and `bcm2835_write()` to do MMIO access
These adjust for the differing location of the MMIO regions on different Pis
- Don't use the `mmio_read()/write()` routines unless you have to (I renamed them to make this hard to do accidentally). The only code that uses them is the new gic-400 irq controller code on Pi4 because it lives in a different mmio region separate from the peripheral IO



Changes for HW#4

- In case you want to re-use your HW#3 code
- Autodetects Pi model
 - sets up io_base for you
 - sets up various pi4 things if it sees you have one
 - This required adding a lot of code, including device tree support
- `uart_write()` will automatically insert the carriage return (`\r`) for you if it sees a linefeed



HW#4 – Device Tree

- The “new” way of providing hardware info to the kernel for an ARM machine
- Replaces ATAGS
- Microsoft is pushing ACPI support instead :(
- Parser isn't too horrible, it's mostly a bunch of key/value pairs
- There's a long complex spec, it's based on powerpc stuff from a while ago
- For now we just grab the device type but in future we



could do things like actually grab the io_base directly from it



HW#4 – include and quotes

- What is the difference between `#include <string.h>` and `#include "string.h"`
- The first looks at the system includes
- The second looks in your local directory (or what you specify with `-I` on the command line)



HW#4 – String manipulation

- Most C-based OSes quickly obtain string manipulation functions
- `strncmp()`, `strlen()`, `strncpy()`, `memcpy()`, `memset()`, `memcopy()`
- What's the different between `strncpy` and `memcpy`?
- How optimized do these routines need to be?
- `memcpy()` is often short blast of C



```
for(i=0;i<n;i++) { *d=*s; d++; s++;}
```

but it can be optimized to death.

- memcpu() / memmove() difference? Why it's there, hazard when you don't use it right? (memmove the areas can overlap) (what happens if you copy backwards)



HW#4 – Writing a shell

- What is a shell, or monitor routine?
- How can you parse a command line?
- Read values into a buffer. When enter pressed, check for a command. `strcmp()`? By hand? `strtok()` if fancy?
- Do whatever the command indicates, then reset buffer pointer.
- Print an error if unknown command.



HW#4 – LED routines

- I added LED routines in led.c along with gpio.c
- This abstracts the code away, so it should work on any kind of Pi transparently (though very slightly slower than direct coding it)
- Good for you, but also makes grading easier for me.



Interrupt Roundup

Any questions on interrupts?



Interrupts On Raspberry Pis

- 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
- On Pi2/Pi3 multicore. 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?



- 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



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?

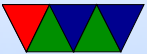


- Scaled on Cortex A7?
- Note, access to these is not via the peripheral area but in the special local cpu area



Interrupts Schemes

- More info on nested interrupts
- More info on interrupt priority



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`?
- Top Half / Bottom Half
Have interrupt routine be bare minimum short. ACK

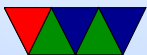


interrupt, handle super pressing thing (copy data out of FIFO) Then tell the kernel to handle the rest later.

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)

- Timer interrupt. How often? 100Hz originally. Up to



1000Hz (why?) now configurable, often at 250Hz.



Userspace

- Why use userspace (why not everything in kernel like DOS?)
Slower, but has some protections from bad programs/security
- Can't access all of CPSR register
Can't turn off interrupts
Can't switch to privileged modes
- If virtual memory enabled, can't access protected/kernel



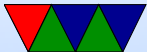
memory

- Can you still access MMIO?



Entering User Mode

```
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.
- How can we manually call into the kernel when we need to?
- System calls!
- Can watch system calls with `strace` command on Linux



ARM 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.



Linux ARM 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.



Application Binary Interface

What is an ABI and why is it necessary?



Linux GNU EABI

- Procedure Call Standard for the ARM architecture
- ABI, agreed on way to interface with system.
- Arguments to registers. r0 through r3.
- Return value in r0.
- How to return float, double, pointers, 64-bit values?
(There's a new ABI on ARM, hf (hard floating point) that's mostly about how to pass floating point values around)
- How to pass the above?



- What if more than 4 arguments? (stack)
- Is there a stack, how aligned?
- Structs, bitfields, endianness?
- Callee vs Caller saved registers? (A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP)
- Frame Pointer?



ABI Purpose

- An ABI is used so that code written by different groups knows how to communicate (code to c-library, c-library to kernel, etc)
- If you are writing your own OS from scratch can write own ABI, but then not compatible with existing code
- Writing in assembly you can ignore the ABI for speed, but only if you do not call out to anyone else's code



Calling a Syscall

```
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;  
  
}
```

