ECE 531 – Advanced Operating Systems
Lecture 11

Vince Weaver

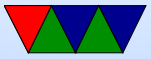
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

26 September 2025

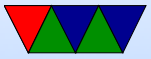
Announcements

- Homework #3 Due Today
- Homework #4 will be posted



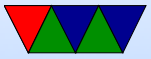
Last minute HW#3 Hints

- If get blank screen in terminal, try typing some stuff or hitting enter a few times, possibly the boot messages printed before your USB serial port came up



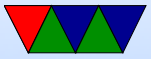
HW#2 Code Graded

- Mostly fine, but many people had timings for the blink that was way off of 1Hz
- Note you'll need different constants for ASM and C. Look at the code, the C code generally is doing more instructions which will take longer



Timer interrupt

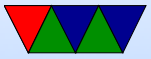
- Most OSes have some sort of internal timer keeping things going
- Tracks time of day, triggers scheduler (for context switching), uptime, time accounting
- Ideally triggered as a regular interrupt



Timer interrupt – Linux

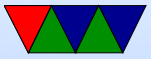
- Traditionally 100Hz, these days 250Hz?
  - Too slow and the delay in context switching is noticeable
  - Too fast and the overhead from each interrupt adds up (have to stop CPU, save/restore state, etc)
- Interrupts can waste power, especially if machine is mostly idle/sleeping

  Linux these days has ways to run without timer-ticks
- Trivia, timer tick on Linux called a jiffie



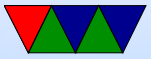
Timer interrupt – Other Operating Systems

- DOS = 18.2Hz
- Windows = 64Hz

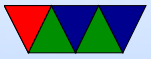


OS Timekeeping

- Hardware will often have a battery-backed RTC (real time clock) that tracks time/date, often this is only read once at boot
- The OS generally has the job of tracking time
- Often it's just a second timer counting from an "epoch" (start date). On 100Hz system, every 100 interrupts increment the seconds count.
- There are routines that convert this to the date/time. Date/time code is a huge mess a bit beyond this class

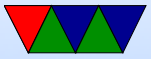


(handle things like leap years, leap seconds, timezones, calendar changes, etc)

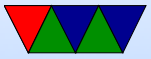


Linux 2038 Problem

- For Linux/UNIX the epoch is Jan 1st 1970
- On 32-bit systems the seconds counter (returned by `time()`) is a signed integer which will overflow when it hits 2^{31} (a bit more than 2 billion seconds)
- This happens in January 2038 which is sooner than you'd think
- There is work to avoid this problem (if you are all 64-bit you avoid most of it) but still working
- This generations Y2K

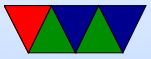


- Why didn't they think again? The Unix developers probably didn't expect people would still be using their code 60 years later.



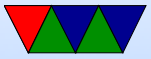
Configuring an ARM Timer

- Section 14 of BCM2835 Peripheral manual.
- Similar, but not exactly the same, as an ARM SP804
- There are also the system timers (4 timers described in Section 12).
- Note that the timer we use is based on the APB clock which ticks at 250MHz
- Limitations: it scales with the system clock, so frequency might change
- Also has free running timer (we don't use)



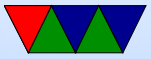
BCM2835 Timer Registers

- TIMER_LOAD: set a value and it will count down on each tick and give interrupt when zero. Automatically re-loaded after interrupt.
- TIMER_CONTROL: start/stop, IRQs on/off, scaling
- TIMER_RELOAD: queue a different value to be loaded into TIMER_LOAD automatically when current hits zero
- TIMER_IRQ_CLEAR: clears the interrupt
- TIMER_PRE_DIVIDE: another divider, as original design was for 1MHz clock

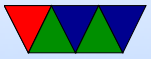


BCM2835 Timer Initialization Code

```
 /* Timer is based on the APB bus clock which is 250MHz on Rasp-Pi */

uint32_t old;

/* Disable the clock before changing config */
old=bcm2835_read(TIMER_CONTROL);
old&=~(TIMER_CONTROL_ENABLE|TIMER_CONTROL_INT_ENABLE);
bcm2835_write(TIMER_CONTROL,old);

/* First we scale this down to 1MHz using the pre-divider */
/* We want to /250.  The pre-divider adds one, so 249 = 0xf9 */
bcm2835_write(TIMER_PREDIVIDER,0xf9);

/* We enable the /256 prescalar */
/* So final frequency = 1MHz/256/61 = 64.04 Hz */

/* The value is loaded into TIMER_LOAD and then it counts down */
/* and interrupts once it hits zero. */
/* Then this value is automatically reloaded and restarted */
```



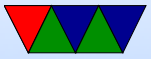
```
bcm2835_write(TIMER_LOAD,61);

/* Enable the timer in 32-bit mode, enable interrupts */
/* And pre-scale the clock down by 256 */
bcm2835_write(TIMER_CONTROL,
              TIMER_CONTROL_32BIT | /* In manual 23bit typo */
              TIMER_CONTROL_ENABLE |
              TIMER_CONTROL_INT_ENABLE |
              TIMER_CONTROL_PRESCALE_256);
```



BCM2835 Enable IRQ Controller

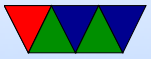
- In addition to enabling interrupts on the UART (see previous slide) we also have to tell the BCM2835 IRQ controller to let serial interrupts through

```
/* Enable timer interrupt */
bcm2835_write(IRQ_ENABLE_BASIC_IRQ,IRQ_ENABLE_BASIC_IRQ_ARM_TIMER);
```

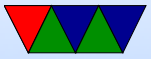


Pi Sample Path through an Interrupt

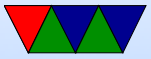
- HARDWARE: something triggers an interrupt
- DEVICE: passes IRQ request to IRQ controller
- IRQ CONTROLLER: passes IRQ request to CPU
- CPU:
  - Saves CPSR to proper SPSR
  - Saves PC to banked LR, loads banked stack pointer
  - Switches CPSR to correct mode
  - Jumps to proper entry in IRQ vector table
- IRQ HANDLER (your code):



- Code should save any registers you overwrite (note if FIR then some are auto-saved)
  (no need to save SPSR unless nested)
  gcc can do this for you
- Check interrupt source and call the proper handler (or give error if unknown)
- ACKnowledge (turn off) the interrupt at the device level
- Handle the interrupt
- Restore registers, return to proper address (LR + some offset)

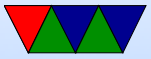


- CPU: switches back mode, restores registers, restart execution where we were when interrupted.



Sample Interrupt Handler

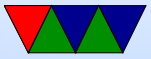
```
void __attribute__((interrupt("IRQ"))) interrupt_vector(void) {

   static int num_irqs = 0;
   int which;

   /* Check to see which interrupt happened */
   which=bcm2835_read(IRQ_BASIC_PENDING);
   if (which & IRQ_BASIC_PENDING_TIMER) {

      /* Clear (ACK) the Timer interrupt */
      bcm2835_write(TIMER_IRQ_CLEAR,0x1);

      /* increment IRQ count */
      num_irqs++;

      /* Every 64th IRQ print a message */
      if (num_irqs%64==0) {
         printk("We had an interrupt!\n");
      }

}
```



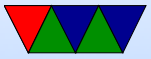
Enabling/Disabling IRQs on CPU – ARMv5 and Earlier

This works by reading out the CPSR register and using bit manipulation to set/clear the interrupt enable flags

```
static inline uint32_t get_CPSR(void) {
        uint32_t temp;
        asm volatile ("mrs␣%0,CPSR":"=r" (temp):) ;
        return temp;
}

static inline void set_CPSR(uint32_t new_cpsr) {
        asm volatile ("msr␣CPSR_cxsf,%0"::"r"(new_cpsr) );
}

/* enable interrupts  */
static inline void enable_interrupts(void){
        uint32_t temp;
        temp = get_CPSR();
```



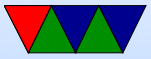
```
        set_CPSR(temp & ~0x80);
}
```



Enabling/Disabling IRQs on CPU – ARMv6 and Later

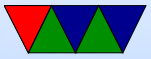
- CPSIE (Change Processor State Interrupt Enable)
- CPSID (Change Processor State Interrupt Disable)
- List which to enable/disable from list I=interrupts, F=fast interrupt, A=aborts

```
asm volatile ("cpsid␣i"::);  // disable irqs
asm volatile ("cpsie␣i"::);  // enable irqs
```



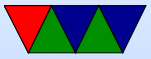
HW#4

- Now we know enough to start HW#4
- Set up a system timer that ticks at 64Hz
- Have the timer interrupt handler toggle the ACT LED on and off
- Additionally enhance the command line interpreter



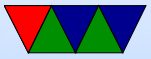
Changes for HW#4

- 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



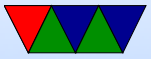
HW#4 – Device Tree

- 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, mostly 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 could also grab io_base, etc



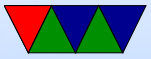
HW#4 – Device Tree

- You can find the device files on /boot with the dts ending
- If on a raspberry pi you can use the `dtc` tool to look at the contents (they are packed before using)
  `dtc -O dts FILE.DTC`
- TODO: provide a sample chunk from the Pi-1B+



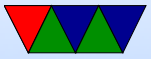
HW#4 – Writing a shell

- 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.



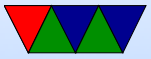
HW#4 – Reading a Line

- Have a buffer
- Do a uart_getc();
- If nothing special, put it in buffer and increment
- If CR/LF then done with line. Be sure to NUL terminate
- Call parser to parse line
- When gets back be sure to reset pointer to start of buffer



HW#4 – Parsing a Line

- How do you check what was entered using C?
- `strcmp()? strtok()?`
  Remember we have to write the C library ourself
- I provide a `string.c` that provides some routines
- For this homework it might be easy enough to just check manually instead
- Simple way to do things is to manually check, like
  `if ((buffer[0]=='l') && (buffer[1]=='s')) somet`

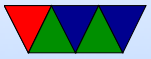


HW#4 – String manipulation

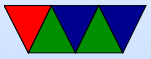
- Most C-based OSes quickly obtain string manipulation functions
- strncmp(), strlen(), strncpy(), memcpy(), memset(), memcpy()
- What's the different between strncpy and memcpy?
- How optimized do these routines need to be?
- memcpy() is often short blast of C

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

  but it can be optimized to death.

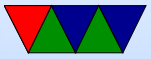


- memcpu() / memmove() difference? Why it's there, hazard when you don't use it right? (memmove the areas can overlap) (what happens if you copy backwards)



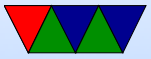
HW#4 – First Command

- Ask you to implement “print” which just prints Hello World
- If anything else typed, print Unknown command



HW#4 – Something Cool

- Add command of your choice
- Also add this to the provided "help" command so I can see what you did when grading.



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)