ECE 598 – Advanced Operating Systems
Lecture 9

Vince Weaver

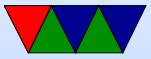
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

20 February 2018

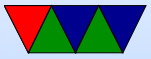
Announcements

- Homework #4 was posted
- Homework #3 was graded



HW#3 Overview

- Wasn't as picky this time, but comment code!
- Serial port: most got value right
  forgot to warn about floating point (say you wanted this to be parameterizable)
  Can you use floating point in kernel?
- printk: instead of /10, print remainder plus '0' instead /16 (which converts to shift) and two cases. 0-9 same as before, but A-F (you can just add 'A'-10 which I think is 55)



Beauty of ASCII. Often complicated use of ternary operator

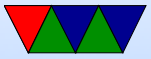
technically upper vs lowercase %X vs %x

Can also use lookup table.

Be careful shifting, what if print 0xfffffff? Shift right? Be sure unsigned! Also in C, shift right by 32?

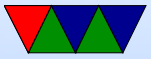
- Be sure print hardware info (r1)
- Questions
  - Why serial port?
  - What is parity? Why is it disabled?

    Faster (one fewer bit per byte), much bigger



infrastructure to handle, not even that great (only detect one bit flip). Not that critical for text. What would a file transfer do? (checksum?)
Some systems might not support? True, but why don't they support it?

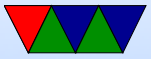
- inline asm lets you write code that's not possible in C. Also lets you bypass compiler (if you think you can do better)
  Don't confuse it with the volatile keyword.
- Why no strtok?
  string.h is the header, contains no code. Just describes



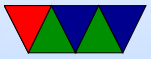
the routine in the C library.

- Problems with OSX?

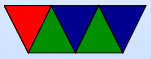
  screen, need to plug and unplug? Be sure to turn off HW flow control. Weird keybindings minicom. DOS was ALT-whatever, but ALT values don't go over connection (and are not easy to get from text console Linux) so used Control-A first to mean ALT.



Things added for HW#4

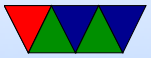


Device Tree



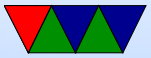
include and quotes

- 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)



string manipulation

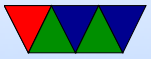
- Most C-based OSes quickly obtain string manipulation functions
- strncmp(), strlen(), strncpy(), memcpy(), memset(), memcpy()
- What's the different between strncpy and memcpy?
- How optimized do these routines need to be?
- memcpy() is often short blast of C



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

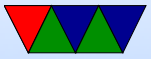
but it can be optimized to death.

- Should I mention 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)



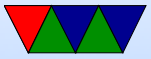
no more \r

- I've modified uart_write() so you no longer need to do \r.



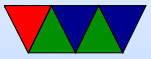
writing a shell

- 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.



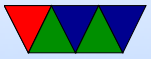
LED routines

- 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.



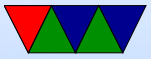
Interrupt Roundup

Any questions on interrupts?



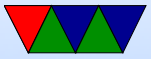
Interrupts Schemes

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



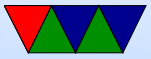
Interrupts on Linux

- 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

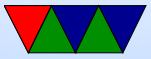


interrupt, handle super pressing thing (copy data out of FIFO) Then tell the kernel to handle the rest later.
So you might have a tasklet/kernel thread that runs occasionally (and is fully interruptible) that will do the rest.
For example, network packet comes in, important to read the packet and ACK 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.

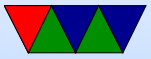


Userspace

- Why use userspace (why not everything in kernel like DOS?)
  Slower, but has some protections from bad programs/security

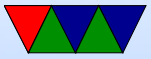
- Can't access all of CPSR register
  Can't turn off interrupts
  Can't switch to privileged modes

- If virtual memory enabled, can't access protected/kernel



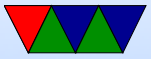
memory

- Can you still access MMIO?



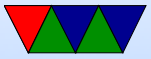
Entering User Mode

```
mov r0, #0x10
msr SPSR, r0
ldr lr, =first
movs pc, lr
```



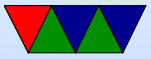
System Calls

- 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.



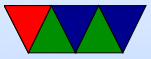
Linux ARM System Call Interface

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

  `swi 0`

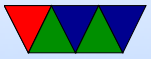
  Return value in r0

- OABI: Arguments in r0 through r6. `swi SYSBASE+SYSCALLNUM`. Why bad? No way to get swi value except parsing back in instruction stream.



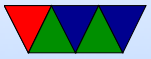
SWI Interrupt Handler

```
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;
}
```



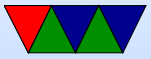
Linux System Call Results

- 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?)



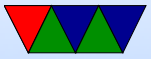
Non-ARM syscalls

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



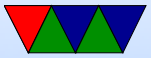
Application Binary Interface

What is an ABI and why is it necessary?

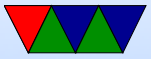


Linux GNU EABI

- 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?

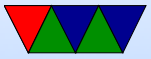


- What if more than 4 arguments? (stack)
- Is there a stack, how aligned?
- Structs, bitfields, endianess?
- Callee vs Caller saved registers? (A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP)
- Frame Pointer?



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;

}
```