ECE 598 – Advanced Operating Systems Lecture 8

Vince Weaver

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

vincent.weaver@maine.edu

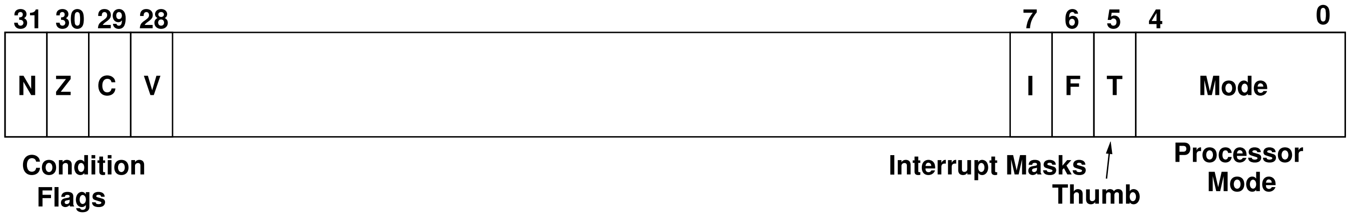
10 February 2015

Announcements

- Homework #1 grades will be out soon
- Homework #2 : Don't put it off until the last minute!
- Putty and Minicom seem to work. MacWise can work for OSX, be sure to install the PL2303 driver.
  Try putting a `uart_getc()` before your boot code so it will wait for a keypress before displaying.
- Inline asssembly problem with older compiler version.

- Homework #3 coming out probably on Wednesday again

ARM CPSR Register



- Current Program Status Register
- There are seven processor modes
  six privledged: abort, fast interrupt, interrupt, supervisor, system, undefined
  one nonprivledged: user
- unprivledged cannot write CPSR

ARM Interrupt Registers

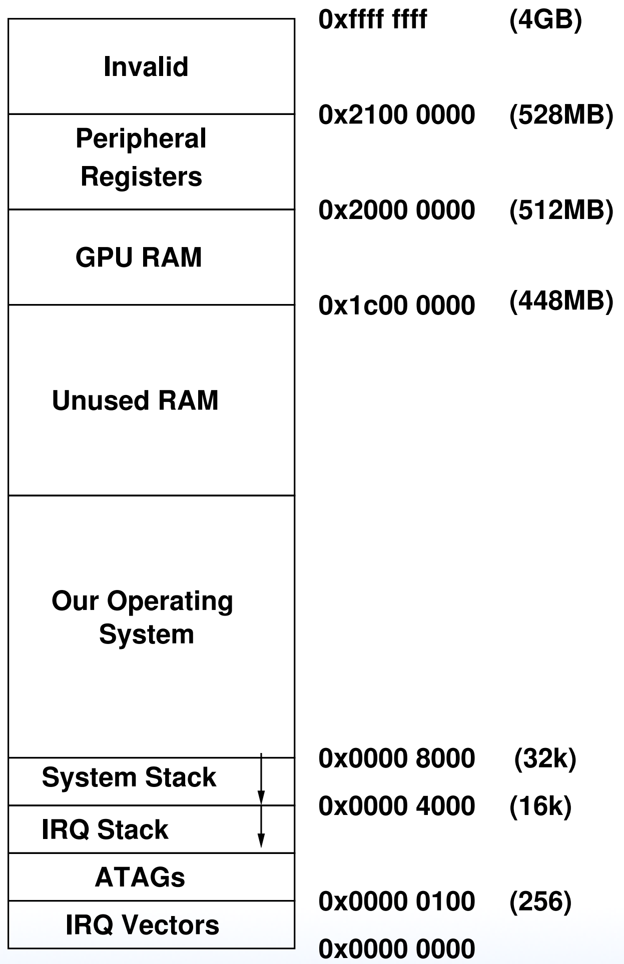
User/Sys	Fast	IRQ	Supervisor	Undefined	Abort
r0					
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8_fiq				
r9	r9_fiq				
r10	r10_fiq				
r11	r11_fiq				
r12	r12_fiq				
r13/sp	r13_fiq	r13_irq	r13_svc	r13_undef	r13_abt
r14/lr	r14_fiq	r13_irq	r14_svc	r14_undef	r14_abt
r15/pc					
cpsr	spsr_fiq	spsr_irq	spsr_svc	spsr_undef	spsr_abt

Setting up the Stacks

```
/* Set up the Interrupt Mode Stack       */
/* First switch to interrupt mode, then update stack pointer */
mov     r3, #(CPSR_MODE_IRQ | CPSR_MODE_IRQ_DISABLE | CPSR_MODE_FIQ_DISA
BLE )
msr     cpsr_c, r3
mov     sp, #0x4000

/* Switch back to supervisor mode */
mov     r3, #(CPSR_MODE_SVR | CPSR_MODE_IRQ_DISABLE | CPSR_MODE_FIQ_DISA
BLE )
msr     cpsr_c, r3
```

Our Memory Map



Timer Interrupt

- Section 14 of peripheral manual.
  There are also the system timers (4 timers described in Section 12).

```
int timer_init(void) {

        mmio_write(IRQ_ENABLE_BASIC_IRQ,IRQ_ENABLE_BASIC_IRQ_ARM_TIMER);

        /* Timer frequency = Clk/256 * 0x400 */
        mmio_write(TIMER_LOAD,0x400);

        /* Setup the ARM Timer */
        mmio_write(TIMER_CONTROL,
                TIMER_CONTROL_32BIT | /* typo 23 */
                TIMER_CONTROL_ENABLE |
                TIMER_CONTROL_INT_ENABLE |
                TIMER_CONTROL_PRESCALE_256);
```

```
    /* Enable interrupts! */
//    _enable_interrupts();
}
```

Enabling Interrupts

```
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);
}
```

SWI Interrupt

```
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);
        return 42;
}
```

System Calls

- EABI: Arguments in r0 through r6. System call number in r7.
  `swi 0`

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

ABI

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 throgh r4.
- Return value in r0.
- How to return float, double, pointers, 64-bit values?
- 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?

Executables

Executable Format

- ELF (Executable and Linkable Format, Extensible Linking Format)
  Default for Linux and some other similar OSes
  header, then header table describing chunks and where they go

- Other executable formats: a.out, COFF, binary blob

ELF Layout

ELF Header
Program header
Text (Machine Code)
Data (Initialized Data)
Symbols
Debugging Info
....
Section header

ELF Description

- ELF Header includes a "magic number" saying it's 0x7f,ELF, architecture type, OS type, etc. Also location of program header and section header and entry point.

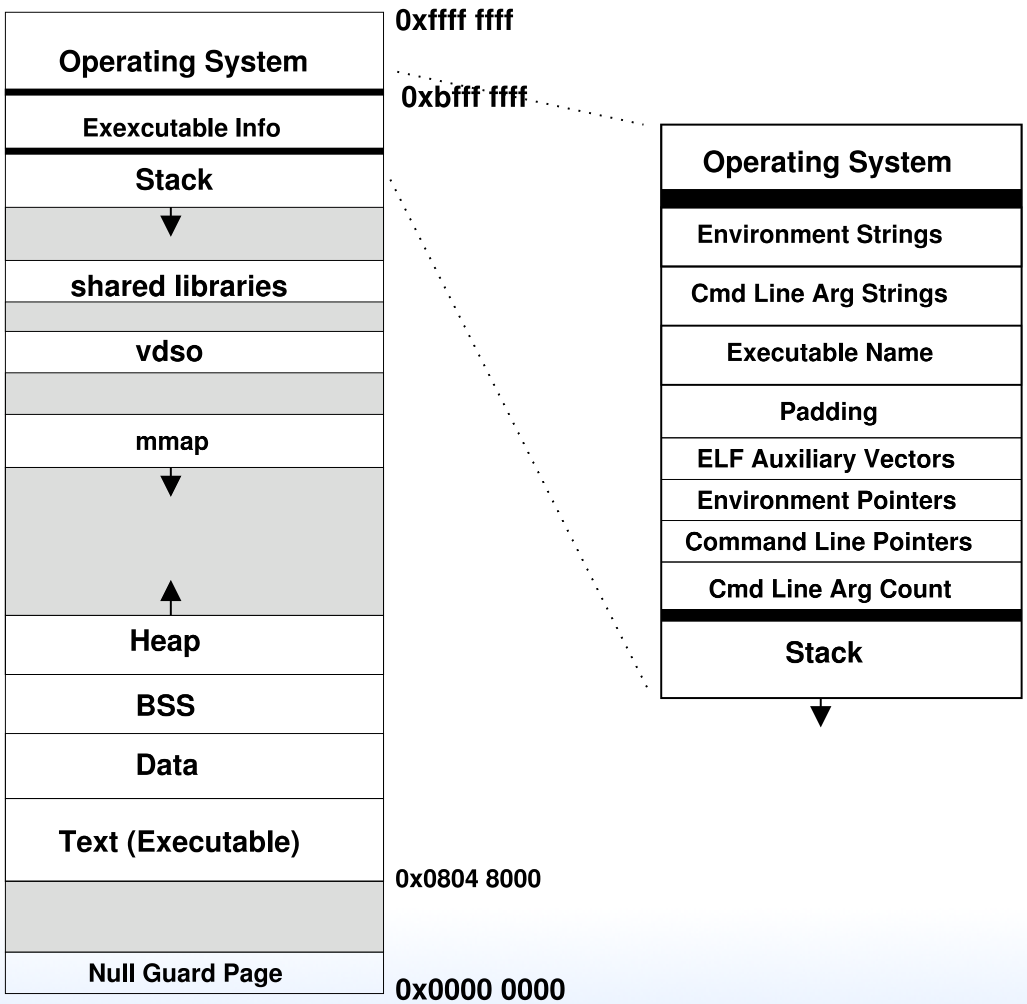
- Program Header, used for execution:
  has info telling the OS what parts to load, how, and where (address, permission, size, alignment)

- Program Data follows, describes data actually loaded into memory: machine code, initialized data

- Other data: things like symbol names, debugging info (DWARF), etc.
  DWARF backronym = "Debugging with Attributed Record Formats"

- Section Header, used when linking:
  has info on the additional segments in code that aren't loaded into memory, such as debugging, symbols, etc.

Linux Virtual Memory Map



Program Memory Layout on Linux

- Text: the program's raw machine code
- Data: Initialized data
- BSS: uninitialized data; on Linux this is all set to 0.
- Heap: dynamic memory. `malloc()` and `brk()`. Grows up
- Stack: LIFO memory structure. Grows down.

Program Layout

- Kernel: is mapped into top of address space, for performance reasons
- Command Line arguments, Environment, AUX vectors, etc., available above stack
- For security reasons "ASLR" (Address Space Layout Randomization) is often enabled. From run to run the exact addresses of all the sections is randomized, to make it harder for hackers to compromise your system.

Loader

- /lib/ld-linux.so.2
- loads the executable

Static vs Dynamic Libraries

- Static: includes all code in one binary.
  Large binaries, need to recompile to update library code, self-contained

- Dynamic: library routines linked at load time.
  Smaller binaries, share code across system, automatically links against newer/bugfixes

How a Program is Loaded on Linux

- Kernel Boots
- `init` started
- `init` calls `fork()`
- child calls `exec()`
- Kernel checks if valid ELF. Passes to loader
- Loader loads it. Clears out BSS. Sets up stack. Jumps

to entry address (specified by executable)

- Program runs until complete.
- Parent process returned to if waiting. Otherwise, init.

UCLinux

Flat File Format

- http://retired.beyondlogic.org/uClinux/bflt.htm
- bFLT or 0x62, 0x46, 0x4C, 0x54
-
```
struct flat_hdr {
    char magic[4];
    unsigned long rev;          /* version */
    unsigned long entry;        /* Offset of first executable instruction
                                   with text segment from beginning of file */
    unsigned long data_start;   /* Offset of data segment from beginning of
                                   file */
    unsigned long data_end;     /* Offset of end of data segment
                                   from beginning of file */
    unsigned long bss_end;      /* Offset of end of bss segment from beginning
                                   of file */

    /* (It is assumed that data_end through bss_end forms the bss segment.) */
```

```
    unsigned long stack_size;   /* Size of stack, in bytes */
    unsigned long reloc_start;  /* Offset of relocation records from
                                   beginning of file */
    unsigned long reloc_count;  /* Number of relocation records */
    unsigned long flags;
    unsigned long filler[6];    /* Reserved, set to zero */
};
```