ECE 598 – Advanced Operating Systems
Lecture 10

Vince Weaver

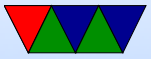
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

18 February 2016

Announcements

- Homework #5 will be posted

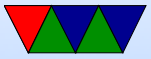


Userspace

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

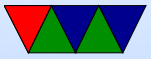
- Can't access all of CPSR register
  Can't turn off interrupts
  Can't switch to priviledged modes

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



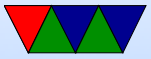
memory

- Can you still access MMIO?



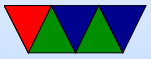
Entering User Mode

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



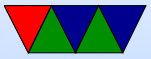
System Calls

- 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!



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.



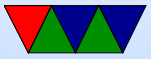
Linux ARM System Call Interface

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

  `swi 0`

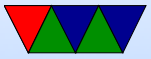
  Return value in r0

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



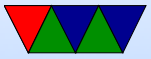
SWI Interrupt Handler

```
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;
}
```



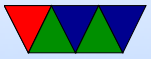
Linux System Call Results

- 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?)



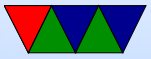
Non-ARM syscalls

- 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.



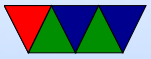
Application Binary Interface

What is an ABI and why is it necessary?

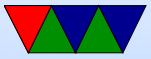


Linux GNU EABI

- Procedure Call Standard for the ARM architecture
- ABI, agreed on way to interface with system.
- Arguments to registers. r0 throgh 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?

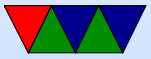


- What if more than 4 arguments? (stack)
- Is there a stack, how aligned?
- Structs, bitfields, endianess?
- Callee vs Caller saved registers? (A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP)
- Frame Pointer?



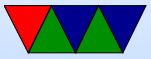
ABI Purpose

- 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

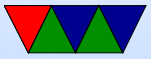


Calling a Syscall

```
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;

}
```



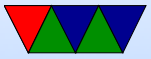
Userspace Executables



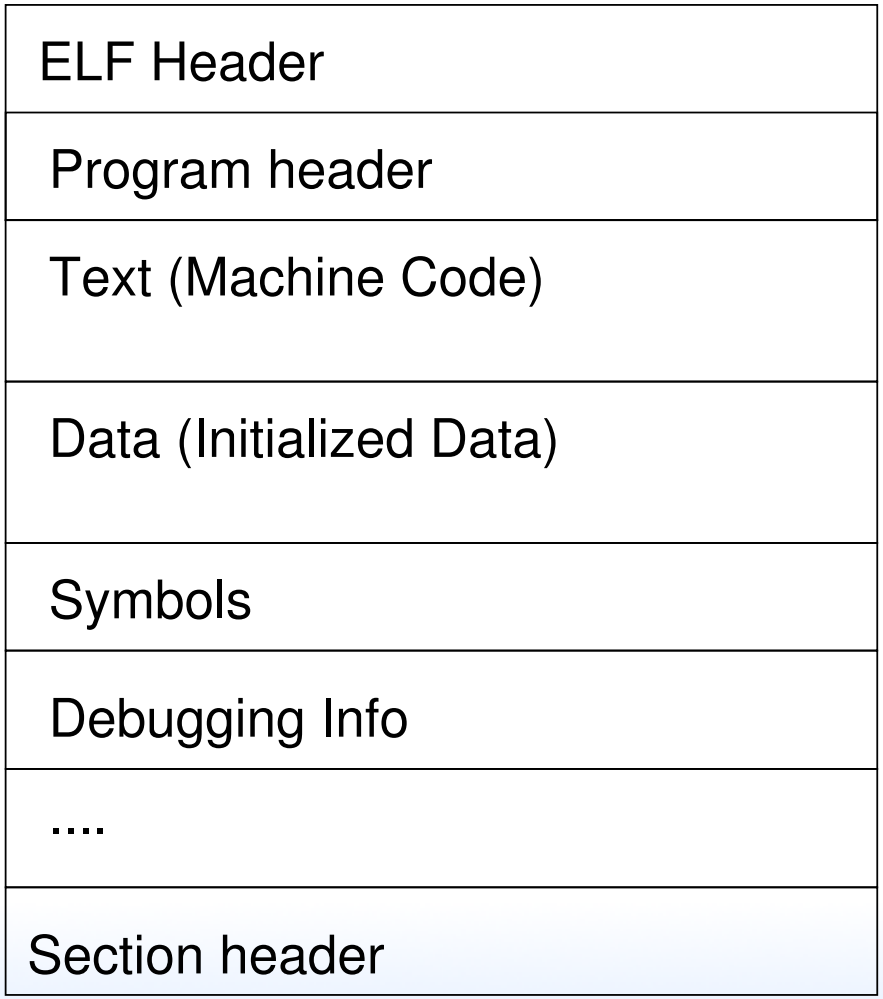
Executable Format

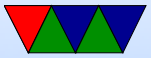
- ELF (Executable and Linkable Format, Extensible Linking Format)
  Default for Linux and some other similar OSes
  header, then header table describing chunks and where they go

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



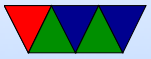
ELF Layout


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



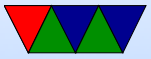
ELF Description

- 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



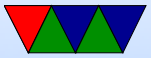
- Other data: things like symbol names, debugging info (DWARF), etc.
  DWARF backronym = "Debugging with Attributed Record Formats"

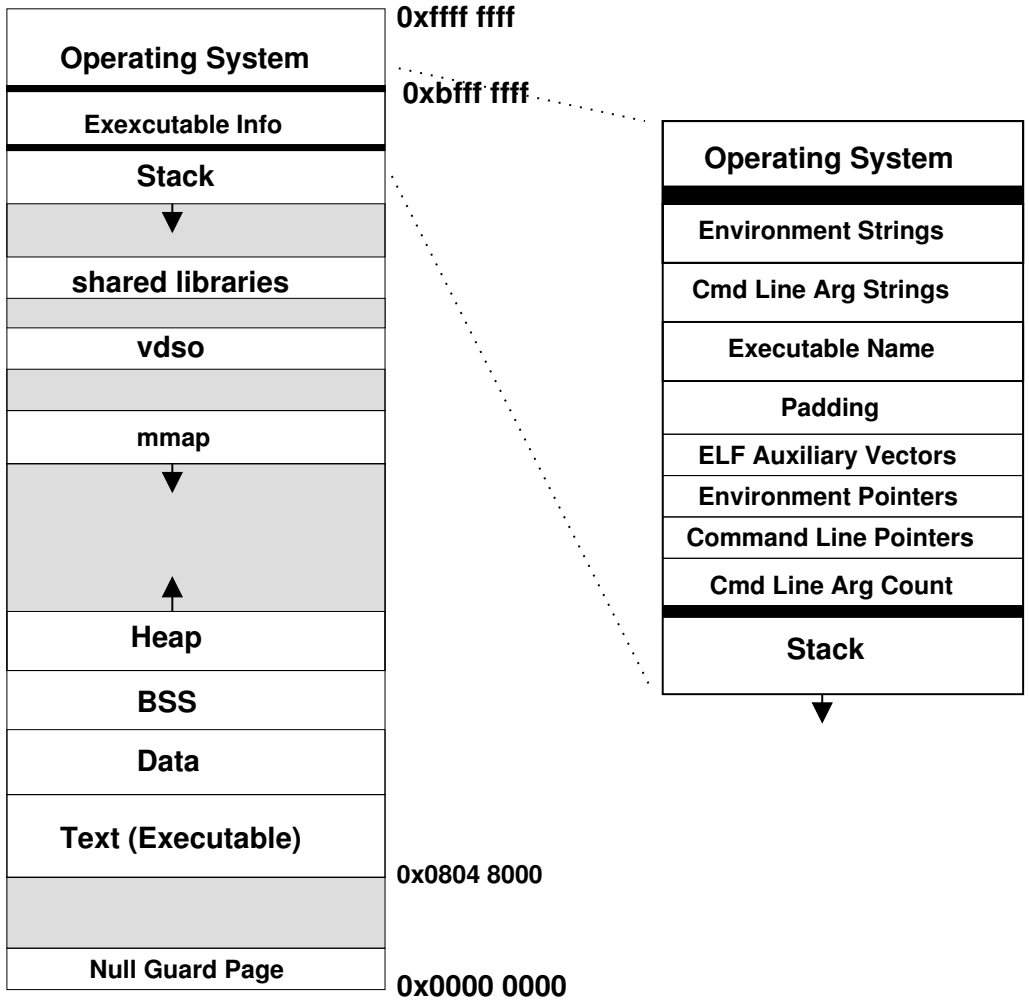
- Section Header, used when linking:
  has info on the additional segments in code that aren't loaded into memory, such as debugging, symbols, etc.

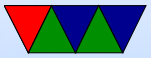


Linux Virtual Memory Map

We will go over virtual memory in much greater detail later.

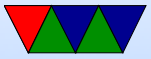



Operating System
Exexcutable Info
Stack
shared libraries
vdso
mmap
Heap
BSS
Data
Text (Executable)
Null Guard Page
0xffff ffff
0xbfff ffff
0x0804 8000
0x0000 0000
Operating System
Environment Strings
Cmd Line Arg Strings
Executable Name
Padding
ELF Auxiliary Vectors
Environment Pointers
Command Line Pointers
Cmd Line Arg Count
Stack



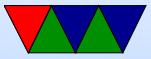
Program Memory Layout on Linux

- 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.



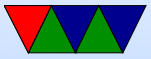
Program Layout

- 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