ECE 598 – Advanced Operating Systems
Lecture 11

Vince Weaver

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

vincent.weaver@maine.edu

23 February 2016

Announcements

- Homework #5 Posted
- Some notes, discovered the hard way:
  - Do not call a syscall while in SVC mode. Why? SWI mode and SVC mode share the same stack pointer
  - Also, what happens if you forget to set up a user stack?
  - The gcc swi handler won't do the right thing with regards to returning a value from a syscall. Especially if you use local variables.

HW#4 Review

- Forgot to include README in Makefile.

- Same issue with HW#5 if you downloaded before noon Monday. Can manually attach README if you downloaded before then.

- Be careful using `&0x1` vs `&&0x1`

- Be sure your code compiles

- FIQ vs IRQ difference? FIQ banks some registers, so is faster and higher priority.

- `BASIC_PENDING` bit 19 is interrupt 57 which is uart
- How to change modes? Write to the mode field of CPSR register.

Syscall Summary (From Last Time)

- Want to run in userspace usually, safer
- What two ways to get from user back to kernel space?
- How do you call into a syscall?

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?)

ABI/Executable Review

- What's an ABI? Is it necessary?
- ELF executable format
- Static vs Dynamic libraries

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

Linux typically relies on MMU (virtual memory). You can run it on systems w/o virtual memory, this version is called ucLinux (micro-controller Linux).

Our OS in the homework is similar in design to this.

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 */
};
```

Figuring out how it actually works

- Spec isn't worth much
  Your best bet is various Wikis and blog postings (TI-nspire?)

- Actual code more useful

- `fs/binfmt_flat.c` in kernel source.

- Making the binaries hard. Not just a simple matter of telling gcc or linker (no one has bothered yet). Most people use “elf2flt” but not-standard and hard to even find which code repository to use.

Loading a flat binary

- load_flat_binary()
- adjust stack space for arguments (argv and envp)
  - loading header. Uses `ntohl()`. Why?
    Endian issues.
  - check for bFLT magic
  - check version
  - check rlimits() [stack, etc]
  - setup_new_exec()

  - allocate mem for our binary (separately handle XIP and compressed format)
  - read_code()
  - put all of our values in mm struct (Start/stop of all sections)
  - RELOCATION – fix up any symbols that changed due to being moved. (HOW DOES THIS WORK)
  - flush_icache()
  - zero the BSS and STACK areas

- setup shared libraries

- install_exec_creds()
- set_binfmt()
- actually copy command line args, etc, at front of stack
- put stack pointer in mm structure
- start_thread()

PIC/PIE

- Position independent code
- Instead of loading from absolute address, uses an offset, usually in a register or PC-relative.
- gcc has an option `-fPIC` to generate

Relocation

- List of offsets to pointers
- PIC compiles things with zero offset
- At load time the pointers are fixed up to have the load address
- Separate relocation for GOT (global offset table) which is a list of pointers at the beginning of the data segment, ending with -1

Flat Shared Libraries

- Like mini executables, can have up to 256 of them

- Libraries loaded in place, then the callsites are fixed up to have the right address.

- Also at start time the various library init routines are called

Execute in Place

- Want our text in ROM. Why? Save space, save copying. Why bad? ROM often slow, more complicated binaries (data not follow text)

RAM Disk

- How to load our code?
- Can we load from disk? No driver yet.
- We can create a RAM disk, will be loaded by our bootloader right after. Sometimes called an initrd.

Context switching

Starting a Process and Context switching

r14	the process LR
r13	
r12	
r11	
r10	
r9	
r8	
r7	
r6	
r5	
r4	
r3	
r2	
r1	
r0	PCB pointer points here (for stm instruction)
lr	pc from process to return to
spsr	

Process Control Block

- PCB – process control block. One for each process
- r0-r14 saved. PC. cpsr
- Pid, uid
- Memory ranges
- Process accounting
- Ready, sleeping, waiting, etc

Entering User Mode

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

ARM Context Switch

r12 = new process PCB, r13 = old

```
STM     sp,{R0-lr}^                     ; Dump user registers above R13.
                ; ^ means get user register
MRS     R0, SPSR                        ; get the svaed user status
STMDB   sp, {R0, lr}                    ; and dump with return address below.
                ; lr is the handler lr, pointing
                ; to pc we came fom
LDR     sp, [R12], #4                   ; Load next process info pointer.
CMP     sp, #0                          ; If it is zero, it is invalid
LDMDBNE sp, {R0, lr}                    ; Pick up status and return address.
MSRNE   SPSR_cxsf, R0                   ; Restore the status.
LDMNE   sp, {R0 - lr}^                  ; Get the rest of the registers
NOP
SUBSNE pc, lr, #4                       ; and return and restore CPSR.
                                        ; Insert "no␣next␣process␣code" here.
```

Storing

```
ldmfd   r13!,{r0-r3,r12,r14}
ldr r13,=PCB_PtrCurrentTask
ldr r13,[r13]
sub r13,r13,#offset15regs
stmia   r13,{r0-r14}^
mrs r0,spsr
stmdb   r13,{r0,r14}
```

Loading

```
ldr r13,=PCB\_PtrNextTask
ldr r13,[r13]
sub r13,r13,#offset15regs
ldmdb   r13,{r0,r14}
msr spsr_cxsf,r0
ldmia   r13,{r0=r14}^    ; ^ means update user regs
ldr r13,=PCB_IRQstack
ldr r13,[r13]
movs    pc,r14
```