LAKSMI
Some OS Implementation on ARM

Peijie You 13307130325

January 12, 2016

Contents

1 Opening

In this project for Operating System course, we are told to *implement our own operating system on ARM*. However, we have not finish all features of it by now. In this report, we are going to take a review of what we have done during the semester, and then describe our version of AIM operating system (I called it LAKSMI) here.

2 Previously on OS

During the course, we are guided by our TA to build program we need for our final OS. We will review these thing quickly here.

2.1 STEP 0 — Some Introduction on ARM or the Zedboard

Before that, I have nearly **no** experience on ARM or other architecture.

Notable things on ARM and the zedboard.

- Core register, r0-r12, sp, lr, pc, some of them are **banked**.
- CPSR, and SPSR, program status register, it will remember all kind of flag, and the mode.
- CP15(co-processor?) thing, like SCTLR, TTBR0 .... And the special instruction MCR MRC to load/storage the register value. *They are bank for the processor.*
- Mode, USR, IRQ, FIQ, Supervisor, Monitor, Abort, Hyp, Undefined, System, we mainly need SYS and USR here.
- Routine while exception arose, we should talk about it latter.
- Complicated number is not supported, like 0x12345678, you should separate them into two operation instruction if needed. (Since the ARM instruction is **fixed size**)

2.2 STEP 1 — run the program

To run program on the board, you may want to compile and link the program on your own computer and load to some specific location. (No compiler on the board).

The procedure are as follows:

1. compile with arm-gcc
2. link with arm-ld
3. load to somewhere on the SD card by “dd” instruction on linux.

When the firmware program finish its work, it will load the first block in SD card to the (physical) address 0x100000. So our program should satisfy the following requirement:

- saved on the first block on sd card (can be done by proper dd instruction)
- know that it will execute at address 0x100000 (done by ldscript)
- have length less than the MBR size(like 446B)

After this experiment, I come to know about the MBR thing.

2.3 STEP 2 — build bootloader

Since the length is MBR is limited and our kernel may not fit in it, we have to load our own kernel into memory. Here in our OS, a kernel is a ELF file stored from the first block of the **second** partition.

ELF is a file format, for Executable and Linkable Format, according to the manual, we have to:

1. read the elf header, know some information about the program header
2. load program header entry(a segment?), it will specified where to load this segment thing, just load as it says, I will load to the *paddr* for kernel and *vaddr* for user program
3. jump to the program entry specified in the elf header

Noted that the validation of elf header is not implemented here.

2.4 STEP 3 — memory management

An important things that we have not brought up yet is the **MMU**, so no virtual address - physical address translation yet. Our job here is to enable MMU and clean things that we no longer need.

In ARM32, we have 2 level of page table structure, and 4 kinds of page(4KB, 64KB, 1MB, 16MB), in this project we will only use page of 4KB(for user) and 1MB(for kernel).

We divide the virtual space into *2G:2G*, the lower 2G is user space is higher 2G is kernel space.

VIRTUAL memory for now:

- 3M, code for kernel before MMU(will be clear after MMU is enabled).
- 2G to 2G+512M, image for RAM
- 2G+2M, where the rest of kernel is load.
- 0xE0000000 to 0xFFFFFFFF, mapping to the corresponding place, care for the device and other things.

- 0xDFF00000, kernel stack during initialization, mapping physical RAM 511-512M
- 0xFFFF0000, interrupt table.

PHYSICAL memory for now:

- 3M, code for kernel before MMU
- 2M, rest of the kernel
- 0xE0000000 to 0xFFFFFFFF, care for the device and other things.
- 511-512M, first the firmware, then the stack for sys
- 4M, interrupt table

2.5 STEP 4 — exception handler

3 Recently on OS

Here we will first conclude the overall project and then high light some parts. The following part is a teamwork output, with —todo— implement sched related thing, —todo— code the file system, and three of us design the system call together.

3.1 How the program is run

1. Go from firmware.bin
2. Fall in the MBR code, it automatically load 2 parts(before and after MMU) to the RAM.
3. Init for all kind of device again.
4. Enable MMU.
5. Enable l1cache.
6. Enable SCU.
7. Enable page manage, for page alloc
8. Init for slab allocator.
9. Init for interrupt
10. Init for file system
11. Init for sheduler
12. Fork and exec a user program.

3.2 System Call

Here we support all kind of system call, *some of which is not debugged*, see the file *syscall.h* for detail.

1. process related, *fork(), exec(), exit(), wait(), getpid(), getppid()...*
2. I/O and file system related, *gets(), puts(), open(), close(), write(), create(), delete()...*

A system call is actually a "SVC ID" instruction here, with register R0 (and probably R1 R2) to pass parameter, and R0 serves as the return value.

3.3 Scheduler

Here we use kind of Round Robin policy, with no priority. When private timer ticks, we run scheduler to choose next process.

3.4 File System

3.5 Exception

3.6 Device Update

We also made some modification to the given device, like:

1. Manage the cross frame interrupt, in sd_dma_spin_read(), we have discuss it in class, and the final solution is provided by —todo—.
2. For easy to use and debug, we have *printf()* thing implemented.
3. a function called *puthex()* failed on my OS, because the string that puts *is not end with* \0.
4. *gets()*, provided by —todo—.

3.7 Division on ARM

ARM architecture alone did not provide integer division, here I implement my division.[**?**]

You can click that link and view the source code. The function I implemented are *__aeabi_idiv()* or *__aeabi_uidiv()*.

User in our OS can simply use division, and the compiler will care about the rest. For divided by zero, we will return 0. You are welcome the raise error for this.

4 Others

Necessary *user guides and manual* during the development.

1. ARM Architecture Reference Manual(ARMv7-A and ARMv7-R edition), *manual for the architecture.*
2. CortexTM-A9 MPCore Revision: r3p0 Technical Reference Manual, *manual for the specific mpcore things, such as timer, snoop control unit...*
3. Zynq-7000 All Programmable SoC Technical Reference Manual, *about the board.*
4. Executable and Linkable Format.PDF, *to help the load elf thing.*
5. ARM Generic Interrupt Controller v1.0, *may be good for the interrupt handling.*

5 Summary

6 Thank You

Report generated in LaTeX
View this project on GitHub.

References