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 — MMU

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, and the 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

See the code (like pte.h) for the page table entry detail.

2.5 STEP 3 — Memory Management

The following 3 component are mm provided to the kernel developer:

Page Management As we have agreed during the discussion, page management is implemented as a *list of free extent*. And allocation is done by *first fit* algorithm. The link list information is kept inside the free page(unit in 4KB).

Slab Management My slab offer a easy way for kernel developers to get memory, the interface is as follows:

- *slb_alloc(osize)*, kernel wants memory for their object(size of osize), will return the start address for that.
- *slb_alloc_align(osize, align)*, kernel wants memory, along with align option (mainly used for page table here.)
- use *slb_free(p, osize)* or *slb_free_align(p, osize, align)* to free the memory, noted you should know the object size and align option for that pointer(as my slab is mainly design for *object* allocation.)

My "slab" is a strange concept here, as I have different size mixed up together. But in detail, they use different pool.

Kernel Memory My *kmalloc()* is implemented base on the slab allocator. In *kfree()*, we have no idea of the *memory length*, so it should be a meta data and keep by the allocator. So I just pack the metadata(len) and the raw data into some kind of "object" and use slb_alloc() to allocate memory.

Here I think that no information on SIZE is need when kfree(), is a favorable feature.

mmap? This function is to map the corresponding virtual page to some physical memory. Our implementation is naive, which traverse the page table and build a L2 PAGE for every of them.

2.6 STEP 4 — exception handle

As far as my knowledge, when an exception comes, the following things will happen:

- Change pc to the corresponding interrupt vector address.
- Previous cpsr goes to spsr, and we have new cpsr.
- Bank register is different now.
- The prefer return address, which indicate the return route of exception handling, is save in lr(which is “prefer” is defined in arm-v7a manual).

And then we are in the exception handler, and feel free to do anything, as we are in *privileged mode* inside the exception handler. Other routine we used are included in the following section.

For IRQ, and SGI (software generated irq), we also have to care about the interrupt controller, which decide how to send irq to our processors. Some control register that I have set according to the zynq-7000 manual are:

1. ICDDCR, set to enable distributor
2. ICDIPTR, define which processor should this interrupt go
3. ICDIPR, priority for the interrupt
4. ICDISER, set enable for every irq.
5. ICCPMR, set interrupt mask
6. ICCICR.

Plus, in exception handle, we have to signal the acknowledge of signal in ICCIAR and signal the finish of exception handling in ICCEOIR.

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 and we (mainly) share the code, with Xiaotao Liang implement sched related thing, Haoming Xing 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()...*

3.2.1 System Call

I describe the system calls which relate to process here, and the the system calls which relate to file system will be discussed in the next section.

fork To create a new child process, we request a unused PID, create a new PCB and a new page table, copy the whole user image of the parent process(as we have no copy on write), and put the child process into the ready queue. For the parent process, the system call returns PID of the child process; for the child process, the system call returns 0.

We will always return the father process first.

exec This system call removes the memory mapping of user space, and then load the ELF program to be executed into memory just like how bootloader does, excepte that we need to allocate memory for each segment and take care of the memory mapping here. After that, we set up the context of CPU.

wait If there is any child process of the current process in the zombie queue, the current process deallocates the resources of its child process and return, otherwise it will be put into the waiting queue and wait to be wake up.

exit Change the status of the calling process to ZOMBIE. If the current process's parent is waiting for it, then wake up the parent process to deallocated its resources. If the current process's children have exited, then pass the children process to init.

getpid, getppid Pick the calling process in running queue and return the PID or PPID which is stored in the process's PCB.

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

This part is provided by —todo—, and please refer to his report for detail.

Here are status value for our pcb:

`NEW, READY, RUNNING, WAIT, ZOMBIE, ABORT.`

Other than that, our pcb have a pointer to L1 page table, a cpu context, pid, ppid things and other information(see code for detail).

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

How to context switch? Here we have two functions named *switch_mm()* and *switch_to()*, the first is used to change the TTBR0, and the second to switch cpu register context.

3.4 File System

This part is provided by Haoming Xing, and please refer to his report for detail.

The file system is implemented using link-based allocation. Each file is described as a linked list of disk blocks. The kernel supports basic operations of the file system, providing following system calls:

```
int open(char *name, int nbyte);
int close(int fd);
int read(int fd, char *buf, int nbyte);
int write(int fd, char *buf, int nbyte);
int create(char *name, int nbyte);
int delete(int fd);
```

3.5 Exception

Here is the road map of our OS exception handling.

1. exception arise, switch to other mode like svc, irq.
2. save some not bank register value(in assembly language), we change our cpsr manually, then jump to c program
3. save the rest cpu context, noted that we have no stack for other mode, only a **structue** (rather than all kind of stack here) to store the context
4. execute the svc / irq handler in c language, system will go down for other exception.
5. on return, load the context, and the assembly language will change the cpsr automatically while we update pc

3.6 Timer

Here we use private timer as our timer for scheduler. It will count down periodically from certain value then send irq. It is memory mapped and therefore easy to set and control.

3.7 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 Haoming Xing.
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 Haoming Xing.

3.8 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.

3.9 How to run it?

It may take time to initialization. Our OS can run user program, please place it start at 0x003b9ad4 block. (For example, user.elf here)

1. copy the firmware.bin
2. dd to load the mbr.bin to the block 0
3. dd kernel.elf to second partition
4. dd user.elf to 0x003b9ad4 block

```
sudo dd if=mbr.bin of=/dev/disk1 bs=440 count=1 && sync
sudo dd if=kernel.elf of=/dev/disk1s2 bs=512 && sync
sudo dd if=user.elf of=/dev/disk1s3 bs=512 && sync
```

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. You can see code for detail.
Sorry about all comment and debug printf.
I will show my appreciation to two of my teammate, Xiaotao Liang and Haoming Xing, and our TA.