A Simple Kernel for ARM

Xiaotao Liang
13307130319

January 12, 2016

1 Introduction

This is a simple kernel on the architecture ARM.

1.1 Overview

1.2 Components of the Kernel

1.2.1 Bootloader

Create 4 primary partitions on the SD card. The first partition stores the firmware, the 2nd partition stores the kernel, and the 3rd partition serves as a file system for the kernel.

A binary program will run as MBR to load and boot the kernel. The program first reads the partition table to find out the start address of the 2nd partition, then loads the kernel program which is in ELF format to memory, and finally calls the entry of the kernel to boot it. Loading the kernel needs to read the ELF header, the program header table and all segments of contents successively.

1.2.2 Memory Manager

Divide the virtual address space into two parts, 2GB/2GB, one for user space and the other for kernel space.

Build a basic first-level page table: high address has mapping `VA = PA + KERN_BASE`, and low address has mapping `VA = PA`. Set the TTB register, enable MMU, invalidate TLB, jump to high memory, and remove all low memory mappings. With all these finished, finally the MMU has been started.

I use a first-fit algorithm for page allocation. A linked list is used to manage empty pages. The physical memory allocator has following interfaces:

```
char *alloc_pages(int num);  // allocate num pages
void free_pages(char *addr, int num);  // free num pages starting from
     addr
```

I use a slab-allocation algorithm to allocate memory to kernel objects. Each kind of fixed-size kernel objects has a corresponding cache. Each cache consists of one or more slabs, and each slab consists of several physically contiguous pages. When creating a cache for a new kind of kernel objects, the size of objects, the size of slabs and the boundary for objects to align must be specified. The object allocator has following interfaces:

```c
void init_slab_cache(slab_cache_t *cache, int obj_size, int obj_num,
    int align_shift, int pages_of_slab);  // initialize a cache
char *alloc_obj(slab_cache_t *cache);  // allocate a object from the
    corresponding cache
void free_obj(slab_cache_t *cache, void *addr)  // free the object at
    addr
```

1.2.3 CPU Scheduler

To support different scheduling algorithms, policy is separated from mechanism. I implement the round-robin (RR) algorithm in the kernel, but other scheduling algorithms can be added easily.

The status of each process is one of following:

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

And RR scheduling maintains 6 FIFO queues of corresponding processes. Each queue contains all processes with the same status. When the status of a process is changed, the process is moved from the current queue to the tail of another queue.

The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. The process continues this cycle until it terminates, at which time it is moved to the zombie queue. Then its parent process deallocates the process's PCB and resources. Finally the process will be removed from all queues.

1.2.4 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 at the subsection 1.2.5 .

fork To create a new child process, we request a unused PID, create a new PCB and a new page table, copy the whole image of the parent process, 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.

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.

1.2.5 File System

The file system is implemented using chain-based allocation. Each file is described as a linked list of disk blocks. Therefore, data is read or wrote sequentially. The kernel supports basic operations of the file system, providing following system calls:

```c
int open(char *name, int nbyte);  // open the file, given the name
int close(int fd);  // close the file
int read(int fd, char *buf, int nbyte);  // read n bytes from the file
     into bufffer
int write(int fd, char *buf, int nbyte);  // append n bytes stored in
    bufffer to the file
int create(char *name, int nbyte);  // create a file, given the name
int delete(int fd);  //  delete the file
```

1.2.6 Others

- The function `printf()` is implemented, which helps to debug.
- Integer division is implemented. If divided by zero, 0 will be returned.

2 Contribution

The development of the kernel consists of two phases. During the first phase, I developed the bootloader and memory manager of the kernel by myself. During the second phase, I developed the kernel with Peijie You and Haoming Xing. Because our previous design of the kernel were quite different, the latter development was based on the work of Peijie You, and I didn't merge others' code into my code. The major work I did in second phase was development of the RR scheduler and some system calls. I submit two versions of code. One is for the first phase, and the other is for the second phase.