K-OS PROJECT PROPOSAL
CS134c, Spring 2003

Kai Chen

1 Overview

We propose to write an operating system **K-OS** (**K**ai's **O**perating **S**ystem) from scratch. **K-OS** is designed for the IA-32 Intel architecture, and will be developed and tested on machines with Intel Pentium processors. In this proposal, we carefully specify the goals we hope to achieve and describe the approach we will take. We also give a rough timeline for the completion of each of the several phases.

2 Objective

From this project, we hope to gain experience with low-level system design, interfacing with hardware and to understand concepts and details of various aspects of the operating system.

For this class, we aim for a rather complete implementation of the basic features of an operating system in a *simple* manner. We hope to design **K-OS** in such a way that it can later be extended in various directions when researchers need a simple enough system that they fully understand to support researches in different fields.

Features we plan to support for **K-OS** and the related components include:

- **Boot Sector**: **K-OS** will of course need to be able to boot itself at the very least.
- **Interrupt Handler**: Necessary for supporting I/O, system calls, and handling exceptions.
- **Memory management**: We need to at least keep track of which memory is being used and which is free. We also need to provide simple mechanisms to allocate and free memory. We also hope to support some simplistic virtual memory with paging if time permits.

- **I/O**: We need low level mechanisms to generate and handle interrupts and to read from and write to hardware ports. We will also provide highlevel I/O routines for system and application usage.
- **Process Management**: Some notion of "process" is necessary. We also hope to support some simple multi-tasking.
- **Application**: We want to be able to run applications. The kernel needs to load application into memory. The application will interface with the kernel via system calls. We might also want to implement some standard library functions for **K-OS**.
- **File System** (Optional): We hope to implement something like DOS FAT12 file system for **K-OS**.

In the next section, we describe our approach to implement each component, and give a rough timeline for each phase.

3 Approach

We have a total of 10 weeks for this project, including design, implementation and testing. The first 2 weeks were spent on researching and designing. As stated earlier, our goal here is to keep everything simple. Next we describe detailedly how we would approach each of the proposed components.

3.1 Boot Sector

The **K-OS** kernel will be loaded from a floppy. We will need to write the boot loader to the first sector (boot sector) of a floppy. After performing *POST*, BIOS will copy the boot loader into memory location $0x7C00$. From there the control is transfered to the boot loader. In the boot sector we will enable the $A20$ line to allow **K-OS** to access the entire memory under protected mode. It will also set up the Global Descriptor Table (GDP), and load rest of the kernel. Before switching to protected mode, we might want to make sure the processor is 80386 or above. We also need to keep in mind that the boot loader must be 512 bytes in size and end with $0xAA55$. We will want to display something on the screen to show progress.

We plan to finish this part by 4/25.

3.2 Interrupt Handler

We will need to set up the Interrupt Descriptor Table (IDT) with interrupt service routines. We at least need to handle keyboard interrupt, video interrupt, clock interrupt, page fault interrupt if we use paging, and interrupts for system calls. A central dispatching function will map different interrupts to the corresponding handlers. Each handler will be implemented in some high level language (likely, C) with an assembly wrapper. The assembly wrapper is necessary when we need to manipulate registers differently from C or when we need to call the IRET instruction. We will also need to initialize the Programmable Interrupt Controller (PIC) to start receiving interrupts.

We plan to set up this part by 5/2. We may add more handlers as the project evolves.

3.3 Memory Management

We need some basic memory management before we can move on. Two different schemes are necessary, one for kernel space, and one for user space. (Best illustrated with $kmalloc()/kfree()$ and $malloc()/free()$.) We hope to support limited virtual memory with paging. We need some efficient data structure to keep track of raw memory usage. The kernel space allocator will allocate memory and return physical address. The user space allocator will allocate memory from the application's memory space, which is protected by paging. If there is no free memory, it will call the kernel allocator and populate its page table with new memory.

This part can get overly complicated. We will make sure to support very basic memory management first before worrying about virtual memory. We plan to finish this part by 5/11

3.4 I/O

At the low level, we work with I/O address space. Unlike the main physical memory, here we use instructions IN and OUT to read from and write to I/O ports.

Printing on the screen is fairly straightforward. We need to keep track of the position of the cursor. The video screen is mapped to physical memory $0xb8000$. The text mode is 80 columns by 25 rows. Each character is represented by 2 bytes. One for ASCII value and one for attribute. After outputing a character, we need to advance the cursor. The screen shall scroll if the cursor passes the bottom of the screen.

Keyboard is interrupt driven. We keep a buffer which the interrupt handler can write to. When we have processes, we also need to take care of those processes that are waiting on keyboard events.

At a higher level, we might want to implement some functions similar to *printf*() to handle I/O, since we cannot use standard C libraries in **K-OS**.

We plan to finish this part by 5/18.

3.5 Process Management

Once we have some basic memory management, we may create some notion of process. A data structure is necessary to maintain the Task State Segment (TSS). With virtual memory, each process will also keep a copy of the page directory for its own memory space.

We want to support some multitasking. At low level, we will need to implement Context Switch. This normally involves adding TSS to GDT, manipulating Task Register (TR) and far jumping to a TSS selector. We also want some high level mechanisms for process synchronization. Simple primitives like semaphores should be sufficient. We need several queues for processes of different running status, and functions like *sleep*() and *wakeup*() to manipulate these queues.

For scheduling, we will use Round-Robin by default. The user can also assign priorities to processes. Processes with higher priority will run with larger quantums. Rescheduling happens either when the task uses up its quantum (caught by timer interrupt) or the task is finished (in which case the *reschedule*() function is called explicitly).

We hope to finish this part by 5/25.

3.6 Application

We want to be able to run some applications on **K-OS**. The tricky part is how to load the process into memory. We might need to implement some sort of linker/loader, but this should be kept as simple as possible.

Application programs will interface with the kernel via system calls. These are captured by interrupts. Each system call will be implemented in some higher level language with an assembly wrapper.

We will finish this part by 5/30.

3.7 File System (Optional)

We might never get to this part, but it's OK, since there is little really exciting about a toy file system and it is way too complicated to implement a real one. So we dont want to spend too much time on this part. The hope is that we can port the DOS FAT12

file system to **K-OS**. We will most likely keep our file system on a floppy. Since BIOS disk calls are not available under the protected mode, we need to minimize disk access. Probably it's a good idea to load the whole floppy image into memory when booting up.

4 Summary

With any luck we will have a working version of **K-OS** by 5/30. We might be overly ambitious about this project. Along the way some features will probably be dropped. And we might find new features more exciting, or realistic to implement. But this proposal shall serve as a pretty complete guideline for how we will approach to implement **K-OS**. The self-imposed deadlines are summarized as following:

- **4/25**: Boot Sector.
- **5/2**: Interrupt Handler.
- **5/11**: Memory Management.
- **5/18**: I/O.
- **5/25**: Process Management.
- **5/30**: Applications.
- **Optional**: File System.