ON THE DESIGN AND IMPLENTATION OF K-OS

Kai Chen

Abstract

K-OS (**K**ai's **O**perating **S**ystem), pronounced as *chaos*, is designed for the Intel IA-32 Architecture. The current version (v0.01) provides fully featured interrupt handling, supports terminal I/O and multi-tasking, implements basic virtual memory with paging and allows multiple user logins and switching between virtual desktops. In the near future it will load user applications, implement a few device drivers and a filesystem, and provide system calls, API, and an interface to the C runtime library. In this paper we describe the design and implementation of **K-OS**, and suggest future extensions.

1 Introduction

With the original goals [1] in mind, we successfully designed and implemented a reasonably sophisticated operating system from scratch. The system provides a rather complete set of features enjoyed by a real OS, and is designed in such a way that it can be easily extended and any future work will be more than pleasant. During the development we studied the source code of various existing OS' including Linux v0.01 and CHAOS [2], as well as the Intel Architecture Manuals and numerous online system resources. We gained much knowledge and experience with hardware interfacing and low-level system design and development.

In the next section we are pleased to present the design and implementation details of the current version of **K-OS**(v0.01).

[1] Please refer to *K-OS PROJECT PROPOSAL*, by *Kai Chen*, Spring 2003.

[2] CHAOS, by *Cheng Hu* and *Alex Adriaanse*, (Spring 2002) was also written for a CS134c project at California Institute of Technology. The naming was coincidental and the two systems follow rather different design philosophies, but we did get much insight from the CHAOS project, and was inspired by its success.

2 Implementation

The current version of **K-OS** is composed of a boot sector, the interrupt handling mechanism, the I/O routines, a memory manager, a task manager, and miscellaneous features such as virtual desktops and user authentication. We basically finished all of the projected milestones in the proposal (which we believed to be a rather ambitious one), and added a more sophisticated virtual memory manager and various features.

In this section we go through each of the components in detail. To help better understand the design and implementation decisions and issues, we also provide a self-contained introduction to the basics of OS development.

2.1 Boot Sector

When the machine is turned on, control is in BIOS. It first performs the Power-On-Self-Test (POST), and if nothing goes wrong, it selects a boot device (floppy, CD ROM, harddrive ...) and copies the boot sector to memory location 0x7C00. The control is then passed to the boot sector.

For the Intel Architecture, a boot sector has to be exactly 512 bytes in size, and must end with 0xAA55.

The first thing **K-OS** boot sector does is that it tests the CPU to make sure we have 80386 or above. This is done by reading bits 12 - 15 in the flags.

Next we switch to the *protected mode*. The major differences between *protected mode* and *real mode* are the segment registers and the interrupts. (We will discuss the latter in the next section.) In *real mode*, it is 16-bit addressable, and we compute the address with:

```
seg:off = seg*16 + off
```

Protected mode is 32-bit addressable. Segment registers are *selectors* that selects entries in the *Global Descriptor Table* (GDT).

In a more precise sense, segment registers are really composed of 4 registers: selector, base, limit, and attributes. In *real mode*, the register value is stored as the selector, and value*16 is stored as the base. In *protected mode*, the value is used to fetch a *descriptor*, and the *descriptor* is unpacked into the 4 registers.

Once understanding these basics, we can switch to the *protected mode*. The first thing we need to do is to set up the GDT. The GDT is a vector of 8-byte descriptors, which contains 32 bits for base, 20 bits for limit, and 12 bits for descriptor type.

In **K-OS**, the GDT has the following layout:

```
gdtr:
   dw GDT_LIMIT      ; limit = GDT size - 1
   db GDT_BASE       ; base = base address of GDT
null_sel:            ; null selector, each entry 0
   ...
code_sel:            ; code selector
   ...
data_sel:            ; data selector, also for stack segment
   ...
tss0_sel:            ; after the data selector are the selectors
   ...               ; for task state segments (tss)
tss1_sel:
   ...
...
```

We made the arbitrary decision to put GDT at memory location 0x500. And statically allocate enough memory for all the selectors. The code selector is used by the code segment, and the data selector is used by the data segment and the stack segment. After those, we have the selectors for Task State Segments (TSS). (More on this later.)

To load the GDT register, we simply call:

```
LGDT    [gdtr]
```

Next we enable the A20 address line. Gate A20 is the 21st address line. With 20 address lines we can access up to 1Mb of memory. Any reference beyond that will cause memory wrap around. A20 is ANDed with some external programmable device (the keyboard controller, to be more precise, which happens to have an extra pin available). To enable A20, we communicate with the keyboard controller at ports 0x60, 0x64.

Now we load the kernel into memory. We obtain the kernel size when compiling, and while still in *real mode*, we have access to the BIOS disk call (INT 13). We only need to set up the registers properly:

```
AH -- BIOS function. (0x02)
AL -- Number of sectors to read
ES:BX -- segment:offset, memory location
CH -- Track number
CL -- Starting sector
DH -- Head number
DL -- Drive number
```

We repeatedly read in sectors and proceed to new tracks or cylinders when necessary,

until we have loaded the whole kernel. In **K-OS**, the kernel is loaded at memory location 0x10000.

The actual code to switch to *protected mode* is straightforward, we simply need to set bit 1 of CR0:

```
MOV   EAX, CR0
OR    AL, 1
MOV   CR0, EAX
```

At this point, the registers still contain old 16-bit segment values. We flush them with the new GDT selectors, and perform several short jumps to empty the prefetched instructions. Finally, we are ready to execute the kernel:

```
JMP   CODE_SEL : KERNEL_ADDR
```

2.2 Interrupts

Here we have 3 goals:

- Set up the *Interrupt Descriptor Table* (IDT),
- Define and install the *Interrupt Service Routines* (ISR), and
- Remap and enable the *Programmable Interrupt Controller* (PIC).

The IDT is fairly similar to GDT. It's a vector of 8-byte descriptors associating interrupts with their service routines. We have at most 256 IDT entries since there are at most 256 interrupts on a PC.

Each descriptor mainly contains the address to the corresponding interrupt handler. It also contains attribute information such as a present bit, the DPL priviledge level, and a selector. Loading IDT is very similar to loading GDT:

```
LIDT   [idtr]
```

where idtr contains the base and limit of the IDT.

We define a separate service routine for each interrupt. The real handlers are implemented in C, but we provide an assembly wrapper to conveniently signal EOI and do IRET.

We have 3 highlevel handlers of interest:

- **Keyboard Interrupt Handler**: Read in the scancode, translate that into ASCII, and write the ASCII to the keyboard buffer.
- **Timer Interrupt Handler**: Decrease the preemption timeout counter, if zero, it calls the scheduler. (More on this later.)
- **Default Interrupt Handler**: This is a panic function, which display a Win98-style blue screen with the interrupt number.

Most of the low level wrappers will simply store the interrupt number and call the panic function. The code has the following structure:

```
push all registers
store interrupt number
call highlevel handler
signal EOI
pop all registers
IRET
```

We mentioned earlier that one major difference between *real mode* and *protected mode* is in interrupts.

Our machine has 16 hardware interrupts, called *Interrupt Request* (IRQ). These IRQs are divided into two blocks, each associated with a *Programmable Interrupt Controller* (PIC).

In *real mode*, IRQs 0 - 7 are mapped to INTs 08h - 0Fh, and IRQs 8 - 15 are mapped to INTs 70h - 77h.

In *protected mode*, this interferes with the software interrupts. So we need to remap PIC.

So we have two PICs. PIC0 is master, and is associated with IRQ0 - IRQ7. PIC1 is slave, and is associated with IRQ8 - IRQ15.

To program PIC, we communicate with the master and slave controllers by sending *command words*. There are *Interrupt Command Word* (ICW) and *Optional Command Word* (OCW). We briefly describe the usage of each:

- **ICW1**: specifies whether ICW4 is sent
- **ICW2**: remapping information
- **ICW3**: specifies how master and slave are connected

- **ICW4**: specifies if the CPU expects EOI
- **OCW1**: masks off unused IRQs
- **OCW2**: signals EOI

Once the PIC is enabled. The CPU starts to accept interrupts.

2.3 I/O

K-OS takes input from the keyboard and outputs to the screen.

When there is a keystroke, a keyboard IRQ is fired, and control is passed to our keyboard event handler. The handler first reads the scancode from the keyboard controller's data port. It translates it into ASCII and writes it to the keyboard buffer. We also handles breakcode to manipulate multiple keystrokes. In particular, a status word is maintained to keep track of whether Ctrl, Alt, or Shift is pressed. We also handle upper and lower cases, and even the LEDs on the keyboard.

The printing is a bit more interesting. At low level, we write directly to the video memory. In x86, the video screen is memory mapped to 0x6B000. The screen is divided into 25 rows and 80 columns. Each character is represented by 2 bytes, one for the ASCII value, and one for the attributes, which specifies the foreground and background color.

So printing onto the screen is fairly straightforward. For example, the following routine outputs a char at a specific location on the screen:

```
void kputc(char c, int row, int col, unsigned char attr)
{
   unsigned char *vidmem = (unsigned char *) 0x6B000;
   int pos = row*80*2 + col*2;

   vidmem[pos] = c;
   vidmem[pos+1] = attr;
}
```

Of course we are not satisfied with such primitive printing routines. We would like to have a kernel version of the printf function. By Linux convention, let's call it printk:

```
   printk(char *fmt, ...)
```

For a variable argument function like this, we need to figure out the pointer to the first argument after *fmt.

Once we have the argument list, the real printing is carried out by the function:

```
vprintk(char *fmt, void *args)
```

Right now we support 4 formats: signed decimal (%d), unsigned hexadecimal (%x), string (%s), char (%c).

To compute the argument list, Linux uses a set of macros:

```
va_start, va_end, va_list ...
```

K-OS takes a simplified approach by using the following piece of assembly code:

```
printk:
   PUSH       EBP
   MOV        EBP, ESP
   MOV        EAX, EBP       ; EBP points to EBP
   ADD        EAX, 12        ; EBP+4 points to return addr
   PUSH       EAX            ; EBP+8 points to the first arg *fmt
   PUSH dword [EBP+8]        ; EBP+12 points to the second arg
   CALL       vprintk
   MOV        ESP, EBP
   POP        EBP
   RET
```

Also, at low level, the cursor is nothing more than a flashing spot on the screen. We would like it to indicate the position the next char is printed to. Hence we need to manipulate the cursor. We do this by sending a command to CRT_CTRL register (0x03D4), and then read the address from or write the address to CRT_DATA register (0x03D5). Finally we translate the address into a position on the screen:

```
addr = row*80 + col;
```

2.4 Memory Management

K-OS supports virtual memory with paging. We first review the basics of paging.

A page directory is a vector of 4-byte page table specifiers, each taking up to 4Kb to store. A page table is a vector of 4-byte page specifiers. One page table maps 4Mb of memory and takes up to 4Kb to store. A page frame is 4Kb of contiguous memory. The starting address of a page frame must be 4Kb aligned, which means the low 12 bits are all zero.

A page directory entry or page table entry may contain a pointer to a physical page. Since page frames are 4Kb aligned, it only takes 20 bits to specify an address. The low 12 bits in a PDE or PTE are thus used to store attribute information. Most interesting are the present bit and the permission bits.

The following example shows how to set up paging. It puts the page directory at 0x70000 and the page tables right after the page directory. It also identity maps first 4Mb of the memory. The actual setup in **K-OS** is a bit more complicated, which we will discuss later:

```
void setup_paging()
{
   Unsigned long * page_dir = (unsigned long *) 0x70000;
   unsigned long * page_table = (unsigned long *) 0x71000;
   unsigned long addr = 0;

   for(i=0; i<1024; i++){
      page_table[i] = addr |  3;
      addr += 4096;
   }
   page_dir[0] = page_table | 3;

   for(i=0; i<1024; i++){
      page_dir[i] = 0 | 2;
   }
}
```

In **K-OS**, physical memory is organized in terms of pages. A byte-vector `page_use_count` is used to keep track of the usage of the pages.

We chose to put the page directory at 0x70000, and the page tables follow the page directory. We also chose to put the kernel heap at 0x100000.

We played a nice trick here to point the last entry of the page directory back to itself. In this way, the page directory looks like a page table to the CPU, and a page table looks like a normal page. Hence we can refer to the page directory and page tables themselves via virtual memory:

```
PAGE_SELF = 1023
PAGE_DIR_VADDR = (PAGE_SELF<<22) | (PAGE_SELF<<12)
PAGE_TABLE_VADDR = (PAGE_SELF << 22)
```

Next we go through the components of the memory manager.

At the lowest level are the page allocator and deallocator. The page allocator looks for a free page in the `page_use_count` vector. Once it finds one, it increments the count and returns the physical address of the page. Deallocation is the natural opposite.

We also need a memory map function to map physical memory to virtual address. It basically computes the page table index and insert the physical address into the page table:

```
mmap(vaddr, paddr, attr)
{
   pde = vaddr >> 22;
   pte = vaddr >> 12;

   check page_dir[pde] present;
   page align paddr (so we dont corrupt attribute bits);

   page_table[pte] = paddr;
}
```

Our low level allocator is the `morecore` function. It allocates a page, performs memory map, and returns the virtual address of the allocated memory. Here memory are allocated in 4Kb blocks. Also, there is no corresponding deallocator, since we do not allow downsizing the kernel heap.

```
morecore(int size)
{
   paddr <- allocate enough pages;
   move along kernel heap to get next available vaddr;
   mmap(vaddr, paddr, attr);
   return vaddr;
}
```

Our highlevel allocator is the familiar `kmalloc`. Here memory is organized in a linked list of blocks. Each block contains a *header* which contains information about the size of the block, whether it is being used, and also a pointer to the next block.

```
struct header {
   size_t size;
   struct header *next;
   unsigned magin : 31;
   unsigned used : 1;
}
```

Upon request, `kmalloc` first goes through the list of blocks to check if there is a free block that is large enough. If so, it splits the block into two to save extra memory, marks the used block and returns its address. If not found, it calls `morecore` to allocate a new block first.

```
kmalloc(size_t size)
{
   look for a free block that is large enough;

   if(found){
      split block into two and save extra memory;
      mark and return the block with right size;
   }

   if(not found){
      block = morecore();
      split block into two and save extra memory;
      mark and return the block with right size;
   }
}
```

Our deallocator `kfree` does the opposite. It marks the block as unused and combines the adjacent free blocks to reduce fragmentation:

```
kfree(void *p)
{
   find the block in the list;
   return error if not found;
   mark block as free;
   combine adjacent free blocks;
}
```

In summary, in the allocation, we allocate physical pages, map them to virtual address, and then reorganize pages of virtual memory into a linked list of finer blocks. In deallocation, we mark a block as free and combine the adjacent free blocks.

2.5 Task Management

Here we discuss the notion of process, context switches, and scheduling in **K-OS**.

In x86, the notion of process is captured by the *Task State Segment* (TSS). A TSS has the following structure:

```
struct tss {
   unsigned short back_link, reserved0;
   unsigned long esp0;
   unsigned short ss0, reserved1;
   unsigned long esp1;
   unsigned short ss1, reserved2;
   unsigned long esp2;
   unsigned short ss2, reserved3;
   unsigned long cr3, eip, eflags;
   unsigned long eax, ecx, edx, ebx, esp, ebp, esi, edi;
   unsigned short es, reserved4;
   unsigned short cs, reserved5;
   unsigned short ss, reserved6;
   unsigned short ds, reserved7;
   unsigned short fs, reserved8;
   unsigned short gs, reserved9;
   unsigned short ldt, reserved10;
   unsigned short tflag, iomap;
}
```

Basically entries in TSS reflects the initial state of the registers of a task. Before any context switch, we at the very least need to set up the following:

- CR3: set to the kernel page directory, until we create a separate address space for each task.
- EIP: set to the entry point of the task.
- Segment Registers: CS set to the code selector; DS, ES, FS, GS, SS set to the data selector.
- ESP: set to the top of the stack we allocated for the task, double word aligned.

Once the TSS is properly set up, we insert it into the GDT. Each GDT entry for TSS is just like a normal GDT entry. It contains the base address of the TSS, its limit, and type and protection informations. A normal TSS has type 9, and a busy TSS has type 11. The TSS entries come right after the code descriptor. In the current version, **K-OS** statically allocates 16 tasks so we have 16 TSS descriptors in the GDT.

After installing the TSS in the GDT, we can perform a context switch. We simply do a far jump to the TSS selector. For example, TSS0 has selector 0x18, TSS1 is 0x20 ...

In **K-OS**, the task structure contains its TSS, a TSS selector, a status word, and a task pointer for queue implementations:

```
struct task_t {
   tss_t tss;
   unsigned short tss_sel;
   enum{
      TASK_RUNNING = 0,
      TASK_INTERRUPTIBLE = 1,
      TASK_UNINTERRUPTIBLE = 2,
      TASK_NON_EXIST = 3
   } status;
   struct task_t *next;  // for runqueue, wait_queue
}
```

The current scheduler implements a simple round-robin algorithm. We have a timeout counter for preemption. In the timer interrupt, the counter is decreased and if it reaches 0, the scheduler is invoked. The scheduler simply performs a context switch to the next task in the runqueue.

The current version has some issues in context switches, we will discuss more about this in a later section.

2.6 Miscellaneous

K-OS implements 4 virtual desktops. Each VD has its own command buffer, so this creates 4 separate work spaces for the user. The virtual desktops will become increasingly useful when we have multiple tasks running.

The shell runs as a separate thread. It constantly reads bytes in from the keyboard buffer, writes them to the current virtual desktop's command buffer, and interprets the command.

We also provide basic user authentications. Of course this will only be useful when we have a real filesystem. In the current version, on start up, the user should log in as root. The root can then create new users and delete existing users (except for the root itself). Normal users do not have such priviledges, but have access to all other features of the system.

3 Known Bugs

K-OS is a fairly stable system. But we are having some issues with the task manager. At this point, the context switch generates a general protection fault [3]. According to the

[3]Before, we also got a CPU triple fault in some instances. As this document was being written, David

Intel Architecture Manual Volumn III, a general protection fault (#GP) occurs during task switch when:

- The TSS selector does not reference the GDT or it's not within the limit of the GDT.
- The TSS descriptor is busy.

In theory neither of these should have happened. We suspect this is due to some overlooked technical details, and plan to write a simple kernel debugger to dump memory, stack and register values.

4 Extensions

K-OS was designed in such a way that it can be easily extended. Each of the existing features has a pretty complete implementation. For instance, the keyboard module handles any combination of keystrokes people will ever want to use, although many of them are not being used in the current version. The standard functions such as `printk` and `kmalloc` greatly ease the rest of the development. And we also have a decent support for the string library.

In the near future we plan to provide an interface to the C runtime library. In particular, we will implement the **K-OS** version of `printf`, `scanf`, `malloc`, `free`, and the string operations.

We also want to support several system calls. Since we have a full support for the interrupts, and already have most of the utility functions, this part should be relatively straightforward.

We already have an implementation for the DOS FAT12 file system, which can be ported to **K-OS** fairly easily. However, since we do not have access to the BIOS disk functions under *protected mode*, we need to implement our own disk driver first. The other option is to use direct hardware access, but that is painful and does no good in the long run.

Given the support of system calls, the C runtime library, and a file system, we can start to write and load applications for **K-OS**. From there it's not far away from being a real world operating system.

Moore noticed a typo in the code which solved that problem.

5 Conclusion

The **K-OS** project has been a great success. In 8 we weeks designed and implemented an operating system with most of the basic features a modern OS enjoys. And the design allows a fairly sustainable development. During the course of the project, we obtained tons of knowledge in system design, and gained much experience with low level development. It has been a lot of fun working on this project. This report is not a final one, and there will never be a final one, since the work will continue.

Acknowledgements

I would like to thank Professor Jason Hickey for teaching me this material, and would like to thank David Moore, Justin Smith, and Christian Tapus for their help and instructions on this project. I'm also grateful to everyone who have used **K-OS**, provided comments and suggestions, or showed interest in the project.