FBOS 内核剖析

作者：小虾(xiaoxia.org) 2009-9-13

目录

一、FBOS 的编译与运行

首先，通过 SVN 下载 FBOS 的源代码，自带的 make.txt 是不能编译的 fbos 的，发生很多奇妙的错误，在此不再详述。

然后，我通过修改 makefile 的内容，把 cc，as，ld 都设为我自己的编译器，然后在 fbos 根目录输入 make 编译成功，得到 kernel.bin。

最后，把 kernel.bin 导入一个装有 grub 的软盘，引导后运行。但可惜的是，笔者没有运行成功，GRUB 提示 unsupported format。注意：grub 引导是首先把内核加载到 0x100000 内存地址的，另外根据 multiboot 头部信息，再加载到指定位置。

二、引导

FBOS 的引导部分在 boot 文件夹。

首先查看 boot.s，代码有点眼熟。通过查看 main 处的代码，可以知道一些信息。

```
main:
    ;// set the stack
    movl $STACK_TOP, %esp
    ;// get memory size
    call _memory_get
    ;// init the kernel
    push %eax
    call _kinit
    addl $4, %esp
    ;// sti now
    sti
    ;// call main
    call _kmain
loop:
    hlt
    jmp loop
```

首先是设置堆栈，设置堆栈后就可以无情地 call 和 ret 了。FBOS 首先 call 了 memory_get，这个应该是获取内存大小的，然后把返回值入栈，调用 kinit 对内核进行初始化。初始化完成后，就开中断，调用 kmain。kmain 估计是不返回的了，因为返回的话就 hlt 停机了。

三、内核进行初始化

查看 init.c 文件内容:

```
void kinit(unsigned int magic_num)
{
    display_clear();
    if (magic_num != 0x2badb002)
    {
        kprintf("%s", 0xf, "Kernel isn't boot from multiboot loader.\n");
        cli();
        hlt();
    }
```

```
    else
    {
        set_gdt();
        set_idt();
        set_start_time();
        set_paging();
        set_irq();
        set_clock();
    }
}
```

首先是清屏，然后判断是否从 grub 引导，否则停机。因为 grub 已经关了中断，所以这里的 cli 估计是多余的，可能是为了装饰或凑字数。因此，笔者在此提醒读者，以免 FB 企图误导前途无量的各位的阴谋的得逞。

接着，设置 gdt，idt 表，获取开机时间，开启内存分页，设置 irq 硬件中断，再设置时钟频率，开启时钟。

四、i386 的初始化

设置 gdt 和设置 idt 等都是在 dt 目录下实现的。dt 就是 descriptor table，而不是地图，描述符表包括 global descriptor table 和 interrupt descriptor table 两种。

首先看 gdt.c 的代码：

```
void set_gdt(void)
{
    // zero memory first
    memset(gdt, 0, GDT_MAX_COUNT * sizeof(struct gdt_entry));
    // null gdt, reserved for Intel
    set_gdt_gate(0, 0, 0, 0);
    // code
    set_gdt_gate(1, 0xffff, 0x9a, 0xcf);
    // data
    set_gdt_gate(2, 0xffff, 0x92, 0xcf);
    // set the limit and base
    pgdt.limit = GDT_MAX_COUNT * sizeof(struct gdt_entry) - 1;
    pgdt.base = (int)gdt;
    // load gdt
    __asm__ __volatile__ ("lgdt %0": "=m" (pgdt));
}
```

首先内存清 0 是必要的，保持 gdt 表的干净和无比圣洁。然后设置了一个 null gdt，这是 intel 建议设置的，另外也是我建议设的，还有一个代码段和一个数据段，是给内核使用的。接着就加载设置好的 gdt 表了。

下面看 idt.c 的代码：

```
void set_idt(void)
{
    // zero memory first
    memset(idt, 0, IDT_MAX_COUNT * sizeof(struct idt_entry));
    // now set trap gates
    set_idt_gate(0, exception0, 0x8, 0x8E);
    set_idt_gate(1, exception1, 0x8, 0x8E);
    set_idt_gate(2, exception2, 0x8, 0x8E);
    set_idt_gate(3, exception3, 0x8, 0x8E);
    set_idt_gate(4, exception4, 0x8, 0x8E);
    set_idt_gate(5, exception5, 0x8, 0x8E);
    set_idt_gate(6, exception6, 0x8, 0x8E);
    set_idt_gate(7, exception7, 0x8, 0x8E);
    set_idt_gate(8, exception8, 0x8, 0x8E);
    set_idt_gate(9, exception9, 0x8, 0x8E);
    set_idt_gate(10, exception10, 0x8, 0x8E);
    set_idt_gate(11, exception11, 0x8, 0x8E);
    set_idt_gate(12, exception12, 0x8, 0x8E);
    set_idt_gate(13, exception13, 0x8, 0x8E);
    set_idt_gate(14, exception14, 0x8, 0x8E);
    set_idt_gate(16, exception16, 0x8, 0x8E);
    // set the limit and base
    pidt.limit = IDT_MAX_COUNT * sizeof(struct idt_entry) - 1;
    pidt.base = (int)idt;
    // load idt
    __asm__ __volatile__ ("lidt %0": "=m" (pidt));
}
```

还真长呢！ 这里在 idt 表上设置了 17 个异常处理函数，对应 idt 向量 0 到 16，然后加载之。

还是顺着启动流程看代码吧，接着到了 kernel 下的 time.c，

```
void set_start_time(void)
{
    // zero memory
    memset(&start_time, 0, sizeof(struct time_entry));
    // read cmos to get current time
    do
    {
        start_time.t_second = cmosread(0x0);
        start_time.t_minute = cmosread(0x2);
        start_time.t_hour = cmosread(0x4);
        start_time.t_day = cmosread(0x7);
```

```
        start_time.t_month = cmosread(0x8);
        start_time.t_year = cmosread(0x9) + cmosread(0x32) * 100;
    } while (start_time.t_second != cmosread(0x0));
    // display the start time
    kprintf("Start time: %d.%d.%d %d:%d:%d\n", 0xf,
            start_time.t_year,
            start_time.t_month,
            start_time.t_day,
            start_time.t_hour,
            start_time.t_minute,
            start_time.t_second);
}
```

从代码很容易得知 FBOS 是利用 BIOS 得到当前时间，然后打印出来。while (start_time.t_second != cmosread(0x0))是为了保持获取的时间的精确度，即你获取完毕后，电脑时间不能变为下 1 秒了，要与获取了的一致。

接着看 mm 里的 page.c 的 set_paging（）：

```
// enable paging and other works for memory
void set_paging(void)
{
    int *dir = (int *)PAGE_DIR; // point to the page dir
    int *table = (int *)PAGE_TABLE; // point to the page table
    int i;

    // check memory first
    check_memory();
    // set memory map point
    mem_map = (short *)PAGE_MAP;
    // zero memory
    memset(table, 0, 0x100000);

    // set PTE & PDE
    for (i = 0; i < LOW_MEM / PAGE_SIZE; i++) // fill kernel page tables
        table[i] = (i * PAGE_SIZE) | 7;
    for (i = LOW_MEM / PAGE_SIZE; i < MEM_SIZE / PAGE_SIZE; i++) // fill
remaining page tables
        table[i] = (i * PAGE_SIZE) | 6;  // present = 0
    for (i = 0; i < LOW_MEM / PAGE_SIZE; i++) // fill kernel page tables
        table[i] = (i * PAGE_SIZE) | 7;
    for (i = 0; i < MEM_SIZE / PAGE_SIZE / 1024; i++) // fill used page dirs
        dir[i] = (PAGE_TABLE + PAGE_SIZE * i) | 7;
    for (i = MEM_SIZE / PAGE_SIZE / 1024; i < 1024; i++) // fill remaining page dirs
        dir[i] = 6; // present = 0
```

```
    // mark memory free
    memset_short(mem_map, PAGE_FREE, MEM_SIZE / PAGE_SIZE);
    // set maps
    for (i = 0; i < LOW_MEM / PAGE_SIZE; i++)
        mem_map[i] = PAGE_USED;

    // at last, load cr3 and set the PG WP bit of cr0
    __asm__ __volatile__ ("movl %%eax, %%cr3\n\t"
                          "movl %%cr0, %%eax\n\t"
                          "orl $0x80000000, %%eax\n\t"
                          "movl %%eax, %%cr0"::"a"(PAGE_DIR));
}
```

这里首先调用了 check_memory()，好像是检查一下内存对齐和大小，如果内存小于 16MB，就停机，如果大于 256MB，就 bark 一下。

接着设置新的页目录和页表，加载之。 mem_map 是记录哪个页面被使用，哪个空闲。一开始的时候，设置 LOW_MEM 部分的内存为使用，这些是供内核使用的。所以动态分配的是大于 LOW_MEM 的内存。

下面看 dt 下的 irq.c 的 set_irq():

```
void set_irq(void)
{
    // reset i8259a first
    reset_i8259a_irq();
    // zero irqs
    memset(irq, 0, IRQ_MAX_COUNT * sizeof(void *));
    // set the idt
    set_idt_gate(IRQ_START_IN_IDT + 0, irq0, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 1, irq1, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 2, irq2, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 3, irq3, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 4, irq4, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 5, irq5, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 6, irq6, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 7, irq7, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 8, irq8, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 9, irq9, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 10, irq10, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 11, irq11, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 12, irq12, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 13, irq13, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 14, irq14, 0x8, 0x8E);
    set_idt_gate(IRQ_START_IN_IDT + 15, irq15, 0x8, 0x8E);
}
```

首先重新映射 i8259a 芯片。然后设置新的硬件中断映射，这样就确保了硬件中断发生时，传送的中断号是 IRQ_START_IN_IDT 之后的，因为之前的要保留给 Intel CPU 异常等使用。

最后看 kernel/clock.c 的

```
void set_clock(void)
{
    set_irq_routine(0, clock_handler);
}
```

这里是设置一个时钟中断的捕捉句柄。当中断发生时，会调用 clock_handler。至于 set_irq_routine 如何处理就不再分析了。

五、内核工作

话说，kinit 完成后就跳入 kmain。kmain 里做的似乎暂时都是测试的函数。例如分配一个内存，打印一些信息，然后打印内存的分配信息。最后进入死循环，企图耗尽 CPU，使用电量最大化，让使用者因交巨额电费而倾家荡产，最终使电力资源枯竭，整个国家进入瘫痪状态。

BTW，那个内存分配使用的是链表，效率应该还不错，赞一下 FB！

六、剖析结束

提醒读者，剖析结束。

七、参考文献

无。100%原创！

八、FB 剖析

FB 是神牛！