一、内核初始化

（1）initSerial()

（2）initIdt()

初始化 IDT 表，为中断设置中断处理函数

```
.global vector0
vector0:   pushl $0;  pushl    $0; jmp asmDoIrq
.global vector1
vector1:   pushl $0;  pushl    $1; jmp asmDoIrq
.global vector2
vector2:   pushl $0;  pushl    $2; jmp asmDoIrq
.global vector3
vector3:   pushl $0;  pushl    $3; jmp asmDoIrq
.global vector4
vector4:   pushl $0;  pushl    $4; jmp asmDoIrq
.global vector5
vector5:   pushl $0;  pushl    $5; jmp asmDoIrq
.global vector6
vector6:   pushl $0;  pushl    $6; jmp asmDoIrq
.global vector7
vector7:   pushl $0;  pushl    $7; jmp asmDoIrq
.global vector8
vector8:              pushl    $8; jmp asmDoIrq
.global vector9
vector9:   pushl $0;  pushl    $9; jmp asmDoIrq
.global vector10
vector10:             pushl   $10; jmp asmDoIrq
.global vector11
vector11:             pushl   $11; jmp asmDoIrq
.global vector12
vector12:             pushl   $12; jmp asmDoIrq
.global vector14
vector14:             pushl   $14; jmp asmDoIrq
.global vector16
vector16:  pushl $0;  pushl   $14; jmp asmDoIrq
```

```
);
_KERN);
_KERN);
_KERN);
_KERN);
_KERN);
_KERN);
_KERN);
_KERN);
L_KERN);
L_KERN);
L_KERN);
ault, DPL_KERN);
L_KERN);
PL_KERN);

 DPL_USER); // for int 0x80, interrupt
```

（3）initIntr()

（4）initSeg()

初始化 GDT 表。为了之后写现存输出，添加视频段门描述符

初始化 TSS。tss.ss0=KSEL(SEG_KDATA)，tss.esp=0x1ffff0，并使用 ltr 指令

初始化段寄存器。将 cs、ds 赋值为 KSEL(SEG_KDATA)

（5）loadMain()

加载用户程序至内存，方法和加载内核相似

完成用户程序加载后，调用 enterUserSpace()函数进入用户程序。

在 enterUserSpace 中，sti 开中断，依次压栈用户态 ss、esp、eflags、cs、eip，通过 iret 指令设置各寄存器，进入用户态。

```
void enterUserSpace(uint32_t entry) {
        /*
         * Before enter user space
         * you should set the right segment registers here
         * and use 'iret' to jump to ring3
         */
        clr_screen();
        asm volatile("sti");
        asm volatile("push %0"::"r"(USEL(SEG_UDATA)));
        asm volatile("push %0"::"r"(0x1000000));
        asm volatile("pushfl");
        asm volatile("pushl %%eax"::"a"(USEL(SEG_UCODE)));
        asm volatile("pushl %%eax"::"a"(entry));
        asm volatile("movw %%ax, %%ds"::"a"(USEL(SEG_UDATA)));
        asm volatile("iret");
}
```

二、加载内核至内存

基本思路为：读取其 ELF文件头以及程序头，设置正确的内存加载地址以及跳转地址。

具体实现方式为定义两个指针变量 ELFHeader *elf，ProgramHeader *ph，elf 指向 kMain.elf 文件头，根据 elf->phoff 可以得到第一个程序头的位置，用 ph 指向它。遍历所有程序头，对于 type 为 PT_LOAD（也就是 1）的程序头，将相对 ELF 文件头偏移 ph->off 开始 ph->filesz

```
#include "boot.h"

#define SECTSIZE 512

void bootMain(void) {
        /* 加载内核至内存，并跳转执行 */
        struct ELFHeader *elf=(void *)0x200000;
        struct ProgramHeader *ph,*eph;
        for(int i=1;i<=200;i++){
                readSect((void *)(0x200000+(i-1)*512),i);
        }
        ph = (void *)elf + elf->phoff;
        eph = ph + elf->phnum;
        for(;ph < eph;ph++){
                if(ph->type == 1){
                        for(int i=0;i<ph->filesz;i++)
                                *(unsigned char *)(ph->vaddr+i) = *(unsigned
char*)(0x200000+ph->off+i);
                        for(int i=ph->filesz;i<ph->memsz;i++)
                                *(unsigned char *)(ph->vaddr+i) = 0;
                }
        }
        void (*entry)(void);
        entry=(void *)elf->entry;
        entry();

}
```

字节的内容，拷贝到内存地址 ph->vaddr 开始的区域，并将内存地址区间[ph->vaddr+filesz，ph->vaddr+ph->memsz-1]的内容清零。装载完成后跳转至 elf->entry 位置执行。

三、printf 实现

实现 printf 前，自定义 0 号系统调用 puts，用于向屏幕输出字符串，约定 ecx 中存放待输出字符串地址，edx 中存放输出字符串长度。对于每个字符，puts 调用 printc 函数通过写显存实现屏幕输出。全局变量 vp 中存放下一次输出的起始位置。

```
}
void printc(unsigned char ch){
        if(ch=='\n'){
                vp = (vp/160+1)*160;
                return;
        }
        unsigned short tmp = ch+0x0c00;
        asm volatile ("movl %%ebx, %%edi"::"b"(vp));
        asm volatile ("movw %0, %%ax"::"a"(tmp));
        asm volatile ("movw %ax, %gs:(%edi)");
        vp+=2;
}
int puts(struct TrapFrame *tf){
        asm volatile ("movw %%ax, %%gs"::"a"KSEL(SEG_VIDEO));
        int len=0;
        for(int i=0;i<tf->edx;i++){
                if(*(char *)(tf->ecx+i)!='\0'){
                        printc(*(char *)(tf->ecx+i));
                        len++;
                }
                else break;

        }
        return len;
}
```

printf 实现（仅实现%d、%s、%c、%x）：

函数原型 void printf(const char*format，…)

由于函数调用时参数是从右向左依次压栈，因此 format 所在的地址+4 即是第二个参数所在地址。基本思路是先申请 buf 数组用于存放待输出字符串，扫描 format，每遇到%就将此前的字符串内容拼接至 buf 尾，解析%后的字符，用相应的方式将参数转化为字符串拼接至 buf 尾。format 解析完成后陷入中断通过系统调用输出 buf 至屏幕。

itoa、hextoa、strcat 为自编函数，分别实现十进制整数转字符串、十六进制数转字符串、字符串拼接。

```
void printf(const char *format,...){
        uint32_t addr = (uint32_t)&format;addr+=4;
        uint32_t start=0;
        uint32_t i=0;
        for(int j=0;j<4096;j++)
                buf[j]='\0';
        while(format[i]!='\0'){
                if(format[i]=='%'){
                        if(i!=start){
                                const_strcat(buf,format+start,i-start);
                        }
                        if(format[i+1]=='s'){
                                strcat(buf,(char *)(*(uint32_t *)addr),0);
                                addr+=4;
                        }
                        else if(format[i+1]=='c'){
                                char tmp[2];
                                tmp[0]=*(uint32_t *)addr;
                                tmp[1]='\0';
                                strcat(buf,tmp,0);
                                addr+=4;
                        }
                        else if(format[i+1]=='d'){
                                itoa(*(int *)addr);
                                strcat(buf,int_res,0);
                                addr+=4;
                        }
                        else if(format[i+1]=='x'){
                                hextoa(*(uint32_t *)addr);
                                strcat(buf,hex_res,0);
                                addr+=4;
                        }
                        start=i+2;
                        i+=2;
                }
                else i++;
        }
        if(start<i){
                const_strcat(buf,format+start,i-start);
        }
        syscall(PUTS,0,(uint32_t)buf,4096,0,0);
}
```