北京郵電大學

BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Beijing University of Posts and Telecommunications

OS Lab report

soOS 操作系统内核报告

Author:
管昇、周煊、万帅兵

Supervisor:
赵方

2024 年 6 月 29 日

目录

第 1 章　soOS 环境与项目组织

本章介绍 soOS 所支持的硬件平台与构建、运行，并介绍开发过程中使用到的部分工具。

1.1　操作系统平台

soOS 底层代码依赖 Intel® i386 系列 CPU，能够运行在兼容 Intel® i386 的系列机上。在实际开发中，采用 qemu 开源工具模拟 Intel® i386 裸机。注意，soOS 仅实现了单核 cpu 工作。

1.2　构建与运行

构建与运行仅介绍 x86 或 x86_64 位 Linux 平台下操作。

1.2.1　前置条件

1. 确保已安装 gcc 编译器。对于 DEB 系列包管理（下文同），可使用 `sudo apt install gcc` 进行 gcc 工具链的下载。对于非 x86 或 x86_64 位平台，需要安装交叉编译环境。

   此外，还会使用到 GNU binutils 工具中的 objcopy，dd 等工具。一般情况下，Linux 平台已预安装 GNU binutils 工具。

2. 构建依赖 Make 工具，使用 `sudo apt install make` 进行 Make 工具的安装。

3. 为能在 Linux 平台模拟 Intel® i386 裸机，可使用 `sudo apt install qemu` 进行 qemu 模拟器的安装。

1.2.2　构建

1. 由于涉及到文件系统，确保在构建前创建了/mnt/t 目录，Makefile 中将会借助该目录实现文件系统的挂载。使用 `sudo mkdir /mnt/t` 创建该目录。

2. 在项目目录下，执行 `make` 命令，将会在当前目录下生成 `build` 和 `image` 文件夹，在 `build` 文件夹下为编译后的各二进制目标代码。在 `image` 目录下为生成的包含 boot 以及操作系统代码的磁盘镜像。

1.2.3 运行

执行 `make run` 命令，将会启动 qemu i386 系列虚拟机，并加载 image 目录下的磁盘镜像文件 disk.img，进入 soOS 操作系统。（由于 `make run` 命令依赖了编译命令，此步命令可无需事先执行 `make` 命令）

1.2.4 修改源码再次运行

若需修改源码再次运行，请确保 Makefile 完成了正确的修改，并建议在下次执行 `make` 或 `make run` 前执行 `make clean`，该命令将删除 build 以及 image 文件夹。

1.3 调试

若需进行代码调试，执行 `make debug` 命令，将会启动 qemu i386 系列模拟器，加载 image 目录下的磁盘镜像，并暂停 cpu 执行。

此时需在另一终端同目录下执行 `gdb` 命令，启动 gdb 程序。项目目录下包含.gdbinit 文件，定义了 gdb 执行远程连接 qemu 以及加载操作系统代码符号信息脚本。gdb 启动后将会根据.gdbinit 中的命令开启远程调试 qemu 模拟器中的代码，继续 cpu 执行，并在.gdbinit 中给定的断点停止执行代码。此时可以使用正常的 gdb 命令进行断点调试，查看变量等操作。

1.4 项目组织与 Makefile 文件

1.4.1 ./

项目根目录下的文件主要为 head.S，main.c，Makefile，.gdbinit，.gitignore，hello.txt。

其中，head.S 将被编译放与为内核代码最开始部分，以 AT&T 语法标准编写的 x86 汇编代码，使用汇编代码确保在引导阶段能够正确跳转到操作系统代码。

main.c 为操作系统主函数，将调用其它函数完成操作系统内存分配，文件系统管理等初始化工作。

.gitignore 用作 git 管理忽略不必要的二进制文件。

hello.txt 将被用作文件系统的简单测试。

Makefile，.gdbinit 如前所述。

1.4.2 include

include 目录包含了所有的头文件。主要包括数据结构定义，函数声明，宏定义等信息。在 Makefile 编译命令选项中已添加 gcc 编译命令`-Iinclude`。

1.4.3 lib

lib 目录包含了在开发过程中可能用到的工具，如 string.c 等。

1.4.4 boot

boot 目录下为 boot.S，用作 boot 引导程序。此外，还将用作文件系统的实现部分。

1.4.5 int

int 目录下为中断处理相关代码，实现 32 位模式下的中断处理例程。

1.4.6 gdt

gdt 目录用于实现进入 32 位模式下 gdt 的重新加载。

1.4.7 mm

mm 目录为内存管理 (Memory Management) 模块，主要负责物理内存和虚拟内存的管理。

1.4.8 task

task 目录实现任务调度模块，负责任务的创建与调度。

1.4.9 fs

fs 目录实现文件系统模块。包含 FAT 文件系统，VFS 虚拟文件系统实现。

1.4.10 disk

disk 目录实现磁盘操作，作为文件系统的底层依赖。

1.4.11 isr80h

作为 80h 号中断即用户中断的处理函数，以实现各种系统调用功能。

1.4.12 Makefile

Makefile 文件集成项目构建所需的命令。内容如下：

```
1 CC=gcc
2 CPPFLAGS=-Iinclude
3 CFLAGS=-g -c -O0 -m32 -MMD $(CPPFLAGS) -fno-pic -fno-pie -fno-stack-protector -fno-builtin -
      nostdlib -nostdinc -nodefaultlibs -nostartfiles
```

```
BOOT_ENTRY=0x7c00
BOOT_LDFLAGS=-m elf_i386 -Ttext $(BOOT_ENTRY)

KERNEL_ENTRY=0x100000
KERNEL_LDFLAGS=-m elf_i386 -Ttext $(KERNEL_ENTRY)

KERNEL_OBJS=build/head.o build/main.o \
build/int/int.o build/int/interrupt.o \
build/lib/print.o build/lib/string.o \
build/mm/page.o build/mm/paging.o \
build/mm/heap.o build/disk/disk.o \
build/fs/path.o build/fs/vfs.o \
build/fs/fat16/fat16.o build/gdt/gdt.o \
build/gdt/gdt_c.o build/task/load_tss.o \
build/task/tss.o


.PHONY:all

all:builddir imagedir  imagefile
-include **/*.d
builddir:
    mkdir -p build
    mkdir -p build/boot
    mkdir -p build/int
    mkdir -p build/lib
    mkdir -p build/mm
    mkdir -p build/disk
    mkdir -p build/fs
    mkdir -p build/fs/fat16
    mkdir -p build/gdt
    mkdir -p build/task
imagedir:
    mkdir -p image
imagefile:./build/boot/boot.bin ./build/kernel.bin
    dd if=/dev/zero of=image/disk.img bs=16M count=1
    dd if=./build/boot/boot.bin of=image/disk.img conv=notrunc
    dd if=./build/kernel.bin of=image/disk.img conv=notrunc seek=1
    # Copy a hello.txt into the disk image
    sudo mount -t vfat ./image/disk.img /mnt/t
    sudo cp ./hello.txt /mnt/t

    sudo umount /mnt/t
./build/boot/boot.bin: boot/boot.S
    gcc $(CFLAGS) -o build/boot/boot.o boot/boot.S
    ld  $(BOOT_LDFLAGS) -o build/boot/boot.elf build/boot/boot.o
    objcopy -O binary build/boot/boot.elf build/boot/boot.bin
./build/kernel.bin:$(KERNEL_OBJS)
```

```
52      ld $(KERNEL_LDFLAGS) -o build/kernel.elf $(KERNEL_OBJS)
53      objcopy -O binary build/kernel.elf build/kernel.bin
54  ./build/head.o:head.S
55      gcc $(CFLAGS) -o $@ $<
56  ./build/main.o:main.c
57      gcc $(CFLAGS) -o $@ $<
58  ./build/int/int.o:int/int.S
59      gcc $(CFLAGS) -o $@ $<
60  ./build/int/%.o:int/%.c
61      gcc $(CFLAGS) -o $@ $<
62  ./build/lib/%.o:lib/%.c
63      gcc $(CFLAGS) -o $@ $<
64  ./build/mm/%.o:mm/%.c
65      gcc $(CFLAGS) -o $@ $<
66  ./build/mm/paging.o:mm/paging.S
67      gcc $(CFLAGS) -o $@ $<
68  ./build/disk/%.o:disk/%.c
69      gcc $(CFLAGS) -o $@ $<
70  ./build/fs/%.o:fs/%.c
71      gcc $(CFLAGS) -o $@ $<
72  ./build/fs/fat16/%.o:fs/fat16/%.c
73      gcc $(CFLAGS) -o $@ $<
74  ./build/gdt/gdt.o:gdt/gdt.S
75      gcc $(CFLAGS) -o $@ $<
76  ./build/gdt/%.o:gdt/%.c
77      gcc $(CFLAGS) -o $@ $<
78  ./build/task/load_tss.o:task/load_tss.S
79      gcc $(CFLAGS) -o $@ $<
80  ./build/task/%.o:task/%.c
81      gcc $(CFLAGS) -o $@ $<
82
83  .PHONY:clean debug run
84  clean:
85      rm -rf build
86      rm -rf image
87  debug:all
88      qemu-system-i386  -smp 1 -m 128M -s -S -drive file=image/disk.img,index=0,media=disk,format
        =raw -monitor stdio -no-reboot
89  run:all
90      qemu-system-i386 -display sdl -boot menu=on,splash-time=5000 -smp 1 -m 128M       -drive
        file=image/disk.img,index=0,media=disk,format=raw -monitor stdio -no-reboot
```

其中，CFLAGS 用于定义 gcc 编译选项，-g 与-O0 指定生成调试信息，0 级优化。根据需要可取消调试选项并设置不同优化级别。-m32 指定生成 32 位架构代码。-MMD 用于生成.d 依赖说明文件，防止执行 make 命令时，头文件修改而不重新编译。 -fno-pic -fno-pie -fno-stack-protector -fno-builtin -nostdlib -nostdinc -nodefaultlibs -nostartfiles 等选项用于指明不要生成位置无关代码，不要开启栈保护，不要使用内建函数、标准库

等要求。

`all` 为最终生成的目标，依赖 build 目录，image 目录以及 imagefile 目标。

`imagefile` 依赖于 boot.bin 与 kernel.bin 文件，此二文件为纯二进制代码，不含 elf 头信息，boot 将直接根据地址跳转到 kernel.bin 的 head 部分执行。并且通过 dd 命令制作磁盘镜像 disk.img 文件，其中 boot.bin 将被放置在第一扇区，`kernel.bin` 将被放在随后的若干扇区。

`kernel.bin` 依赖 `KERNEL_OBJS` 指定的所有目标文件，并链接生成最终的二进制 OS 代码，并且设置了 head.o 为连接最开始的部分，确保 head.S 中的代码为内核最开始部分。在使用 gcc 编译生成 elf 格式的可执行文件 kernel.elf 后使用 objcopy 工具将其转为不含 elf 头信息的二进制代码 kernel.bin。

`clean` 目标执行删除 build 和 image 目录命令。

`debug` 与 `run` 目标执行运行 qemu 指令，并依赖于 all 目标的完成。

第 2 章　boot 与物理内存管理

2.1　Intel® x86 开机过程

Intel® x86 系列 CPU 在开机时首先工作在 16 位实模式 (Real Mode) 状态，CS:IP 被设为 0xffff:0，执行该内存处的指令，此处指令为跳转指令，将使 CPU 进入 ROM 内置的 BIOS 程序。

当 BIOS 完成初始化和硬件自检后，将会在磁盘中逐个寻找可引导扇区，引导扇区即我们需要编写的 boot 程序。标志为扇区最后两字节为 0x55, 0xAA。BIOS 将引导扇区的一扇区内容读入内存 0x7c00 地址开始处，最后跳转到 0x7c00 处执行 boot 引导代码。

实模式下内存布局如下图所示[1]：

2.2　boot

由上述内容可知，boot 代码应至少完成如下两个功能：

1. 使该扇区成为引导扇区。
2. 加载操作系统剩余部分并跳转执行。

对于 Intel® i386 系列机，还应当使 CPU 从 16 位实模式进入 32 位保护模式。

2.2.1　进入 boot

boot.S 开头内容如下。其中，#include 的 config.h 文件中预定义了若干宏常量。

第一条指令为 `jmp _start`,将跳转到 _start 处继续执行,开头数据用于定义 FAT16 文件系统（参见第 ?? 章）所要求的元信息。_start 处将会进行若干寄存器的初始化操作。

```
1 #include "config.h"
2     .global _start
3     .text
4 fat16_header:
5     .code16
6     # Byte Offset: 0x000
7     # Length: 3 bytes
8     # Opstring: 0XEB(jmp short) 0Xxx(start) 0X90(nop)
9     jmp _start
```

[1]References: OSDEVWIKI(https://wiki.osdev.org/Memory_Map_(x86)#Overview)

```
10      nop
11
12      # BIOS Parameter Block
13      # Byte Offset: 0x003
14      .ascii "soOSv1.0"           # OEM名称
15      .word 512                   # 每扇区字节数
16      .byte 64                    # 每个簇的扇区数
17      .word 200                   # 保留扇区数
18      .byte 2                     # FAT表数
19      .word 64                    # 根目录项数
20      .word 0x00                  # 扇区数
21      .byte 0xf8                  # 媒体描述符
22      .word 0x100                 # 每FAT扇区数
23      .word 0x20                  # 每道扇区数
24      .word 64                    # 磁头数
25      .long 0                     # 隐藏扇区数
26      .long 0x400000              # 逻辑扇区总数
27
28      # Extended BIOS Parameter Block
29      # Byte Offset: 0x024
30      .byte 0x80                  # 驱动器号
31      .byte 0                     # 保留(WINNT BIT)
32      .byte 0x29                  # 扩展引导标志
33      .long 0xD105                # 卷序列号
34      .ascii "SOOS BOOTIN"        # 卷标
35      .ascii "FAT16   "           # 文件系统类型
36
37  _start:
38      xor %eax,%eax
39      xor %ebx,%ebx
40      xor %ecx,%ecx
41      xor %edx,%edx
42      xor %ebp,%ebp
43      xor %esi,%esi
44      mov %ax,%ds
45      mov %ax,%es
46      mov %ax,%ss
47      mov $0x7c00,%sp
48      mov %sp,%bp
```

2.2.2 进入 32 位保护模式

进入 32 位保护模式的标志为 CR0 寄存器中 PE 位（即第 0 位）。在此之前需要还要完成两项工作。即打开 A20 地址总线，设置 GDT 表。

由于 i386 开机时工作在 16 位实模式，20 号地址总线置为 0，会出现地址绕回现象（即地址对 1MB 取余），无法对高于 1MB 的地址进行寻址。代码采用 Fast A20 Gate 法打开 20 号地址总线。

32 位保护模式下，寻址方式将会发生改变，寻址不再是 16 位模式下段基址左移 4 位加上段偏移地址。在不考虑分页开启的情况下，段基址寄存器（cs,ds,es,ss,fs,gs 等），作为选择子，其高 14 位的数值将作为 GDT 表的索引值，从 GDT 表中选择索引值对应的 GDT 表项，GDT 表项的数据定义了段基址和段界限。获取的段基址和与偏移地址相加得到最终地址。在这过程中还会对结果是否超过 GDT 表项定义的段界限。而段选择子寄存器的低 2 位将作为特权级的标志。0 表示最高特权，3 表示最低特权，GDT 表项包含了 DPL(descriptor privilege level)，在此计算最终地址时会对请求的段选择子寄存器权限进行检查，若段选择子寄存器的特权级过低，会被禁止访问。

下图取自 *Intel® 64 and IA-32 Architectures Software Developer' s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*

```
start_enter_32:
    #关中断，32位中断表尚未建立
    cli

    #打开A20地址线，Fast A20 Gate法
    inb $0x92,%al
    orb $0x2,%al
    outb %al, $0x92

    #加载GDT
    lgdt gdt_ptr

    #置cr0,PE位为1，进入保护模式
    movl %cr0,%eax
    orl $0x1,%eax
    movl %eax,%cr0
    /*
    * Segment Selector:
    * 15-3  Index
    * 2     TI table Idicator 0=GDT,1=LDT
    * 1-0   RPL requested privilege level
    */
```

```
23      ljmp $0x8, $start32 #通过jmp设置cs选择子, 使用jmp可清空流水线, 刷新
            gdt缓存
24
25  start32:
26      .code32
27      movw $0x10,%ax
28      movw %ax,%ds     #设置数据段选择子为2号gdt项
29      movw %ax,%es
30      movw %ax,%ss
31      movw %ax,%fs
32      movw %ax,%gs
33      movl $0x200000,%esp #设置栈顶
34      movl %esp,%ebp
```

其中 gdt_ptr 标号以及 gdt 内容定义在后文内容：

```
1   .p2align 2  #内存对齐到2^2字节处
2
3   gdt:
4       /*
5       * 高地址到低地址依次为:
6       * base 31_24      (8bit)
7
8       * G               Granualrity
9       * D/B             Default operation size
10      * L               64-bit code seg
11      * AVL             Availabel for use by system software
12
13      * Seg Limit 19_16(4bit)
14
15      * P               Segment present
16      * DPL(2bit)       Descriptor privilege level
17      * S               Descriptor type(0=system;1=code or data)
18      * Type(4bit)      Segment type
19
20      * Base 23_16      (8bit)
21
22      * Base 15_0
23      * Seg Limit 15_0
24      */
25  NULL_GDT:
26      .quad 0x0
```

```
CODE_DESC:
    #小端，高地址到低地址
    #Base=0,G=1,D/B=1,SegLim=0xfffff,P=1,DPL=0,S=1,Type=8(execute only)
    # Type change to a(Access bit set)
    .quad 0x00cf9a000000ffff
DATA_DESC:
    #Type=2(Read/Write)
    .quad 0x00cf92000000ffff
#STACK_DESC:#可以直接用data的描述符
#    .quad 0x004
GDT_LIMITS:

gdt_ptr:
    .word GDT_LIMITS -gdt-1  #低地址gdt表长度（字节）-1
    .long gdt        #高地址gdt基地址位置
```

2.2.3 加载操作系统剩余部分并跳转执行

进入 32 位模式后，继续读取位于磁盘上的操作系统剩余部分到内存，并将跳转执行。

read_kernel 先为调用函数设置参数，然后调用 read_disk 函数将磁盘上的内容读入内存。`call read_disk` 返回后将会执行 `ljmp $0x8, $KERNEL_START_PADDR`。跳转到在 config.h 中定义的内核代码开始内存地址继续执行。

read_disk 函数使用 io 端口操作实现对磁盘的读写，采用 LBA28 定址方式，传入参数寄存器 eax 为读取起始扇区，寄存器 cl 为读取扇区数，寄存器 es 与 edi 指定将要读取硬盘所到的内存地址。io 操作已有详细的注释在代码中。此处不再赘述。此外，在 boot.S 中直接使用汇编

```
read_kernel:
    #接下来需要将操作系统剩余部分代码从硬盘转到内存，并跳转执行
    movl $BOOT_END_SECTOR, %eax      #起始扇区。LBA从0开始编号，CHS从1开
        始编号
    movl $KERNEL_START_PADDR,%edi    #内核代码存放地址
    movb $255,%cl                    #读取扇区数，目前内核应不会超过255
        *512B约128KB。
    call read_disk

    ljmp $0x8,$KERNEL_START_PADDR
    hlt

```

```
#读取硬盘到内存。参数：eax起始扇区，LBA28定址；cl读取扇区数；es:edi，存放内存位置
read_disk:
    pushl %eax
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %edi
    #LBA方式从IO端口读取硬盘数据到内存
    #端口0x1f3，设置LBA0~7位
    movw $0x1f3,%dx
    out %al,%dx
    #端口0x1f4，设置LBA8~15位
    movw $0x1f4,%dx
    shr $8,%eax
    out %al,%dx
    #端口0x1f5，设置LBA16~23位
    movw $0x1f5,%dx
    shr $8,%eax
    out %al,%dx

    #端口0x1f6，设置device寄存器，选择LBA寻址模式（6位置1），选择主盘（4位置0），0-3位置LBA地址27~24位，5、7位默认为1
    movw $0x1f6,%dx
    shr $8,%eax
    #4位置0
    and $0xf,%al
    #5,6,7位置1
    or $0xe0,%al
    out %al,%dx

    #端口0x1f2，设置读取扇区数
    movw $0x1f2,%dx
    movb %cl,%al
    out %al,%dx

    #端口0x1f7，设置command读取命令
    movw $0x1f7,%dx
    movb $0x20,%al    #command=0x20,read;0x30,write;0xec,identify
    out %al,%dx
next_sector:
```

```
    push %ecx

    #轮询方式等待IO数据
wait_for_disk:
    #端口0x1f7, 读取status  !重新赋值dx!
    movw $0x1f7,%dx
    in %dx,%al
    and $0x89,%al  #只看第0,3,7位。0位错误, 3位完成, 6位就绪, 7位忙
    cmp $0x08,%al
    #nop
    jne wait_for_disk

    #端口0x1f0, 每次读取2字节数据, 一个扇区512字节, 循环256次。

read_port_data:
    mov $256,%ecx
    movw $0x1f0,%dx
    rep insw

    pop %ecx
    loop next_sector

    popl %edi
    popl %edx
    popl %ecx
    popl %ebx
    popl %eax

    ret
```

2.2.4 设置扇区为引导扇区

boot 由 boot.S 生成，在 AT$T 语法中，使用.org 伪命令即可设置编译计数器，并在该位置开始继续代码。使用.byte 伪命令在编译计数器中写入字节数据：

```
endcode:
.org 510  #0x1fe
.byte 0x55,0xaa
```

因此，在编译出的二进制代码中第 510（从 0 开始计）字节为 0x55，第 511 字节为 0xAA，在将来同操作系统代码一同写入 disk.img 磁盘映像中，并保证第一扇区为 boot 生成的代码。

第 3 章　虚拟内存管理

本章将详细介绍 soOS 的虚拟内存管理，包括页表的结构、页目录的创建与切换、以及内存页面的管理等内容。

3.1　页表结构

虚拟内存管理的核心是页表。页表将虚拟地址转换为物理地址，并控制访问权限。以下是页表项（PTE）的结构[2]：

31 Bits Address	AVL	G	D	A	C	W	U	R	P

Bit Definitions:

- **P (Present)**: Indicates if the page is in physical memory.
- **R/W (Read/Write)**: If set, the page is read/write. Otherwise, it is read-only.
- **U/S (User/Supervisor)**: Controls access based on privilege level. If set, accessible by all.
- **PWT (Write-Through)**: Controls write-through caching. If set, write-through is enabled.
- **PCD (Cache Disable)**: If set, the page will not be cached.
- **A (Accessed)**: Indicates if the page has been accessed (read during virtual address translation).
- **D (Dirty)**: Indicates if the page has been written to.
- **G (Global)**: If set, the TLB entry for the page is not invalidated on a MOV to CR3 instruction.
- **PAT (Page Attribute Table)**: Used to determine memory caching type, along with PCD and PWT.
- **AVL (Available)**: Bits available for OS use.

[2]References: OSDEVWIKI

3.2 页目录的创建

页目录是页表的上一级结构，管理多个页表。下面是创建页目录的代码实现：

```
#include "mm.h"
#include "page.h"
#include "errno.h"

static uint32_t* current_directory = 0; // ! Must be static in case of multiple paging directories

struct page_directory* create_page_directory(uint8_t flags)
{
    uint32_t* directory = kmalloc(sizeof(uint32_t) * PAGE_ENTRIES_PER_TABLE);
    int offset = 0;

    for(int i = 0; i < PAGE_ENTRIES_PER_TABLE; i++)
    {
        uint32_t* page_table_entry = kmalloc(sizeof(uint32_t) * PAGE_ENTRIES_PER_TABLE);
        for  (int j = 0; j < PAGE_ENTRIES_PER_TABLE; j++)
        {
            page_table_entry[j] = (offset + PAGE_SIZE * j) | flags;
        }

        offset += PAGE_SIZE * PAGE_ENTRIES_PER_TABLE; // switch to next page table
        directory[i] = (uint32_t)page_table_entry | flags | PAGE_IS_WRITABLE; // set the whole pagetable directory writable
    }

    struct page_directory* paging_directory = kmalloc(sizeof(struct page_directory));

    paging_directory->directory_entry = directory;
    return paging_directory;
}
```

3.3 页目录的切换

为了使用新的页目录，我们需要切换页目录。以下是切换页目录的实现：

```
void switch_page(uint32_t* directory)
{
    load_page_directory(directory);
    current_directory = directory;
}
```

3.3.1 加载页目录

加载页目录是通过将页目录的基地址加载到 CR3 寄存器来实现的。以下是汇编代码实现：

```
.code32

.global load_page_directory
.global enable_paging

.text

load_page_directory:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax # In X86, the first argument is at [ebp + 8]
movl %eax, %cr3 # CR3 = page directory base address
popl %ebp # Restore the caller's original base pointer ebp
ret
```

3.3.2 启用分页

启用分页需要设置 CR0 寄存器的 PG 位。以下是启用分页的汇编代码实现：

```
enable_paging:
pushl %ebp
movl %esp, %ebp
movl %cr0, %eax
orl $0x80000000, %eax # Set the PG bit in CR0
movl %eax, %cr0
popl %ebp
ret
```

3.4 页面管理

页面管理包括页面的对齐检查、页面索引计算以及设置页面映射等操作。

3.4.1 页面对齐检查

页面对齐检查用于确保地址是页大小的倍数。以下是实现代码：

```
bool is_page_aligned(void* addr)
{
    return ((uint32_t)addr % PAGE_SIZE) == 0;
}
```

3.4.2 页面索引计算

计算虚拟地址对应的页目录索引和页表索引。以下是实现代码：

```
int get_page_index(void* virtual_addr, uint32_t* dir_index,
    uint32_t* table_index)
{
    int res = 0;
    if(!is_page_aligned(virtual_addr))
    {
        res = -EINVARG;
        goto out;
    }

    *dir_index = ((uint32_t)virtual_addr / (PAGE_SIZE *
        PAGE_ENTRIES_PER_TABLE));
    *table_index = ((uint32_t)virtual_addr / PAGE_SIZE) %
        PAGE_ENTRIES_PER_TABLE;
    out:
    return res;
}
```

3.4.3 设置页面映射

设置虚拟地址到物理地址的映射关系。以下是实现代码：

```
int set_paging(uint32_t* directory, void* virtual_addr, uint32_t
    physical_addr)
{
    if(!is_page_aligned(virtual_addr))
```

```
        {
            return -EINVARG;
        }

        uint32_t dir_index = 0;
        uint32_t table_index = 0;

        int res = get_page_index(virtual_addr, &dir_index, &table_index
            );
        if(res < 0)
        {
            return res;
        }

        uint32_t entry = directory[dir_index];
        uint32_t* table = (uint32_t*)(entry & 0xFFFFF000); // Only get
            the table address

        table[table_index] = physical_addr; // physical_addr is the
            physical address with flags set
        return res;
    }
```

3.5 头文件定义

以下是页表相关的头文件定义，包含了一些常量定义和函数声明：

```
    #ifndef PAGE_H
    #define PAGE_H

    #include "types.h"

    #define PAGE_CACHE_DISABLED  0B00010000
    #define PAGE_WRITE_THROUGH   0B00001000
    #define PAGE_ACCESS_FROM_ALL 0B00000100
    #define PAGE_IS_WRITABLE     0B00000010
    #define PAGE_IS_PRESENT      0B00000001

    #define PAGE_ENTRIES_PER_TABLE 1024
    #define PAGE_SIZE 4096

```

```
15      struct page_directory
16      {
17          uint32_t* directory_entry;
18      };
19
20      uint32_t* get_page_directory(struct page_directory*
            paging_directory);
21      void switch_page(uint32_t* directory);
22      struct page_directory* create_page_directory(uint8_t flags);
23      int set_paging(uint32_t* directory, void* virtual_addr, uint32_t
            val);
24      bool is_page_aligned(void* addr);
25      void enable_paging();
26      void load_page_directory(uint32_t* directory);
27
28      #endif
```

通过以上章节的详细说明，我们可以清晰地了解 soOS 的虚拟内存管理机制，包括页表结构、页目录的创建与切换，以及内存页面的管理等方面的实现。

第 4 章　文件系统

4.1　底层磁盘驱动

本章节详细介绍了 soOS 的底层磁盘驱动实现，包括磁盘读写的基本原理、数据流的管理以及相关函数的具体实现。

4.1.1　I/O 端口操作

在 x86 架构中，I/O 端口用于与外部设备进行通信。以下是 ‘io.h‘ 中定义的几个最底层的 I/O 操作函数：

```
#ifndef __IO_H
#define __IO_H
#include "types.h"

// 向端口 port 输出一字节数据 data
static inline void __attribute__((always_inline)) outb(uint16_t
    port, uint8_t data) {
     asm volatile ("outb %b0,%1" ::"a"(data),"id"(port));
}

// 向端口 port 输出两字节数据 data
static inline void __attribute__((always_inline)) outw(uint16_t
    port, uint16_t data) {
     asm volatile ("outw %w0,%1" ::"a"(data),"id"(port));
}

// 将 %ds:addr 内存处 word_count 个字写入端口 port
static inline void __attribute__((always_inline)) outsw(uint16_t
    port, void* addr, uint32_t word_count) {
     asm volatile("cld;\n\t""rep outsw;\n\t"
     :"+S"(addr), "+c"(word_count)
     :"d"(port));
}

// 从端口 port 读取一字节数据
static inline uint8_t __attribute__((always_inline)) inb(uint16_t
    port) {
     uint8_t data;
     asm volatile ("inb %1,%b0" :"=a"(data) :"id"(port));
     return data;
```

```
27      }
28
29      // 从端口 port 读取两字节数据
30      static inline uint16_t __attribute__((always_inline)) inw(uint16_t
            port) {
31          uint16_t data;
32          asm volatile ("inw %1,%w0" :"=a"(data) :"id"(port));
33          return data;
34      }
35
36      #endif
```

outb 函数 outb 函数用于向指定的 I/O 端口发送一个字节的数据。其实现如下：

```
1       static inline void __attribute__((always_inline)) outb(uint16_t
            port, uint8_t data) {
2           asm volatile ("outb %b0,%1" ::"a"(data),"id"(port));
3       }
```

该函数使用内联汇编，将数据从寄存器 a 发送到指定的 I/O 端口。

outw 函数 outw 函数用于向指定的 I/O 端口发送两个字节的数据。其实现如下：

```
1       static inline void __attribute__((always_inline)) outw(uint16_t
            port, uint16_t data) {
2           asm volatile ("outw %w0,%1" ::"a"(data),"id"(port));
3       }
```

该函数与 outb 类似，只不过发送的是两个字节的数据。

outsw 函数 outsw 函数用于将内存中的多个字写入到指定的 I/O 端口。其实现如下：

```
1       static inline void __attribute__((always_inline)) outsw(uint16_t
            port, void* addr, uint32_t word_count) {
2           asm volatile("cld;\n\t""rep outsw;\n\t"
3           :"+S"(addr), "+c"(word_count)
4           :"d"(port));
5       }
```

该函数使用了字符串指令 rep outsw，将内存中的多个字连续写入 I/O 端口。

inb 函数 inb 函数用于从指定的 I/O 端口读取一个字节的数据。其实现如下：

```
static inline uint8_t __attribute__((always_inline)) inb(uint16_t
    port) {
    uint8_t data;
    asm volatile ("inb %1,%b0" :"=a"(data) :"id"(port));
    return data;
}
```

该函数使用内联汇编，从指定的 I/O 端口读取一个字节的数据并存储在寄存器 a 中。

inw 函数 inw 函数用于从指定的 I/O 端口读取两个字节的数据。其实现如下：

```
static inline uint16_t __attribute__((always_inline)) inw(uint16_t
    port) {
    uint16_t data;
    asm volatile ("inw %1,%w0" :"=a"(data) :"id"(port));
    return data;
}
```

该函数与 inb 类似，只不过读取的是两个字节的数据。

4.1.2 磁盘结构

在 soOS 中，磁盘通过如下结构体进行描述：

```
typedef unsigned int disk_type;
struct disk{
    disk_type type;
    unsigned int sector_size;
    int disk_id;
    struct filesystem* filesystem;
    void* data;
};
```

其中，各字段含义如下：- type：磁盘类型，REAL_DISK_TYPE 表示物理磁盘。- sector_size：扇区大小。- disk_id：磁盘标识符。- filesystem：文件系统指针。- data：磁盘数据指针。

4.1.3 磁盘初始化

在系统初始化时，需要搜索并初始化磁盘。以下是实现代码：

```
void search_and_init_disk()
```

```
    {
        memset(&disk, 0, sizeof(struct disk));
        disk.type = REAL_DISK_TYPE;
        disk.sector_size = SECTOR_SIZE;
        disk.disk_id = 0; // TODO: More disk support
        disk.filesystem = resolve_fs(&disk);
    }
```

4.1.4 磁盘读写操作

磁盘读写操作是通过 I/O 端口进行的。以下是向端口输出和从端口读取数据的基本函数：

```
    #include "io.h"

    static inline void outb(uint16_t port, uint8_t data) {
        asm volatile ("outb %b0,%1" ::"a"(data),"id"(port));
    }

    static inline uint8_t inb(uint16_t port) {
        uint8_t data;
        asm volatile ("inb %1,%b0" :"=a"(data) :"id"(port));
        return data;
    }

    static inline uint16_t inw(uint16_t port) {
        uint16_t data;
        asm volatile ("inw %1,%w0" :"=a"(data) :"id"(port));
        return data;
    }
```

读取磁盘扇区 读取磁盘扇区是通过 LBA（线性块地址）方式进行的。以下是读取磁盘扇区的实现代码：

```
    int read_disk_sector(int lba, int total, void* buf)
    {
        outb(0x1F6, (lba >> 24) | 0xE0); // LBA 高 4 位和主/从位
        outb(0x1F2, total); // 要读取的扇区数
        outb(0x1F3, (unsigned char)(lba & 0xff)); // LBA 低 8 位
        outb(0x1F4, (unsigned char)(lba >> 8)); // LBA 中 8 位
        outb(0x1F5, (unsigned char)(lba >> 16)); // LBA 高 8 位
```

```
        outb(0x1F7, 0x20); // 读取扇区命令

        unsigned short* ptr = (unsigned short*) buf;
        for (int b = 0; b < total; b++)
        {
            char status = inb(0x1F7);
            while (!(status & 0x08))
            {
                status = inb(0x1F7);
            }

            for (int i = 0; i < 256; i++)
            {
                unsigned short data = inw(0x1F0);
                *ptr = data;
                ptr++;
            }
        }

        return 0;
    }
```

读取磁盘块 读取磁盘块是对读取扇区的进一步封装，检查传入的 disk 结构体是否与全局 disk 结构体相匹配。以下是实现代码：

```
int read_disk_block(struct disk* _disk, unsigned int lba, int total
    , void* buf)
{
    if (_disk != &disk){
        return -EIO;
    }

    return read_disk_sector(lba, total, buf);
}
```

4.2 磁盘流操作

由于磁盘工作原理限制，每次只能以块为单位来读取数据。而磁盘流则是对底层磁盘驱动的一层封装，增加了一个指定读写位置的 pos 变量，提供了以字节为单位读取数据的接口，方便上层调用。以下是磁盘流相关的结构体和函数：

```
struct disk_stream
{
    int pos;
    struct disk* disk;
};

struct disk_stream* create_disk_stream(int disk_id)
{
    struct disk_stream* stream = (struct disk_stream*) kmalloc(
        sizeof(struct disk_stream));
    stream->pos = 0;
    stream->disk = (struct disk*)get_disk(disk_id);
    return stream;
}

void destroy_disk_stream(struct disk_stream* stream)
{
    kfree(stream);
}

int seek_disk_stream(struct disk_stream* stream, int pos)
{
    stream->pos = pos;
    return 0;
}
```

4.2.1 读取磁盘流

读取磁盘流支持从当前流位置开始读取指定字节的数据。以下是实现代码：

```
int read_disk_stream(struct disk_stream* stream, int total, void*
    out)
{
    int sector = stream->pos / SECTOR_SIZE;
    int offset = stream->pos % SECTOR_SIZE;

    char buf[SECTOR_SIZE];
    int res = read_disk_block(stream->disk, sector, 1, buf);
    if(res < 0)
    {
        return res;
```

```
        }

        int total_to_read = total > SECTOR_SIZE? SECTOR_SIZE: total;
        for(int i = 0; i < total_to_read; i++)
        {
            *(char*)out++ = buf[offset + i]; // MOD 06-21: out指针也要跟着走
        }

        stream->pos += total_to_read;
        if(total > SECTOR_SIZE)
        {
            return read_disk_stream(stream, total - total_to_read, (char*)out);
        }

        return res;
    }
```

通过以上章节的详细说明，我们可以清晰地了解 soOS 的底层磁盘驱动实现，包括磁盘结构、磁盘初始化、磁盘读写操作以及磁盘流操作等方面的实现。

4.3 FAT16 文件系统

soOS 操作系统实现了对 FAT16 文件系统的支持。FAT16 是一种常见的文件系统格式，广泛应用于各种存储设备。本节将详细介绍 soOS 中 FAT16 文件系统的实现原理和主要功能。

4.3.1 FAT16 文件系统结构

FAT16 文件系统的主要组成部分包括：引导扇区、文件分配表、根目录区和数据区。在 soOS 中，我们使用以下结构体来表示 FAT16 文件系统的各个组成部分：

引导扇区 引导扇区包含了文件系统的基本信息，使用 `struct fat_header` 和 `struct extended_bios_parameter_block` 来表示：

```
    struct fat_header {
        u8 jump[3];
        u8 oem[8];
        u16 bytes_per_sector;
        u8 sectors_per_cluster;
        u16 reserved_sectors;
```

```
    u8 fat_count;
    u16 root_entry_count;
    u16 total_sectors;
    u8 media_type;
    u16 sectors_per_fat;
    u16 sectors_per_track;
    u16 head_count;
    u32 hidden_sectors;
    u32 total_sectors_large;
} __attribute__((packed));

struct extended_bios_parameter_block {
    u8 drive_number;
    u8 reserved;
    u8 boot_signature;
    u32 volume_id;
    u8 volume_label[11];
    u8 system_id[8];
} __attribute__((packed));
```

这些结构体成员变量包含了扇区大小、簇大小、FAT 表数量等重要信息，用于解析和操作 FAT16 文件系统。

目录项 目录项用于存储文件和目录的元数据，使用 `struct fat16_entry` 表示：

```
struct fat16_entry {
    u8 name[8];
    u8 ext[3];
    u8 attr;
    u8 reserved;
    u8 creation_time_tenths;
    u16 creation_time;
    u16 creation_date;
    u16 last_access_date;
    u16 first_cluster_high;
    u16 last_mod_time;
    u16 last_mod_date;
    u16 first_cluster_low;
    u32 size;
} __attribute__((packed));
```

这个结构体包含了文件名、扩展名、属性、创建时间、最后修改时间、文件大小等

信息。

目录 目录用于组织文件和子目录，使用 `struct fat16_directory` 表示：

```
struct fat16_directory {
    struct fat16_entry* entries;
    int totel_entries;
    int sector_begin;
    int sector_end;
};
```

这个结构体包含了目录中的所有条目、总条目数以及目录在磁盘上的起始和结束扇区。

文件项 文件项是对文件或目录的抽象，使用 `struct fat16_item` 表示：

```
struct fat16_item {
    union {
        struct fat16_directory* directory;
        struct fat16_entry* entry;
    };
    FAT16_ITEM_TYPE type;
};
```

这个结构体使用联合体来表示文件或目录，并用 type 字段来区分类型。

文件描述符 文件描述符用于跟踪打开的文件，使用 `struct fat16_file_descriptor` 表示：

```
struct fat16_file_descriptor {
    struct fat16_item* item;
    u32 offset;
};
```

这个结构体包含了指向文件项的指针和当前文件偏移量。

4.3.2 文件系统解析

FAT16 文件系统的解析主要通过 `resolve_fat16` 函数完成。这个函数的主要任务是读取磁盘的引导扇区，初始化 FAT16 数据结构并判断是否为 FAT16 格式化的磁盘，若是，则与该磁盘绑定并获取根目录信息。

函数调用逻辑 `resolve_fat16` 函数在文件系统初始化过程中被调用。它会进一步调用 `init_fat16_data`、`read_disk_stream` 和 `get_root_dir` 等函数来完成初始化过程。

函数实现 以下是 resolve_fat16 函数的实现：

```
int resolve_fat16(struct disk* disk)
{
    struct fat16_data* data = (struct fat16_data*) kmalloc(sizeof(
        struct fat16_data));
    init_fat16_data(disk, data);

    disk->data = data;
    disk->filesystem = &fat16_fs;

    struct disk_stream* stream = create_disk_stream(disk->disk_id);
    if(!stream)
    {
        kfree(data);
        return -ENOMEM;
    }

    if(read_disk_stream(stream, sizeof(struct fat16_fs_info), &data
        ->header) != 0)
    {
        kfree(data);
        return -EIO;
    }

    if(get_root_dir(disk, data, &data->root) != 0)
    {
        kfree(data);
        return -EIO;
    }

    return 0;
}
```

代码解析 1. 首先，函数分配内存并初始化 fat16_data 结构。2. 然后，它创建一个磁盘流来读取文件系统信息。3. 接下来，它读取 FAT16 文件系统的头信息。4. 最后，它获取根目录信息。

如果在任何步骤中出现错误，函数会释放已分配的资源并返回相应的错误代码。

相关函数 get_root_dir 函数用于读取 FAT16 文件系统的根目录信息，并将其加载到内存中。这是文件系统初始化过程中的一个关键步骤，因为根目录是访问文件系统中所

有其他文件和目录的起点。其实现如下：

```
int get_root_dir(struct disk* disk, struct fat16_data* data, struct
    fat16_directory* directory)
{
    struct fat16_fs_info* header = &data->header;
    int root_dir_sector_pos = (header->header.fat_count * header->
       header.sectors_per_fat) + header->header.reserved_sectors;
    int root_dir_entries = header->header.root_entry_count;
    int root_dir_size = root_dir_entries * sizeof(struct
       fat16_entry);
    int total_sectors = root_dir_size / disk->sector_size;
    if(root_dir_size % disk->sector_size != 0)
    {
        total_sectors ++;
    }

    int total_entries = get_total_entries(disk, root_dir_sector_pos
       );

    struct fat16_entry* dir = (struct fat16_entry*) kmalloc(
       root_dir_size);
    if(!dir){
        return -ENOMEM;
    }

    struct disk_stream* stream = data->directory_streamer;
    if(seek_disk_stream(stream, sector_to_address(disk,
       root_dir_sector_pos)) != 0)
    {
        return -EIO;
    }

    if(read_disk_stream(stream, root_dir_size, dir) != 0)
    {
        return -EIO;
    }

    directory->entries = dir;
    directory->totel_entries = total_entries;
    directory->sector_begin = root_dir_sector_pos;
    directory->sector_end = root_dir_sector_pos + (root_dir_size /
```

```
            disk->sector_size);
35
36          return 0;
37      }
```

让我们详细分析这个函数的每一步：

1. 函数参数：
   - `disk`: 指向磁盘结构的指针，包含磁盘的基本信息。
   - `data`: 指向 FAT16 文件系统数据结构的指针。
   - `directory`: 指向将要填充根目录信息的结构的指针。
2. 计算根目录的位置和大小：
   - `int root_dir_sector_pos = (header->header.fat_count * header->header.sectors + header->header.reserved_sectors;` 计算根目录的起始扇区位置。这个位置在 FAT 表之后，计算方法是：FAT 表数量 * 每个 FAT 表的扇区数 + 保留扇区数。
   - `int root_dir_entries = header->header.root_entry_count;` 获取根目录中的条目数量。
   - `int root_dir_size = root_dir_entries * sizeof(struct fat16_entry);` 计算根目录的总大小（字节）。
   - 计算根目录占用的扇区数：

     ```
     1   int total_sectors = root_dir_size / disk->
             sector_size;
     2   if(root_dir_size % disk->sector_size != 0)
     3   {
     4       total_sectors ++;
     5   }
     ```

     这里考虑了根目录大小可能不是扇区大小的整数倍的情况。
3. 获取实际的条目数量：
   - `int total_entries = get_total_entries(disk, root_dir_sector_pos);` 调用 `get_total_entries` 函数来获取根目录中实际的条目数量。这个数量可能小于或等于 `root_dir_entries`。
4. 分配内存：

- `struct fat16_entry* dir = (struct fat16_entry*) kmalloc(root_dir_size);` 为根目录条目分配内存。如果内存分配失败，函数返回 `-ENOMEM` 错误。

5. 读取根目录数据：

   - 使用 `data->directory_streamer` 来读取磁盘数据。
   - `seek_disk_stream(stream, sector_to_address(disk, root_dir_sector_pos))` 将磁盘流定位到根目录的起始位置。
   - `read_disk_stream(stream, root_dir_size, dir)` 读取整个根目录的内容到刚才分配的内存中。
   - 如果上述操作失败，函数返回 `-EIO` 错误。

6. 填充目录结构：

   - `directory->entries = dir;` 设置目录条目指针。
   - `directory->totel_entries = total_entries;` 设置实际的条目数量。
   - `directory->sector_begin = root_dir_sector_pos;` 设置起始扇区。
   - `directory->sector_end = root_dir_sector_pos + (root_dir_size / disk->sector_` 设置结束扇区。

7. 返回结果：

   - 如果所有操作成功完成，函数返回 0 表示成功。

这个函数是 FAT16 文件系统初始化过程中的关键步骤。通过加载根目录进入内存，提取磁盘文件系统元数据信息，为后续的所有文件系统操作（如文件查找、目录遍历等）奠定了基础，可以方便地进行后续的操作而减少了对磁盘的访问。

4.3.3 打开文件

打开文件的过程主要通过 `fat16_open_file` 函数实现。这个函数的主要任务是在文件系统中定位指定的文件，并创建一个文件描述符。

函数调用逻辑 `fat16_open_file` 函数在用户请求打开文件时被调用。它会进一步调用 `get_root_directory_item` 来查找文件，然后创建并返回一个文件描述符。

函数实现 以下是 fat16_open_file 函数的实现：

```
void* fat16_open_file(struct disk* disk, struct path_part* path,
    FILE_OPEN_MODE mode)
{
    struct fat16_item* item = get_root_directory_item(disk, path);
    if(!item || item->type != FAT16_ITEM_TYPE_FILE)
    {
        return 0;
    }

    struct fat16_file_descriptor* fd = kmalloc(sizeof(struct
        fat16_file_descriptor));
    fd->item = item;
    fd->offset = 0;

    return fd;
}
```

代码解析 1. 首先，函数调用 get_root_directory_item 来查找指定路径的文件。2. 如果找到文件并且是一个文件（而不是目录），它会创建一个新的文件描述符。3. 文件描述符包含了文件项的指针和初始偏移量（设为 0）。4. 最后，函数返回创建的文件描述符。

相关函数 get_root_directory_item 函数在查找文件过程中起着关键作用。它会遍历目录结构，查找指定路径的文件或目录。这个函数的实现涉及到复杂的目录遍历逻辑，包括处理子目录的情况。

get__root__directory__item 函数 get_root_directory_item 函数用于在 FAT16 文件系统中查找指定路径的文件或目录。它从根目录开始，逐级遍历目录结构，直到找到目标项或确定目标不存在。

函数签名如下：

```
struct fat16_item* get_root_directory_item(struct disk* disk,
    struct path_part* path)
{
    struct fat16_data* data = disk->data;
    struct fat16_item* root_item = find_item_in_root_directory(disk
        , &data->root, path->part);
    if(!root_item)
```

```
6           {
7               return 0;
8           }
9
10          struct path_part* next = path->next;
11          struct fat16_item* current = root_item;
12          while(next)
13          {
14              if(current->type != FAT16_ITEM_TYPE_DIRECTORY)
15              {
16                  free_fat16_item(current);
17                  return 0;
18              }
19
20              struct fat16_item* next_item = find_item_in_root_directory(
                    disk, current->directory, next->part);
21              if(!next_item)
22              {
23                  free_fat16_item(current);
24                  return 0;
25              }
26
27              free_fat16_item(current);
28              current = next_item;
29              next = next->next;
30          }
31
32          return current;
33      }
```

让我们详细分析这个函数的每一步：

1. 函数参数：
   - `disk`: 指向磁盘结构的指针，包含文件系统信息。
   - `path`: 指向路径结构的指针，表示要查找的文件或目录的路径。
2. 初始化：
   - `struct fat16_data* data = disk->data;` 获取 FAT16 文件系统的数据结构。

- `struct fat16_item* root_item = find_item_in_root_directory(disk, &data->root path->part);` 在根目录中查找路径的第一部分。这里调用了 `find_item_in_root_directo` 函数，该函数负责在给定目录中查找特定名称的项。

3. 检查根目录项：
   - 如果 `root_item` 为空，说明在根目录中没有找到匹配的项，函数返回 0（表示未找到）。
4. 遍历路径：
   - `struct path_part* next = path->next;` 获取路径的下一部分。
   - `struct fat16_item* current = root_item;` 将当前项设置为根目录项。
   - 使用 while 循环遍历路径的剩余部分。
5. 在循环中：
   - 检查当前项是否为目录：
     `if(current->type != FAT16_ITEM_TYPE_DIRECTORY)`
     如果不是目录但路径还有下一部分，说明路径无效，释放资源并返回 0。
   - 在当前目录中查找下一级项：
     `struct fat16_item* next_item = find_item_in_root_directory(disk, current->di next->part);`
     这里再次调用 `find_item_in_root_directory`，但这次是在当前目录中查找。
   - 检查是否找到下一级项：
     如果 `next_item` 为空，说明没有找到匹配的项，释放资源并返回 0。
   - 更新当前项：
     释放之前的 `current` 项，将 `next_item` 设为新的 `current`。
   - 移动到路径的下一部分：
     `next = next->next;`
6. 返回结果：
   - 如果循环正常完成（遍历了整个路径），函数返回 `current`，这就是最终找到的文件或目录项。

函数的关键作用 `get_root_directory_item` 函数在 FAT16 文件系统的实现中起着核心作用：

- 路径解析：它能够解析复杂的文件路径，支持多级目录结构。
- 文件/目录查找：它可以在文件系统中定位特定的文件或目录。
- 错误处理：函数在遇到无效路径或找不到项时会正确地处理错误，释放资源并返回适当的结果。
- 资源管理：在遍历过程中，函数会适时释放不再需要的资源，防止内存泄漏。

这个函数是文件系统操作（如打开文件、读取目录内容等）的基础。它提供了一种在 FAT16 文件系统的目录结构中导航的方法，使得系统能够根据用户提供的路径找到正确的文件或目录。

4.3.4 读取文件

读取文件的过程主要通过 `fat16_read_file` 函数实现。这个函数的主要任务是从指定的文件偏移量开始，读取指定数量的数据。

函数调用逻辑 `fat16_read_file` 函数在用户请求读取文件内容时被调用。它会进一步调用 `read_internal_data` 来实际读取数据。`read_internal_data` 函数又会调用 `read_data_from_stream` 来处理可能的跨簇读取情况。

函数实现 以下是 `fat16_read_file` 函数的实现：

```
1  int fat16_read_file(struct disk* disk, void* fd, u32 size, u32 nb,
       char* out)
2  {
3      struct fat16_file_descriptor* descriptor = (struct
           fat16_file_descriptor*)fd;
4      struct fat16_entry* entry = descriptor->item->entry;
5      int offset = descriptor->offset;
6      for(u32 i = 0; i < nb; i++)
7      {
8          if(read_internal_data(disk, get_first_cluster(entry),
               offset, size, out) < 0)
9          {
10             return 0;
11         }
12         out += size;
13         offset += size;
14     }
15     return nb;
```

```
16      }
```

代码解析 1. 函数首先从文件描述符中获取文件条目和当前偏移量。2. 然后，它进入一个循环，读取指定数量（nb）的数据块。3. 对于每个数据块，它调用 `read_internal_data` 函数来读取数据。4. 如果读取成功，它更新输出缓冲区指针和偏移量。5. 如果在任何点发生错误，函数会立即返回。6. 最后，函数返回成功读取的数据块数量。

相关函数 `read_internal_data` 和 `read_data_from_stream` 函数在读取过程中起着关键作用。它们负责处理 FAT16 文件系统的簇链结构，确保能够正确读取跨越多个簇的文件数据。

`read_internal_data` 是 `read_data_from_stream` 的一层封装，其函数实现如下：

```
1   static int read_internal_data(struct disk* disk, int cluster, int
        offset, int total, void* out)
2   {
3       struct fat16_data* data = disk->data;
4       struct disk_stream* stream = data->read_cluster_streamer;
5       if (!stream)
6       {
7           return -EIO;
8       }
9
10      return read_data_from_stream(disk, stream, cluster, offset,
            total, out);
11  }
```

这个函数启用了文件系统上下文结构体 `fat16_data` 中专门用于读取文件所在簇的 `read_cluster_streamer`，并将其传入下层 `read_data_from_stream`.

`read_data_from_stream` 函数的实现如下：

```
1   static int read_data_from_stream(struct disk* disk, struct
        disk_stream* stream, int cluster, int offset, int total, void*
        out)
2   {
3       struct fat16_data* data = disk->data;
4       int cluster_size = data->header.header.sectors_per_cluster *
            disk->sector_size;
5       int cluster_to_use = get_cluster_for_offset(disk, cluster,
            offset);
6       if (cluster_to_use < 0)
7       {
```

```
            return cluster_to_use;
        }

        int cluster_offset = offset % cluster_size;
        int start_sector = cluster_to_sector(data, cluster_to_use);
        int start_pos = (start_sector * disk->sector_size) +
           cluster_offset;
        int total_to_read = total > cluster_size ? cluster_size : total
           ;

        if (seek_disk_stream(stream, start_pos) != 0)
        {
            return -EIO;
        }

        if(read_disk_stream(stream, total_to_read, out) != 0)
        {
            return -EIO;
        }

        total -= total_to_read;
        if (total > 0)
        {
            return read_data_from_stream(disk, stream, cluster, offset
               + total_to_read, total, out + total_to_read);
        }

        return 0;
    }
```

这个函数处理了跨簇读取的情况，确保能够正确读取大文件的内容。让我们详细解析这个函数的每一步：

1. 首先，函数获取文件系统的基本信息：
   - `struct fat16_data* data = disk->data;` 获取 FAT16 文件系统的数据结构。
   - `int cluster_size = data->header.header.sectors_per_cluster * disk->sector_si` 计算每个簇的大小（字节数）。
2. 然后，函数确定要读取的簇：

- `int cluster_to_use = get_cluster_for_offset(disk, cluster, offset);` 根据给定的偏移量，确定应该从哪个簇开始读取。这个函数会遍历 FAT 表，找到对应偏移量的正确簇。`get_cluster_for_offset` 解析在下一段落
- 如果 `cluster_to_use < 0`，说明发生了错误（例如，偏移量超出文件范围），函数直接返回错误码。

3. 接下来，函数计算在磁盘上的具体位置：

- `int cluster_offset = offset % cluster_size;` 计算在簇内的偏移量。
- `int start_sector = cluster_to_sector(data, cluster_to_use);` 将簇号转换为起始扇区号。
- `int start_pos = (start_sector * disk->sector_size) + cluster_offset;` 计算在磁盘上的精确起始位置（字节偏移）。

4. 函数确定本次要读取的数据量：

- `int total_to_read = total > cluster_size ? cluster_size : total;` 如果请求的总量大于一个簇的大小，则只读取一个簇的数据；否则读取请求的全部数据。

5. 然后，函数开始实际的读取操作：

- `if (seek_disk_stream(stream, start_pos) != 0)` 将磁盘流定位到计算出的起始位置。
- `if(read_disk_stream(stream, total_to_read, out) != 0)` 从磁盘流中读取数据到输出缓冲区。
- 如果上述操作任一失败，函数返回 `-EIO`（输入/输出错误）。

6. 最后，函数处理可能的跨簇读取情况：

- `total -= total_to_read;` 更新剩余需要读取的数据量。
- 如果 `total > 0`，说明还有数据需要读取，函数递归调用自身：
  - 更新偏移量：`offset + total_to_read`
  - 更新剩余需读取的总量：`total`
  - 更新输出缓冲区指针：`out + total_to_read`
- 这个递归调用确保了可以跨多个簇连续读取数据，直到读取完所有请求的数据。

7. 如果所有数据都已读取完毕（`total == 0`），函数返回 0 表示成功。

总的来说，`read_data_from_stream` 函数实现了 FAT16 文件系统中复杂的数据读取逻辑。它能够处理跨簇读取的情况，正确计算文件在磁盘上的物理位置，并通过递归调用来实现大文件的连续读取。这个函数是 FAT16 文件系统实现中的核心组件，确保了文件数据能够被正确且高效地读取。

get_cluster_for_offset 函数 `get_cluster_for_offset` 函数用于确定文件中给定偏移量所对应的簇。在 FAT16 文件系统中，文件的数据可能分散存储在多个簇中，这个函数帮助我们找到正确的簇来读取数据。

函数签名如下：

```
1  static int get_cluster_for_offset(struct disk* disk, int
       starting_cluster, int offset)
2  {
3      struct fat16_data* data = disk->data;
4      int cluster_size = data->header.header.sectors_per_cluster *
           disk->sector_size;
5      int cluster_to_use = starting_cluster;
6      int clusters_ahead = offset / cluster_size;
7      for (int i = 0; i < clusters_ahead; i++)
8      {
9          int entry = get_entry(disk, cluster_to_use);
10         if (entry == 0xFF8 || entry == 0xFFF)
11         {
12             return -EIO;
13         }
14
15         if (entry == 0xFF0 || entry == 0xFF6)
16         {
17             return -EIO;
18         }
19
20         if (entry == 0x00)
21         {
22             return -EIO;
23         }
24
25         cluster_to_use = entry;
26     }
27
28     return cluster_to_use;
29 }
```

让我们详细分析这个函数的每一步：

1. 函数参数：
   - `disk`: 指向磁盘结构的指针，包含文件系统信息。
   - `starting_cluster`: 文件的起始簇号。
   - `offset`: 在文件中的偏移量（字节）。
2. 初始化：
   - `struct fat16_data* data = disk->data;` 获取 FAT16 文件系统的数据结构。
   - `int cluster_size = data->header.header.sectors_per_cluster * disk->sector_si` 计算每个簇的大小（字节）。
   - `int cluster_to_use = starting_cluster;` 初始化要使用的簇为起始簇。
   - `int clusters_ahead = offset / cluster_size;` 计算需要前进的簇数。这是通过将偏移量除以簇大小来得到的。
3. 遍历 FAT 表：
   - 函数使用一个 for 循环，遍历 FAT 表 `clusters_ahead` 次。
   - 在每次迭代中，它调用 `get_entry(disk, cluster_to_use)` 来获取下一个簇的 FAT 表项。
4. 处理 FAT 表项：
   - 如果 entry 是 0xFF8 或 0xFFF，表示这是文件的最后一个簇。如果遇到这种情况但还没有到达目标偏移量，说明偏移量超出了文件范围，返回 -EIO 错误。
   - 如果 entry 是 0xFF0 或 0xFF6，这些是保留值，通常不应该遇到。如果遇到，返回 -EIO 错误。
   - 如果 entry 是 0x00，表示这是一个未使用的簇。在正常的文件中不应该遇到这种情况，所以也返回 -EIO 错误。
   - 如果 entry 是其他值，它代表下一个簇的编号。函数更新 `cluster_to_use` 为这个新的簇号。
5. 返回结果：

- 如果循环正常完成（没有遇到错误情况），函数返回 `cluster_to_use`，这就是包含目标偏移量的簇号。

这个函数的核心思想是通过 FAT 表来”遍历”文件的簇链。它从文件的起始簇开始，然后根据偏移量计算需要遍历多少个簇。在遍历过程中，它检查每个 FAT 表项，确保文件结构的完整性，并最终找到包含目标偏移量的正确簇。

这个函数在 FAT16 文件系统的实现中起着关键作用，它使得系统能够正确地定位和访问文件中的任意位置，即使文件的数据分散存储在多个不连续的簇中。通过这个函数，`read_data_from_stream` 和其他文件操作函数能够准确地找到并读取文件的任何部分。

4.4 VFS 虚拟文件系统

VFS（Virtual File System）虚拟文件系统是操作系统中的一个重要组件，它为不同的文件系统提供了一个统一的接口，使得应用程序可以以相同的方式访问不同类型的文件系统。在 soOS 中，VFS 的实现主要包括路径解析、文件系统初始化、文件打开和读取等功能。

4.4.1 PathParser 路径解析

路径解析是 VFS 的基础功能之一，它负责将用户输入的文件路径转换为系统可以理解和处理的数据结构。

数据结构 路径解析涉及以下主要数据结构：

```
struct path_root {
    int drive_no;
    struct path_part* first_part;
};

struct path_part {
    const char* part;
    struct path_part* next;
};
```

`path_root` 结构体表示一个完整的路径，包含驱动器号和指向第一个路径部分的指针。`path_part` 结构体表示路径的每一个部分，通过 `next` 指针链接成一个链表。

主要函数 路径解析的核心函数是 `parse`：

```
struct path_root* parse(const char* path)
{
    int drive_no = 0;
    const char* tmp = path;
    struct path_root* root = 0;

    if(strlen(path) > MAX_PATH_LEN)
    {
        return root;
    }

    drive_no = get_path_drive(&tmp);

    root = create_root(drive_no);
    if(!root)
    {
        return root;
    }

    struct path_part* first_part = parse_path(0, &tmp);
    if(first_part)
    {
        root->first_part = first_part;
    }

    struct path_part* last_part = first_part;
    while(last_part)
    {
        last_part = parse_path(last_part, &tmp);
    }

    return root;
}
```

函数调用逻辑 1. parse 函数首先调用 get_path_drive 获取驱动器号。2. 然后调用 create_root 创建根路径结构。3. 接着反复调用 parse_path 解析路径的每一部分。

功能与角色 parse 函数的主要作用是将字符串形式的路径转换为结构化的 path_root 对象。它通过分解路径字符串，创建一系列的 path_part 对象，并将它们链接在一起。

代码解析 函数首先检查路径长度，然后获取驱动器号。之后创建根路径结构，并开始解析路径的各个部分。每解析出一个部分，就创建一个新的 path_part 对象并加入到链表中。最后返回完整的 path_root 结构。

4.4.2 初始化文件系统

文件系统的初始化是 VFS 正常工作的前提，它设置了必要的数据结构和函数指针。

数据结构 文件系统初始化涉及以下主要数据结构：

```
struct filesystem {
    FS_OPEN_FILE open_file;
    FS_READ_FILE read_file;
    FS_CLOSE_FUNCTION close;
    FS_RESOLVE_FUNC resolve;
    FS_STAT_FUNCTION stat;
    char name[20];
};

struct filesystem* filesystems[MAX_FILESYSTEMS];
struct file_descriptor* file_descriptors[MAX_FILE_DESCRIPTORS];
```

filesystem 结构体定义了一个文件系统应该实现的接口函数。filesystems 数组存储了系统支持的所有文件系统，而 file_descriptors 数组存储了所有打开的文件描述符。

主要函数 文件系统初始化的核心函数是 init_fs：

```
void init_fs()
{
    memset(file_descriptors, 0, sizeof(file_descriptors));
    load_fs();
}

static void load_fs()
{
    memset(filesystems, 0, sizeof(filesystems));
    insert_filesystem(init_fat16());
}
```

函数调用逻辑 1. `init_fs` 函数首先清空文件描述符数组。2. 然后调用 `load_fs` 函数。3. `load_fs` 函数清空文件系统数组，并调用 `insert_filesystem` 插入 FAT16 文件系统。

功能与角色 `init_fs` 函数的主要作用是初始化 VFS 系统，包括清空文件描述符数组和加载支持的文件系统。

代码解析 函数首先使用 `memset` 清空 `file_descriptors` 数组，确保没有残留的文件描述符。然后调用 `load_fs` 函数，该函数清空 `filesystems` 数组，并通过 `insert_filesystem` 函数插入 FAT16 文件系统。这里只插入了 FAT16 文件系统，但设计允许未来 easily 添加更多文件系统支持。

4.4.3 打开文件

文件打开是 VFS 的核心功能之一，它负责定位文件并创建文件描述符。

数据结构 文件打开涉及以下主要数据结构：

```
struct file_descriptor {
    int index;
    struct filesystem* fs;
    void* data;
    struct disk* disk;
};
```

`file_descriptor` 结构体表示一个打开的文件，包含文件描述符索引、使用的文件系统、私有数据和所在的磁盘。

主要函数 文件打开的核心函数是 `fopen`：

```
int fopen(const char* filename, const char* mode)
{
    struct path_root* root = parse(filename);
    if(!root || !root->first_part)
    {
        return -1;
    }

    struct disk* disk = get_disk(root->drive_no);
    if(!disk || !disk->filesystem)
    {
```

```
12              return -1;
13          }
14
15          FILE_OPEN_MODE open_mode = get_file_open_mode(mode);
16          if(open_mode == FILE_MODE_INVALID)
17          {
18              return -1;
19          }
20
21          void* data_to_descriptor = (*disk->filesystem->open_file)(disk,
                root->first_part, open_mode);
22
23          struct file_descriptor* fd = 0;
24          if(get_new_file_descriptor(&fd) != 0)
25          {
26              return -1;
27          }
28
29          fd->fs = disk->filesystem;
30          fd->data = data_to_descriptor;
31          fd->disk = disk;
32          return fd->index;
33      }
```

函数调用逻辑 1. `fopen` 函数首先调用 `parse` 解析文件路径。2. 然后调用 `get_disk` 获取对应的磁盘。3. 调用 `get_file_open_mode` 获取文件打开模式。4. 调用文件系统的 `open_file` 函数打开文件。5. 调用 `get_new_file_descriptor` 获取新的文件描述符。

功能与角色 `fopen` 函数的主要作用是打开一个文件并返回文件描述符。它负责解析文件路径、确定文件所在的磁盘和文件系统，然后调用具体文件系统的打开函数，最后创建并返回一个文件描述符。

代码解析 函数首先解析文件路径，然后获取对应的磁盘和文件系统。之后它确定文件打开模式，并调用具体文件系统的 `open_file` 函数打开文件。最后，它创建一个新的文件描述符，填充相关信息，并返回描述符的索引。

4.4.4 读取文件

文件读取是 VFS 的另一个核心功能，它负责从打开的文件中读取数据。

数据结构 文件读取主要使用前面定义的 `file_descriptor` 结构。

主要函数 文件读取的核心函数是 `fread`：

```
int fread(void* ptr, uint32_t size, uint32_t count, int fd)
{
    if(fd < 0 || fd >= MAX_FILE_DESCRIPTORS || size <= 0 || count
       <= 0)
    {
        return -1;
    }

    struct file_descriptor* file = get_file_descriptor(fd);
    if(!file)
    {
        return -1;
    }

    return (*file->fs->read_file)(file->disk, file->data, size,
       count, ptr);
}
```

函数调用逻辑 1. `fread` 函数首先进行参数检查。2. 然后调用 `get_file_descriptor` 获取文件描述符。3. 最后调用文件系统的 `read_file` 函数读取文件内容。

功能与角色 `fread` 函数的主要作用是从指定的文件中读取数据。它负责验证文件描述符，然后调用具体文件系统的读取函数来完成实际的数据读取操作。

代码解析 函数首先检查输入参数的有效性，包括文件描述符、读取大小和数量。然后它获取对应的文件描述符结构。最后，它调用文件系统特定的 `read_file` 函数来执行实际的读取操作，并返回读取的结果。

通过这种设计，VFS 实现了对不同文件系统的统一抽象，使得上层应用可以用统一的接口访问不同类型的文件系统，大大提高了系统的灵活性和可扩展性。

第 5 章　进程管理

5.1　初始化 GDT

encode_gdt_entry 函数

功能：编码 GDT（全局描述符表）条目。

- **参数**：
  - `uint8_t* dest`：目标数组，用于存储编码后的 GDT 条目。
  - `struct gdt_structure src`：包含 GDT 条目信息的结构体。
- **过程**：
  - **检查限制**：
    - 如果 `limit` 大于 65536 且未正确对齐，调用 `panic` 函数报告错误。
  - **设置粒度**：
    - 如果 `limit` 大于 65536，右移 12 位，并设置粒度位（`dest[6]`）。
  - **编码**：
    - **限制（limit）**：编码在 `dest[0]`、`dest[1]` 和 `dest[6]` 的低 4 位。
    - **基址（base）**：编码在 `dest[2]` 到 `dest[4]` 和 `dest[7]`。
    - **类型（type）**：编码在 `dest[5]`。

gdt_structure_to_gdt_entry 函数

功能：将 GDT 结构数组转换为 GDT 条目。

- **参数**：
  - `struct gdt_structure *gdt`：指向源 GDT 结构的指针。
  - `struct gdt_entry *entry`：指向目标 GDT 条目的指针。
  - `int total_entries`：总条目数。
- **过程**：
  - 遍历每个 GDT 结构，调用 `encode_gdt_entry` 进行编码。

汇编代码（load_gdt）

功能：加载 GDT 到处理器中。

- **过程**：
  - 从堆栈（`%esp`）加载参数：
    - 基址和长度。
  - 将这些值移动到 `gdt_desc`。
  - 使用 `lgdt` 指令加载 GDT 描述符。

数据段（gdt_desc）

- **结构**：
  - `.word 0`：GDT 的长度。
  - `.long 0`：GDT 的基址。

这个代码用于设置和加载 GDT，为操作系统的段管理提供基础支持。

5.2 process 处理

process_load_for_slot

- **功能**：将指定文件加载到进程指定的槽位中。
- **参数**：
  - `filename`：要加载的程序文件名。
  - `process`：指向指针的指针，用于返回加载后的进程结构体指针。
  - `process_slot`：表示要加载进程的槽位索引。
- **返回值**：操作结果的返回值，可能是成功的返回值或错误码（例如 `-EISTKN` 表示槽位已被占用，`ENOMEM` 表示内存分配失败等）。

process_load

- **功能**：将指定文件加载到第一个可用的进程槽位中。
- **参数**：
  - `filename`：要加载的程序文件名。

  - process：指向指针的指针，用于返回加载后的进程结构体指针。

- **返回值**：操作结果的返回值，可能是成功的返回值或错误码（例如 -EISTKN 表示没有可用的槽位，ENOMEM 表示内存分配失败等）。

struct process 结构体

- **描述**：表示一个进程的结构体。
- **成员变量**：
  - id：进程的唯一标识符，类型为 uint16_t。
  - filename：进程关联的文件名，长度不超过 MAX_PATH。
  - task：指向进程主任务的指针，类型为 struct task*。
  - allocations：进程的内存分配数组，存储了多个 void* 类型的指针，最多 MAX_PROGRAM_ALLOCATIONS 个。
  - ptr：指向进程内存的物理指针，类型为 void*。
  - stack：指向进程栈内存的物理指针，类型为 void*。
  - size：指针 ptr 所指数据的大小，类型为 uint32_t。

process_init

- **功能**：对给定的 struct process 进行初始化，将其内存清零。
- **参数**： struct process* process，需要被初始化的进程结构体指针。
- **实现**：使用 memset 函数将 process 指针所指向的内存清零，确保所有成员变量初始状态为零。

process_current

- **功能**：返回当前正在运行的进程的指针。
- **返回值**： struct process*，当前正在运行的进程的指针。

process_get

- **功能**：根据进程 ID 获取进程结构体指针。
- **参数**： int process_id，需要获取的进程的 ID。

- **返回值：**
  - 如果 process_id 不在有效范围内（小于 0 或大于等于 MAX_PROCESSES），返回 NULL。
  - 否则，返回 processes 数组中对应 process_id 的进程结构体指针。

process_load_binary

- **功能：**加载二进制文件到进程的内存中。
- **参数：**
  - const char* filename：要加载的文件名。
  - struct process* process：目标进程结构体指针，加载后将文件内容保存在其中。
- **返回值：**加载操作的结果，可能是成功的返回值或错误码（如 -EIO、-ENOMEM、-EISTKN 等）。

process_load_data

- **功能：**调用 process_load_binary 加载数据文件到进程的内存中。
- **参数：**
  - const char* filename：要加载的文件名。
  - struct process* process：目标进程结构体指针，加载后将文件内容保存在其中。
- **返回值：**同 process_load_binary 的返回值。

process_map_binary

- **功能：**将二进制程序映射到进程的虚拟地址空间。
- **参数：** struct process* process，目标进程结构体指针。
- **返回值：**操作结果的返回值。

`process_map_memory`

- **功能**：将二进制程序和堆栈映射到进程的虚拟地址空间。
- **参数**： `struct process* process`，目标进程结构体指针。
- **返回值**：操作结果的返回值。

`process_load_for_slot`

- **功能**：为给定的进程槽位加载文件并创建进程。
- **参数**：
  - `const char* filename`：要加载的文件名。
  - `struct process** process`：输出参数，返回加载的进程结构体指针。
  - `int process_slot`：进程槽位索引。
- **返回值**：操作结果的返回值，可能是成功的返回值或错误码。

`process_get_free_slot`

- **功能**：获取一个空闲的进程槽位索引。
- **返回值**：
  - 如果找到空闲槽位，返回该索引。
  - 如果没有空闲槽位，返回 `-EISTKN`。

`process_load`

- **功能**：加载程序文件到一个新的进程中。
- **参数**：
  - `const char* filename`：要加载的程序文件名。
  - `struct process** process`：输出参数，用于返回加载的进程结构体指针。
- **返回值**：操作结果的返回值，可能是成功的返回值或错误码。

task_new

- **功能**：创建一个新的任务并关联给定的进程。
- **参数**：
  - process：指向要关联的进程的指针。
- **返回值**：返回一个指向新创建任务结构体的指针，或者返回相应的错误码（如 -ENOMEM 表示内存分配失败）。

task_current

- **功能**：获取当前运行的任务。
- **返回值**：返回指向当前任务结构体的指针。

task_get_next

- **功能**：获取下一个任务。
- **返回值**：返回指向下一个任务结构体的指针。

task_free

- **功能**：释放任务占用的内存和资源。
- **参数**：
  - task：要释放的任务结构体指针。
- **返回值**：返回释放操作的结果，通常为零（表示成功）。

task_init

- **功能**：初始化任务结构体。
- **参数**：
  - task：要初始化的任务结构体指针。
  - process：指向要关联的进程的指针。
- **返回值**：返回初始化操作的结果，通常为零（表示成功）。

task_switch

- **功能**：切换当前运行的任务到指定的任务。
- **参数**：
  - `task`：要切换到的目标任务结构体指针。
- **返回值**：返回切换操作的结果，通常为零（表示成功）。

task_page

- **功能**：切换到用户模式并执行当前任务。
- **返回值**：返回切换操作的结果，通常为零（表示成功）。

task_run_first_ever_task

- **功能**：运行第一个任务。
- **注意**：此函数要求当前任务必须存在。

save_task_state

- **功能**：保存任务的状态。
- **参数**：
  - `task`：要保存状态的任务结构体指针。
  - `frame`：中断帧，包含了任务状态的详细信息。

save_current_task_state

- **功能**：保存当前任务的状态。
- **注意**：此函数要求当前任务必须存在。

copy_string_from_task

- **功能**：从指定任务的虚拟地址复制字符串到物理地址。
- **参数**：
  - `task`：要复制字符串的任务结构体指针。
  - `virt_addr`：源字符串的虚拟地址。

  - `phys_addr`：目标字符串的物理地址。
  - `max`：最大允许复制的字符串长度。

- **返回值**：返回复制操作的结果，通常为零（表示成功）。

汇编函数

以下是几个汇编函数，用于处理任务返回和寄存器恢复：

- **`task_return`**：任务返回函数的汇编实现，用于切换到用户模式执行。
- **`restore_general_purpose_registers`**：恢复通用寄存器状态的汇编函数。
- **`user_registers`**：设置用户模式下的段寄存器的汇编函数。
- **`kernel_registers`**：设置内核模式下的段寄存器的汇编函数。

5.3 tss

进行提权的栈切换和切换大量寄存器

I/O Map Base Address	Reserved	T
Reserved	LDT Segment Selector	
Reserved	GS	
Reserved	FS	
Reserved	DS	
Reserved	SS	
Reserved	CS	
Reserved	ES	
EDI		
ESI		
EBP		
ESP		
EBX		
EDX		
ECX		
EAX		
EFLAGS		
EIP		
CR3 (PDBR)		
Reserved	SS2	
ESP2		
Reserved	SS1	
ESP1		
Reserved	SS0	
ESP0		
Reserved	Previous Task Link	

图 5.1: tss

- `prev_tss`：指向前一个 TSS（任务状态段）的链接字段。
- `esp0`：内核栈的栈顶指针。
- `ss0`：内核数据段选择子。
- `esp1`：环境 1 的栈顶指针。
- `esp2`：环境 2 的栈顶指针。
- `ss2`：环境 2 的数据段选择子。
- `sr3`：PDBR（页目录基址寄存器）的高 32 位。
- `eip`：指令指针。
- `eflags`：标志寄存器。
- `eax`、`ecx`、`edx`、`ebx`：通用寄存器。
- `esp`：用户栈的栈顶指针。
- `ebp`：栈基指针。
- `esi`、`edi`：源索引和目的索引寄存器。
- `es`、`cs`、`ss`、`ds`、`fs`、`gs`：段寄存器，分别为额外、代码、堆栈、数据、附加数据和附加代码段。
- `ldtr`：局部描述符表的选择子。
- `iopb`：I/O 位图基址偏移量。

5.4 process 流程

先获取空闲槽，之后进行对文件的加载读取，process 的初始化，之后进行映射。

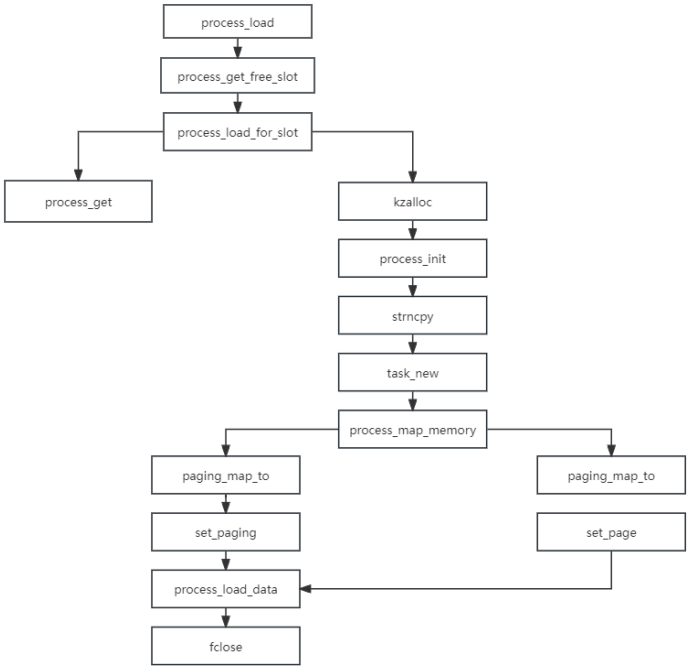

图 5.2: process 流程图

5.5 进程切换流程

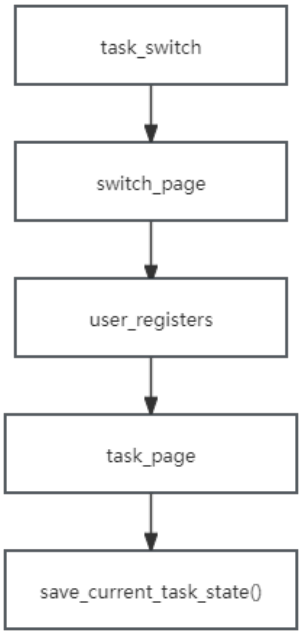

图 5.3: 进程切换流程图

先保护现场，之后上下文切换，最后恢复现场。通过 run_first_ever_task 函数运行第一个任务。调用切换和返回函数来维护进程调度

5.6 中断处理

1. **硬件中断发生**

   - CPU 根据中断号跳转到相应的中断处理程序入口。

2. **int21h 和 ignore_int 中断处理程序**

```
int21h:
    pushal
    call int21h_handler
    popal
    iret

ignore_int:
    pushal
    call ignore_int_handler
    popal
    iret
```

3. **isr80h_wrapper 中断处理程序**

```
isr80h_wrapper:
    pushal
    pushl %esp
    pushl %eax
    call isr80h_handler
    movl %eax, tmp_res
    addl $8, %esp
    popal
    movl tmp_res, %eax
    iret
```

4. **具体的中断处理函数**

```
void int21h_handler() {
    print("Keyboard pressed!\n");
```

```
    outb(0xa0, 0x20);
    outb(0x20, 0x20);
}

void ignore_int_handler() {
    outb(0xa0, 0x20);
    outb(0x20, 0x20);
}

void* isr80h_handler(int command, struct interrupt_frame* frame) {
    void* res = 0;
    page_kernel();
    save_current_task_state(frame);
    res = isr80h_handle_command(command, frame);
    task_page();
    return res;
}
```

5. 系统调用注册与处理

```
void isr80h_register_command(int command_id, ISR80H_COMMAND command)
    if(command_id >= MAX_ISR80H_COMMANDS || command_id < 0) {
        panic("command_id out of range");
    }
    if(isr80h_commands[command_id] != 0) {
        panic("command_id already registered");
    }
    isr80h_commands[command_id] = command;
}

void* isr80h_handle_command(int command, struct interrupt_frame* fram
    if(command >= MAX_ISR80H_COMMANDS || command < 0) {
        panic("command out of range");
    }
    if(isr80h_commands[command] == 0) {
        panic("command not registered");
    }
    return isr80h_commands[command](frame);
```

}

以下是中断处理的流程图，用于展示上述函数调用的关系和执行顺序：

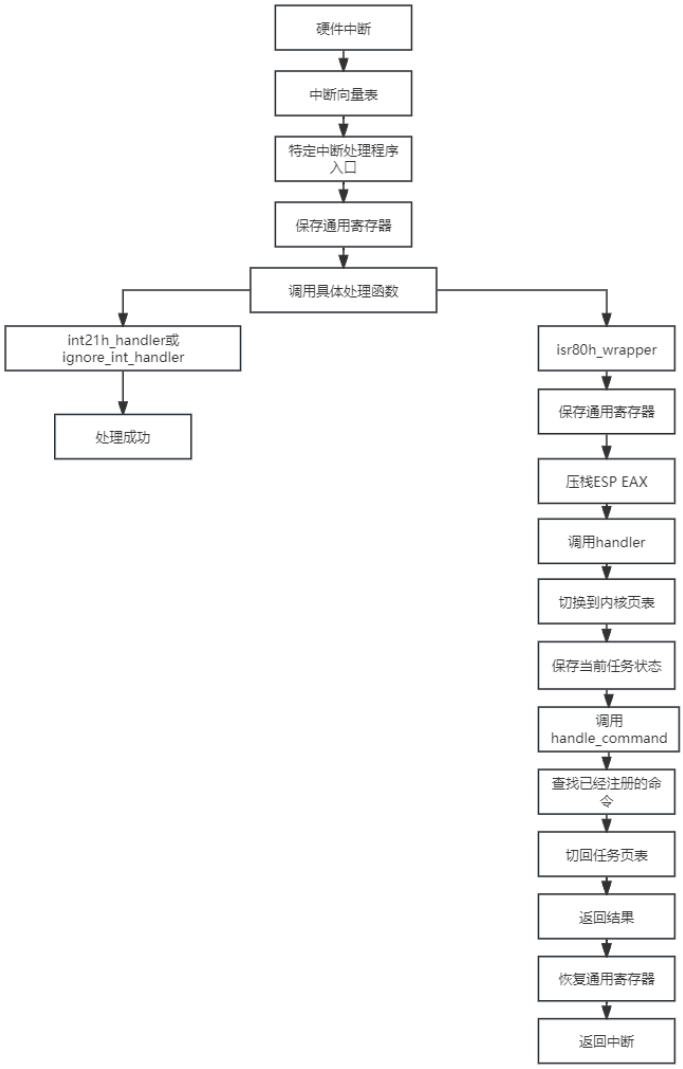

图 5.4: 中断处理流程图

1. **硬件中断发生：**当外部设备或软件触发中断时，CPU 会根据中断向量表（IDT）跳转到相应的中断处理程序。

2. **中断处理程序：**`int21h` 和 `ignore_int` 是简单的中断处理程序，它们主要保存当前寄存器状态，调用相应的处理函数，最后恢复寄存器并返回中断。`isr80h_wrapper` 更为复杂，用于处理系统调用，包括保存寄存器，管理栈，调用处理函数，并返回结果。

3. **具体的中断处理函数：**这些函数执行特定的任务，例如处理键盘输入（`int21h_handler`），忽略特定中断（`ignore_int_handler`），或处理系统调用（`isr80h_handler`）。

4. **系统调用注册与处理:**系统调用通过 `isr80h_register_command` 注册,通过 `isr80h_handle_comm` 执行。命令处理函数如 `isr80h_command0_sum` 和 `isr80h_command1_print` 负责执行具体的系统调用操作。

运行结果如下

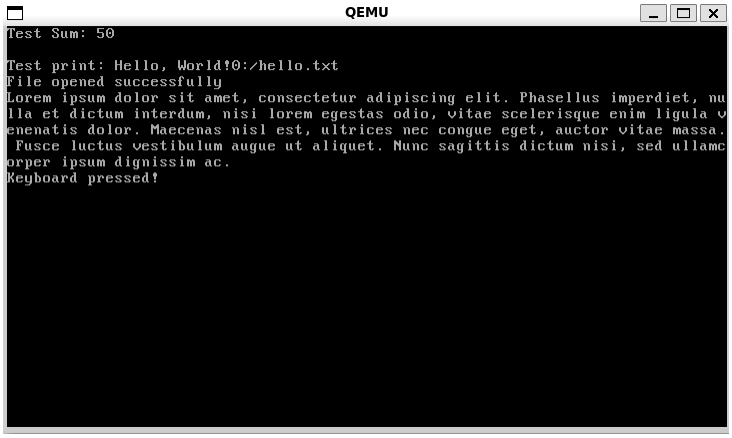

图 5.5: 运行结果

成功执行三个系统调用