**《操作系统课程设计》**

**（2023/2024学年第一学期）**

**1引言**

操作系统是计算机系统的核心软件，它管理和控制计算机的硬件和软件资源，为用户和应用程序提供服务和接口。操作系统的设计和实现涉及到计算机科学的多个领域，如计算机体系结构、编译原理、数据结构、算法、并发编程、系统安全等。因此，学习操作系统的原理和实现对于提高计算机专业学生的综合素养和能力具有重要的意义。

本课程设计参考学习《操作系统真象还原》，掌握操作系统的基本原理和实现方法，从零开始开发一个简易的操作系统。本课程设计的内容包括理解操作系统的概念、功能和结构，学习保护模式、中断、内存管理、线程、用户进程、文件系统等操作系统的核心技术，使用汇编语言和C语言编写操作系统的启动代码、内核代码和应用程序代码，使用Bochs虚拟机进行操作系统的编译、运行和调试，实现一个键盘输入、鼠标操作、多任务切换、文件操作等功能的简易操作系统。

本课程设计的目的是通过实践的方式，深入了解操作系统的工作原理和实现方法，提高自己的编程能力和系统思维，为将来的计算机研究和开发打下坚实的基础。本课程设计的问题是如何从零开始开发一个简易的操作系统，实现操作系统的基本功能和特性，以及学习如何对操作系统进行测试和优化。

**1.1任务要求**

在计算机科学领域，操作系统是计算机系统的核心软件，扮演着管理和控制硬件与软件资源、为用户及应用程序提供服务与接口的关键角色。学习操作系统的原理和实现方法是提高计算机专业学生综合素养和能力的关键步骤。

我们的任务是基于《操作系统真象还原》的学习和参考，从零开始开发一个简易的操作系统。这包括理解操作系统的概念、功能和结构，掌握保护模式、中断、内存管理、线程、用户进程、文件系统等核心技术。我们将使用汇编语言和C语言编写启动代码、内核代码和应用程序代码，并在Bochs虚拟机中进行编译、运行和调试。

通过实践这一任务，我们将深入了解操作系统的工作原理和实现方法，提高编程能力和系统思维。这有助于培养对计算机系统的整体理解，为将来的计算机研究和开发奠定坚实基础。将通过亲身经历，巩固课堂学习，使理论知识更具实际应用性。

本报告将会从整体上介绍该简易操作系统中的Boot启动模块、Memory内存管理以及文件系统，展示这几个模块设计的关键思路。

**1.2选题**

D自主命题：在x86平台模拟软件bochs上搭建简易操作系统

组队完成，本报告只负责介绍Boot模块、内存管理以及文件系统

**2需求分析与设计**

**2.1需求分析**

本项目的选题是自制一个简易的操作系统，目的是学习和实践操作系统的设计和实现的过程和方法，实现操作系统的基本功能和服务，展示操作系统的工作过程和效果，验证操作系统的理论知识和技术技巧，提高操作系统的开发能力和水平。本项目的用户主要是计算机专业的学生和教师，以及对操作系统感兴趣的开发者和爱好者。本项目的场景主要是在虚拟机上运行和调试本项目的操作系统。本项目的需求分析如下：

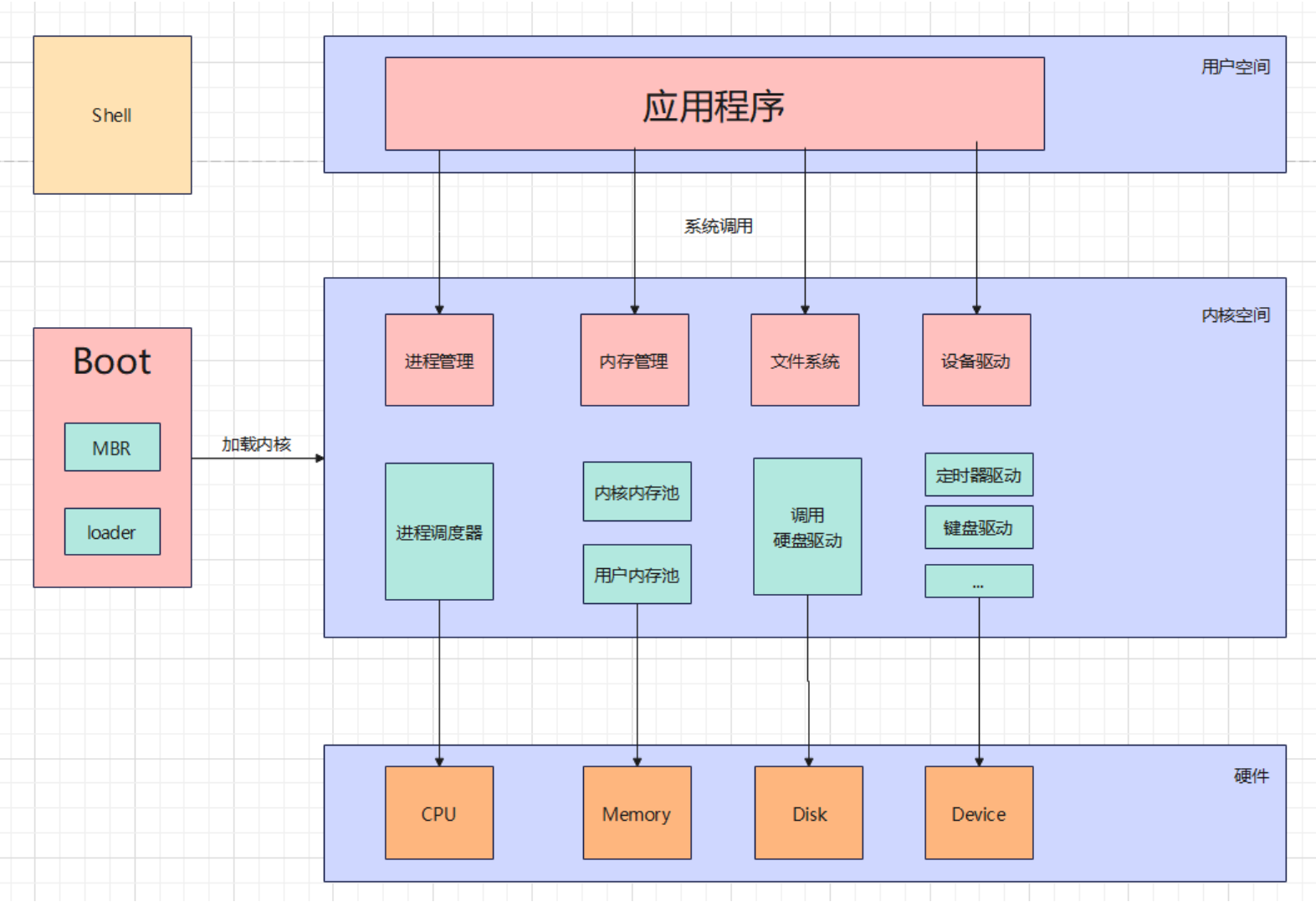
引导加载需求：本项目需要实现操作系统的引导加载，即将操作系统的内核从磁盘加载到内存中，并跳转到保护模式，初始化全局描述符表和中断描述符表，以及内核分页。

内存管理需求：本项目需要实现操作系统的内存管理，即分配和回收内存，实现内存的保护和隔离，实现内存的映射和共享，实现虚拟内存和分页机制等。

文件系统需求：本项目需要实现操作系统的文件系统，即存储和管理文件，实现文件的创建和删除，实现文件的读写和定位，实现文件的目录和索引，实现文件的缓存和安全等。

**2.2系统整体功能框架图**

SimpleOS整体框架图

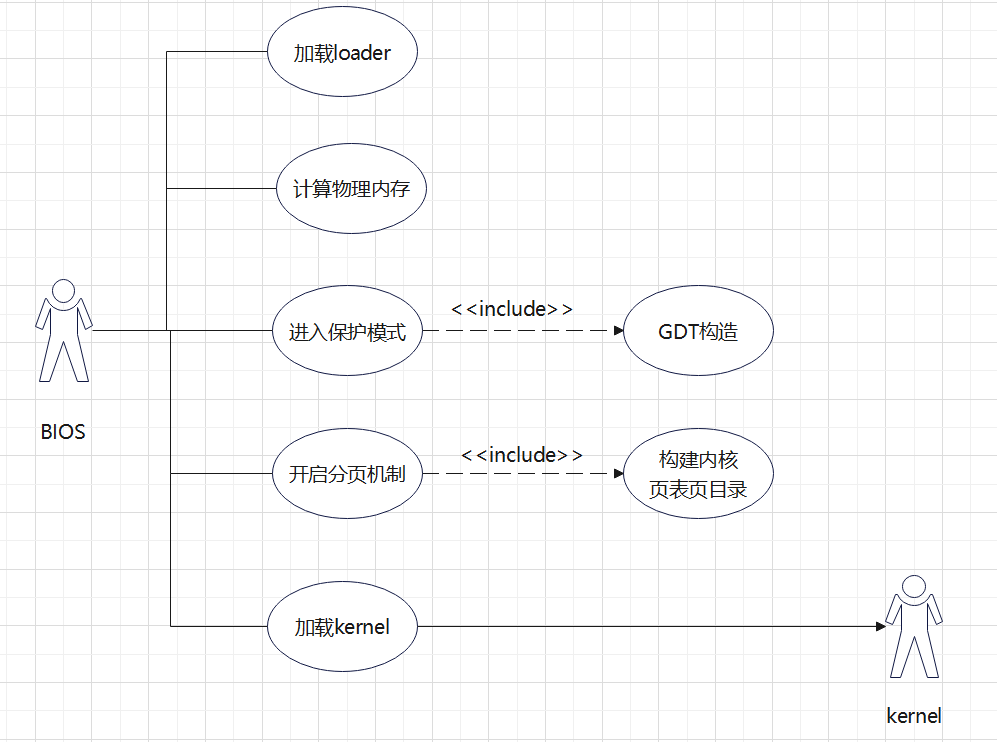


系统整体功能大致可以分为6个部分，负责启动加载内核的boot模块，负责提供用户交互的Shell模块，以及进程管理、内存管理、文件系统和设备驱动4个子系统模块

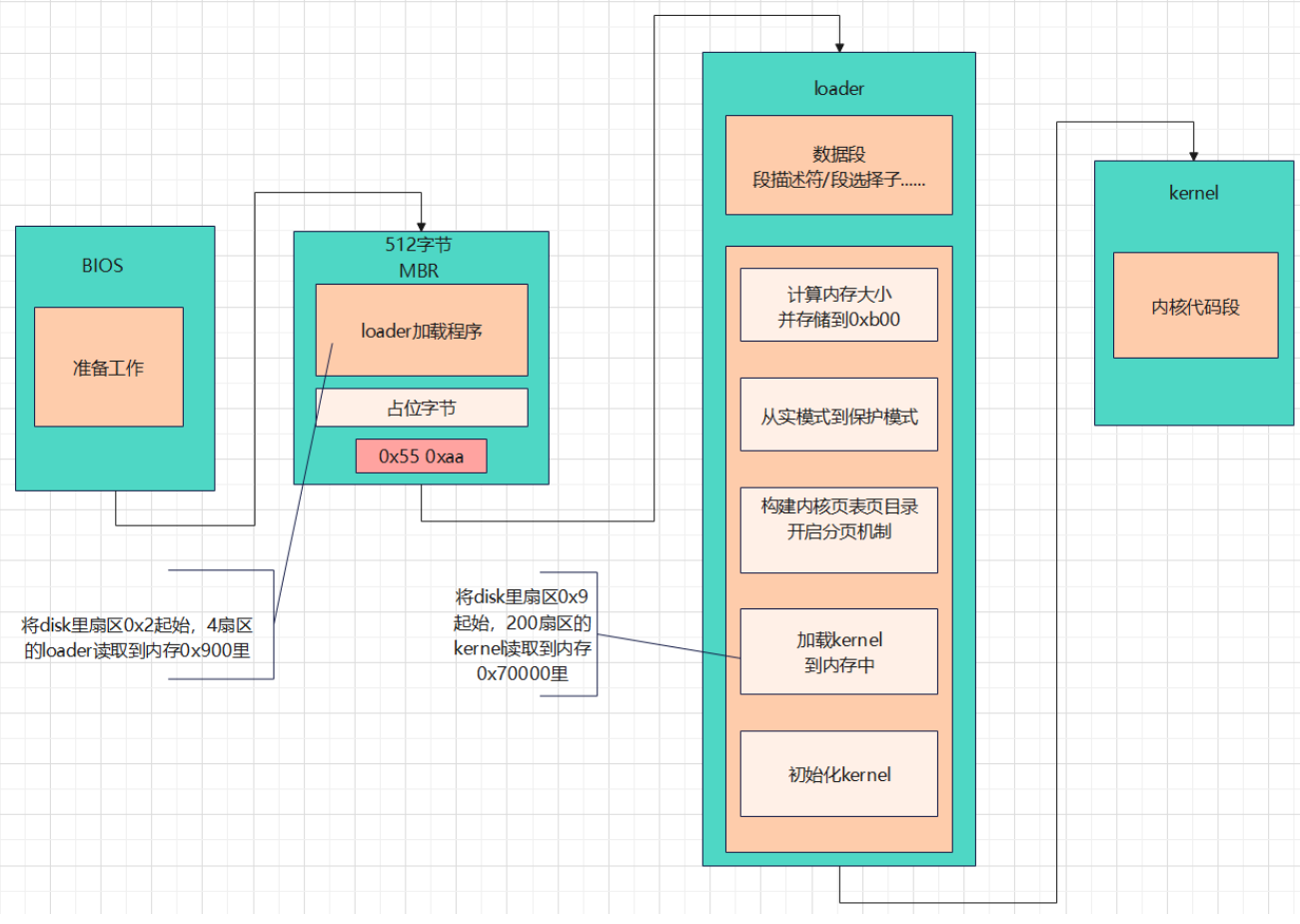
**2.3模块描述和框架展示**

**2.3.1 SimpleOS-Boot模块**

用例图：



系统功能结构图：



Boot模块：boot模块是系统启动时执行的模块

BIOS：基本输入输出系统。PC机通电后接力的第一棒就是交到BIOS手中，它负责检测和初始化硬件的准备工作，然后把MBR装载到内存后，将接力棒交给MBR。

MBR：主引导记录，主要负责把位于扇区0x2起始,大小4扇区的loader程序加载到内存中，并将接力棒交给loader。MBR的末尾两字节必须是0x55和0xaa，保证BIOS可以正确检测并加载MBR

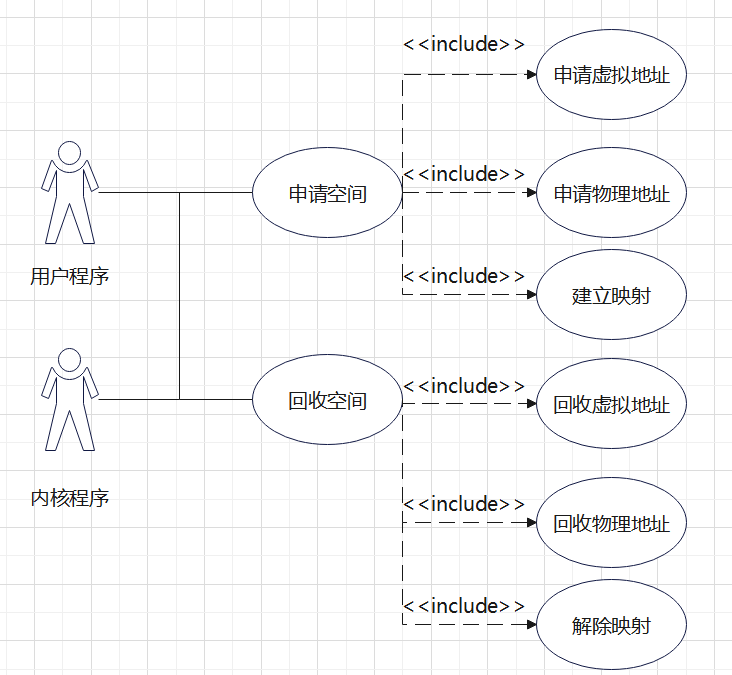
loader：内核程序加载程序，最主要的功能时加载内核程序。除此之外还要做一些准备工作，如：计算内存大小并存储、从实模式切换到保护模式、构建内核页表页目录开启分页。最后加载位于扇区0x9起始,大小200扇区内核代码到内存中，并安装elf文件格式拓展(初始化)kernel.bin的代码

kernel:内核代码，本质上是一个while循环

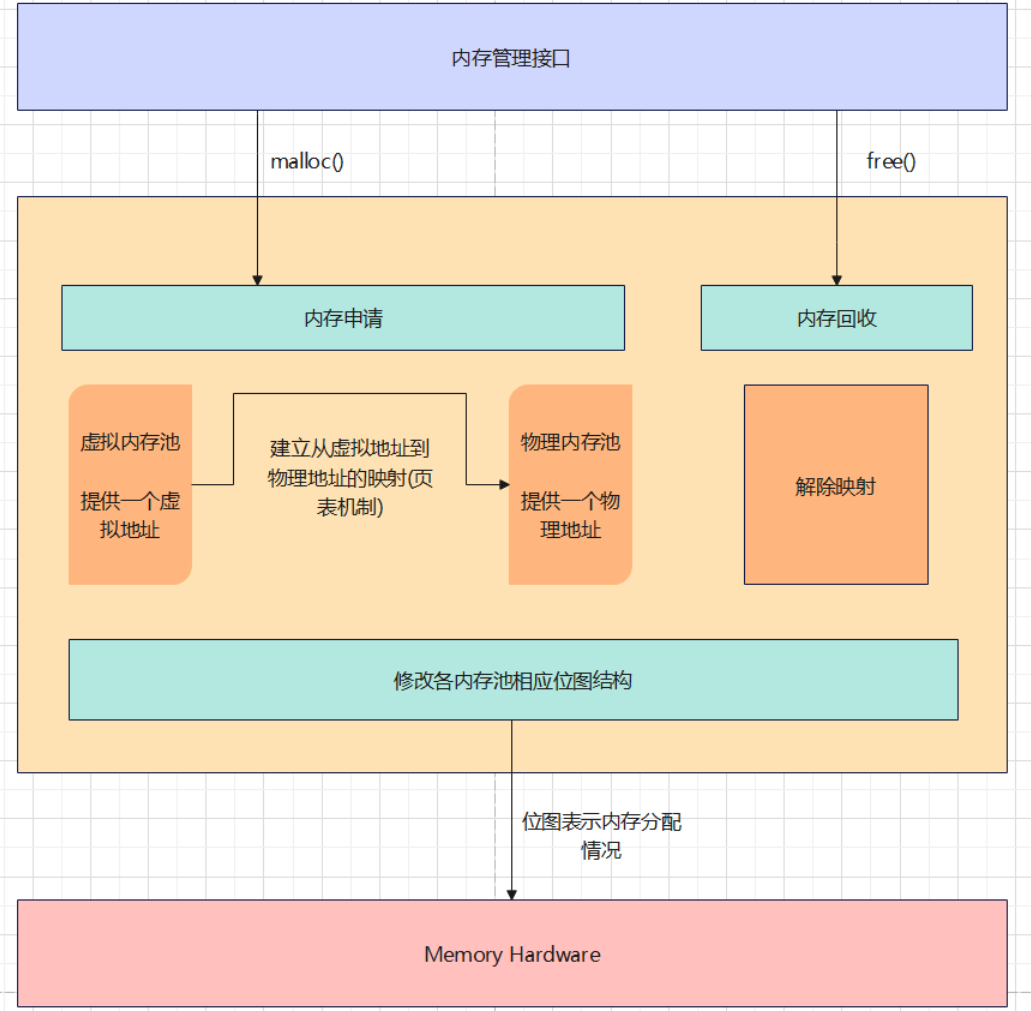
：

**2.3.2 SimpleOS-内存管理模块**

用例图：

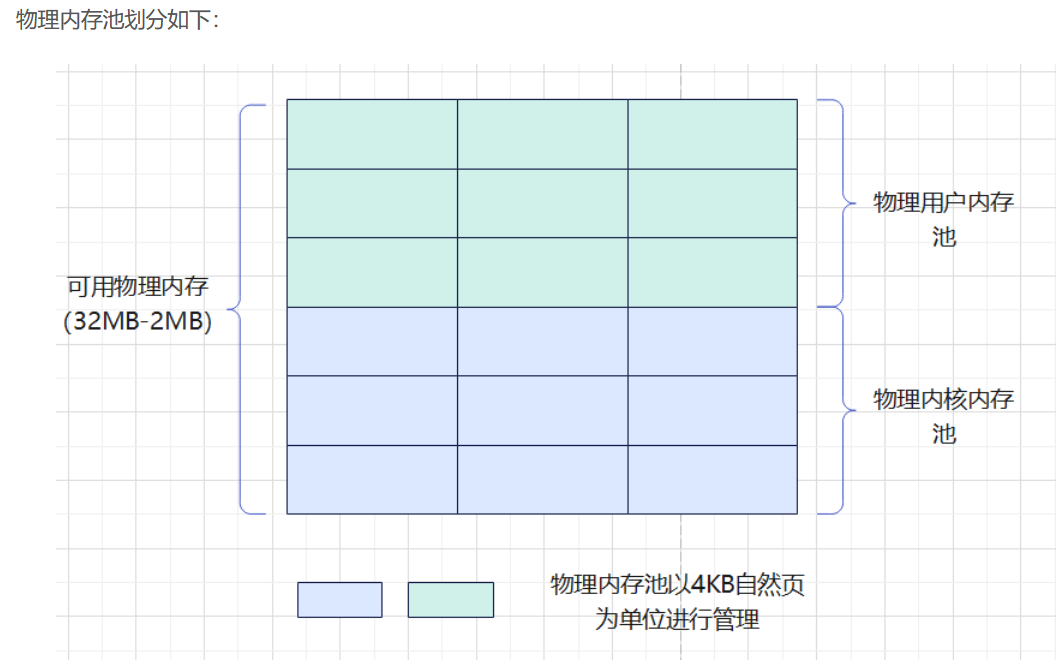


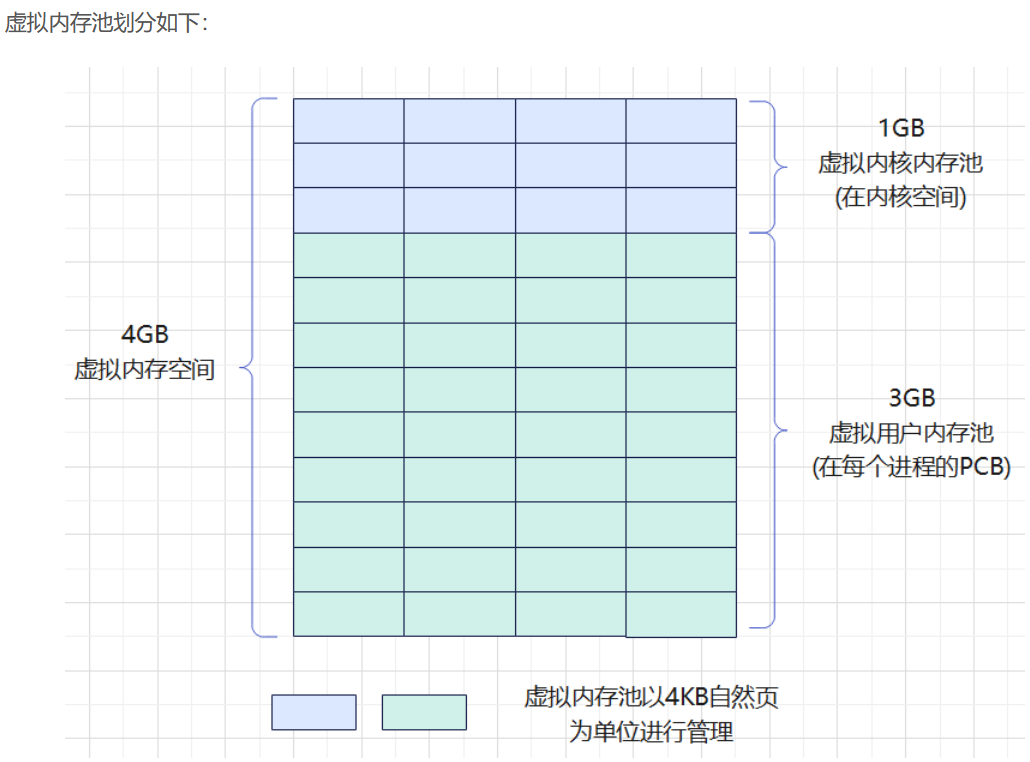
系统功能结构图：



内存管理的核心是维护内存池，利用位图结构来实现物理内存池、虚拟内存池对内存进行管理。其中物理内存池以及虚拟内存池中又分别可以划分为用户内存池和内核内存池，总共4种内存池。内存管理的本质其实是地址的提供以及映射的建立和删除。内存池提供物理地址和虚拟地址，并在中间使用二级页表机制建立或删除映射，实现内存的分配。

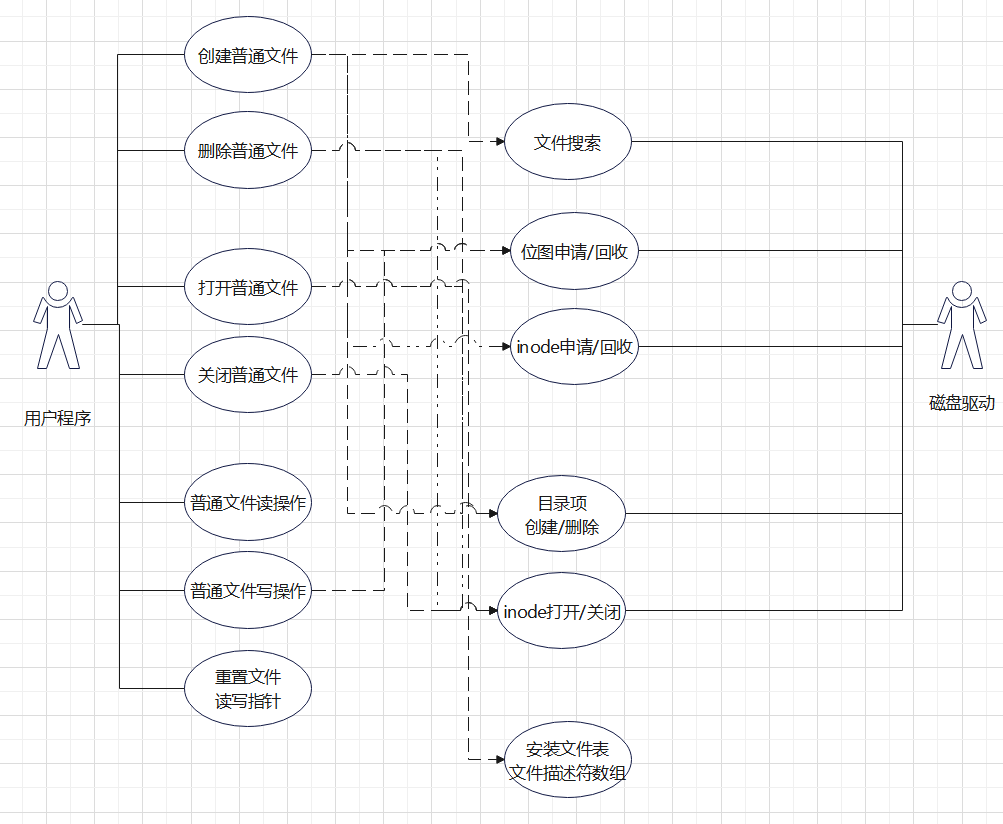
内存管理的内存池空间划分如下：

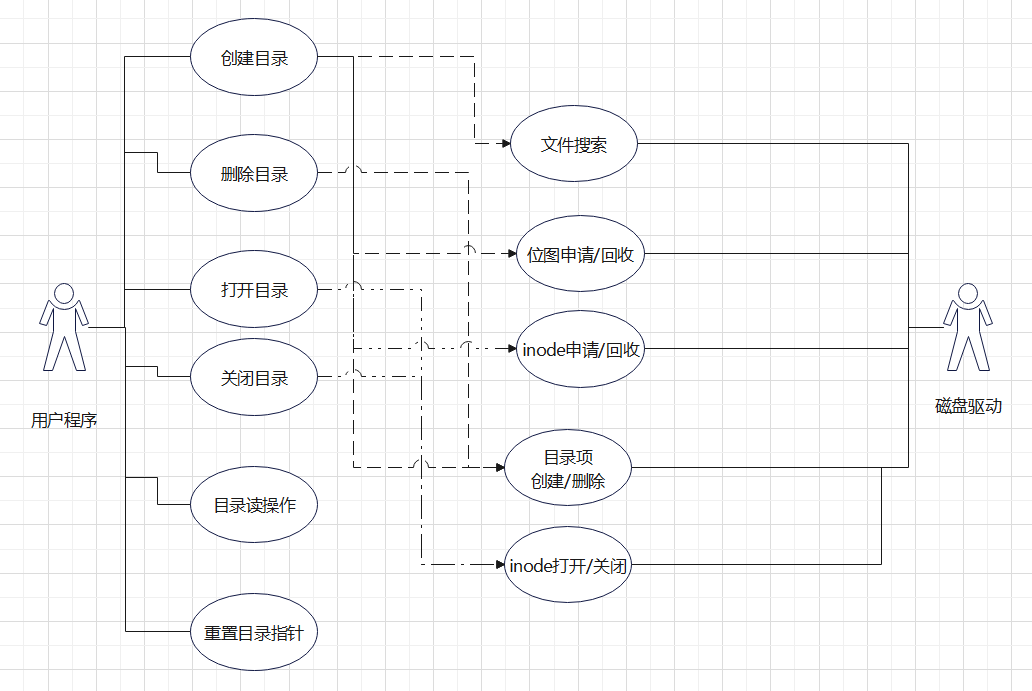




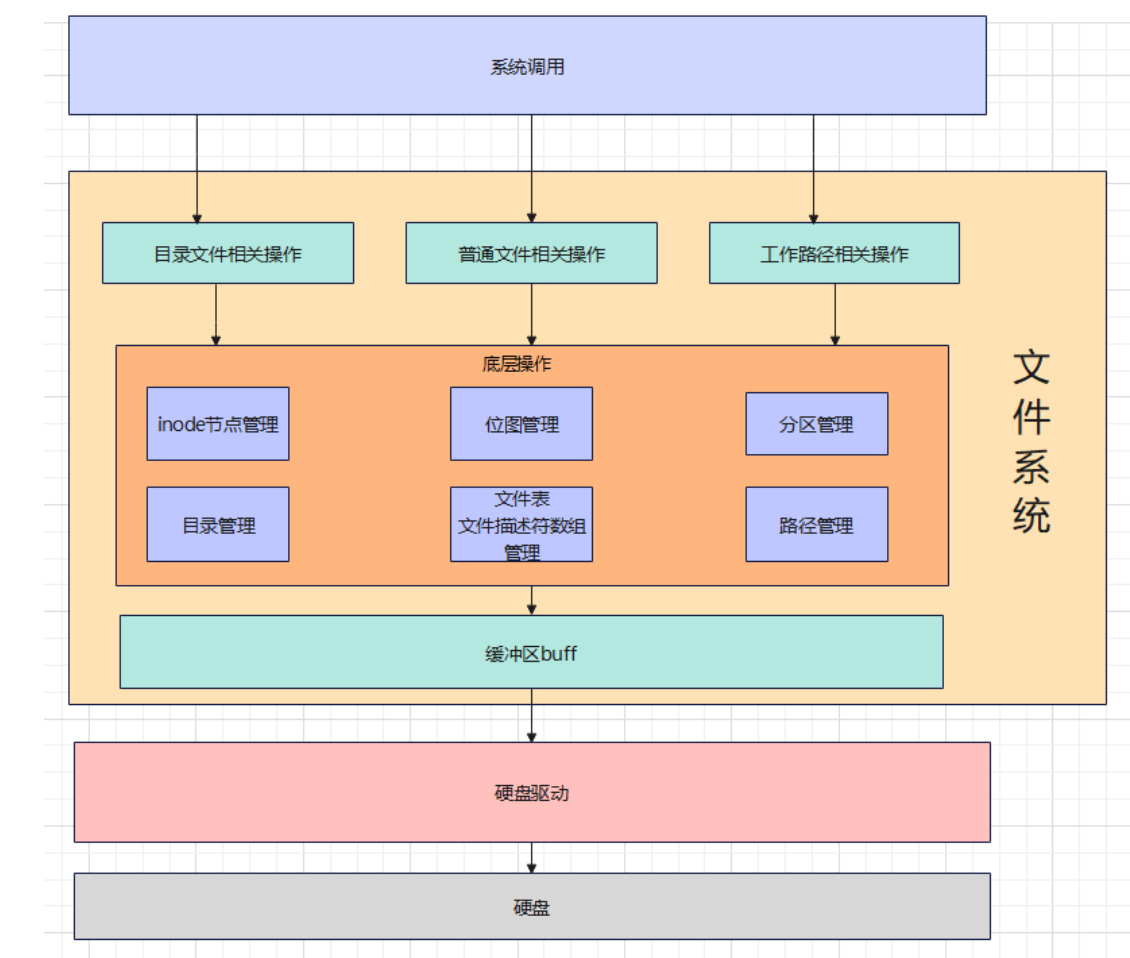
**2.3.3 SimpleOS-文件系统模块**

用例图：





系统功能结构图：



**3数据结构**

**3.1 SimpleOS-Boot模块**

Boot模块由汇编代码编写，并无较为复杂的数据结构，Boot模块的数据结构主要是进入实模式时所需要构造的GDT以及选择子，还有开启页表机制时需要的页表项构造和页目录项构造。GDT、选择子、页表项以及页目录项均需要按照规定的格式组装而成。

;构建gdt及其内部的描述符

;一共放入4个段描述符，分别是空段描述符、代码段描述符、数据/栈段描述符（共用一个段）、显存段描述符

;由于小端序，所以高位字节应放置在高地址(越靠下面地址越高),低位字节放置在低地址

GDT\_BASE:   dd 0x00000000           ;我们说过gdt的第一个段描述符不能用，所以全部置0

            dd 0x00000000

CODE\_DESC:  dd 0x0000FFFF           ;代码段的段界限应该为0xfffff，段基址应该为0

            dd DESC\_CODE\_HIGH4

DATA\_STACK\_DESC:    dd 0x0000FFFF   ;数据/栈段的段界限应该为0xfffff，段基址应该为0

                    dd DESC\_DATA\_HIGH4

VIDEO\_DESC: dd 0x80000007        ;显存基址是0xB8000（B在高32位），界限是(bFFFF-b8000)/4k = 0x7

            dd DESC\_VIDEO\_HIGH4

;构造16位的段选择子

SELECTOR\_CODE        equ (0X0001<<3) + TI\_GDT + RPL0

SELECTOR\_DATA     equ (0X0002<<3) + TI\_GDT + RPL0

SELECTOR\_VIDEO       equ (0X0003<<3) + TI\_GDT + RPL0

**3.2 SimpleOS-内存管理模块**

内存管理数据结构代码：

//物理内存池

struct pool{

    struct bitmap pool\_bitmap;  //物理内存池位图

    uint32\_t phy\_addr\_start;    //管理空间起始地址

    uint32\_t pool\_size;         //管理空间长度

    struct lock lock;           //进程申请物理内存时需要上锁

};

//虚拟内存池

struct virtual\_addr{

    struct bitmap vaddr\_bitmap; //虚拟内存池位图

    uint32\_t vaddr\_start;       //管理空间起始地址

};

/\* 内存块 \*/

struct mem\_block {

    struct list\_elem free\_elem;

};

/\* 内存块描述符 \*/

struct mem\_block\_desc {

    uint32\_t block\_size;         // 内存块大小

    uint32\_t blocks\_per\_arena;   // 本arena中可容纳此mem\_block的数量.

    struct list free\_list;   // 目前可用的mem\_block链表

};

/\* 内存仓库arena元信息 \*/

struct arena {

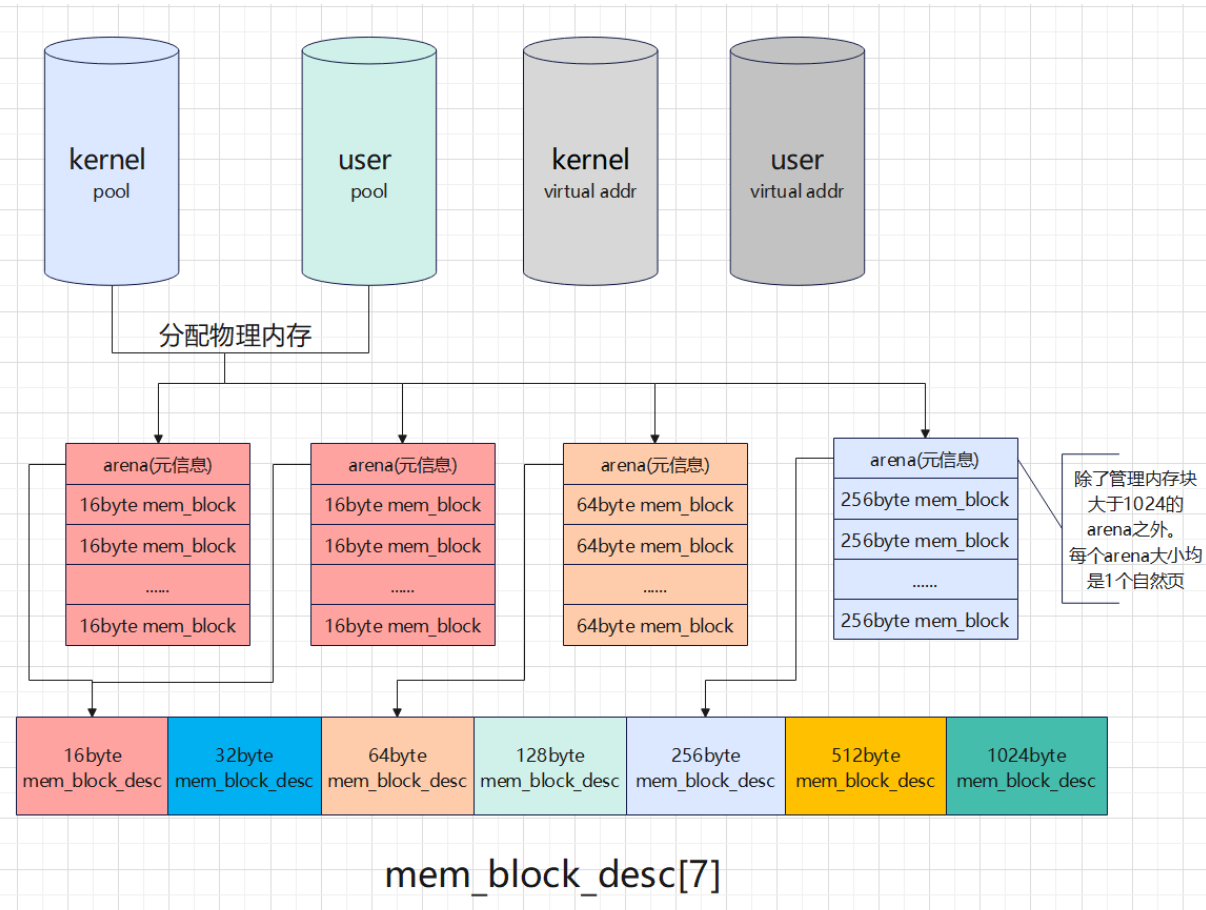
    struct mem\_block\_desc\* desc;     // 此arena关联的mem\_block\_desc

    uint32\_t cnt;

    bool large;                     // large为ture时,cnt表示的是页框数。否则cnt表示空闲mem\_block数量

};

内存管理数据结构可视化：



最上面的4个内存池是内存管理的核心，仅由4个内存池便可以实现以页(4KB)为单位的内存管理和分配。

但是实际上操作系统需要有更灵活的内存分配方案，分配小块空间，所以引用了arena来组织内存仓库，mem\_block\_desc[7]用来描述不同粒度的内存块。  
 对于每一页内存空间都由元信息arena来组织，并将剩余空间切分成小块。每一个元信息arena组织的内存仓库 都按照块的大小有着相应的mem\_block\_desc。由mem\_block\_desc中的free\_list来串起所有同一大小的空闲内存块进行同一管理

**3.3 SimpleOS-文件系统模块**

数据结构代码如下：

/\* 分区结构:一个分区就是一个文件系统 \*/

struct partition {

    uint32\_t start\_lba;         // 起始扇区

    uint32\_t sec\_cnt;           // 扇区数

    struct disk\* my\_disk;       // 分区所属的硬盘

    struct list\_elem part\_tag;  // 用于队列中的标记，用于将分区形成链表进行管理

    char name[8];               // 分区名称

    struct super\_block\* sb;     // 本分区的超级块

    struct bitmap block\_bitmap; // 块位图

    struct bitmap inode\_bitmap; // inode位图

    struct list open\_inodes;    // 本分区打开的inode缓冲队列

};

/\* 超级块：文件系统元信息的配置文件 \*/

struct super\_block {

    uint32\_t magic;                 // 用来标识文件系统类型,支持多文件系统的操作系统通过此标志来识别文件系统类型

    uint32\_t sec\_cnt;               // 本分区总共的扇区数

    uint32\_t inode\_cnt;             // 本分区中inode数量

    uint32\_t part\_lba\_base;         // 本分区的起始lba地址

    uint32\_t block\_bitmap\_lba;      // 块位图本身起始扇区地址

    uint32\_t block\_bitmap\_sects;    // 扇区位图本身占用的扇区数量

    uint32\_t inode\_bitmap\_lba;      // i结点位图起始扇区lba地址

    uint32\_t inode\_bitmap\_sects;    // i结点位图占用的扇区数量

    uint32\_t inode\_table\_lba;       // i结点表起始扇区lba地址

    uint32\_t inode\_table\_sects;     // i结点表占用的扇区数量

    uint32\_t data\_start\_lba;        // 数据区开始的第一个扇区号

    uint32\_t root\_inode\_no;         // 根目录所在的I结点号

    uint32\_t dir\_entry\_size;        // 目录项大小

    uint8\_t  pad[460];              // 加上460字节,凑够512字节1扇区大小

} \_\_attribute\_\_ ((packed));

/\* inode：文件的实质 \*/

struct inode {

    uint32\_t i\_no;                  // inode编号

    uint32\_t i\_size;                //当此inode是普通文件时,i\_size是指普通文件大小

                                    //若此inode是目录,i\_size是指该目录下所有目录项大小之和\*/

    uint32\_t i\_open\_cnts;           // 记录此文件被打开的次数

    bool write\_deny;                // 写文件不能并行,进程写文件前检查此标识

    uint32\_t i\_sectors[13];         // i\_sectors[0-11]是直接块,

                                    // i\_sectors[12]用来存储一级间接块指针

    struct list\_elem inode\_tag;     //用于文件缓冲队列中

};

/\* 文件结构 \*/

struct file

{

    uint32\_t fd\_pos;        // 记录当前文件操作的偏移地址，文件尾为-1

    uint32\_t fd\_flag;       // 文件操作标识符

    struct inode \*fd\_inode;

};

/\* 目录结构 \*/

struct dir {

    struct inode\* inode;

    uint32\_t dir\_pos;       // 记录在目录内的偏移

    uint8\_t dir\_buf[512];   // 目录的数据缓存

};

/\* 目录项结构 \*/

struct dir\_entry {

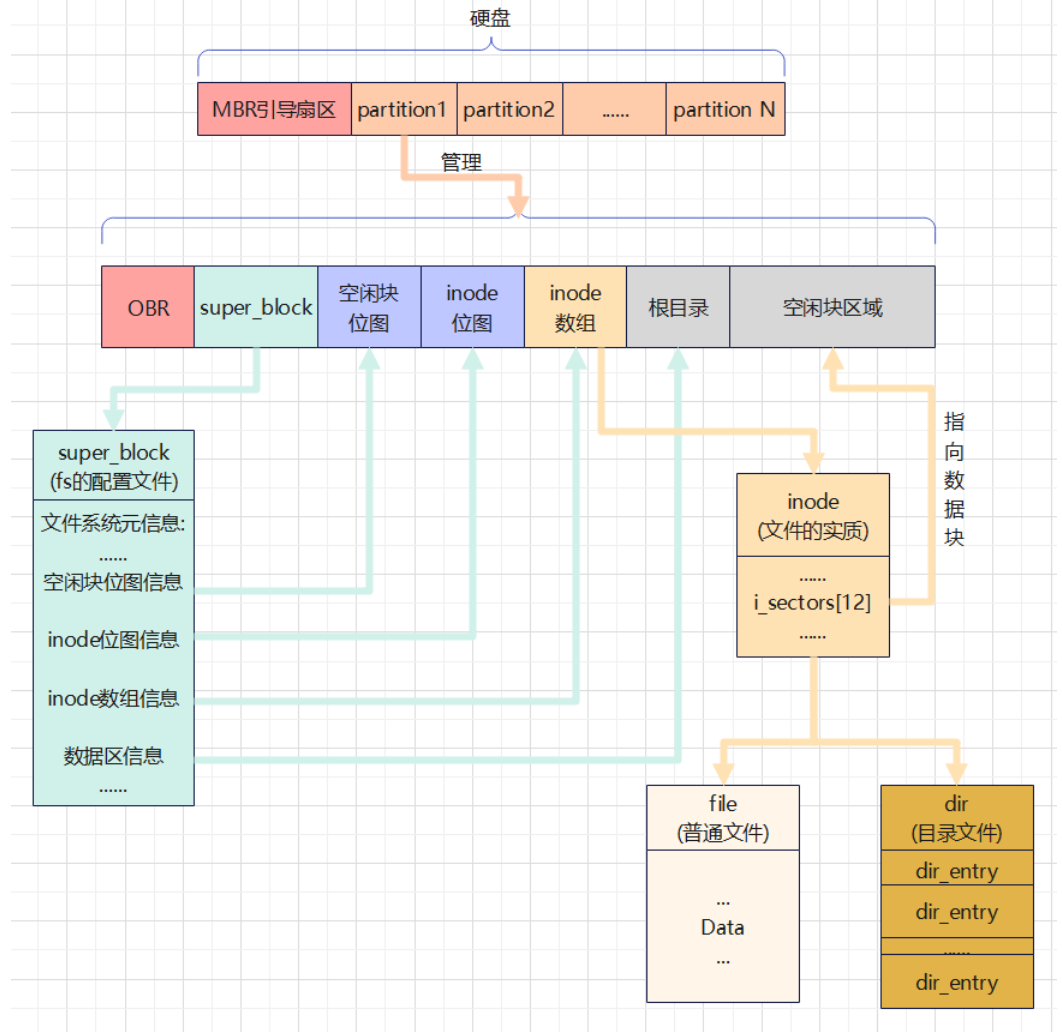
    char filename[MAX\_FILE\_NAME\_LEN];   // 普通文件或目录名称

    uint32\_t i\_no;                      // 普通文件或目录对应的inode编号

    enum file\_types f\_type;             // 文件类型

};

数据结构可视化：

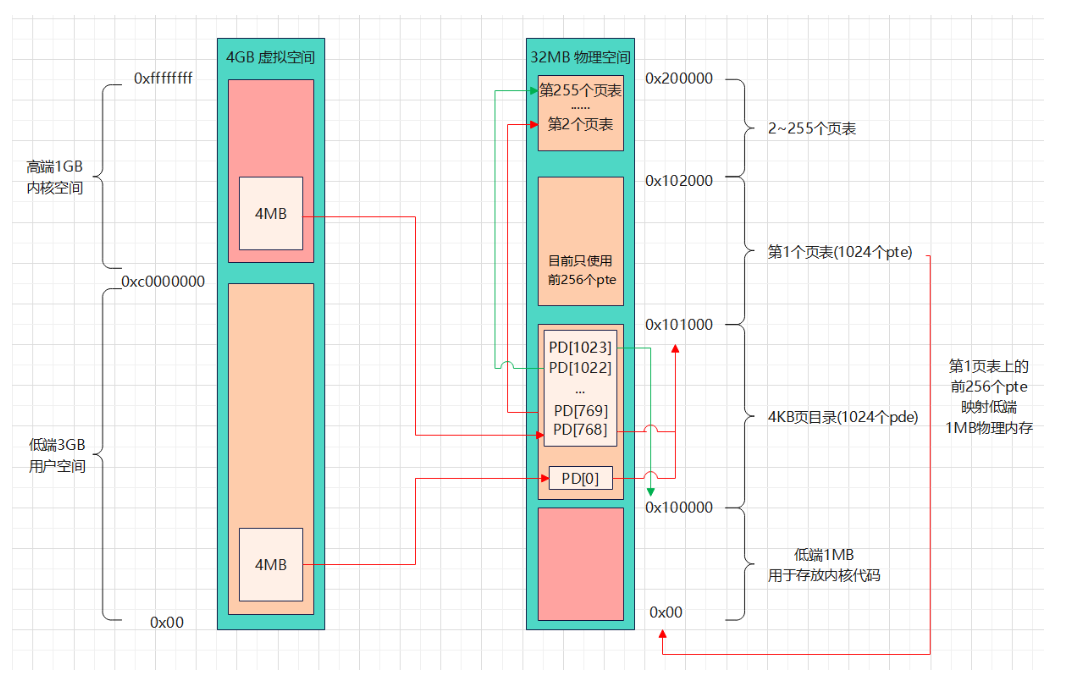


**4关键技术**

**4.1 SimpleOS-Boot模块**

#### 4.1.1内核的二级分页

Boot模块最重要的功能是装载内核，但最困难的部分是在于对内核使用的二级页表的建立。开启分页机制的前提是先为内核建立好一个初始的足够使用的二级页表，二级页表的映射结构图如下：



该二级页表实现了两大功能：

1、通过虚拟地址对内核代码区(低端1MB)的访问：虚拟空间0x00~0x100000和0xc0000000~0xc0100000两个区间都映射到物理空间的低端1MB内核代码区间

2、实现使用虚拟地址访问页表/页目录功能。

我们知道虚拟地址需要经过页表和页目录的访问后才能修改，有一个问题是开启分页后无法访问页表和页目录，原本访问页表和页目录的虚拟地址也会经过映射变成一个新的地址。

所以我们需要使用一个方法，使得可以在开启分页后通过虚拟地址来访问页表和页目录。

方法如下：

PD[1023]指向页目录本身，为的是实现在开启分页机制后还能正确访问页表和页目录

如果虚地址高10位全为1、虚地址中10位全为0，就把PD[0]当成自己的页表项，最终指向物理页地址0x101000  
如果虚地址高10位全为1、虚地址中10位全为1，就把PD[1023]当成自己的页表项,最终指向物理页地址0x100000  
如果虚地址高10位全为1、虚地址中10位处于一定范围内，就把PD[768]~PD[1022]当成自己的页表项目，最终指向物理地址0x101000及以上空间

总结出不变的规律如下：

要获取页目录表物理地址：让虚位高20位地址全为1，低12位全为0，即0xfffff000。这就是页目录自身的起始物理地址

要访问页目录中的页目录项，即获取页表物理地址：使虚拟地址为0xfffffxxx,其中xxx是页目录项的索引\*4

访问页表中的页表项:虚拟地址公式为 0x3ff<<22+中间10位<<12+低12位（中间10位是页表的索引，低12位为页表内的偏移地址）

#### 4.1.2平坦模型

操作系统中的GDT（全局描述符表）是一种用于存储内存段的访问权限、基址、大小等信息的数据结构。GDT可以选择采用的平坦模型或多段模型。

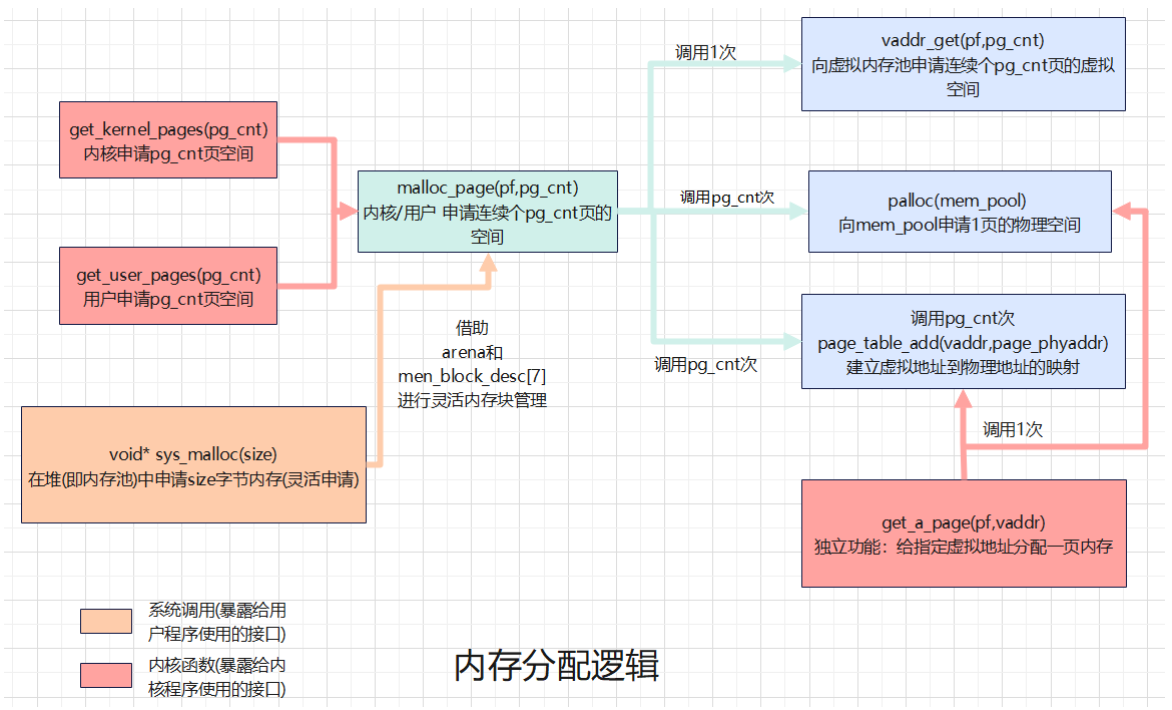
在较为传统的多段模型中，内存被划分为多个不同的段，每个段有自己的基址和限制，需要通过段寄存器和偏移量来寻址。这样会增加内存管理的复杂度和开销，也会限制内存的可用空间。

我们采用的平坦模型则是将整个内存看作一个单一的段，只使用一个GDT项来描述它。这样就可以直接使用32位的线性地址来访问内存，不需要进行段的切换和转换。、

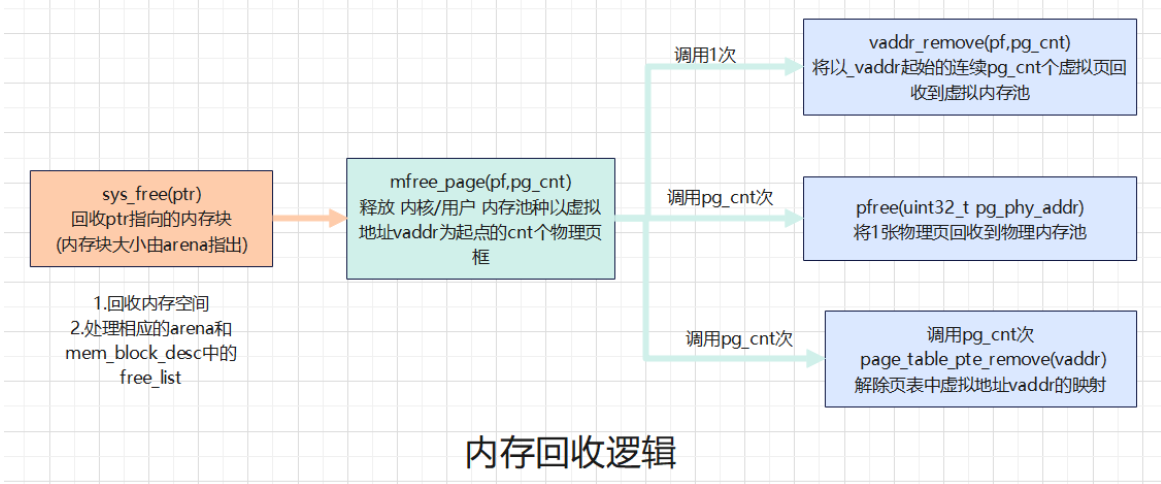
平坦模型的优势在于简化了内存管理，提高了内存的利用率和访问效率，也方便了保护模式和分页机制的实现。

**4.2 SimpleOS-内存管理模块**

#### 4.2.1内存分配/回收逻辑



内存分配中橙色模块的sys\_malloc是暴露给应用程序使用的系统调用接口，红色模块则是一般在内核函数中自己使用的接口。二者接口的本质都是调用malloc\_page()来实现内存分配。malloc\_page()则会先根据申请内存的线程的PCB来判断究竟是该去内核内存池还是用户内存池去申请空间，判断完毕后则调用vaddr\_get()向虚拟内存池申请虚拟空间，调用palloc()向物理内存池申请物理空间，最后再调用page\_table\_add()在虚拟空间和物理空间中一一建立映射（即二级页表），最后再将虚拟地址层层返回，实现了内存分配。

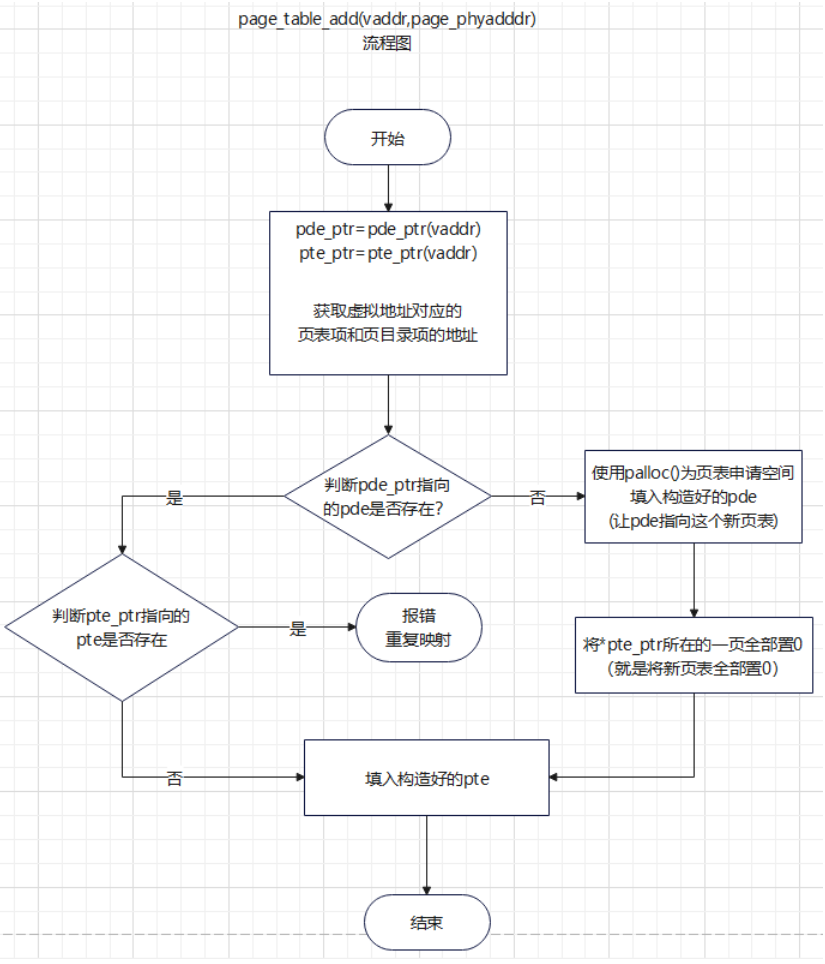


内存回收的逻辑可以说是内存分配的逆向行为，内存回收的核心函数是mfree\_page(),它先调用vaddr\_remove()回收虚拟地址，再调用pfree()去回收物理地址，最后调用page\_table\_pte()一一解除物理地址和虚拟地址之间的映射（解除映射的本质是将对应页表的存在位置0）。

#### 4.2.2映射建立函数

内存分配的一大核心在于使用二级页表机制进行映射的建立和删除

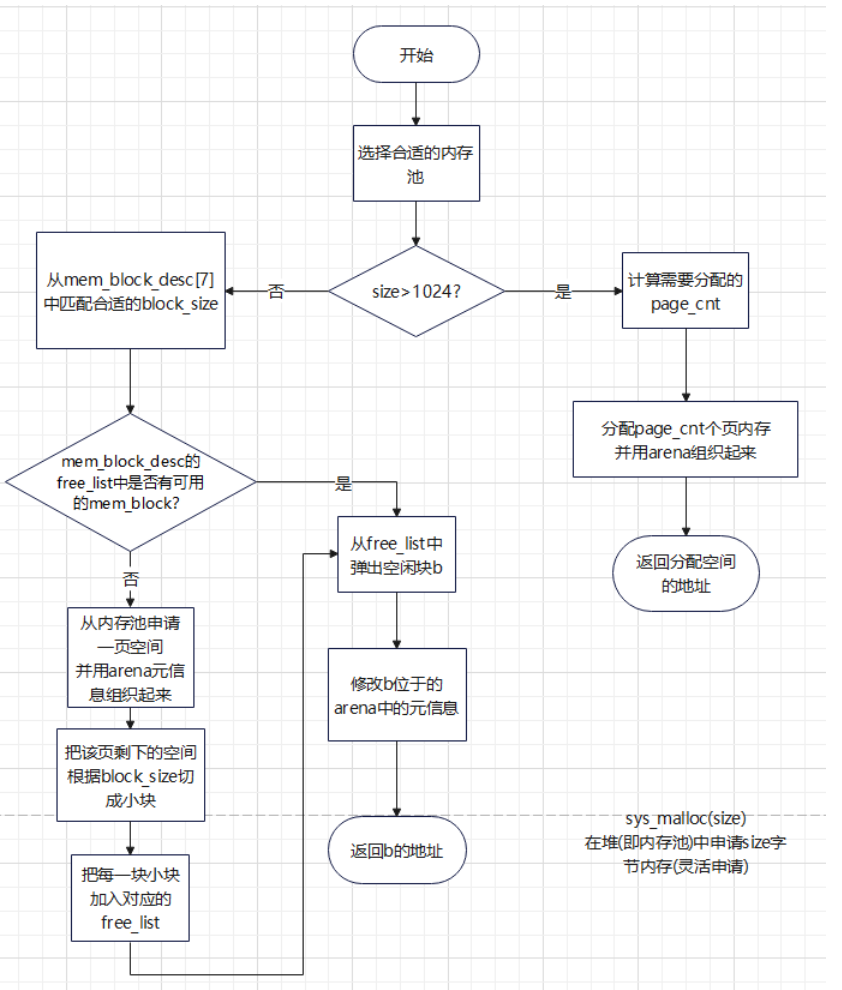
Page\_table\_add()是映射建立函数，流程图如下：



#### 4.2.3内存分配函数

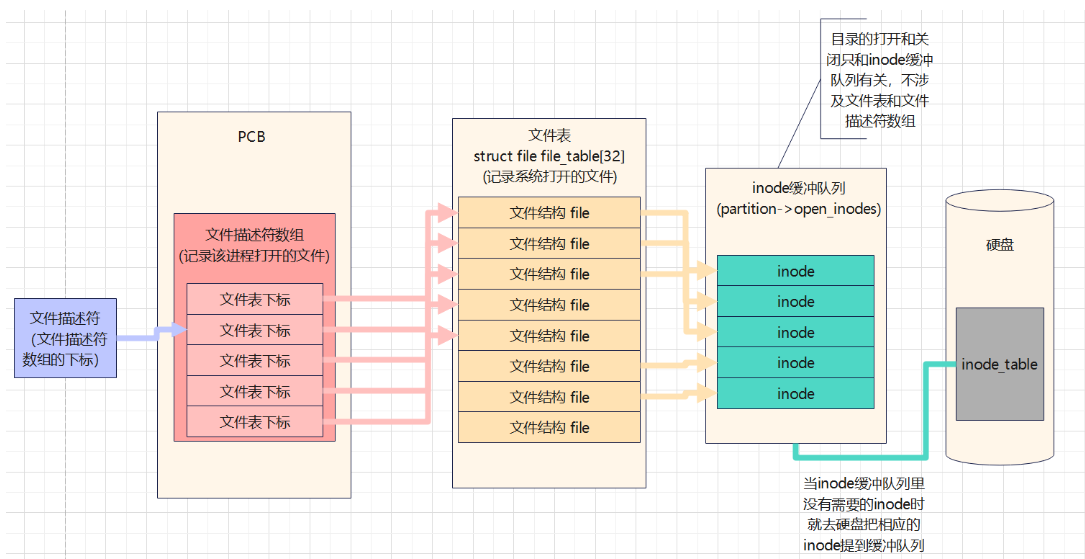
Sys\_malloc()是实现了细粒度内存分配的系统调用

流程图如下：



**4.3 SimpleOS-文件系统模块**

#### 4.3.1文件和目录的管理体系



引入文件表、文件描述符数组、文件描述符、inode缓冲队列的概念来共同管理目录文件和普通文件。

文件描述符数组表示进程中以打开的普通文件

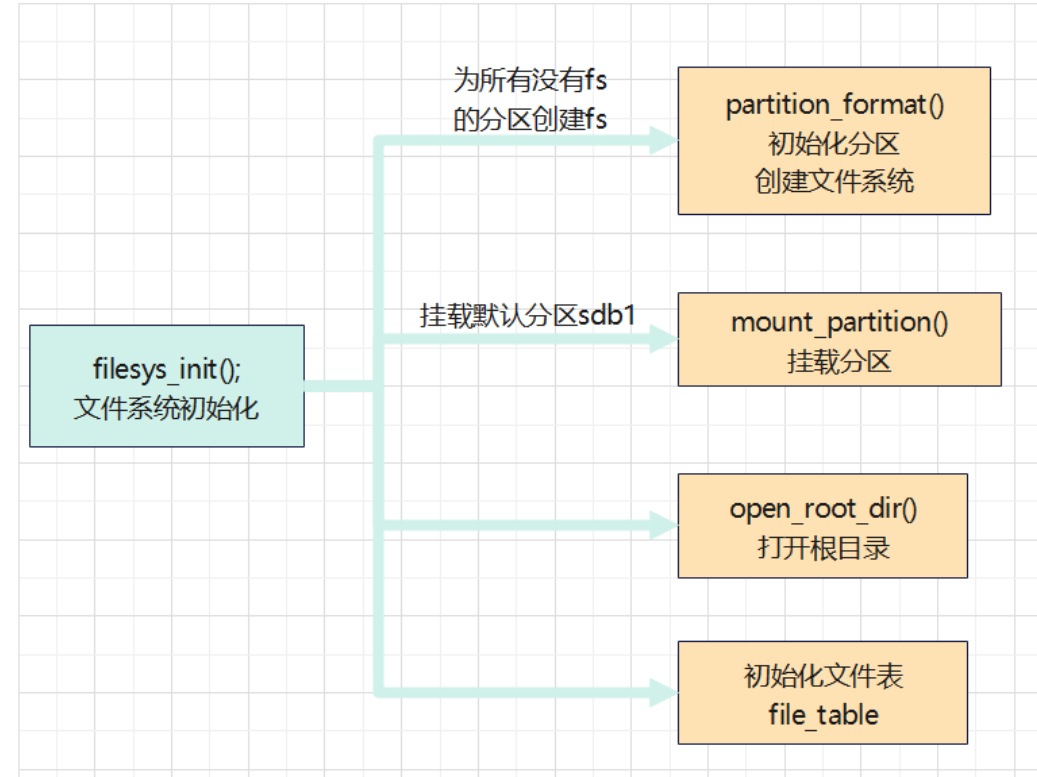
文件表表示系统中以打开的普通文件

inode缓冲队列代表已打开的inode(包括目录文件和普通文件)

注意：已打开的目录文件只加入inode缓冲队列，并不涉及文件表和文件描述符数组。已打开的普通文件则除了放入inode缓冲队列外还要添加到相应的文件表和文件描述符数组里

#### 4.3.2文件系统初始化逻辑

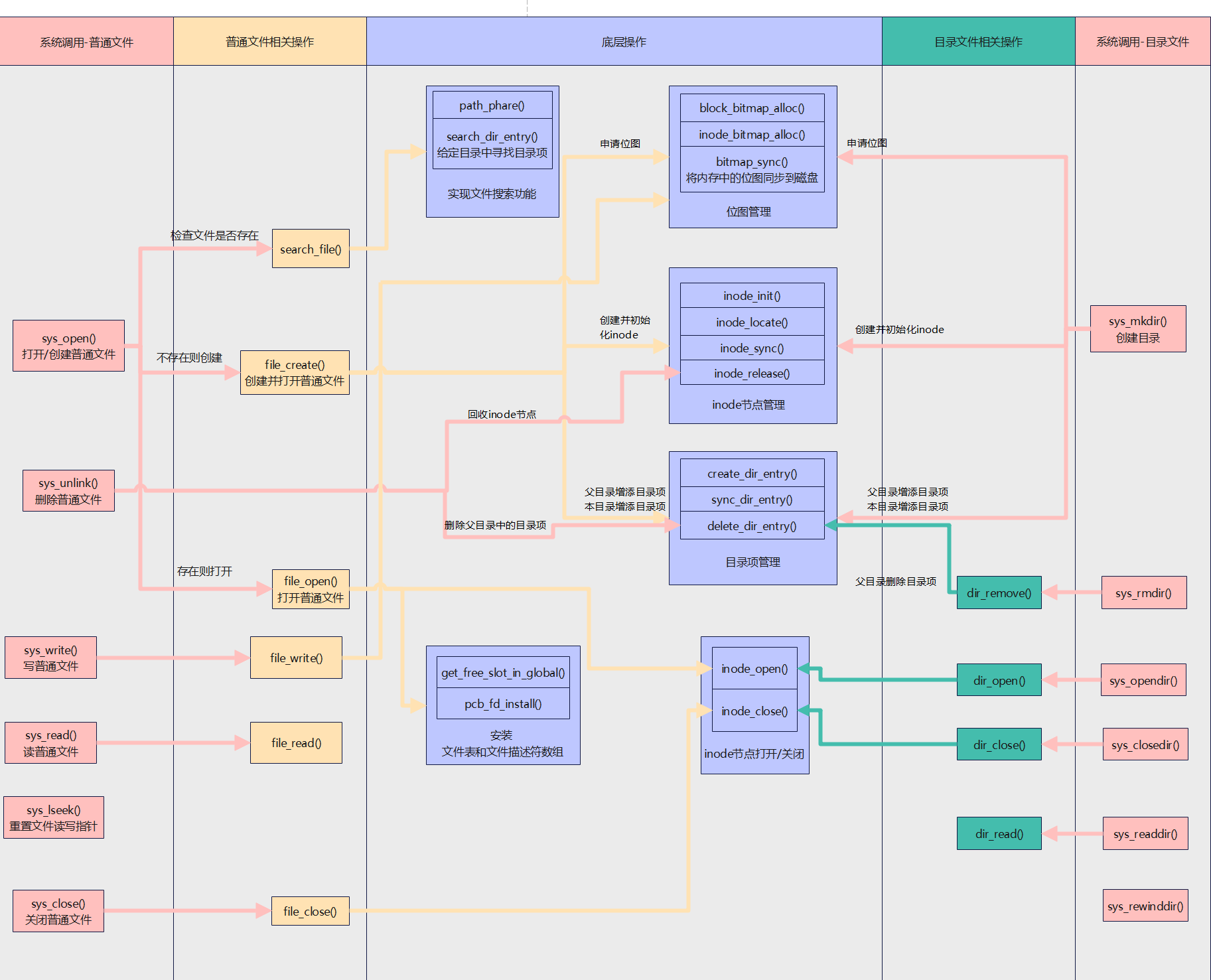
文件系统初始化函数逻辑如下：



#### 4.3.3文件和目录操作逻辑

文件系统的函数逻辑结构非常复杂，大致上可分为系统调用函数、普通文件操作相关函数、目录文件操作相关函数、工作路径操作相关函数以及底层操作函数。

其中普通文件和目录文件操作相关函数息息相关，也是整个文件系统的核心部分，逻辑结构如下图所示:

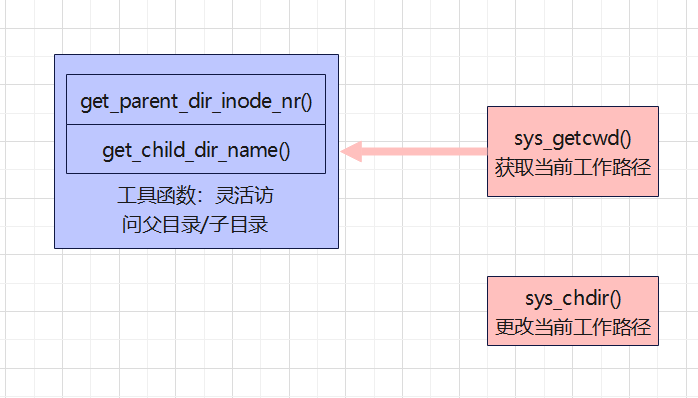


图中红色代表系统调用，黄色是普通文件相关操作、绿色是目录文件相关操作。蓝色是底层相关操作。而底层相关操作又是建立在文件驱动函数上实现的(文件驱动属于驱动模块，不在该模块讲述)。

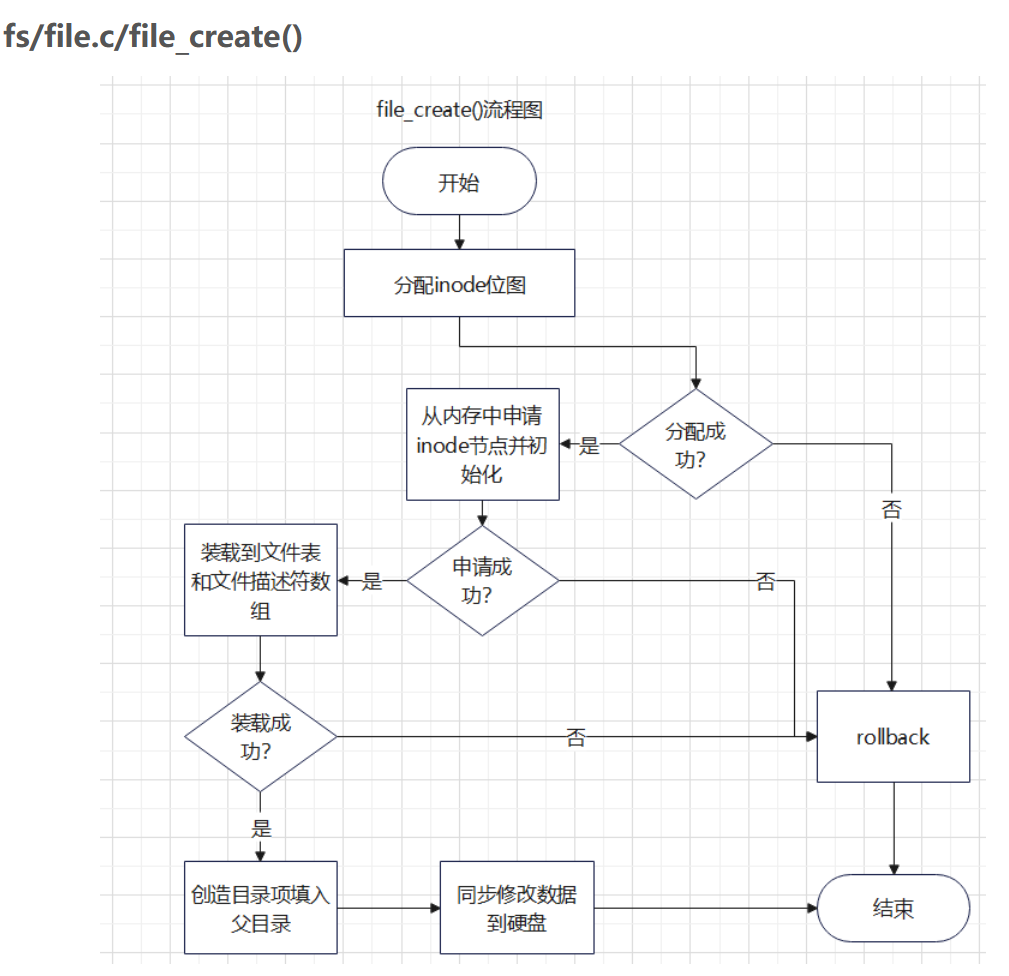
如图可见，我们将文件分为普通文件和目录文件，普通文件和目录文件的操作并不相同，但在底层上却会调用相同的模块进行管理。

整张图的最左放置关于文件的系统调用，右边则是关于目录的基本调用，二者通过抽象和层层调用像中间辐射，中间的蓝色模块则代表较为底层的操作，涉及到inode管理、位图管理、目录项管理、文件描述符管理等模块

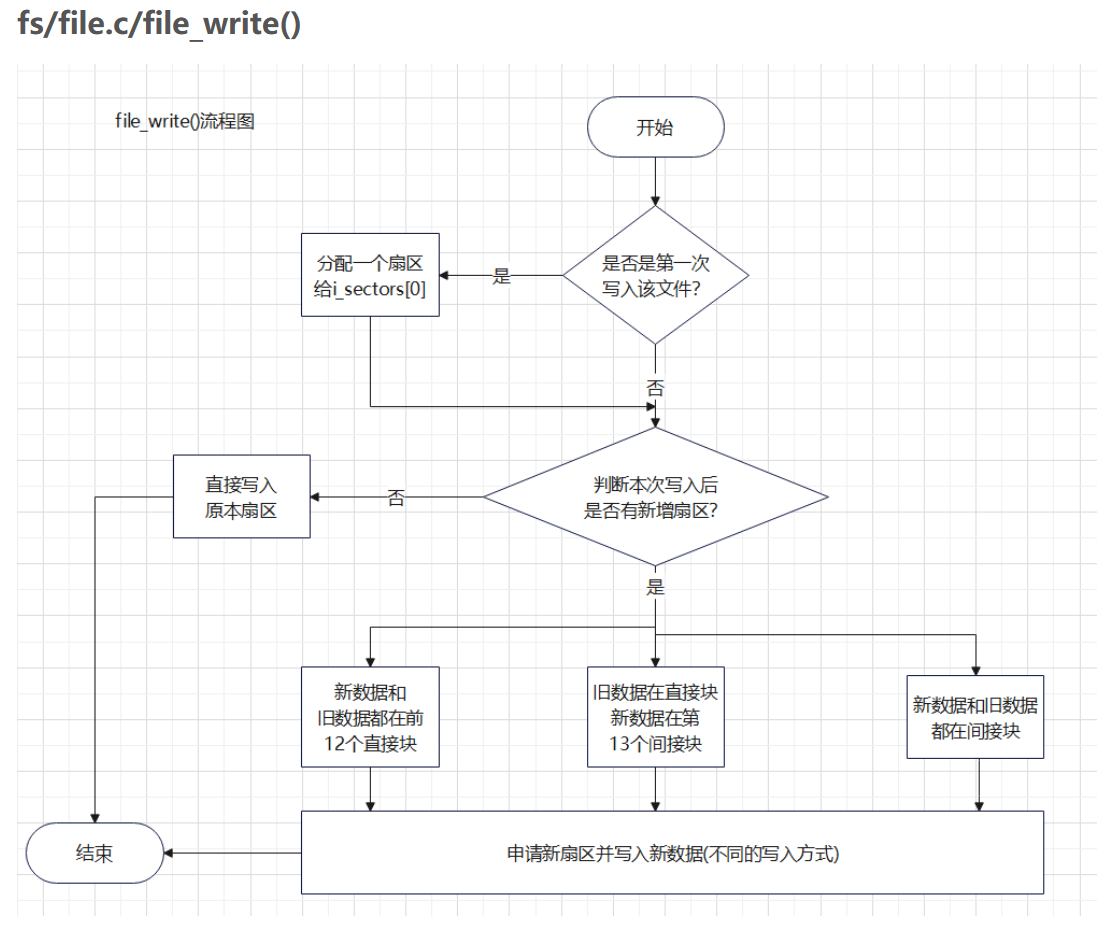
工作路径操作相关函数则较为简单且独立于上述操作，函数调用逻辑如下：



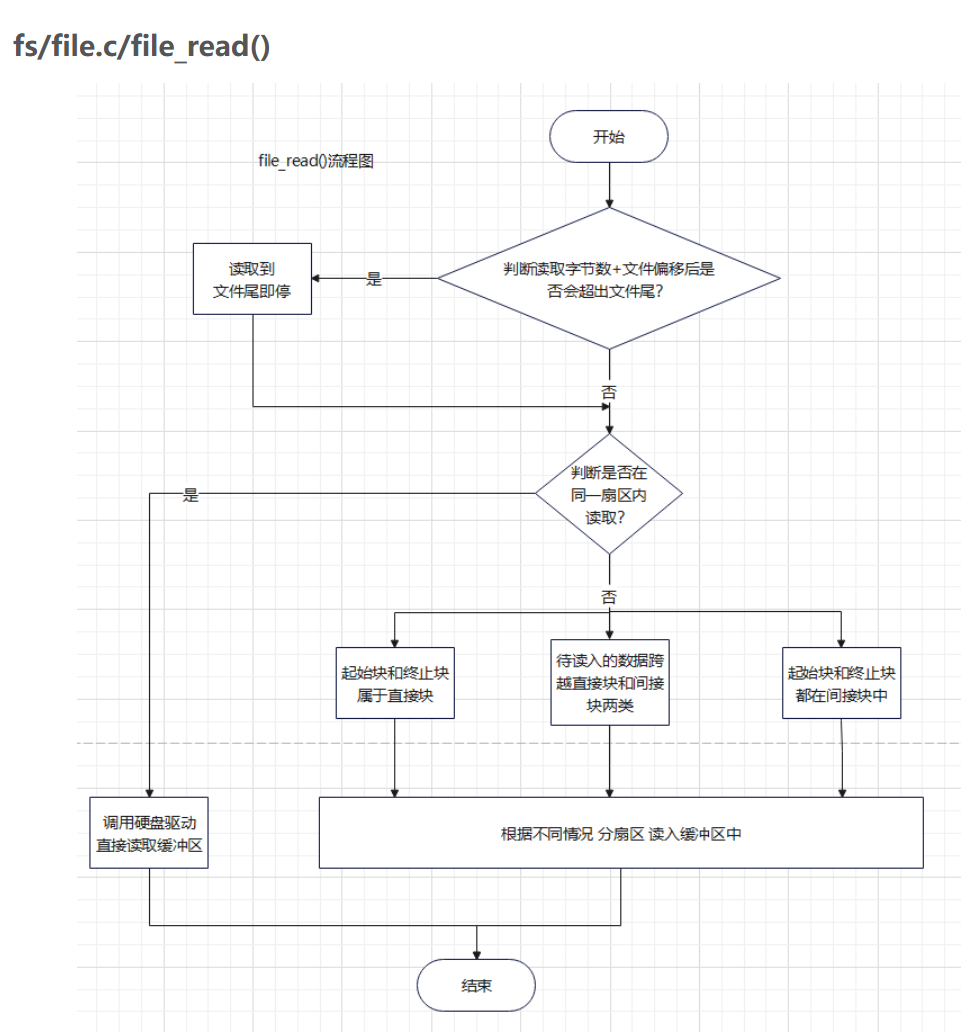
#### 4.3.4普通文件创建



#### 4.3.5普通文件写操作



#### 4.3.6普通文件读操作



**5运行结果**

**5.1 运行环境**

C编译器使用gcc4

PC模拟软件使用Bochs2.6.8

Bochs2.6.8需预先建立两个虚拟磁盘，bochsrc.disk配置文件如下：

megs: 32

romimage: file=/home/sparkle2/bochs/share/bochs/BIOS-bochs-latest

vgaromimage: file=/home/sparkle2/bochs/share/bochs/VGABIOS-lgpl-latest

boot: disk

log: bochs.out

mouse:enabled=0

keyboard:keymap=/home/sparkle2/bochs/share/bochs/keymaps/x11-pc-us.map

ata0:enabled=1,ioaddr1=0x1f0,ioaddr2=0x3f0,irq=14

#新加入的代码

ata0-master: type=disk, path="hd60M.img", mode=flat,cylinders=121,heads=16,spt=63

ata0-slave: type=disk, path="hd80M.img", mode=flat,cylinders=162,heads=16,spt=63

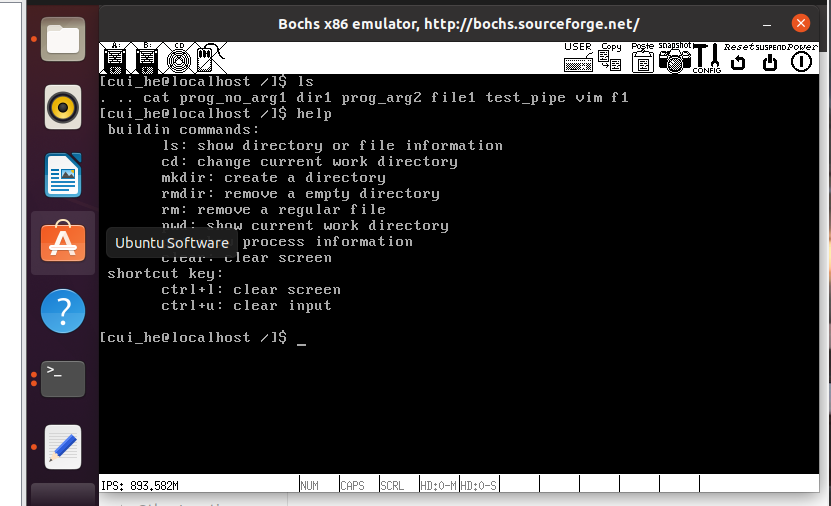
#gdbstub:enabled=1,port=1234,text\_base=0,data\_base=0,bss\_base=0

源代码需经过gcc4编译过后才能正确生成.bin文件（过高版本gcc将会因为elf文件格式解析不合而产生错误）。源代码编译后生成的.bin文件将加载到Bochs2.6.8中，操作系统将会被装载在Bochs2.6.8中。

**5.2运行结果和分析**

运行结果分析部分并非是单独的Boot、内存管理和文件系统系统模块。单独的三个模块无法运行。因此运行结果分析是完备的整个操作系统代码的运行结果

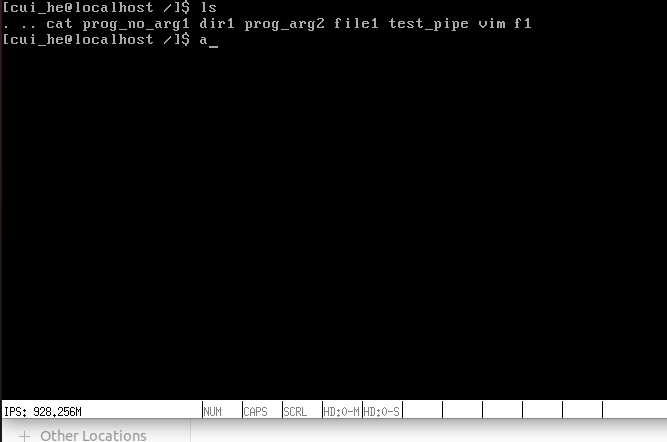
**5.2.1 SimpleOS主界面**



5. 系统主界面

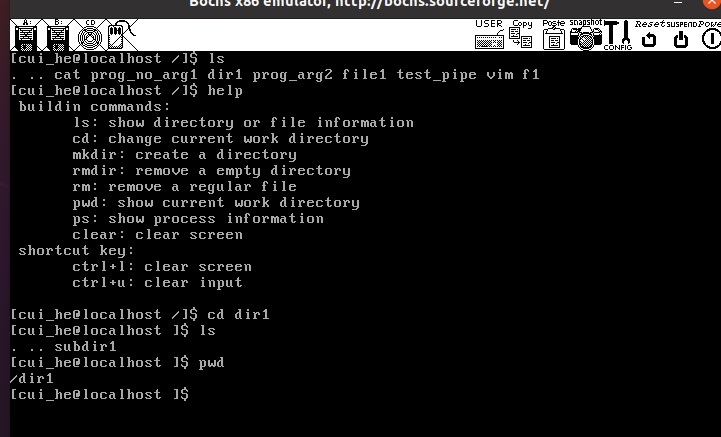
**5.2.2 SimpleOS支持的基础命令**

本项目支持的基础命令操作仿照Linxu的部分命令



5. ls

ls命令：展示当前文件夹下的目录和文件



5. cd、pwd

cd命令：进入选择的文件夹

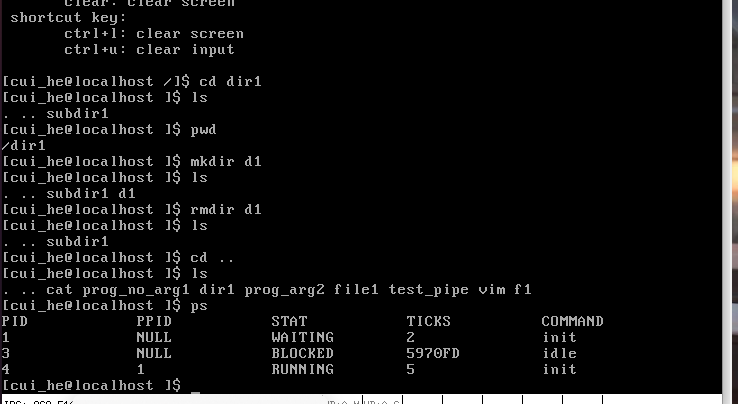
pwd命令：显示当前的工作目录路径



5. mkdir、rmdir

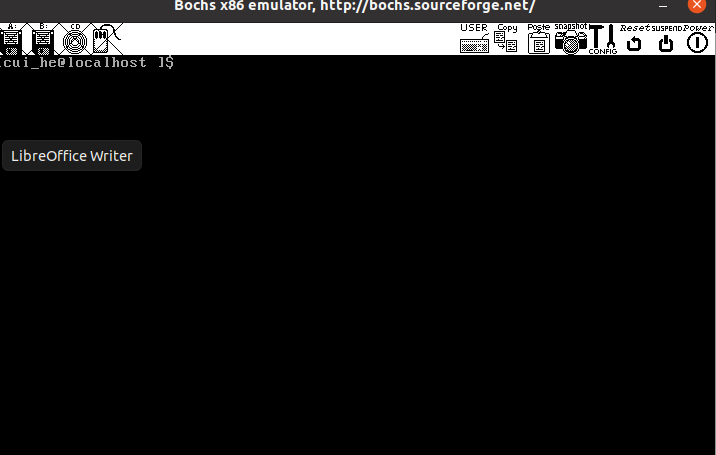
mkdir：创建目录

rmdir：删除目录



5. ps

ps：查看进程信息



5. clear/ctrl+l清屏/ctrl+u单行清除

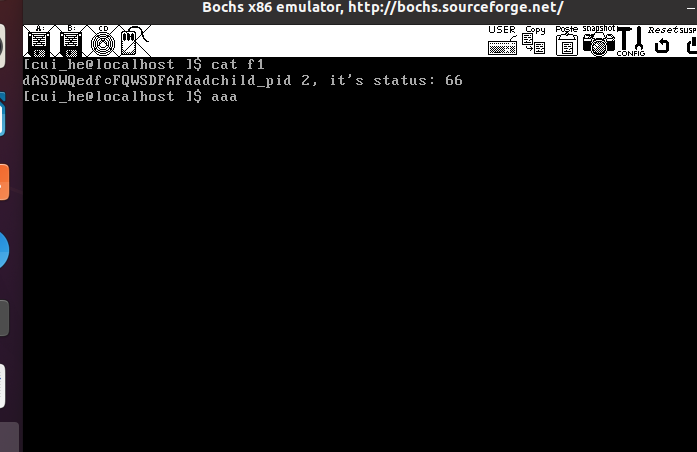
/ctrl+l支持清屏功能

/ctrl+u则是单行清除功能

**5.2.3 SimpleOS支持的外部命令**

本操作系统可以装载C程序代码并运行，我们一共准备4个示例C程序来体现系统的功能。4个示例C程序相当于是用户进程，并且被我们装载到文件系统中，我们可使用外部命令来运行4个用户进程。

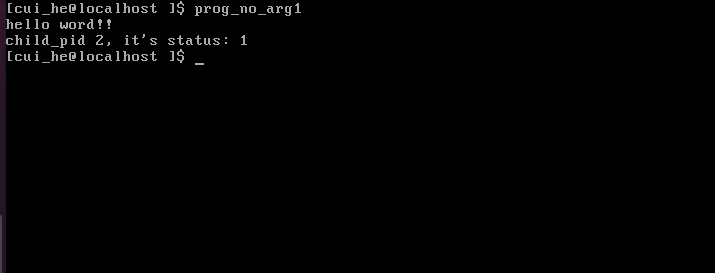
#### 5.2.3.1 Cat程序



5. cat

Cat命令查看文件内容，返回进程号以及cat子进程状态

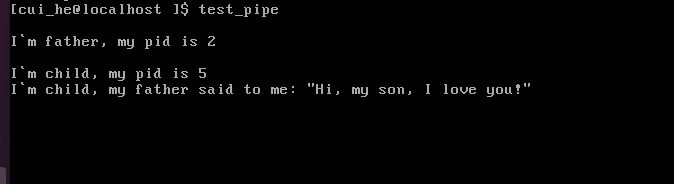
#### 5.2.3.2 示例程序prog\_no\_arg1



5. 执行c程序输出字符串

C可执行程序输出helloworld!

#### 5.2.3.3 示例程序test\_pipe



5. 进程通信

源代码如下：

进程通信示例代码：

#include "stdio.h"

#include "syscall.h"

#include "string.h"

#include "stdint.h"

int main(int argc, char \*\*argv)

{

    int32\_t fd[2] = {-1};

    pipe(fd);

    int32\_t pid = fork();

    if (pid)

    {                 // 父进程

        close(fd[0]); // 关闭输入

        write(fd[1], "Hi, my son, I love you!", 24);

        printf("\nI`m father, my pid is %d\n", getpid());

        exit(8);

    }

    else

    {

        close(fd[1]); // 关闭输出

        char buf[32] = {0};

        read(fd[0], buf, 24);

        printf("\nI`m child, my pid is %d\n", getpid());

        printf("I`m child, my father said to me: \"%s\"\n", buf);

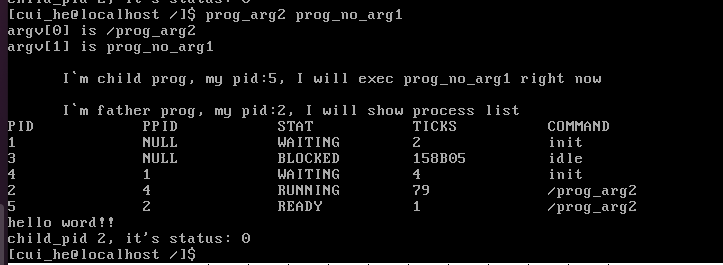
        exit(9);

    }

}

实现了进程之间的通信

#### 5.2.3.4 示例程序prog\_arg2



5.

源代码如下：

测试程序执行其他程序示例代码：

#include "stdio.h"

#include "syscall.h"

#include "string.h"

int main(int argc, char \*\*argv)

{

    int arg\_idx = 0;

    while (arg\_idx < argc)

    {

        printf("argv[%d] is %s\n", arg\_idx, argv[arg\_idx]);

        arg\_idx++;

    }

    int pid = fork();

    if (pid)

    {

        int delay = 900000;

        while (delay--);

        printf("\n      I`m father prog, my pid:%d, I will show process list\n", getpid());

        ps();

        if(argc!=1){

        int32\_t status;

        int32\_t child\_id=wait(&status);

        }

    }

    else

    {

        char abs\_path[512] = {0};

        printf("\n      I`m child prog, my pid:%d, I will exec %s right now\n", getpid(), argv[1]);

        if (argv[1][0] != '/')

        {

            getcwd(abs\_path, 512);

            strcat(abs\_path, "/");

            strcat(abs\_path, argv[1]);

            execv(abs\_path, argv);

        }

        else

        {

            execv(argv[1], argv);

        }

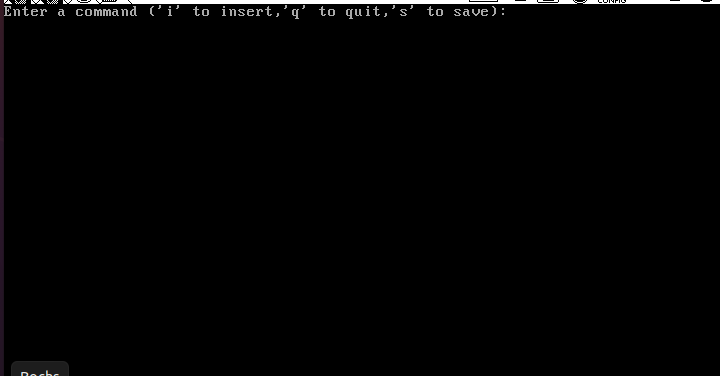
    }

    return 0;

}

进程执行子程序输出helloword

#### 5.2.3.5 示例程序 简易vim编辑器



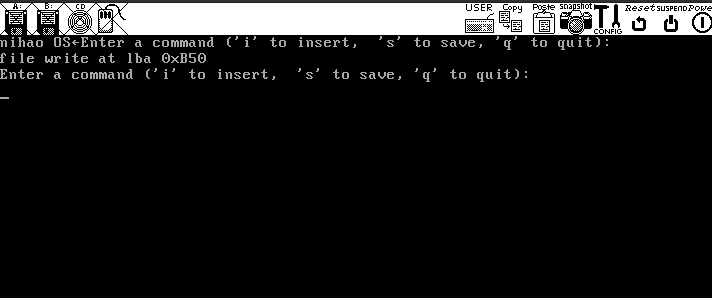
5. 11

简易VIM主界面



5.

输入I进入编辑模式 写入nihao OS



5.

esc键退出编辑模式，输入S保存写入文件系统，输入q退出vim



5.

写入成功

程序源代码如下：

#include "syscall.h"

#include "stdio.h"

#include "string.h"

#include "fs.h"

#include"file.h"

int main(int argc, char \*\*argv)

{

    if (argc > 2||argc==1)

    {

        printf("cat: argument error\n");

        exit(-2);

    }

    int buf\_size = 1024;

    char abs\_path[512] = {0};

    void\*buf=malloc(buf\_size);

    char op[3]={0};

    if (buf == NULL||op==NULL)

    {

        printf("cat: malloc memory failed\n");

        return -1;

    }

    if (argv[1][0] != '/')

    {

        getcwd(abs\_path, 512);

        strcat(abs\_path, "/");

        strcat(abs\_path, argv[1]);

    }

    else

    {

        strcpy(abs\_path, argv[1]);

    }

    int fd = open(abs\_path,O\_CREAT|O\_RDWR);

    if (fd == -1)

    {

        printf("cat: open: open %s failed\n", argv[1]);

        return -1;

    }

    clear();

    int read\_bytes = 0;

    printf("Enter a command ('i' to insert,'q' to quit,'s' to save): \n");

    read(stdin\_no, op, 1);

    clear();

   // char\* str="a";

    while(1){

        switch (op[0]) {

            case 'i':

                while(1){

                    read(stdin\_no, op, 1);

                    op[1]='\0';

                    write(stdout\_no,op,1);

                    if(op[0]=='\033'){

                        break;

                    }

                    strcat(buf, op);

                    read\_bytes++;

                }

                break;

            case 's':

                write(fd, buf, read\_bytes);

                break;

            case 'q':

                free(buf);

                close(fd);

                //free(op);

                exit(-1);

            default:

                printf("Invalid command\n");

        }

        printf("Enter a command ('i' to insert,  's' to save, 'q' to quit): \n");

        read(stdin\_no, op, 1);

    }

    return 8;

}

**6调试和改进**

**6.1 远跳转指令清空流水线**

我遇到了一个关于指令译码的问题。问题的原因是CPU的流水线技术，它使得指令在CPU中并非是顺序执行的，而是利用了流水线技术追求效率使得指令乱序执行。在loader中，我们需要从实模式切换到保护模式，这涉及到指令长度的变化，从16位指令变成了32位指令。由于流水线技术的影响，一些提前被加载进流水线的32位指令被错误地译码成了16位指令，导致程序出现了无法理解的错误。

为了解决这个问题，我首先通过调试工具查看了指令的译码结果，发现了译码错误的现象。然后，我通过查阅相关资料，了解了CPU的流水线技术的原理和工作机制，找到了问题的根源。最后，我采用了一个简单而有效的解决方案，就是在loader开启保护模式后，使用无条件跳转指令jmp来进行一次无意义的跳转，目的是让跳转过后CPU自动清空流水线，从而避免了译码错误的发生。通过这样的方法，我成功地解决了这个问题，使得程序能够正常地运行。

**6.2 屏幕输出的竞态问题**

当只有一个线程的情况时，往显存段0xb8000~0xB7FFF写入的字符可以正常显示在屏幕中。但在引入多线程的过程中，我遇到了一个严重的问题，就是多个线程同时在屏幕上输出字符时，出现了乱码的现象。我分析了一下原因，发现是因为两个线程同时往显存区域0xb8000~0xB7FFF里写入字符，导致了竞态条件的产生。竞态条件是指多个线程在访问和操作同一个共享资源时，最终的结果取决于线程的执行顺序，而这个顺序是不可预测的。这样就会造成显存区域里的字符乱序，甚至超出了显存段的范围，引发了段错误。

为了解决这个问题，我采用了互斥锁的方法，给显存段的输入输出加上了锁，并引入了以下函数。

void console\_put\_char(uint8\_t char\_asci);

void console\_put\_str(char\* str);

void console\_put\_int(uint32\_t num);

以下函数利用锁保证了同一时刻只有一个线程可以访问显存段。之后在所有和屏幕输出相关的线程中，都使用了这些函数来输出字符和整数，而不是直接往显存段里写入。这样，就避免了多个线程之间的竞态条件，实现了正确的屏幕输出。

**6.3 read()读取操作字符中的错误**

在操作系统课程设计中，我需要编写一些示例用户进程来展示我的操作系统的功能。其中一个示例用户进程是一个简易的VIM程序，它可以让用户在终端中编辑和保存文本文件。这个程序的主要功能是通过read()和write()系统调用来实现输入和输出，以及通过open()和close()系统调用来打开和关闭文件。程序的核心部分是一个while循环，它根据用户输入的命令来执行相应的操作，例如'i'表示插入模式，'s'表示保存文件，'q'表示退出程序等。

在编写和测试这个程序的过程中，我遇到了一个问题，就是当我输入'i'进入插入模式后，我输入的字符会出现一些奇怪的符号，而不是我想要的内容。我仔细检查了我的代码，发现问题出在这句代码上：

read(stdin\_no, op, 1);

这两句代码的作用是从标准输入文件中读取一个字符到op数组中，但是从键盘缓冲区读取到op的字符串数组中没有’\0’字符，而是一些随机的值。这就使得op数组不是一个合法的字符串，而是一个未定义的内存区域。当我用write()系统调用将op数组输出到标准输出文件时，它不会停止在op数组的第一个元素，而是会继续输出后面的内存内容，直到遇到一个’\0’字符为止。这就导致了我看到的一些奇怪的符号，其实是op数组后面的内存内容。

为了解决这个问题，我在read()系统调用后面添加了一句代码，即op[1]=‘\0’，来强制将op数组的第二个元素设为’\0’字符，从而使得op数组成为一个合法的字符串，只包含我输入的一个字符。这样，当我用write()系统调用将op数组输出到标准输出文件时，它就会正确地停止在op数组的第一个元素，而不会输出后面的内存内容。这样，我就可以正常地输入和显示我想要的字符了。

**7心得与结论**

操作系统是计算机科学的一个重要分支，它涉及到许多复杂的算法、策略和机制，如进程管理、内存管理、文件系统、设备驱动等。在操作系统的理论课上，我们学习了这些内容的基本原理和实现方法，但是这些理论知识并不能让我们完全掌握操作系统的精髓。只有通过亲自动手编写一个操作系统，我们才能真正体会到操作系统的设计思想和运行原理，才能把理论知识和实践技能有效地结合起来。

在本次课程设计中，我完成了一个仿linux风格的简易操作系统，它具有基本的内核、用户态程序、系统调用、中断处理、进程调度、内存分配、文件系统等功能。在这个过程中，我不仅运用了理论课上学习的最基础最简单的算法，如RR进程调度、信号量实现锁功能等，而且还学习了大量的底层硬件知识，如CPU的寄存器、指令、模式、中断、异常等，以及与之相关的汇编语言、链接器、加载器等工具。这些知识让我能够更深入地理解操作系统的运作过程，也让我能够更灵活地设计和优化我的操作系统。

以全局描述符表（GDT）和选择子为例，理论课上只介绍了GDT的概念和作用，即用来存储段描述符，从而实现段式存储管理。但是在实际编程中，我还需要正确了解选择子的格式、段描述符的具体结构、LGDT指令的用法等细节，才能正确地设置GDT，并在保护模式下访问内存。这些细节虽然看似琐碎，但却是操作系统的基石，没有它们，就无法构建一个完整的操作系统。

除了收获了许多操作系统相关的知识，我还对C语言有了更深的理解。C语言是一种高效而灵活的编程语言，它可以直接操作硬件，也可以实现复杂的逻辑。C语言的核心概念之一就是指针，它可以让我们访问和修改内存中的任意位置。在编写操作系统的过程中，我充分利用了指针的特性，利用各种指针类型转换和访问实现了各种功能，如内核加载、进程切换、系统调用等。

如果说Python里一切皆对象的话，那么我认为C中一切皆是地址。无论是变量、函数、结构体，还是其他类型的数据。这些标号就是内存中的地址，它们可以用不同的方式来解释和使用。不同的类型只不过代表着不同的偏移量的单位罢了，利用基址+偏移量这一简单组合，就可以自由访问或转换C中的一切标号。例如，一个int类型的变量，它的地址可以用一个char类型的指针来访问，从而得到它的每个字节的值；一个函数的地址，它可以用一个函数指针来调用，从而执行它的代码；一个结构体的地址，它可以用一个结构体指针来访问，从而得到它的每个成员的值。这些地址的转换，虽然看似违背了类型的规则，但却是C语言的强大之处，它让我们可以自由地控制内存，实现各种功能。当然，这种自由也带来了危险，如果我们不小心，就可能造成内存泄漏、越界访问、非法操作等错误，导致程序崩溃或者系统崩溃。

本次课程设计中，我还接触了大量关于x86系列的指令、硬件、格式等底层的内容，但在这个过程中我见到了大量让人难以理解的设计，而这些难以理解的设计皆是由于“兼容“。计算机的发展是一个不断迭代和创新的过程，但同时也是一个不断兼容和适应的过程。在这个过程中，兼容性是一个非常重要的因素，它决定了计算机的可用性和普及性。

以x86系列的段描述符为例，它是用来实现段式存储管理的一种数据结构，它包含了段的基址、限长、属性等信息。在32位系统中，每个段的基址是32位的，但是在段描述符中，段基址被分成了三个部分，分别存放在不同的位置。这样做的原因是为了兼容之前的系统，如实模式下的16位段基址，以及80286时代的24位段基址。为了保持与这些系统的一致性，段基址被拆分成了16-8-8的形式，而不是连续的32位。这样做的好处是可以让不同的系统之间互相访问和转换，提高了计算机的兼容性和灵活性。

兼容性是计算机发展的一个必要条件，它让计算机可以利用前人的成果，也让计算机可以适应不同的环境和需求。没有兼容性，计算机就无法实现跨平台、跨时代、跨领域的应用，也无法满足市场的多样化和个性化的需求。可以说，兼容性是计算机产品的一种生命力，它让计算机可以不断地进化和创新。但是，兼容性也有它的代价，它会让计算机的设计变得复杂和冗余，也会让计算机的性能和效率受到影响。为了兼容旧的系统，计算机必须保留一些不必要或者过时的功能和结构，这些功能和结构会占用计算机的资源和空间，也会增加计算机的错误和风险。为了兼容不同的系统，计算机必须适应一些不统一或者不合理的规则和标准，这些规则和标准会限制计算机的发挥和优化，也会降低计算机的可读性和可维护性。因此，兼容性是一把双刃剑，它既有利于计算机的发展，也有阻碍计算机的发展。

总而言之，我认为本次课程设计是一次非常有意义和有趣的学习经历，它不仅提高了我的编程能力，也拓展了我的视野，让我对操作系统有了更深的认识和兴趣。我希望在今后的学习中，我能够继续探索操作系统的奥秘，也能够把我学到的知识应用到实际的项目中，为计算机科学的发展做出贡献。

**主要参考文献**

[1] 操作系统真象还原[M]. 北京：人民邮电出版社，2016.

[2]涂凌鸣,廖雁.简易操作系统的设计与实现[J].电脑知识与技术,2011,7(30):7522-7525.

[3]曾建周,李尧.一个简易操作系统的设计与实现[J].内江师范学院学报,2006(S1):238-242.