实　　验　　报　　告

学　　号	21030031009	姓　　名	惠欣宇	专业班级	计算机 4 班
课程名称	**操作系统**			学期	**2023 年秋季学期**
任课教师	窦金凤	完成日期	2023.12.15	上机课时间	周五 3-6
实 验 名 称	**文件系统**				

Exercise 部分：

Exercise 5.1

> Exercise 5.1 请根据 lib/syscall_all.c 中的说明，完成 sys_write_dev 函数和 sys_read_dev 函数的，并且在 user/lib.h, user/syscall_lib.c 中完成用户态的相应系统调用的接口。
>
> 编写这两个系统调用时需要注意物理地址、用户进程虚拟地址同内核虚拟地址之间的转换。
>
> 同时还要检查物理地址的有效性，在实验中允许访问的地址范围为：console: [0x10000000, 0x10000020)，disk: [0x13000000, 0x13004200)，rtc: [0x15000000, 0x15000200)，当出现越界时，应返回指定的错误码。 ■

sys_write_dev 的目的是将用户空间缓冲区 (va) 的数据写入到设备内存区 (dev)。在将数据复制之前，该实现检查指定的设备地址范围 (dev) 是否在某些有效范围内。如果 dev 范围在有效范围内，就使用 bcopy 函数将数据从用户空间缓冲区复制到设备内存。

系统调用接受参数：va（用户空间缓冲区地址），dev（设备地址），和 len（要写入的数据长度）。

它检查 dev 地址是否落在特定有效范围内，这些范围对应于某些设备。

如果 dev 地址在有效范围内，就使用 bcopy 将数据从用户空间缓冲区复制到设备内存。

如果 dev 地址不在有效范围内，就返回错误码 (-E_INVAL)。

代码如图 1 所示。

```
611 int sys_write_dev(int sysno, u_int va, u_int dev, u_int len)
612 {
613         // Your code here
614     if((0x10000000 <= dev && dev + len <= 0x10000000 + 0x20)
615             || (0x13000000 <= dev && dev + len <= 0x13000000 + 0x4200)
616             || (0x15000000 <= dev && dev + len <= 0x15000000 + 0x200)){
617         bcopy((char *)va, (char *)(dev + 0xA0000000), len);
618         return 0;
619     }
620     return -E_INVAL;
621 }
```

图 1　sys_write_dev 函数实现

sys_read_dev 目的是从设备内存中读取数据，并将其复制到用户空间缓冲区。实现检查指定的设备地址范围 (dev) 是否在某些有效范围内。如果 dev 范围在有效范围内，就使用 bcopy 函数将数据从设备内存复制到用户空间缓冲区。

系统调用接受参数：va，dev 和 len（要读取的数据长度）。

它检查 dev 地址是否落在特定有效范围内，这些范围对应于某些设备。

如果 dev 地址在有效范围内，就使用 bcopy 将数据从设备内存复制到用户空间缓冲区。

如果 dev 地址不在有效范围内，就返回错误码 (-E_INVAL)。

代码如图 2 所示。

```
640 int sys_read_dev(int sysno, u_int va, u_int dev, u_int len)
641 {
642         // Your code here
643         if((0x10000000 <= dev && dev + len <= 0x10000000 + 0x20)
644                 || (0x13000000 <= dev && dev + len <= 0x13000000 + 0x4200)
645                 || (0x15000000 <= dev && dev + len <= 0x15000000 + 0x200)){
646             bcopy((char *)(dev + 0xA0000000), (char *)va, len);
647             return 0;
648         }
649         return -E_INVAL;
650 }
```

图 2　sys_read_dev 函数实现

Exercise 5.2

> **Exercise 5.2** 参考这个内核态驱动，完成 fs/ide.c 中的 `ide_write` 函数，以及 `ide_read` 函数，实现对磁盘的读写操作。 ■

从设备读取数据的函数 ide_read。函数的主要流程是在循环中多次执行以下步骤：

使用 syscall_write_dev 将磁盘号写入到 IDE 控制寄存器（0x13000010）中，通知 IDE 设备。

使用 syscall_write_dev 将要读取的扇区偏移地址 (offset_begin + offset) 写入到 IDE 地址寄存器（0x13000000）中，指定要读取的数据的位置。

使用 syscall_write_dev 将一个标志位 (0) 写入到 IDE 控制寄存器（0x13000020），通知 IDE 开始读取。

使用 syscall_read_dev 读取 IDE 状态寄存器（0x13000030），检查是否读取成功。如果状态为 0，表示出错，调用 user_panic 报告错误。

使用 syscall_read_dev 读取实际数据到目标缓冲区 (dst + offset)，从 IDE 数据寄存器（0x13004000）读取。

该函数循环的次数由 nsecs 决定，每次读取一个扇区。在循环的每次迭代中，offset 会逐渐增加，指示下一个要读取的数据位置。

ide_read 函数实现如图 3 所示。

```c
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs)
{
    int offset_begin = secno * 0x200;
    int offset_end = offset_begin + nsecs * 0x200;
    int offset = 0;

    int temp;
    while (offset_begin + offset < offset_end) {
        // Your code here
        temp = diskno;
        if(syscall_write_dev((int)&temp, 0x13000010, 4))
            user_panic("error in ide_read()\n");
        temp = offset_begin + offset;
        if(syscall_write_dev((int)&temp, 0x13000000, 4))
            user_panic("error_in_read()\n");
        temp = 0;
        if(syscall_write_dev((int)&temp, 0x13000020, 4))
            user_panic("error in ide_read()\n");
        if(syscall_read_dev((int)&temp, 0x13000030, 4))
            user_panic("error in ide_read()\n");
        if (temp == 0) user_panic("error in ide_read()\n");
        if(syscall_read_dev((int)(dst + offset), 0x13004000, 512))
            user_panic("error in ide_read()\n");
        offset += 0x200;
    }
}
```

图 3　ide_read 函数实现

向设备写入数据的函数 ide_write。函数的主要流程是在循环中多次执行以下步骤：
使用 syscall_write_dev 将要写入的数据写入到 IDE 数据寄存器（0x13004000），写入一个扇区的数据。
使用 syscall_write_dev 将磁盘号写入到 IDE 控制寄存器（0x13000010）中，通知 IDE 设备。
使用 syscall_write_dev 将要写入的扇区偏移地址 (offset_begin + offset) 写入到 IDE 地址寄存器（0x13000000）中，指定要写入数据的位置。
使用 syscall_write_dev 将一个标志位 (1) 写入到 IDE 控制寄存器(0x13000020)，通知 IDE 开始写入。
使用 syscall_read_dev 读取 IDE 状态寄存器（0x13000030），检查是否写入成功。如果状态为 0，表示出错，调用 user_panic 报告错误。

该函数循环的次数由 nsecs 决定，每次写入一个扇区。在循环的每次迭代中，offset 会逐渐增加，指示下一个要写入的数据位置。

ide_write 函数实现如图 4 所示。

```c
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs)
{
    // Your code here
    int offset_begin = secno * 0x200 ;
    int offset_end = offset_begin + nsecs * 0x200 ;
    int offset = 0 ;
    int temp;
    while(offset_begin + offset < offset_end){
        if(syscall_write_dev((int)(src + offset), 0x13004000, 512))
            user_panic("error in ide_write()");
        temp = diskno;
        if(syscall_write_dev((int)&temp, 0x13000010, 4))
            user_panic("error in ide_write()");
        temp = offset_begin + offset;
        if(syscall_write_dev((int)&temp, 0x13000000, 4))
            user_panic("error in ide_write()");
        temp = 1;
        if(syscall_write_dev((int)&temp, 0x13000020, 4))
            user_panic("error in ide_write()");
        if(syscall_read_dev((int)&temp,0x13000030,4))
            user_panic("error in ide_write()");
        if(temp == 0)
            user_panic("error in ide_write()");
        offset += 0x200;
    }
}
```

图 4　ide_write 函数实现

Exercise 5.3

Exercise 5.3 文件系统需要负责维护磁盘块的申请和释放，在回收一个磁盘块时，需要更改 bitmap 中的标志位。如果要将一个磁盘块设置为 free，只需要将 bitmap 中对应的 bit 的值设置为 0x1 即可。请完成 fs/fs.c 中的 `free_block` 函数，实现这一功能。同时思考为什么参数 `blockno` 的值不能为 0 ？

```c
// Overview:
//  Mark a block as free in the bitmap.
void
free_block(u_int blockno)
{
    // Step 1: Check if the parameter `blockno` is valid (`blockno` can't be zero).

    // Step 2: Update the flag bit in bitmap.

}
```

free_block 函数实现了释放一个数据块：

检查参数 blockno 是否有效，即不能为零，且必须小于文件系统中数据块总数。如果不满足条件，调用 user_panic 报告错误。

计算数据块在位图中的索引，然后更新相应的位，将对应的标志位置为 1，表示该数据块已被释放。

代码如图 5 所示。

```
215 void free_block(u_int blockno)
216 {
217     // Step 1: Check if the parameter `blockno` is valid (`blockno` can't be zero).
218     if(blockno == 0 || blockno >= super->s_nblocks){
219         user_panic("blockno is zero");
220     }
221     // Step 2: Update the flag bit in bitmap.
222     bitmap[blockno / 32] |= (1 << (blockno % 32));
223 }
```

图 5　free_block 函数实现

Exercise 5.4

Exercise 5.4 请文件系统的设计，完成 fsformat.c 中的 `create_file`函数，并按照个人兴趣完成 `write_directory` 函数（不作为考察点），实现将一个文件或指定目录下的文件按照目录结构写入到 fs/fs.img 的根目录下的功能。关于如何创建二进制文件的镜像，请参考 fs/Makefile。

在实现的过程中，你可以将你的实现同我们给出的参考可执行文件 tools/fsformat 进行对比。具体来讲，你可以通过 Linux 提供的 xxd 命令将两个 fsformat 产生的二进制镜像转化为可阅读的文本文件，手工进行查看或使用 diff 等工具进行对比。 ■

create_file 函数实现了在给定目录文件 dirf 下创建一个新文件：

首先，根据目录文件已占用的块数 nblk 进行分类讨论，计算正确的块号 bno。对于直接块和间接块，根据 nblk 计算出对应的块号。

遍历目录块 dirblk，查找一个未被使用的目录项，即 dirblk[j].f_name[0] == '\0' 的项。

如果找到未使用的目录项，返回该目录项的指针，表示创建成功。

如果在当前块中找不到未使用的目录项，则通过 make_link_block 函数创建新的块，并返回新块的指针。

create_file 函数实现代码如图 6 所示。

```
209 struct File *create_file(struct File *dirf) {
210     struct File *dirblk;
211     int i,j, bno, found;
212     int nblk = dirf->f_size / BY2BLK;
213     // Your code here
214     // Step1: According to different range of nblk, make classified discussion to
215     //        calculate the correct block number.
216     for(i = 0; i < nblk; i++){
217         if(i < NDIRECT){
218             bno = dirf->f_direct[i];
219         }else{
220             bno = ((uint32_t *)(disk[dirf->f_indirect].data))[i];
221         }
222         dirblk = (struct File *)(disk[bno].data);
223         for(j = 0; j< FILE2BLK; j++){
224             if(dirblk[j].f_name[0] == '\0'){
225                 return &dirblk[j];
226             }
227         }
228 
229     }
230     // Step2: Find an unused pointer
231     bno = make_link_block(dirf, nblk);
232     return (struct File *)disk[bno].data;
233 }
```

图 6　create_file 函数实现

Exercise 5.5

> **Exercise 5.5** fs/fs.c 中的 diskaddr 函数用来计算指定磁盘块对应的虚存地址。完成 `diskaddr` 函数，根据一个块的序号 (block number)，计算这一磁盘块对应的 512 bytes 虚存的起始地址。(提示：fs/fs.h 中的宏 `DISKMAP` 和 `DISKMAX` 定义了磁盘映射虚存的地址空间)。 ■

diskaddr 函数实现了一个用于计算磁盘块号对应的虚拟地址的。函数首先检查给定的磁盘块号是否有效，如果无效，则通过 user_panic 抛出错误。接着，函数通过简单的乘法和加法操作计算磁盘块在内存中的虚拟地址。这个虚拟地址是通过将磁盘块号乘以每个块的字节数（BY2BLK）得到的，再加上一个表示磁盘在内存中的映射起始地址（DISKMAP）。最终，函数返回计算得到的虚拟地址。

diskaddr 函数实现代码如图 7 所示。

```
16 u_int diskaddr(u_int blockno)
17 {
18     if(super && blockno >= super->s_nblocks){
19         user_panic("reading non-existent block %08x\n", blockno);
20     }
21     return DISKMAP + (blockno * BY2BLK);
22 }
```

图 7　diskaddr 函数实现

Exercise 5.6

Exercise 5.6 实现 `map_block` 函数，检查指定的磁盘块是否已经映射到内存，如果没有，分配一页内存来保存磁盘上的数据。对应地，完成 `unmap_block` 函数，用于解除磁盘块和物理内存之间的映射关系，回收内存。(提示：注意磁盘虚拟内存地址空间和磁盘块之间的对应关系)。 ■

map_block 函数实现了一个用于映射磁盘块到物理内存的。函数首先通过 block_is_mapped 判断指定的磁盘块是否已经映射到物理内存中，如果已经映射，则直接返回 0，表示无需再次映射。如果磁盘块尚未映射，函数调用 syscall_mem_alloc 来为该磁盘块分配一页物理内存，并将其映射到该物理页上，标志位设置为可读可写。函数返回 syscall_mem_alloc 的执行结果，为成功分配物理内存返回 0，否则返回相应的错误码。

map_block 函数实现如图 8 所示。

```
69  int map_block(u_int blockno)
70  {
71      // Step 1: Decide whether this block has already mapped to a page of physical memory.
72      if(block_is_mapped(blockno)){
73          return 0;
74      }
75      // Step 2: Alloc a page of memory for this block via syscall.
76      return syscall_mem_alloc(0, diskaddr(blockno), PTE_V | PTE_R);
77  }
```

图 8　map_block 函数实现

unmap_block 函数实现了一个用于解除磁盘块映射的。函数首先检查指定的磁盘块是否已经映射，如果未映射，则直接返回。接着，函数使用 block_is_free 和 block_is_dirty 检查磁盘块的状态。如果磁盘块既不是空闲的，又是脏的（有修改未写回），则调用 write_block 将其写回磁盘。然后，函数使用 syscall_mem_unmap 解除虚拟内存中对应的映射。最后，函数验证解除映射操作的结果，确保磁盘块不再被映射。

unmap_block 函数实现如图 9 所示。

```
82  void unmap_block(u_int blockno)
83  {
84      int r;
85      u_int addr;
86      // Step 1: check if this block is mapped.
87      addr = block_is_mapped(blockno);
88      if(addr == 0){
89          return;
90      }
91      // Step 2: use block_is_free, block_is_dirty to check block,
92      if(!block_is_free(blockno) && block_is_dirty(blockno)){
93          write_block(blockno);
94      }
95      // Step 3: use 'syscall_mem_unmap' to unmap corresponding virtual memory.
96      r = syscall_mem_unmap(0, addr);
97      if(r < 0){
98          user_panic("umap_block failed");
99      }
100     // Step 4: validate result of this unmap operation.
101     user_assert(!block_is_mapped(blockno));
102 }
```

图 9　unmap_block 函数实现

Exercise 5.7

Exercise 5.7 补完 dir_lookup 函数，查找某个目录下是否存在指定的文件。(提示：使用file_get_block可以将某个指定文件指向的磁盘块读入内存)。 ■

dir_lookup 函数实现了一个目录查找，用于在指定的目录中查找文件名为 name 的文件。函数首先计算目录所占用的块数 nblock，然后在每个块上进行循环查找。在每个块上，通过 file_get_block 读取块的内容，然后在该块的文件项中遍历，比对文件名，如果找到匹配的文件，则将文件项赋值给 *file，并返回 0 表示成功找到文件。如果在整个目录中都没有找到匹配的文件，则返回错误码 -E_NOT_FOUND。

dir_lookup 函数实现如图 10 所示。

```
560 int dir_lookup(struct File *dir, char *name, struct File **file)
561 {
562     int r;
563     u_int i, j, nblock;
564     void *blk;
565     struct File *f;
566     // Step 1: Calculate nblock: how many blocks are there in this dir?
567     nblock = ROUND(dir->f_size, BY2BLK)/BY2BLK;
568     for (i = 0; i < nblock; i++) {
569         // Step 2: Read the i'th block of the dir.
570         // Hint: Use file_get_block.
571         r = file_get_block(dir, i, &blk);
572         if(r < 0){
573             return r;
574         }
575         // Step 3: Find target file by file name in all files on this block.
576         for(j = 0; j < FILE2BLK; j++){
577             f = ((struct File *)blk) + j;
578             if(strcmp((char *)f->f_name, name) == 0 ){
579                 f->f_dir = dir;
580                 *file = f;
581                 return 0;
582             }
583         }
584     }
585     return -E_NOT_FOUND;
586 }
```

图 10　dir_lookup 函数实现

Exercise 5.8

Exercise 5.8 完成 user/file.c 中的 open 函数。(提示：若成功打开文件则该函数返回文件描述符的编号)。 ■

open 函数实现了文件打开的过程。

文件描述符分配：调用 fd_alloc 函数分配一个文件描述符（struct Fd 类型），如果分配失败则返回错误。文件描述符是操作系统内核维护的结构，用于跟踪进程打开的文件。

打开文件：调用 fsipc_open 函数，该函数通过 IPC 与文件系统进行通信，根据给定的文件路径 path 和打开模式 mode 获取文件的描述符信息，包括文件大小、inode 等。如果打开文件失败，则返回相应的错误码。

文件内容映射到内存：使用 fd2data 获取文件内容的起始虚拟地址 va，以及文件的大小 size 和文件编号 fileid。如果打开模式中包含 O_APPEND，则设置文件偏移为文件的大小。

循环映射文件内容：通过循环，每次映射一个块大小（BY2BLK）的文件内容到虚拟地址空间。这是通

过调用 fsipc_map 函数实现的，该函数通过 IPC 与文件系统通信，将文件中指定偏移处的数据映射到指定的虚拟地址。如果映射失败，则返回相应的错误码。

返回文件描述符编号：返回通过 fd2num 获取的文件描述符的编号。

open 函数代码如下：

```
int open(const char *path, int mode)
{
	struct Fd *fd;
	struct Filefd *ffd;
	u_int size, fileid;
	int r;
	u_int va;
	u_int i;
	// Step 1: Alloc a new Fd, return error code when fail to alloc.
	// Hint: Please use fd_alloc.
	r = fd_alloc(&fd);
	if(r < 0){
		return r;
	}
	// Step 2: Get the file descriptor of the file to open.
	// Hint: Read fsipc.c, and choose a function.
	r = fsipc_open(path, mode, fd);
	if(r < 0){
		return r;
	}
	// Step 3: Set the start address storing the file's content. Set size and fileid correctly.
	// Hint: Use fd2data to get the start address.
	ffd = (struct Filefd *)fd;
	va = fd2data(fd);
	fileid = ffd->f_fileid;
	size = ffd->f_file.f_size;
	if((mode & O_APPEND) != 0){
		ffd->f_fd.fd_offset = ffd->f_file.f_size;
	}
	// Step 4: Alloc memory, map the file content into memory.
	for(i = 0; i < size; i += BY2BLK){
		r = fsipc_map(fileid, i, va + i);
		if(r < 0){
			return r;
		}
	}
	// Step 5: Return the number of file descriptor.
	return fd2num(fd);
}
```

Exercise 5.9

Exercise 5.9 参考 user/fd.c 中的 `write` 函数，完成 `read` 函数。

实现了用户空间的 read 系统调用，用于从文件描述符指定的文件中读取数据：

首先调用 fd_lookup 函数通过文件描述符编号 fdnum 获取文件描述符结构 fd，并调用 dev_lookup 函数通过设备 ID 获取设备结构 dev。如果获取失败，则返回相应的错误码。

检查文件描述符的打开模式，如果打开模式是只写（O_WRONLY），则返回错误，因为不允许在只写模式下执行读操作。

调用设备结构中的 dev_read 函数，该函数通过设备的读取接口从文件中读取数据。传递给设备读取函数的参数包括文件描述符 fd、目标缓冲区 buf、要读取的字节数 n 以及文件的当前偏移量 fd->fd_offset。

如果读取成功（返回的字节数 r 大于零），则更新文件描述符的偏移量 fd->fd_offset。这个偏移量表示下一次读写操作应该在文件中的何处进行。

在读取到的数据后面追加一个空字符 '\0'，以便在用户空间中将其视为 C 字符串。

返回实际读取的字节数。如果读取失败，可能返回相应的错误码。

read 函数的代码如图 12 所示。

```
185 int read(int fdnum, void *buf, u_int n)
186 {
187     int r;
188     struct Dev *dev;
189     struct Fd *fd;
190     // Step 1: Get fd and dev.
191     if((r = fd_lookup(fdnum, &fd)) < 0 || (r = dev_lookup(fd->fd_dev_id, &dev)) < 0 ){
192         return r;
193     }
194     // Step 2: Check open mode.
195     if((fd->fd_omode & O_ACCMODE) == O_WRONLY){
196         writef("[%08x] read %d -- bad mode\n", env->env_id, fdnum);
197         return -E_INVAL;
198     }
199     if(debug) writef("read %d %p %d via dev %s\n", fdnum,buf,n,dev->dev_name);
200     // Step 3: Read starting from seek position.
201     r = (*dev->dev_read)(fd, buf, n, fd->fd_offset);
202     // Step 4: Update seek position and set '\0' at the end of buf.
203     if(r > 0){
204         fd->fd_offset += r;
205     }
206     ((char *)buf)[r] = '\0';
207     return r;
208 }
```

图 11　read 函数实现

Exercise 5.10

Exercise 5.10 文件 user/fsipc.c 中定义了请求文件系统时用到的 IPC 操作，user/file.c 文件中定义了用户程序读写、创建、删除和修改文件的接口。完成 user/fsipc.c 中的 `fsipc_remove` 函数、user/file.c 中的 `remove` 函数，以及 fs/serv.c 中的 `serve_remove` 函数，实现删除指定路径的文件的功能。

fsipc_remove 函数实现了文件系统客户端发送删除文件请求的功能：

使用 strlen 函数检查路径的长度是否超过了最大路径长度 MAXPATHLEN，如果超过则返回错误码 -E_BAD_PATH 表示路径无效。

将 fsipcbuf 转换为 struct Fsreq_remove* 类型，这是为了构建删除文件请求的结构体。

使用 strcpy 函数将待删除文件的路径复制到请求结构体 req 中的 req_path 字段中。

调用 fsipc 函数发送删除文件请求给文件系统服务器。FSREQ_REMOVE 是请求的类型，req 是请求结构体，后两个参数为附加的信息，这里没有使用。

返回文件系统服务器的响应，通常是删除操作的结果。如果返回值为 0 表示成功，其他值表示失败。

```
144 int fsipc_remove(const char *path)
145 {
146     struct Fsreq_remove *req;
147     // Step 1: Check the length of path, decide if the path is valid.
148     if(strlen(path) >= MAXPATHLEN){
149         return -E_BAD_PATH;
150     }
151     // Step 2: Transform fsipcbuf to struct Fsreq_remove*
152     req = (struct Fsreq_remove *)fsipcbuf;
153 
154     // Step 3: Copy path to path in req.
155     strcpy((char *)req->req_path, path);
156     // Step 4: Send request to fs server with IPC.
157     return fsipc(FSREQ_REMOVE, req, 0, 0);
158 }
```

图 12　fsipc_remove 函数实现

Remove 函数中调用 fsipc_remove 函数，删除当前路径。

```
273 int remove(const char *path)
274 {
275     // Your code here.
276     return fsipc_remove(path);
277 }
```

图 13　remove 函数实现

serve_remove 函数实现了服务端处理用户进程删除文件请求的功能：

使用 strcpy 函数将请求中的路径 rq->req_path 复制到本地的 path 数组中。这确保路径以 null 字符（\0）终止。

调用 file_remove 函数，该函数根据给定的路径从文件系统中删除相应的文件。返回值 r 表示删除操作的结果，通常为 0 表示成功，负值表示失败。

使用 ipc_send 函数将删除操作的结果 r 发送给用户进程。这是通过调用方传递的 envid 来标识用户进程的。

```
239 void serve_remove(u_int envid, struct Fsreq_remove *rq)
240 {
241     int r;
242     u_char path[MAXPATHLEN];
243     // Step 1: Copy in the path, making sure it's terminated.
244     strcpy(path, (char *)rq->req_path);
245     // Step 2: Remove file from file system and response to user-level process.
246     r = file_remove(path);
247     ipc_send(envid, r, 0, 0);
248 }
```

图 14　serve_remove 函数实现

Thinking 部分：

Thinking 5.1

> **Thinking 5.1**　查阅资料，了解 Linux/Unix 的 /proc 文件系统是什么？有什么作用？Windows 操作系统又是如何实现这些功能的？proc 文件系统这样的设计有什么好处和可以改进的地方？ ■

1. /proc 文件系统是一个虚拟文件系统，主要用于提供内核和进程信息的访问接口。在 Linux/Unix 系统中，/proc 文件系统允许用户和系统工具通过文件的方式访问内核和进程的运行时信息，而这些信息通常存储在内核的数据结构中。

2. /proc 文件系统的主要作用：

进程信息： /proc 提供了一个目录结构，其中包含了系统中每个运行进程的目录。这些目录中包含了有关进程的详细信息，如进程 ID、状态、命令行参数、文件描述符等。

系统信息： /proc 还包含一些系统级的信息，例如系统的内存使用情况、CPU 信息、设备信息等。这些信息可以帮助用户和系统管理员了解系统的运行状态。

内核参数： /proc 文件系统提供了一些用于配置内核参数的文件，用户可以通过修改这些文件来调整系统的行为，而无需重新编译内核。

3. 在 Windows 操作系统中，类似 /proc 的功能是由注册表（Registry）和性能计数器（Performance Counter）等机制来实现的。注册表包含了系统和应用程序的配置信息，而性能计数器用于收集和显示系统性能相关的数据。通过这些机制，Windows 提供了类似于 /proc 的系统和进程信息。

4. 有关 /proc 文件系统设计的一些优点和改进的方向包括：

/proc 文件系统提供了一个透明的访问接口，使得用户和系统工具能够以一致的方式访问系统和进程信息。这种透明性使得开发者和系统管理员能够更容易地了解和调试系统。/proc 中的信息是动态更新的，用户可以实时地获取系统和进程的运行时状态。这对于性能分析、故障排除和监控是非常有用的。/proc 文件系统的设计使得添加新的信息和功能相对容易。这种灵活性允许开发者在不修改内核源代码的情况下扩展和改进 /proc。/proc 提供了一种标准化的接口，使得用户和工具能够通过读取和写入文件的方式与内核和进程交互。这种标准化简化了用户空间工具的开发和维护。

Thinking 5.2

> **Thinking 5.2** 如果我们通过 kseg0 读写设备，我们对于设备的写入会缓存到 Cache 中。通过 kseg0 访问设备是一种**错误**的行为，在实际编写代码的时候这么做会引发不可预知的问题。请你思考：这么做这会引起什么问题？对于不同种类的设备（如我们提到的串口设备和 IDE 磁盘）的操作会有差异吗？可以从缓存的性质和缓存刷新的策略来考虑。 ■

kseg0 缓存：在 kseg0 地址空间范围内，对设备的写入操作可能会被缓存到 CPU 缓存中，而不会立即写入到设备。这可能导致设备的实际状态和 CPU 缓存中的状态不一致，特别是在进行连续写入时。
seg0 缓存：在使用 seg0 地址空间范围时，操作系统通常会采用不同的策略来处理设备写入。通常，对于 seg0 读写设备的操作会绕过 CPU 缓存，确保数据直接传递到设备而不经过缓存。

缓存刷新的策略：
kseg0 缓存刷新：对于 kseg0 地址范围内的设备写入，操作系统可能需要采取额外的步骤来刷新 CPU 缓存，以确保数据被正确写入到设备。这通常涉及到特殊的缓存刷新指令或者使用 cache 指令来清除或刷新缓存。
seg0 缓存刷新：在使用 seg0 地址范围时，由于通常采用了不缓存的策略，可能不需要显式刷新缓存，因为数据不会被缓存在 CPU 中。

对于不同种类的设备（例如串口设备和 IDE 磁盘），可能存在一些差异。串口设备通常对写入的响应较为迅速，且对数据的一致性要求较低。在这种情况下，对于缓存的处理可能相对简单，可以通过适当的缓存刷新策略来确保数据正确传递到串口设备。对于 IDE 磁盘等存储设备，写入操作的一致性更为关键。在使用 kseg0 地址范围时，可能需要更复杂的缓存管理和刷新策略，以确保数据的一致性和可靠性。

Thinking 5.3

> **Thinking 5.3** 一个 Block 最多存储 1024 个指向其他磁盘块的指针，试计算，我们的文件系统支持的单个文件的最大大小为多大？ ■

单个文件的最大大小为 4KB*1024=4MB

Thinking 5.4

> **Thinking 5.4** 查找代码中的相关定义，试回答一个磁盘块中最多能存储多少个文件控制块？一个目录最多能有多少个子文件？ ■

一个磁盘块中最多能存储 16 个文件控制块。
一个目录下最多能有 1024*16=16384 个文件

Thinking 5.5

> **Thinking 5.5** 请思考，在满足磁盘块缓存的设计的前提下，我们实验使用的内核支持的最大磁盘大小是多少？ ■

我们实验使用的内核支持的最大磁盘大小为 0x6f3fd000。

Thinking 5.6

> **Thinking 5.6** 阅读 user/file.c 中的众多代码，发现很多函数中都会将一个 struct Fd * 型的指针转换为 struct Filefd * 型的指针，请解释为什么这样的转换可行。 ■

有可能为一下两点：

内存布局连续性： C 语言规定结构体的字段是按照声明顺序依次排列的，而结构体的首地址即为第一个字段的地址。如果 struct Filefd 的定义包含了 struct Fd 的定义，并且 struct Fd 是 struct Filefd 的前缀部分，那么两者的字段在内存中是连续排列的。

指针类型的转换： 在 C 语言中，可以通过将一个指针类型强制转换为另一个类型的指针，以改变该指针的类型。这种转换在编译时是合法的，但需要确保运行时的实际内存布局符合预期，否则可能导致未定义的行为。

Thinking 5.7

> **Thinking 5.7** 请解释 File, Fd, Filefd, Open 结构体及其各个域的作用。比如各个结构体会在哪些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。 ■

Fd 结构体用于表示文件描述符，包含以下字段：

fd_dev_id: 文件所在设备的 id。

fd_offset: 读或写文件时距离文件开头的偏移量。

fd_omode: 描述文件打开的读写模式。

它主要用于在打开文件之后记录文件的状态，以便对文件进行管理/读写，不对应物理实体，只是单纯的内存数据。

Filefd 结构体是文件描述符和文件的组合形式，包含以下字段：

f_fd: 记录文件描述符的部分，包括设备 id、偏移量和打开模式。

f_fileid: 记录文件的 id。

f_file: 记录文件控制块，包含文件的信息以及指向储存文件的磁盘块的指针。它对应了磁盘的物理实体，同时也包含内存中的数据。

Open 结构体在文件系统进程中用于存储文件相关信息，包含以下字段：

o_file: 指向对应的文件控制块。
o_fileid: 表示文件 id，用于在数组 opentab 中查找对应的 Open。
o_mode: 记录文件打开的状态。
o_ff: 指向对应的 Filefd 结构体。

Thinking 5.8

Thinking 5.8 阅读serve函数的代码,我们注意到函数中包含了一个死循环for (;;) {...}，为什么这段代码不会导致整个内核进入 panic 状态? ■

在 serve 函数中，for (; ;) 循环并不会导致整个内核进入 panic 状态的原因在于这是一个无限循环的用户态服务处理循环，不同于内核态的异常或中断处理。

用户态服务处理循环一直在等待并处理用户级进程发来的系统调用请求。在这个循环中，如果用户级进程发送了一个有效的系统调用，内核会根据系统调用号调用相应的系统调用处理函数。处理完用户请求后，serve 函数会继续等待下一个系统调用请求。

由于这是用户态服务循环，而不是内核异常处理循环，因此不会导致整个内核进入 panic 状态。只有在内核态处理中出现严重错误、不可恢复的异常，或者用户态进程发送的请求有问题时，才可能导致内核 panic。在正常情况下，serve 函数会不断处理用户请求而不中断整个内核的正常运行。

本次实验耗时 7 小时 40 分