哈爾濱工業大學 (深圳)

HARBIN INSTITUTE OF TECHNOLOGY

实验报告）

开课学期:	2024 秋季学期
课程名称:	操作系统
实验名称:	基于 FUSE 的青春版 EXT2 文件系统
学生班级:	9 班
学生学号:	220110915
学生姓名:	马奕斌
评阅教师:	
报告成绩:	

2024 年 11 月

一、 实验详细设计

图文并茂地描述实验实现的所有功能和详细的设计方案及实验过程中的特色部分。

hitFuse 是一个基于 FUSE 的文件系统，支持以下功能：

- 创建、删除、重命名文件和目录 `mkdir, touch, ls, rm, mv ...`
- 读取和写入文件内容 `cat, echo ...`
- 查看文件和目录的属性 `ls ...`

该系统在设计上尽可能采用了模块化设计，于 `Simplefs` 的基础上更加明确地划分了具体的功能模块，并实现了对 FUSE 的封装，使得文件系统的实现更加简洁和高效。

1 总体设计方案

详细阐述文件系统的总体设计思路，包括系统架构图和关键组件的说明

hitFuse 的整体架构图如下：

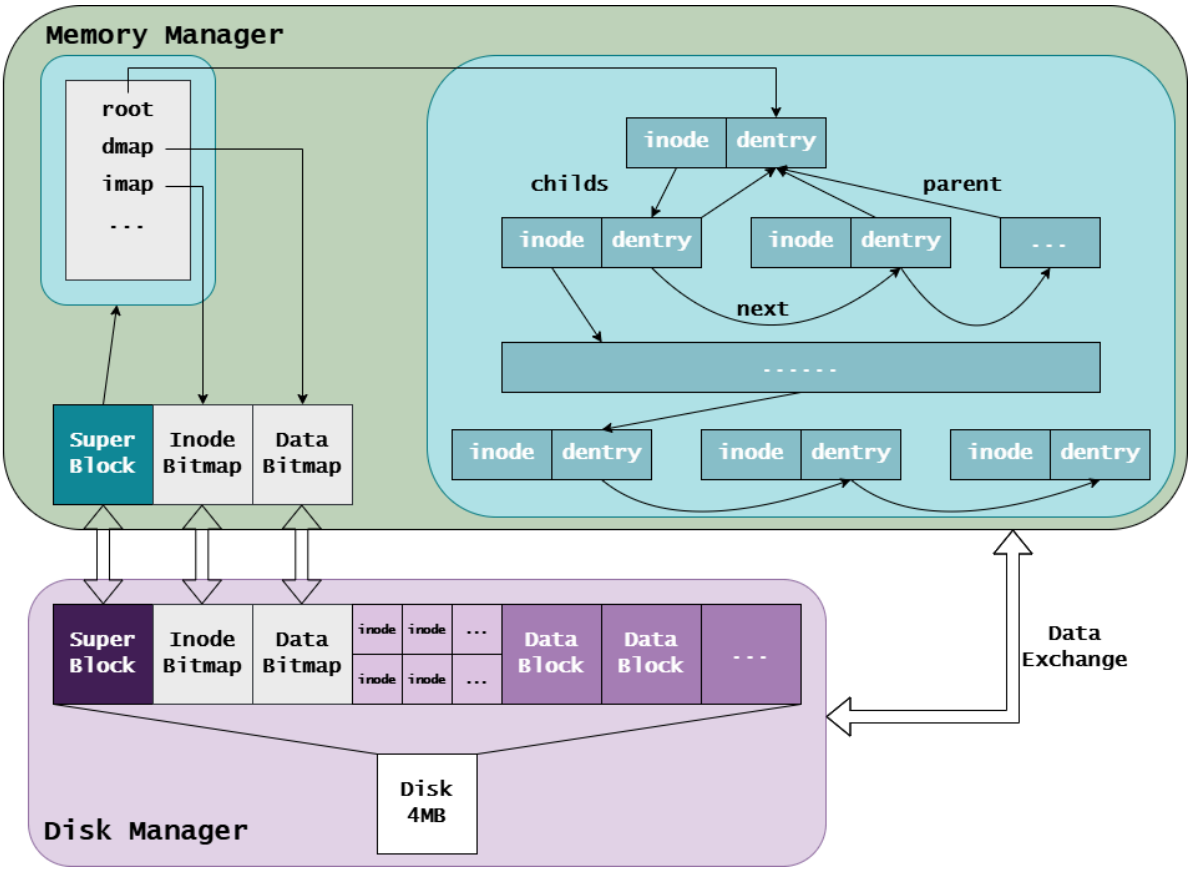

Figure 1: hitFuse 整体架构

hitFuse 的整体架构可以分为以下几个部分：

- **Disk Manager**: 负责管理磁盘空间，包括磁盘读写、内存与磁盘的信息同步、文件读写等。
- **Memory Manager**: 负责管理文件系统的元数据
  - 通过 `inode` 与 `dentry` 维护文件系统的层次结构
  - 通过 `Inode Bitmap` 与 `Data Bitmap` 维护磁盘空间的使用情况

所有来自用户空间的操作都会经过 Memory Manager，由 Memory Manager 进行元数据操作后再进行必要的磁盘操作，从而保证文件系统的正确性和一致性。

2 数据结构说明

本系统中涉及到的数据结构如下：

- **Inode Bitmap**: 磁盘上 inode 的使用情况，用于维护 inode 的分配和释放。
- **Data Bitmap**: 磁盘上数据块的使用情况，用于维护数据块的分配和释放。
- **Dentry**: 目录项，包括文件名、文件类型等。
- **Inode**: 文件或目录的元数据，包括其使用的数据块、大小、子目录等。
- **Super Block**: 全局的元数据，包括文件系统的总大小、已使用空间、根目录等。
- **Disk** : 磁盘，用于实际存储文件系统的数据。

2.1 Bitmap

Bitmap 是一个位图，用于表示磁盘空间的使用情况。每个位表示一个磁盘块的使用情况，0 表示未使用，1 表示已使用。本系统中的数据位图包括 Inode Bitmap 和 Data Bitmap。

任何对磁盘存储空间的操作（除 Super Block 外）都需要先对相应的位图进行操作，以维护磁盘空间的使用情况。

Bitmap 提供以下方法：

- `bitmap_init`: 初始化位图，将所有位设置为 0。
- `bitmap_alloc`: 分配一个未使用的磁盘块，并返回其索引。
- `bitmap_clear`: 将位图中指定索引的位设置为 0，表示释放该磁盘块。

> 💡 **Design Choice**
>
> 在 alloc 和 clear 时，可以对磁盘相应的数据块进行初始化或清除，以防止隐私泄露。
>
> 在本实现中，仅通过对位图进行操作，标记相应的数据块可用。

相应方法的实现均为 bit-level 操作，具体实现见代码，此处仅以 `bitmap_clear` 为例：

```
void bitmap_clear(uint8_t *bitmap, uint32_t index) {
    uint32_t byte_index = index / 8;
    uint32_t bit_index = index % 8;
    bitmap[byte_index] &= ~(1 << bit_index);
}
```

通过将 imap 或 dmap 传入该函数，可以实现对相应位图的修改。

2.2 Dentry && Inode

Dentry 是目录项，主要用于维护文件系统的层次结构。其结构如下：

```
struct fs_dentry {
    FileType ftype;
    char     name[MAX_NAME_LEN];

    uint32_t ino;
    struct fs_inode *self;

    struct fs_dentry *parent;
    struct fs_dentry *next;
};
```

该结构主要包括三个部分：

- 文件类型 ftype 与文件名 name
- 文件对应的 inode 索引 ino 与 inode 结构体指针 self。该部分将 inode 与 dentry 关联起来，使得我们可以通过 dentry 快速找到对应的 inode
- 父目录 parent 与兄弟节点 next。该部分为维护文件系统层次结构的链表节点，使得我们可以通过 dentry 快速找到其父目录和兄弟节点

Inode 用于维护文件或目录的元数据，包括其使用的数据块、大小、子目录等。其结构如下：

```
struct fs_inode {
    uint32_t ino;
    struct fs_dentry *self;

    // * Directory Structure *
    int dir_cnt; // number of sub dentries
    struct fs_dentry *childs; // linked list of sub dentries
    uint32_t dno_dir; // data block number

    // * Regular File Structure *
    int size;
    uint32_t dno_reg[MAX_BLOCK_PER_INODE];
};
```

该结构主要包括三个部分：

- 文件索引 ino 与 dentry 结构体指针 self。该部分将 inode 与 dentry 关联起来，使得我们可以通过 inode 快速找到对应的 dentry
- 目录结构 Directory Structure。该部分用于维护目录的子目录，包括子目录的数量 dir_cnt、子目录的链表 childs 以及子目录的数据块索引 dno_dir
- 文件结构 Regular File Structure。该部分用于维护文件的元数据，包括文件大小 size 以及文件的数据块索引 dno_reg

> 💡 **Design Choice**
>
> 本实现中，每个文件最多只能包含 MAX_BLOCK_PER_INODE 个数据块。
>
> 事实上，如果将 dno_reg 改为指针数组，则可以作进一步扩展，如支持间接数据块索引，从而支持更大的文件。

以上两个数据结构共同实现了以下接口：

- dentry_create 与 inode_create: 创建新的 dentry 与 inode，未分配磁盘空间
- dentry_bind(inode): 将 inode 与 dentry 绑定
- dentry_register/unregister: 将 dentry 注册到父目录中，或从父目录中移除
- dentry_get(index): 获取指定索引的子目录项
- dentry_delete: 删除 dentry，递归清除位图上的对应位，标记为未使用
- inode_alloc: 在磁盘上分配新的数据块用于存放 inode 的子目录项

2.3 Path Resolution

Path Resolution 是文件系统的一个重要功能，用于将用户输入的路径解析为对应的 dentry。通过前面所定义的数据结构，再加上以下的辅助函数：

- get_fname: 获取路径中的文件名

我们可以提供以下接口：

- dentries_find(fname): 在当前目录的子目录中查找文件名对应的 dentry

  - 遍历当前目录项 dentry->self->childs，即存储在 inode->childs 中的链表，查找文件名对应的 dentry。
- dentry_lookup(path)：从根目录开始，查找路径对应的 dentry
  - 逐层解析路径，调用 dentries_find 查找对应的 dentry

dentry_lookup 的实现如下：

```
int dentry_lookup(char* path, struct fs_dentry** dentry)
{
    struct fs_dentry *ptr = super.root; // * start from root
    *dentry = ptr;

    char *fname;

    char* path_bak = strdup(path);

    int levels = get_path_level(path_bak);
    fname = strtok(path_bak, "/");

    for (int i = 0; i < levels; i++) {
        // Find fname in ptr's subdirecties
        ptr = dentry_find(ptr->self->childs, fname);
        if (ptr == NULL) {
            return ERROR_NOTFOUND;
        }
        *dentry = ptr;
        fname = strtok(NULL, "/");
    }
    if (ptr->self == NULL) {
        dentry_restore(ptr, ptr->ino);
    }
    return 0;
}
```

其中，dentry_restore 函数从磁盘中恢复 dentry，在后文会详细介绍。

2.4 Disk

Disk 为连续的数据块，每个数据块的大小为 BLOCK_SIZE。为了实现磁盘的读写操作，定义了以下接口：

- disk_read 与 disk_write: 读写磁盘数据块
- file_read 与 file_write: 直接同步文件数据到磁盘
- inode_sync: 同步 inode 数据到磁盘
- dentry_restore: 从磁盘数据重建 dentry
- disk_mount: 挂载磁盘，初始化 super 结构体
- disk_umount: 卸载磁盘，同步必要的数据，释放资源

各接口的实现思路如下：

disk_read && disk_write

主要参考 Simplefs 中的实现，通过设置暂存区 buf，将对齐的数据块读入暂存区，再从暂存区写入目标地址。

> 💡 **Design Choice**
>
> 本实现中，暂存区 buf 的大小和读写大小相同，在磁盘读写过程中可能会导致性能下降。
>
> 事实上，不对齐的数据块仅有首尾两块，可以只对首尾两块进行对齐操作，从而提高性能。

file_read && file_write

通过调用 disk_read 与 disk_write，将文件数据同步到磁盘。

inode 结构体中存储了文件的数据块索引 dno_reg[]，因此，file_read 与 file_write 函数可以直接通过 dno_reg[] 找到文件的数据块，进行读写操作。

具体的实现思路与 disk_read 与 disk_write 类似：

- 通过设置暂存区 buf，将对齐的数据块读入暂存区，再从暂存区写入目标地址。
- 主要的区别在于在 file_write 函数中需要实现 **lazy allocation**，即当文件的数据块索引 dno_reg[] 中没有对应的数据块时，需要分配新的数据块，并将数据写入其中。

inode_sync && dentry_restore

inode_sync 和 dentry_restore 主要用于同步文件系统的层次结构到磁盘。

- inode_sync 将 inode 写入磁盘的索引节点区，同时，将 inode 的子目录项递归写入磁盘的数据块中。
- dentry_restore 从磁盘的数据块中读取子目录项，并通过 dentry_register 将子目录项注册到父目录中。

disk_mount && disk_umount

disk_mount 和 disk_umount 主要用于挂载和卸载磁盘。

- disk_mount 的主要流程如下：
  - 读取磁盘的超级块 super，根据 **Magic Number** 判断是否需要格式化磁盘。
  - 同步磁盘的位图到内存，以便进行磁盘空间的分配和回收。
  - 重建根目录
- disk_umount 的主要流程如下：
  - 同步超级块 super 到磁盘
  - 同步位图到磁盘
  - 递归同步根目录到磁盘

3 功能详细说明

每个功能点的详细说明（关键的数据结构、核心代码、流程等）

建立在上述数据结构的基础上，各功能的实现如下：

3.1 挂载与卸载

挂载与卸载磁盘是文件系统的重要功能，用于初始化文件系统，以及同步文件系统的层次结构到磁盘。

该功能的 hook 函数为 fs_init 与 fs_destroy，具体实现如下：

```
void* fs_init(struct fuse_conn_info * conn_info) {
  return disk_mount();
}
```

```
void fs_destroy(void* p) {
  disk_umount();
}
```

3.2 查看文件信息

查看文件信息的功能对应的 hook 函数为 `fs_getattr`，参考 Simplefs 中的实现，具体实现如下：

```
int fs_getattr(const char* path, struct stat * fs_stat) {
  struct fs_dentry* dentry;
  if (dentry_lookup(path, &dentry) != 0) {
    return ERROR_NOTFOUND;
  }
  if (dentry->ftype == FT_DIR) {
    fs_stat->st_mode = S_IFDIR | FS_DEFAULT_PERM;
    fs_stat->st_size = dentry->self->dir_cnt * sizeof(struct fs_dentry_d);
  }
  if (dentry->ftype == FT_REG) {
    fs_stat->st_mode = S_IFREG | FS_DEFAULT_PERM;
    fs_stat->st_size = dentry->self->size;
  }

  fs_stat->st_nlink = 1;
  fs_stat->st_uid    = getuid();
  fs_stat->st_gid    = getgid();
  fs_stat->st_atime   = time(NULL);
  fs_stat->st_mtime   = time(NULL);
  fs_stat->st_blksize = super.params.size_block;

  if (strcmp(path, "/") == 0) {
    fs_stat->st_size  = super.params.size_usage;
    fs_stat->st_blocks = super.params.size_disk / super.params.size_block;
    fs_stat->st_nlink  = 2;    /* !特殊，根目录 link 数为 2 */
  }
  return 0;
}
```

> ⚠ **Warning**
>
> `mkdir` 与 `mknod` 均会经过该函数，必须先实现该函数。

3.3 创建文件/文件夹

创建文件夹的功能对应的 hook 函数为 `fs_mkdir`，实现思路为：

- 创建 dentry 与 inode，并将它们绑定
- 在 imap 中分配新的 inode 索引
- 将新建的 dentry 注册到父目录中

具体实现如下：

```
int fs_mkdir(const char* path, mode_t mode) {
  struct fs_dentry* parent;
  if (dentry_lookup(path, &parent) == 0) {
    return ERROR_EXISTS;
  }
  if (parent->ftype != FT_DIR) {
```

```
    return ERROR_NOTFOUND;
  }
  struct fs_dentry* dir = dentry_create(get_fname(path), FT_DIR);
  struct fs_inode* inode = inode_create();
  inode->ino = bitmap_alloc(super.imap, super.params.max_ino);
  dentry_bind(dir, inode);

  inode_alloc(parent->self);
  dentry_register(dir, parent);
  parent->self->dir_cnt++;

  return ERROR_NONE;
}
```

创建文件功能对应的 hook 函数为 `fs_mknod`，其实现思路与创建文件夹类似，具体实现如下：

```
int fs_mknod(const char* path, mode_t mode, dev_t dev) {

  if (S_ISDIR(mode)) {
    return fs_mkdir(path, mode);
  }

  struct fs_dentry* parent;
  if (dentry_lookup(path, &parent) == 0) {
    return ERROR_EXISTS;
  }
  if (parent->ftype != FT_DIR) {
    return ERROR_NOTFOUND;
  }
  struct fs_dentry* new_file = dentry_create(get_fname(path), FT_REG);
  struct fs_inode* inode = inode_create();
  inode->ino = bitmap_alloc(super.imap, super.params.max_ino);
  dentry_bind(new_file, inode);

  inode_alloc(parent->self);
  dentry_register(new_file, parent);

  parent->self->dir_cnt++;
  return ERROR_NONE;
}
```

> ⚠ **Warning**
>
> - 创建文件时不需要 malloc，所有数据都直接与磁盘进行交互
> - 创建文件时不需要分配数据块，写入数据时会进行动态分配

3.4 查看文件夹内容

查看文件夹内容的功能对应的 hook 函数为 `fs_readdir`，具体实现如下：

```
int fs_readdir(const char * path, void * buf, fuse_fill_dir_t filler, off_t offset,
               struct fuse_file_info * fi)
{
  struct fs_dentry* dentry;
  if (dentry_lookup(path, &dentry) != 0) {
    return ERROR_NOTFOUND;
  }
```

```
  struct fs_dentry* dentrys = dentry->self->childs;
  struct fs_dentry* cur = dentry_get(dentrys, offset);
  if (cur == NULL) {
    return ERROR_NOTFOUND;
  }
  filler(buf, cur->name, NULL, offset + 1);
  return ERROR_NONE;
}
```

以上为本实验必做部分的实现，附加功能在实验特色中进行介绍。

4 实验特色

实验中你认为自己实现的比较有特色的部分，包括设计思路、实现方法和预期效果。

本系统的特色在于一定程度的模块化，在设计上适当采用了 OOP 思想。与 Simplefs 中笼统将所有操作都放在 sfs.utils 中不同，本系统以各操作的对象为主体，在其上抽象出了一系列通用操作，使得代码更加清晰、简洁、易扩展。

下面以实验附加题的实现为例进行说明：

4.1 预备工作

按指导书要求，我们首先完成 fs_open, fs_opendir, fs_access 和 fs_truncate 四个函数，参考 Simplefs 中的实现即可。

```
int fs_open(const char* path, struct fuse_file_info* fi) {
  return ERROR_NONE;
}

int fs_opendir(const char* path, struct fuse_file_info* fi) {
  return ERROR_NONE;
}

int fs_access(const char* path, int type) {
  struct fs_dentry* dentry;
  int fail = 0;
  switch (type)
  {
    case R_OK:
    case W_OK:
    case X_OK:
      fail = 0; break;
    case F_OK:
      fail = dentry_lookup(path, &dentry); break;
  }

  return fail ? ERROR_ACCESS : ERROR_NONE;
}

int fs_truncate(const char* path, off_t offset) {
  struct fs_dentry* file;
  if (dentry_lookup(path, &file) != 0) {
    return ERROR_NOTFOUND;
  }
  if (file->ftype != FT_REG) {
    return ERROR_ISDIR;
```

```
  }
  struct fs_inode* inode = file->self;
  inode->size = offset;
  return ERROR_NONE;
}
```

4.2 读写文件

读写文件对应的 hook 函数为 `fs_read` 和 `fs_write`，调用 `file_read` 和 `file_write` 与磁盘进行直接交互即可，具体实现如下：

```
int fs_read(const char* path, char* buf, size_t size, off_t offset,
            struct fuse_file_info* fi) {
  struct fs_dentry* file;
  if (dentry_lookup(path, &file) != 0) {
    return ERROR_NOTFOUND;
  }
  if (file->ftype != FT_REG) {
    return ERROR_ISDIR;
  }
  struct fs_inode* inode = file->self;
  file_read(inode, offset, buf, size);
  return size;
}

int fs_write(const char* path, const char* buf, size_t size, off_t offset,
             struct fuse_file_info* fi) {
  struct fs_dentry* file;
  if (dentry_lookup(path, &file) != 0) {
    return ERROR_NOTFOUND;
  }
  if (file->ftype != FT_REG) {
    return ERROR_ISDIR;
  }
  struct fs_inode* inode = file->self;

  if (inode->size < offset) {
    return ERROR_SEEK;
  }

  file_write(inode, offset, buf, size);

  inode->size = offset + size > inode->size ? offset + size : inode->size;
  return size;
}
```

4.3 删除文件/目录

删除文件对应的 hook 函数为 `fs_unlink`，调用 `dentry_delete` 即可具体实现如下：

```
int fs_unlink(const char* path) {
  struct fs_dentry* file;
  if (dentry_lookup(path, &file) != 0) {
    return ERROR_NOTFOUND;
  }
  dentry_delete(file);

  return ERROR_NONE;
}
```

4.4 MV 命令

MV 命令对应的 hook 函数为 fs_rename，实现思路如下：

- 通过 dentry_unregister 从父目录中移除
- 通过 dentry_register 将其注册到新的父目录中
- 修改 dentry 中的 name 字段

具体实现如下：

```
int fs_rename(const char* from, const char* to) {
  struct fs_dentry* from_file;
  if (dentry_lookup(from, &from_file) != 0) {
    return ERROR_NOTFOUND;
  }

  if (strcmp(from, to) == 0) {
    return ERROR_NONE;
  }

  struct fs_dentry* parent;
  if (dentry_lookup(to, &parent) == 0) {
    return ERROR_EXISTS;
  }

  dentry_unregister(from_file);

  inode_alloc(parent->self);
  dentry_register(from_file, parent);

  parent->self->dir_cnt++;

  char* fname = get_fname(to);

  memcpy(from_file->name, fname, strlen(fname) + 1);

  return ERROR_NONE;
}
```

相对于指导书中给出的实现，该实现更加符合直觉，易于理解。

由上述实现可以看出，对于附加功能的实现都可按直觉进行，代码简洁且易扩展。

二、 遇到的问题与解决方法

列出实验过程中遇到的主要问题，包括技术难题、设计挑战等。对应每个问题，提供采取的解决方案，以及解决问题后的效果评估。

1 数据结构设计

本实验主要涉及的数据结构为 super_block, inode 和 dentry 在内存和磁盘中的表示。

1.1 内存中的数据结构

由于进程的堆栈空间有限，因此内存中的数据结构应当在支撑功能的前提下尽量简洁。具体来说，super_block 和 inode 的内存表示应当尽量简洁，以减少内存占用。dentry 的内存表示则相对复杂，因为需要维护目录树的结构。

经过慎重考虑，本系统的数据结构在参考 Simplefs 的基础上，移除了文件数据在内存中的表示，转而在需要时通过 read 和 write 操作从磁盘读取和写入文件数据，该设计基于以下考虑：

- 优化内存使用：文件数据在内存中的表示会占用大量内存，而实际上大部分时间文件数据并不需要被加载到内存中。
- 延迟分配：文件数据在需要时才被加载到内存中，可以避免在文件创建时立即分配大量内存。
- 责任转移：将文件读写的优化交由相应的系统调用实现，如使用 mmap 将文件数据映射到内存中。

1.2 磁盘中的数据结构

对于磁盘的数据结构，本系统主要考虑以下因素：

- 空间效率：数据结构应当尽量紧凑，以减少磁盘空间的占用。
- 时间效率：数据结构应当尽可能减少磁盘 I/O 的开销。

经过慎重考虑，本系统通过维护磁盘的 Inode Bitmap 和 Data Bitmap 来管理磁盘空间，通过位图上的标记以及 inode 和 dentry 来限制对于磁盘空间的访问，从而实现空间和时间的优化。

2 接口设计

为提高代码的可读性和可复用性，需要对接口进行合理设计。具体来说，需要考虑以下因素：

- 接口的粒度：接口应当足够细，以便于复用和扩展。
- 接口的命名：接口的命名应当清晰、简洁，以便于理解和使用。
- 接口的参数：接口的参数应当尽量简洁，以便于调用。

经过精心设计，本系统的接口命名采取 object_operation 的形式，其中 object 表示操作的对象，operation 表示操作的具体行为。

> 💡 **Design Choice**
>
> 该设计本质上是在 C 语言中模拟方法调用，通过将第一个参数设置为对象，将后续参数设置为方法参数，从而实现方法的调用，即将 object.method(param1, ...) 转化为 object_operation(object, method, param1, ...) 的形式。

对于重复出现的逻辑，本系统尽可能将其提取为函数，以减少代码重复。如 Bitmap::alloc 函数封装了 Bitmap::find_first_zero 和 Bitmap::set 的逻辑，从而简化了磁盘空间分配的代码。

Example

在 Simplefs 的 sfs_utils.c 中，可以看到 sfs_alloc_inode 与 sfs_drop_inode 中均出现了以下形式的代码：

```
for (byte_cursor = 0; byte_cursor < SFS_BLKS_SZ(sfs_super.map_inode_blks);
     byte_cursor++)
{
    for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
        if((sfs_super.map_inode[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                    /* 当前 ino_cursor 位置空闲 */
            sfs_super.map_inode[byte_cursor] |= (0x1 << bit_cursor);
            is_find_free_entry = TRUE;
            break;
        }
        ino_cursor++;
    }
    if (is_find_free_entry) {
        break;
    }
}
```

从实际出发，该函数不应关注 Bitmap 的分配逻辑。举例来说，Bitmap 即可通过找到第一个 0，再将其置 1 来实现分配，也可以通过在其内部维护一个指针，每次分配时将指针后移一位来实现。

更进一步说，在本次实验中含有 Inode Bitmap 和 Data Bitmap，如果每次分配都需要 hand code 一次，那么代码的维护成本将会非常高。因此，本系统通过封装 Bitmap::alloc 函数，将 Bitmap 的分配逻辑抽象为函数，从而简化了代码的编写和维护。

3 效果评估

经过以上设计，本系统在扩展功能时，可以更加方便地添加新的接口，同时也可以更好地维护代码的可读性和可复用性。

仍以 Bitmap 为例，通过封装 Bitmap::alloc 函数，本系统在扩展功能过程中：

- 若需要申请新的数据块，只需调用 Bitmap::alloc 函数即可，无需关心 Bitmap 的分配逻辑
- 若需要修改 Bitmap 的磁盘空间占用，无需在功能代码中修改，只需修改 Bitmap::alloc 函数即可

三、 实验收获和建议

实验中的收获、感受、问题、建议等。

1 收获与感受

本次实验属于一个设计性的实验，通过本次实验，我收获了很多宝贵的经验：

- 模块接口设计：在本次实验中，我慎重考虑了每个模块的接口设计，以确保模块之间的交互尽可能简单和清晰，尽量将模块之间的依赖关系最小化，以减少模块之间的耦合度。
- 信息存储设计：在本次实验中，我深入思考了如何什么信息是需要存储的，以及如何存储这些信息。具体来说，我学会了如何将信息进行结构化存储，以便于后续的查询和修改。
- 复杂度管理：在本次实验中，我学会了如何管理复杂度。随着实验的进行，工程中的代码结构越来越复杂，我学会了如何将复杂度分解为更小的部分，以便于管理和维护。

2 意见与建议

本实验改编自往年学长学姐们参加比赛的作品，从中可以看到很大的设计空间，就取材而言是一个很好的系统设计实验。

可惜的是，本次实验的参考代码和指导书均未突出这一点。

就 Simplefs 的代码组织来讲

- 其将所有函数都放在 `sfs_utils` 文件中，没有进行任何组织，这导致代码十分难以理解。
- 其大量地使用的宏定义，这导致代码难以阅读和调试。
- 其函数的命名较为混乱，对于实体的创建，既有如 `new_dentry` 的命名，又有 `alloc_inode` 的命名。有趣的是，`alloc_dentry` 的功能并非创建一个目录项，而是将目录项注册到父目录中。

就指导书来讲

- 任务一作用有限。任务一在任务二中没有起到任何实质性的作用。其目的完全可以通过在 `fs_init` 打印 “hello fs”，在 `fs_destroy` 打印 “goodbye fs” 来达到。既能让同学们看到 hook 函数的作用，又和任务二紧密相联。Anyway，当前指导书的任务一体现不出什么价值。
- 内容重复。实验实现与选择阅读（补充） 以及 Simplefs 的源码注释均为相同的内容。选择阅读（补充）的本意应该是将源码进行串接，但在指导书上更像是一个网页版的注释。
- 避重就轻：本次实验的磁盘结构最大的特点就是数据块不需要连续分配，可以按需索引，但实验的实现和测评均未对该部分进行侧重。具体来讲，在实现过程中不需要读写文件，那么就无需分配任何内存给文件，经实际验证也可通过测试，Ridiculous!

总的来说，我认为本次实验还是很有价值的，但指导书和参考代码还需要改进。

> 💡 **Idea**
>
> 或许可以考虑将 Simplefs 的代码进行重构，将 hook 中的操作抽象为函数，让同学们修改具体的函数实现来完成对磁盘结构的修改。

四、 参考资料

实验过程中查找的信息和资料

[1] lab5: 基于 FUSE 的青春版 EXT2 文件系统 https://os-labs.pages.dev/lab5/part1/

[2] D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

[3] Simplefs 文件系统 https://gitee.com/ftutorials/user-land-filesystem/tree/main/fs/simplefs

> ⚠ **Warning**
>
> Don’t Blame Me, 没参考什么文献，格式较为随意