Project 6 File System 设计文档

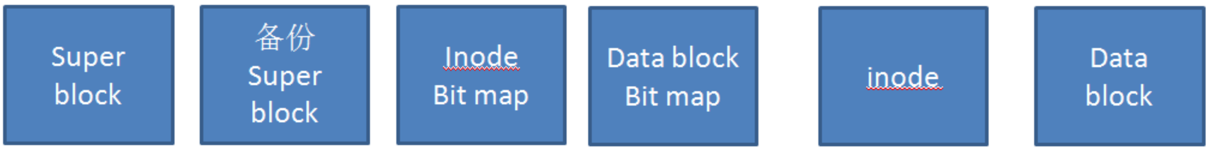
中国科学院大学
[王苑铮]

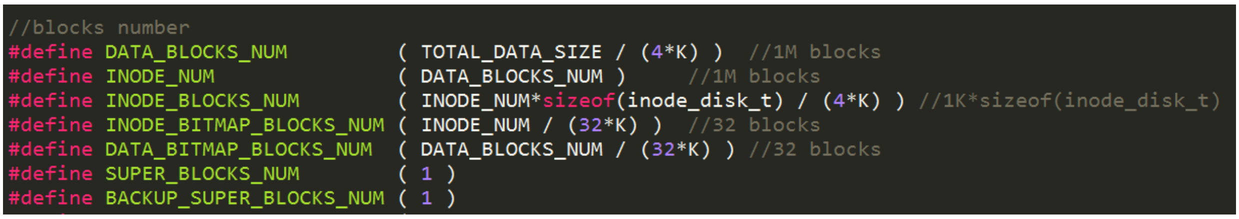
[2018.1.20]

1. 文件系统初始化设计

请至少包含以下内容

（1） 请用图表示你设计的文件系统对磁盘的布局（布局上可以不考虑 boot block 的大小，直接从逻辑地址第 0 块开始），并说明各部分占用的磁盘空间大小，例如 superblock，inode 的元数据等





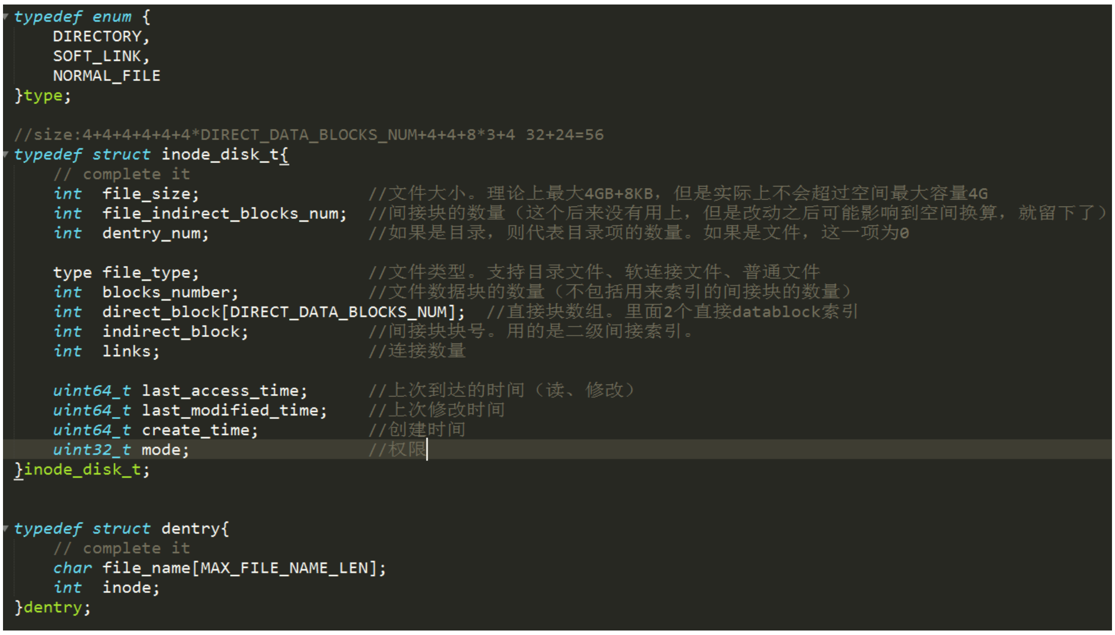
```
//blocks number
#define DATA_BLOCKS_NUM         ( TOTAL_DATA_SIZE / (4*K) )  //1M blocks
#define INODE_NUM               ( DATA_BLOCKS_NUM )      //1M blocks
#define INODE_BLOCKS_NUM        ( INODE_NUM*sizeof(inode_disk_t) / (4*K) ) //1K*sizeof(inode_disk_t)
#define INODE_BITMAP_BLOCKS_NUM ( INODE_NUM / (32*K) )  //32 blocks
#define DATA_BITMAP_BLOCKS_NUM  ( DATA_BLOCKS_NUM / (32*K) ) //32 blocks
#define SUPER_BLOCKS_NUM        ( 1 )
#define BACKUP_SUPER_BLOCKS_NUM ( 1 )
```

布局，以及每部分的大小和换算关系。（单位为块数。4KB 每块）。TOTAL_DATA_SIZE 为最大数据空间，此处为 4G。

（2） 你如何实现 superblock 的备份？如何判断 superblock 损坏，以及当有一个 superblock 损坏时你的文件系统如何正常启动？

备份：另有一个备份的 superblock 块每次更新 superblock 时也一起更新备份的 backup_superblock。判断 superblock 损坏：读出 superblock 后里面的 magic_number 和我定义的 magic_number 不同。当有一个 superblock 损坏时，先读出备份的 backup_ssuperblock，如果备份块里的 magic_number 正确，则认为只是 superblock 坏了，把备份块写给 superblock 就可以了。如果备份块里面的 magicnumber 也不对，则认为是整个文件系统都损坏了，或者新的磁盘尚未初始化文件系统，此时就要 mkfs()，初始化 superblock，inode 数组，bitmap，根目录等数据结构，写进磁盘里。之后 mount()即可启动。

（3） 请列出你设计的 superblock 和 inode 数据结构，并阐明各项含义。请说明你设计的文件系统能支持的最大文件大小，最多文件数目，以及单个目录下能支持的最多文件/子目录数目。



```
typedef enum {
    DIRECTORY,
    SOFT_LINK,
    NORMAL_FILE
}type;

//size:4+4+4+4+4+4*DIRECT_DATA_BLOCKS_NUM+4+4+8*3+4 32+24=56
typedef struct inode_disk_t{
    // complete it
    int  file_size;                  //文件大小。理论上最大4GB+8KB，但是实际上不会超过空间最大容量4G
    int  file_indirect_blocks_num;   //间接块的数量（这个后来没有用上，但是改动之后可能影响到空间换算，就留下了）
    int  dentry_num;                 //如果是目录，则代表目录项的数量。如果是文件，这一项为0

    type file_type;                  //文件类型。支持目录文件、软连接文件、普通文件
    int  blocks_number;              //文件数据块的数量（不包括用来索引的间接块的数量）
    int  direct_block[DIRECT_DATA_BLOCKS_NUM];  //直接块数组。里面2个直接datablock索引
    int  indirect_block;             //间接块块号。用的是二级间接索引。
    int  links;                      //连接数量

    uint64_t last_access_time;       //上次到达的时间（读、修改）
    uint64_t last_modified_time;     //上次修改时间
    uint64_t create_time;            //创建时间
    uint32_t mode;                   //权限
}inode_disk_t;


typedef struct dentry{
    // complete it
    char file_name[MAX_FILE_NAME_LEN];
    int  inode;
}dentry;
```

最多文件数目=datablock 数目=4G/4K=1M 个文件

限定一个目录只能使用一个块。目录项 dentry 中，单个文件名字最长不能超过 60char，加上一个 int 索引 inode 好，一个目录项为 64B。1 块 4K，可以放 64 放文件/子目录。

（4） 请说明你设计的文件系统的块分配策略，按需分配还是有设计其他分配策略？

按需分配，分配的块数为 ceil(文件大小/4K)个块。

（5） 如果完成了 bonus，请通过举例说明你设计的面向 SSD 的文件系统采用异地更新模式后文件数据块在磁盘上的布局，例如新写一个文件后和更新同一个文件后，该文件数据块在磁盘上的布局

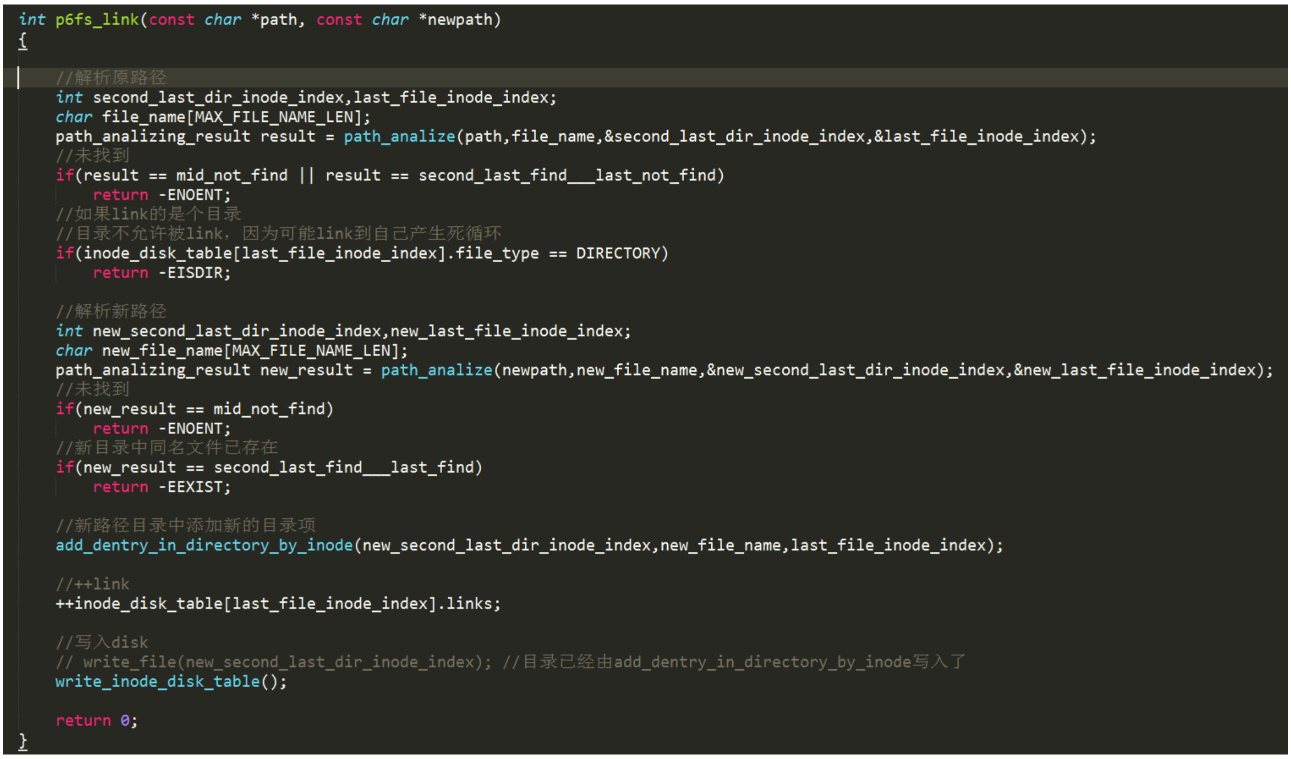
没做

（6）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

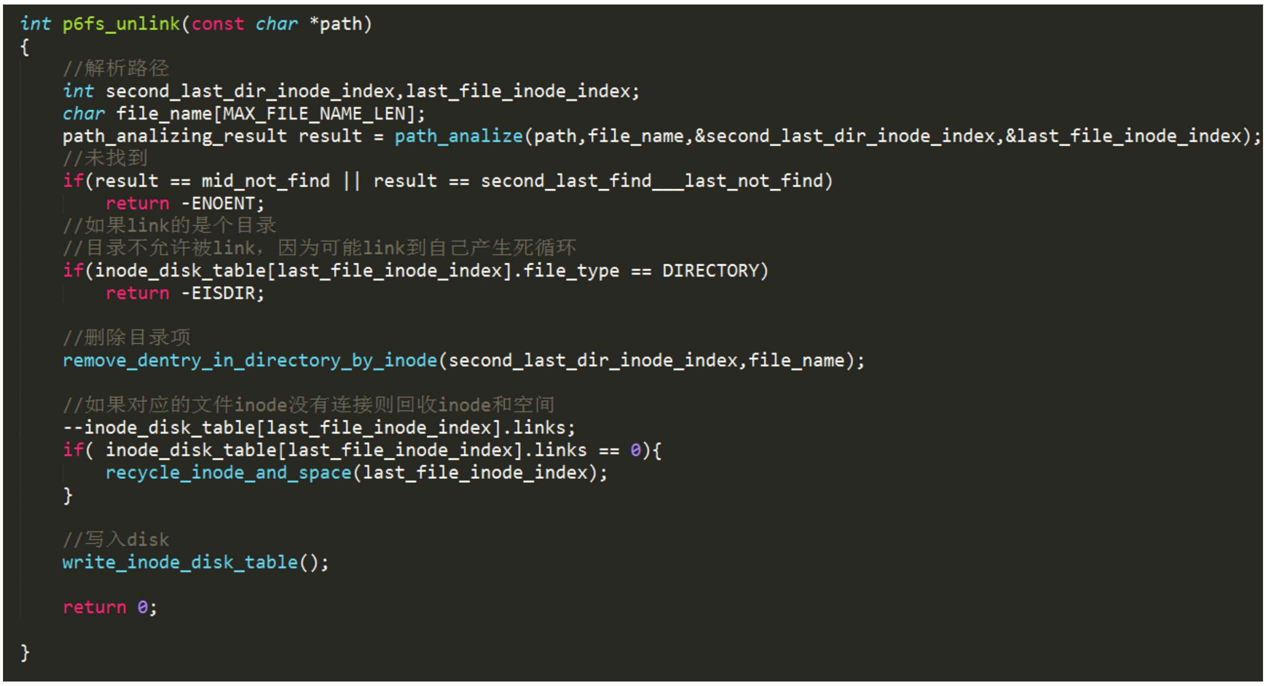
2. 文件操作设计

请至少包含以下内容

（1） 请说明 link 和 unlink 的操作流程



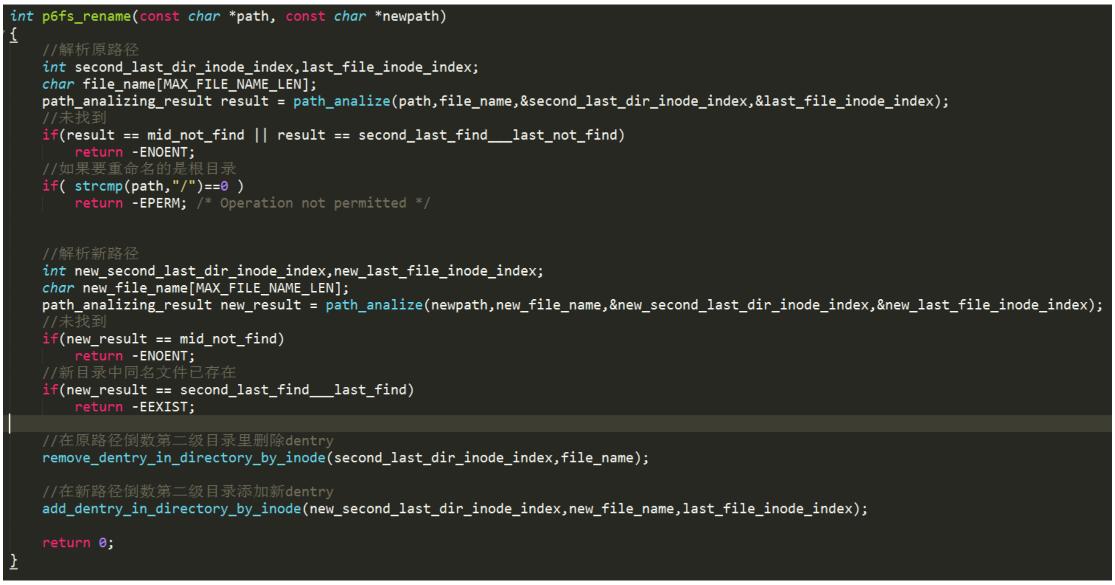
```c
int p6fs_link(const char *path, const char *newpath)
{

    //解析原路径
    int second_last_dir_inode_index,last_file_inode_index;
    char file_name[MAX_FILE_NAME_LEN];
    path_analizing_result result = path_analize(path,file_name,&second_last_dir_inode_index,&last_file_inode_index);
    //未找到
    if(result == mid_not_find || result == second_last_find___last_not_find)
        return -ENOENT;
    //如果link的是个目录
    //目录不允许被link, 因为可能link到自己产生死循环
    if(inode_disk_table[last_file_inode_index].file_type == DIRECTORY)
        return -EISDIR;

    //解析新路径
    int new_second_last_dir_inode_index,new_last_file_inode_index;
    char new_file_name[MAX_FILE_NAME_LEN];
    path_analizing_result new_result = path_analize(newpath,new_file_name,&new_second_last_dir_inode_index,&new_last_file_inode_index);
    //未找到
    if(new_result == mid_not_find)
        return -ENOENT;
    //新目录中同名文件已存在
    if(new_result == second_last_find___last_find)
        return -EEXIST;

    //新路径目录中添加新的目录项
    add_dentry_in_directory_by_inode(new_second_last_dir_inode_index,new_file_name,last_file_inode_index);

    //++link
    ++inode_disk_table[last_file_inode_index].links;

    //写入disk
    // write_file(new_second_last_dir_inode_index); //目录已经由add_dentry_in_directory_by_inode写入了
    write_inode_disk_table();

    return 0;
}
```



```c
int p6fs_unlink(const char *path)
{
    //解析路径
    int second_last_dir_inode_index,last_file_inode_index;
    char file_name[MAX_FILE_NAME_LEN];
    path_analizing_result result = path_analize(path,file_name,&second_last_dir_inode_index,&last_file_inode_index);
    //未找到
    if(result == mid_not_find || result == second_last_find___last_not_find)
        return -ENOENT;
    //如果link的是个目录
    //目录不允许被link, 因为可能link到自己产生死循环
    if(inode_disk_table[last_file_inode_index].file_type == DIRECTORY)
        return -EISDIR;

    //删除目录项
    remove_dentry_in_directory_by_inode(second_last_dir_inode_index,file_name);

    //如果对应的文件inode没有连接则回收inode和空间
    --inode_disk_table[last_file_inode_index].links;
    if( inode_disk_table[last_file_inode_index].links == 0){
        recycle_inode_and_space(last_file_inode_index);
    }

    //写入disk
    write_inode_disk_table();

    return 0;

}
```

见上面代码的注释

（2） 请说明 rename 涉及的操作流程



```c
int p6fs_rename(const char *path, const char *newpath)
{
    //解析原路径
    int second_last_dir_inode_index,last_file_inode_index;
    char file_name[MAX_FILE_NAME_LEN];
    path_analizing_result result = path_analize(path,file_name,&second_last_dir_inode_index,&last_file_inode_index);
    //未找到
    if(result == mid_not_find || result == second_last_find__last_not_find)
        return -ENOENT;
    //如果要重命名的是根目录
    if( strcmp(path,"/")==0 )
        return -EPERM; /* Operation not permitted */


    //解析新路径
    int new_second_last_dir_inode_index,new_last_file_inode_index;
    char new_file_name[MAX_FILE_NAME_LEN];
    path_analizing_result new_result = path_analize(newpath,new_file_name,&new_second_last_dir_inode_index,&new_last_file_inode_index);
    //未找到
    if(new_result == mid_not_find)
        return -ENOENT;
    //新目录中同名文件已存在
    if(new_result == second_last_find__last_find)
        return -EEXIST;

    //在原路径倒数第二级目录里删除dentry
    remove_dentry_in_directory_by_inode(second_last_dir_inode_index,file_name);

    //在新路径倒数第二级目录添加新dentry
    add_dentry_in_directory_by_inode(new_second_last_dir_inode_index,new_file_name,last_file_inode_index);

    return 0;
}
```

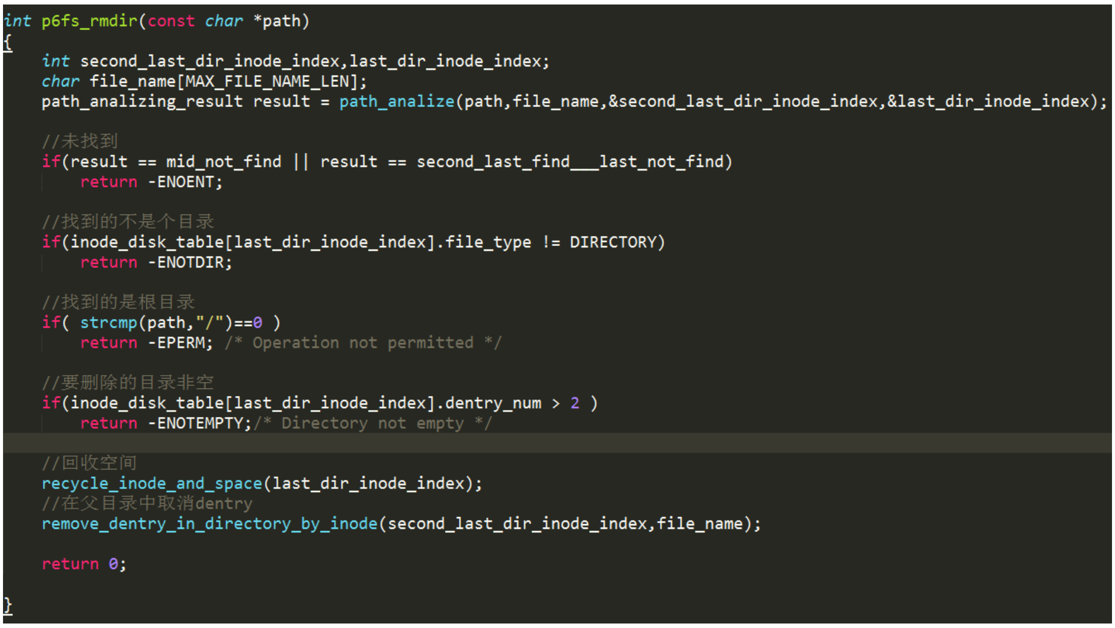
见以上代码的注释

（3）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

3. 目录操作设计

请至少包含以下内容

（1） 请说明 rmdir 的操作流程？



```c
int p6fs_rmdir(const char *path)
{
    int second_last_dir_inode_index,last_dir_inode_index;
    char file_name[MAX_FILE_NAME_LEN];
    path_analizing_result result = path_analize(path,file_name,&second_last_dir_inode_index,&last_dir_inode_index);

    //未找到
    if(result == mid_not_find || result == second_last_find__last_not_find)
        return -ENOENT;

    //找到的不是个目录
    if(inode_disk_table[last_dir_inode_index].file_type != DIRECTORY)
        return -ENOTDIR;

    //找到的是根目录
    if( strcmp(path,"/")==0 )
        return -EPERM; /* Operation not permitted */

    //要删除的目录非空
    if(inode_disk_table[last_dir_inode_index].dentry_num > 2 )
        return -ENOTEMPTY;/* Directory not empty */

    //回收空间
    recycle_inode_and_space(last_dir_inode_index);
    //在父目录中取消dentry
    remove_dentry_in_directory_by_inode(second_last_dir_inode_index,file_name);

    return 0;

}
```

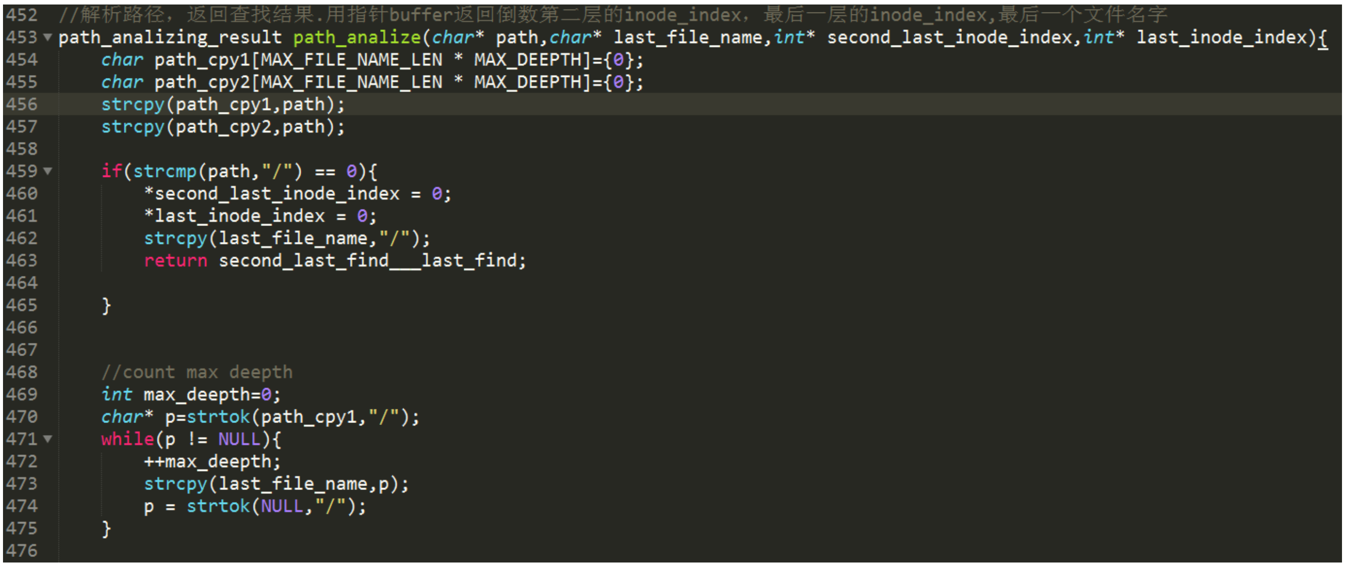
见以上代码的注释

（2）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

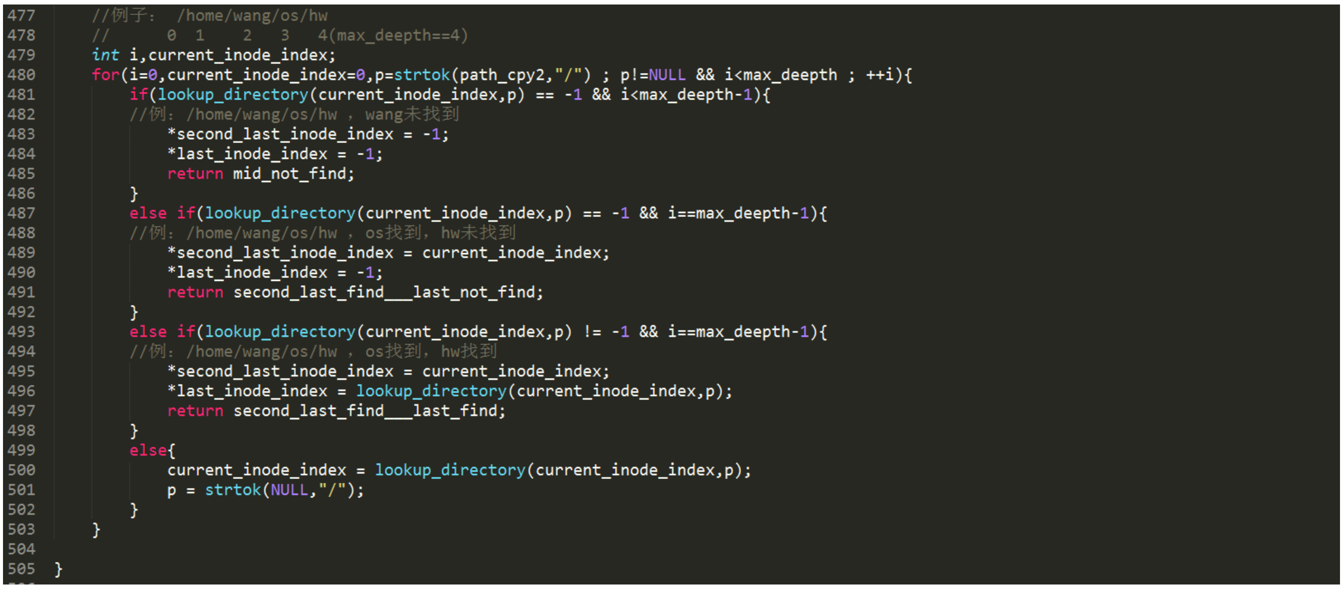
4. 关键函数功能

请列出上述各项功能设计里，你觉得关键的函数或代码块，及其作用

1.绝对路径解析函数。

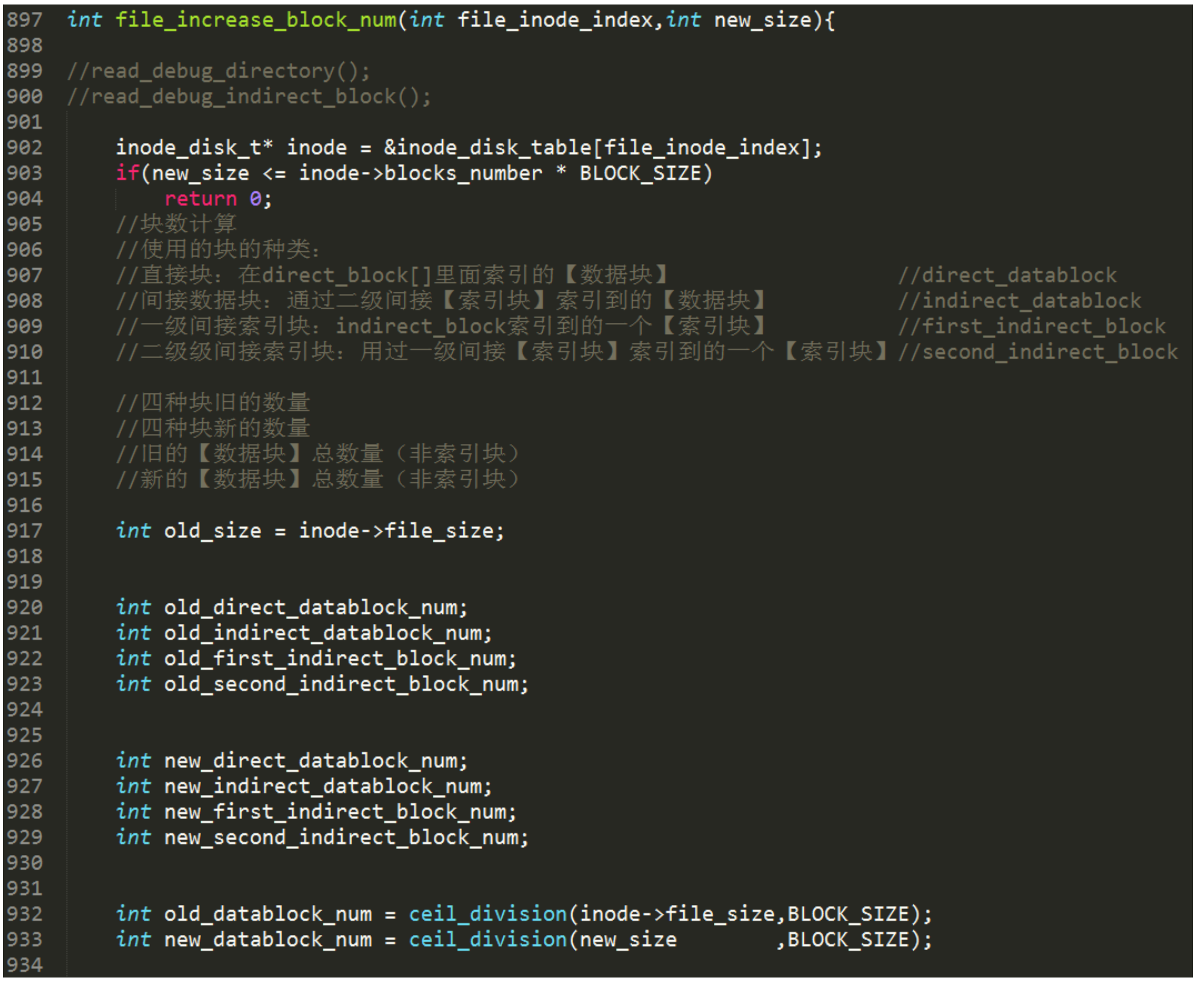


```c
//解析路径，返回查找结果.用指针buffer返回倒数第二层的inode_index，最后一层的inode_index,最后一个文件名字
path_analizing_result path_analize(char* path,char* last_file_name,int* second_last_inode_index,int* last_inode_index){
    char path_cpy1[MAX_FILE_NAME_LEN * MAX_DEEPTH]={0};
    char path_cpy2[MAX_FILE_NAME_LEN * MAX_DEEPTH]={0};
    strcpy(path_cpy1,path);
    strcpy(path_cpy2,path);

    if(strcmp(path,"/") == 0){
        *second_last_inode_index = 0;
        *last_inode_index = 0;
        strcpy(last_file_name,"/");
        return second_last_find___last_find;

    }


    //count max deepth
    int max_deepth=0;
    char* p=strtok(path_cpy1,"/");
    while(p != NULL){
        ++max_deepth;
        strcpy(last_file_name,p);
        p = strtok(NULL,"/");
    }

```

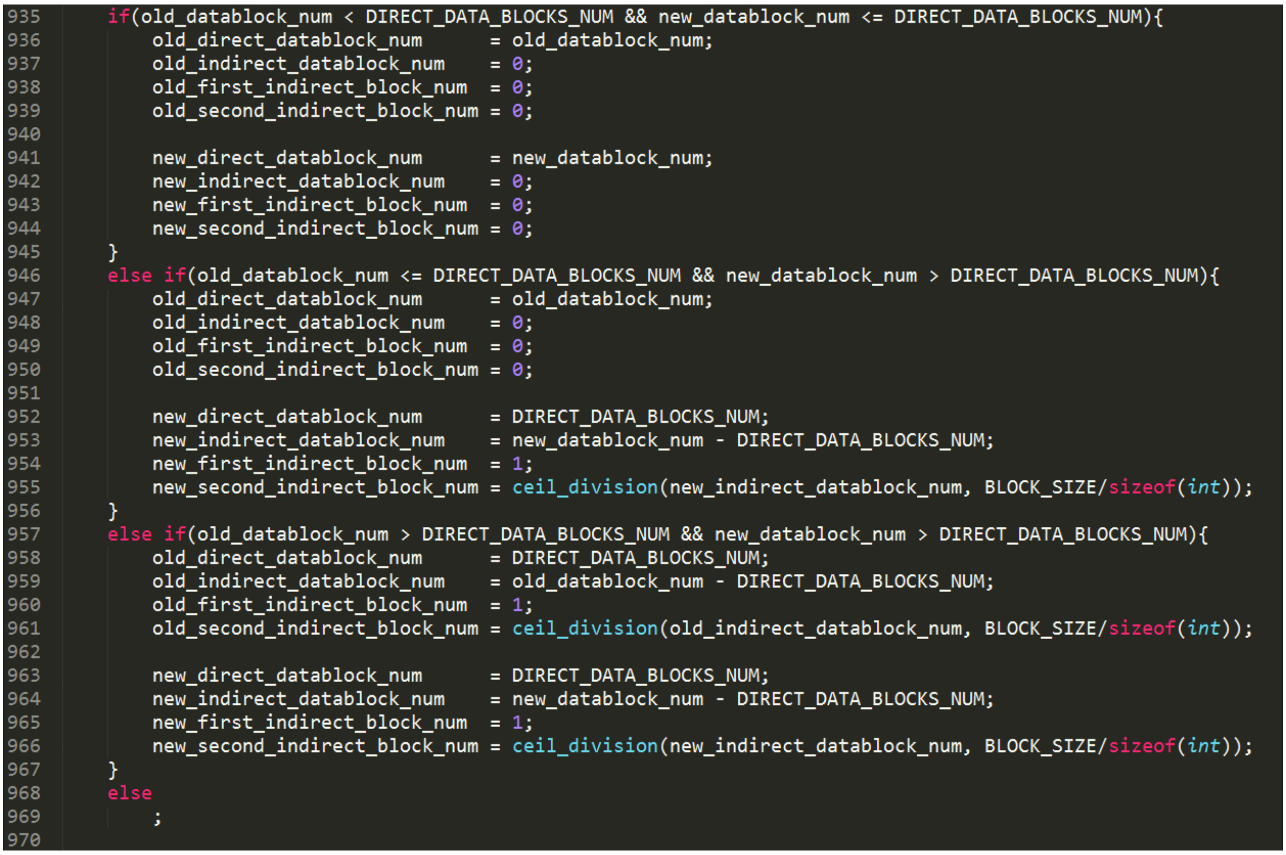


```c
    //例子：  /home/wang/os/hw
    //       0  1    2   3   4(max_deepth==4)
    int i,current_inode_index;
    for(i=0,current_inode_index=0,p=strtok(path_cpy2,"/") ; p!=NULL && i<max_deepth ; ++i){
        if(lookup_directory(current_inode_index,p) == -1 && i<max_deepth-1){
        //例：/home/wang/os/hw ， wang未找到
            *second_last_inode_index = -1;
            *last_inode_index = -1;
            return mid_not_find;
        }
        else if(lookup_directory(current_inode_index,p) == -1 && i==max_deepth-1){
        //例：/home/wang/os/hw ， os找到，hw未找到
            *second_last_inode_index = current_inode_index;
            *last_inode_index = -1;
            return second_last_find___last_not_find;
        }
        else if(lookup_directory(current_inode_index,p) != -1 && i==max_deepth-1){
        //例：/home/wang/os/hw ， os找到，hw找到
            *second_last_inode_index = current_inode_index;
            *last_inode_index = lookup_directory(current_inode_index,p);
            return second_last_find___last_find;
        }
        else{
            current_inode_index = lookup_directory(current_inode_index,p);
            p = strtok(NULL,"/");
        }
    }

}
```

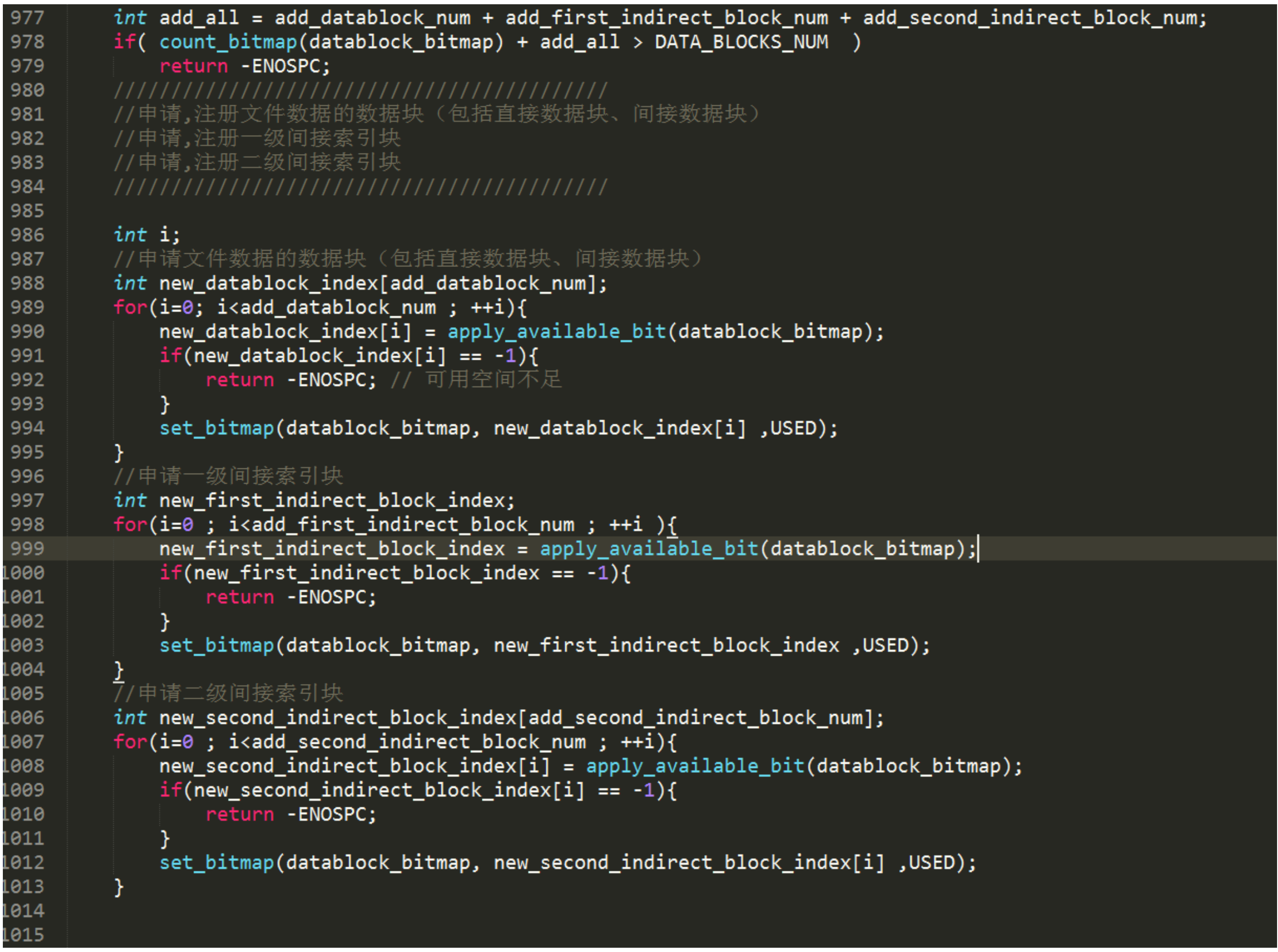
2.当文件需要的 size 大于文件当前的块数的 size 时，申请新的 datablock 对文件扩容



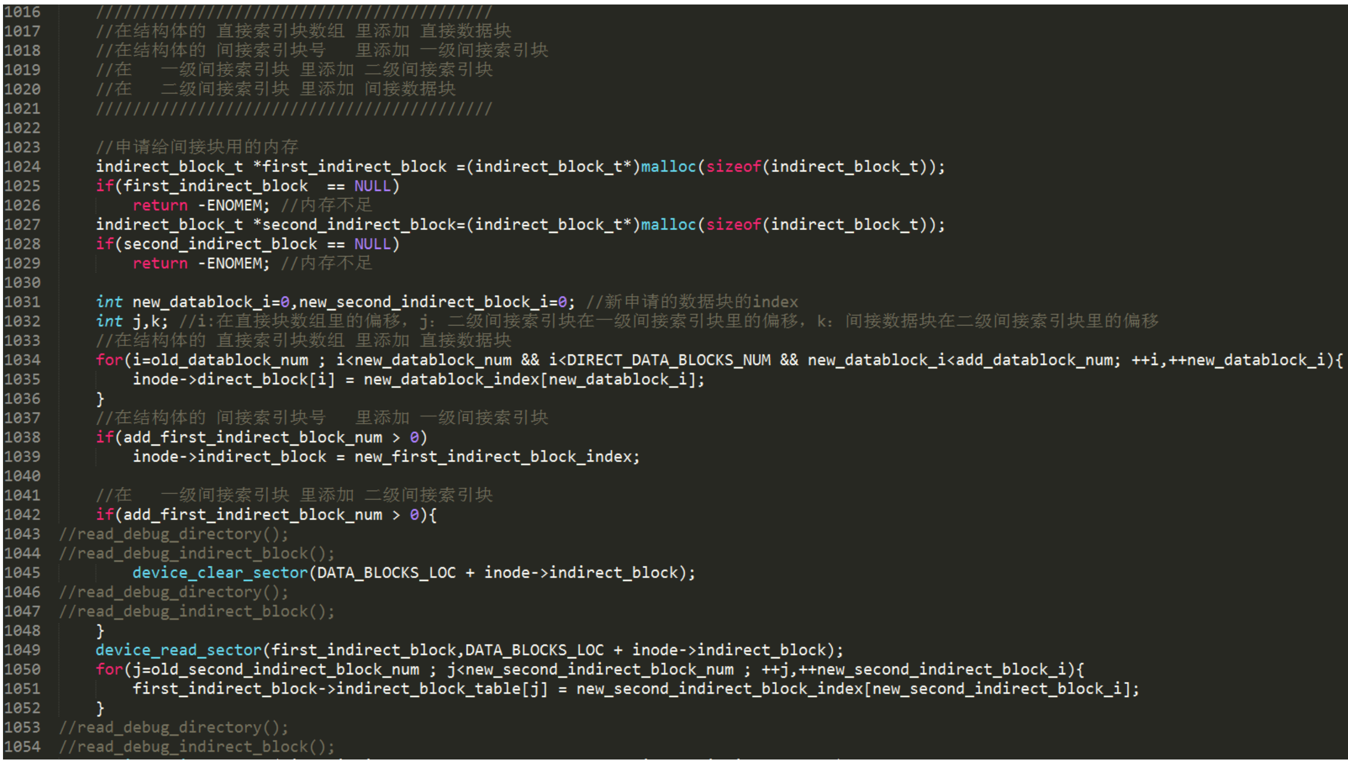
```c
int file_increase_block_num(int file_inode_index,int new_size){

//read_debug_directory();
//read_debug_indirect_block();

    inode_disk_t* inode = &inode_disk_table[file_inode_index];
    if(new_size <= inode->blocks_number * BLOCK_SIZE)
        return 0;
    //块数计算
    //使用的块的种类:
    //直接块: 在direct_block[]里面索引的【数据块】                   //direct_datablock
    //间接数据块: 通过二级间接【索引块】索引到的【数据块】           //indirect_datablock
    //一级间接索引块: indirect_block索引到的一个【索引块】           //first_indirect_block
    //二级级间接索引块: 用过一级间接【索引块】索引到的一个【索引块】 //second_indirect_block

    //四种块旧的数量
    //四种块新的数量
    //旧的【数据块】总数量（非索引块）
    //新的【数据块】总数量（非索引块）

    int old_size = inode->file_size;


    int old_direct_datablock_num;
    int old_indirect_datablock_num;
    int old_first_indirect_block_num;
    int old_second_indirect_block_num;


    int new_direct_datablock_num;
    int new_indirect_datablock_num;
    int new_first_indirect_block_num;
    int new_second_indirect_block_num;


    int old_datablock_num = ceil_division(inode->file_size,BLOCK_SIZE);
    int new_datablock_num = ceil_division(new_size        ,BLOCK_SIZE);

```



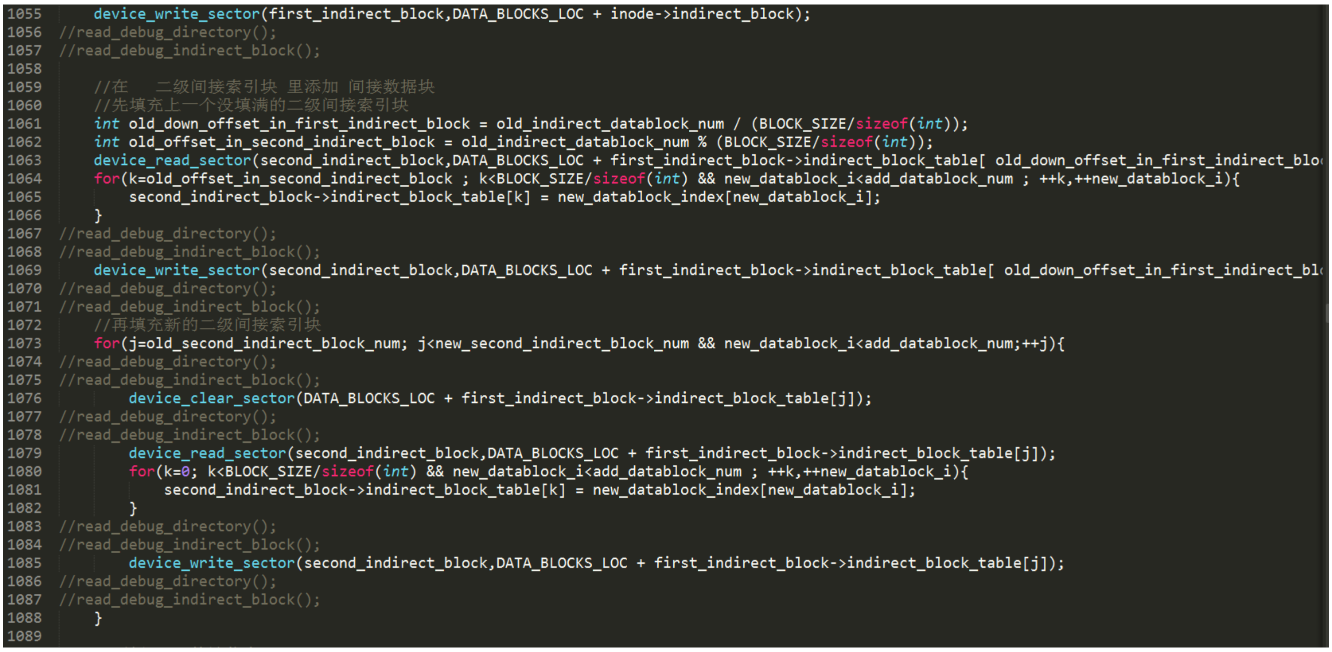
```c
    if(old_datablock_num < DIRECT_DATA_BLOCKS_NUM && new_datablock_num <= DIRECT_DATA_BLOCKS_NUM){
        old_direct_datablock_num      = old_datablock_num;
        old_indirect_datablock_num    = 0;
        old_first_indirect_block_num  = 0;
        old_second_indirect_block_num = 0;

        new_direct_datablock_num      = new_datablock_num;
        new_indirect_datablock_num    = 0;
        new_first_indirect_block_num  = 0;
        new_second_indirect_block_num = 0;
    }
    else if(old_datablock_num <= DIRECT_DATA_BLOCKS_NUM && new_datablock_num > DIRECT_DATA_BLOCKS_NUM){
        old_direct_datablock_num      = old_datablock_num;
        old_indirect_datablock_num    = 0;
        old_first_indirect_block_num  = 0;
        old_second_indirect_block_num = 0;

        new_direct_datablock_num      = DIRECT_DATA_BLOCKS_NUM;
        new_indirect_datablock_num    = new_datablock_num - DIRECT_DATA_BLOCKS_NUM;
        new_first_indirect_block_num  = 1;
        new_second_indirect_block_num = ceil_division(new_indirect_datablock_num, BLOCK_SIZE/sizeof(int));
    }
    else if(old_datablock_num > DIRECT_DATA_BLOCKS_NUM && new_datablock_num > DIRECT_DATA_BLOCKS_NUM){
        old_direct_datablock_num      = DIRECT_DATA_BLOCKS_NUM;
        old_indirect_datablock_num    = old_datablock_num - DIRECT_DATA_BLOCKS_NUM;
        old_first_indirect_block_num  = 1;
        old_second_indirect_block_num = ceil_division(old_indirect_datablock_num, BLOCK_SIZE/sizeof(int));

        new_direct_datablock_num      = DIRECT_DATA_BLOCKS_NUM;
        new_indirect_datablock_num    = new_datablock_num - DIRECT_DATA_BLOCKS_NUM;
        new_first_indirect_block_num  = 1;
        new_second_indirect_block_num = ceil_division(new_indirect_datablock_num, BLOCK_SIZE/sizeof(int));
    }
    else
        ;

```



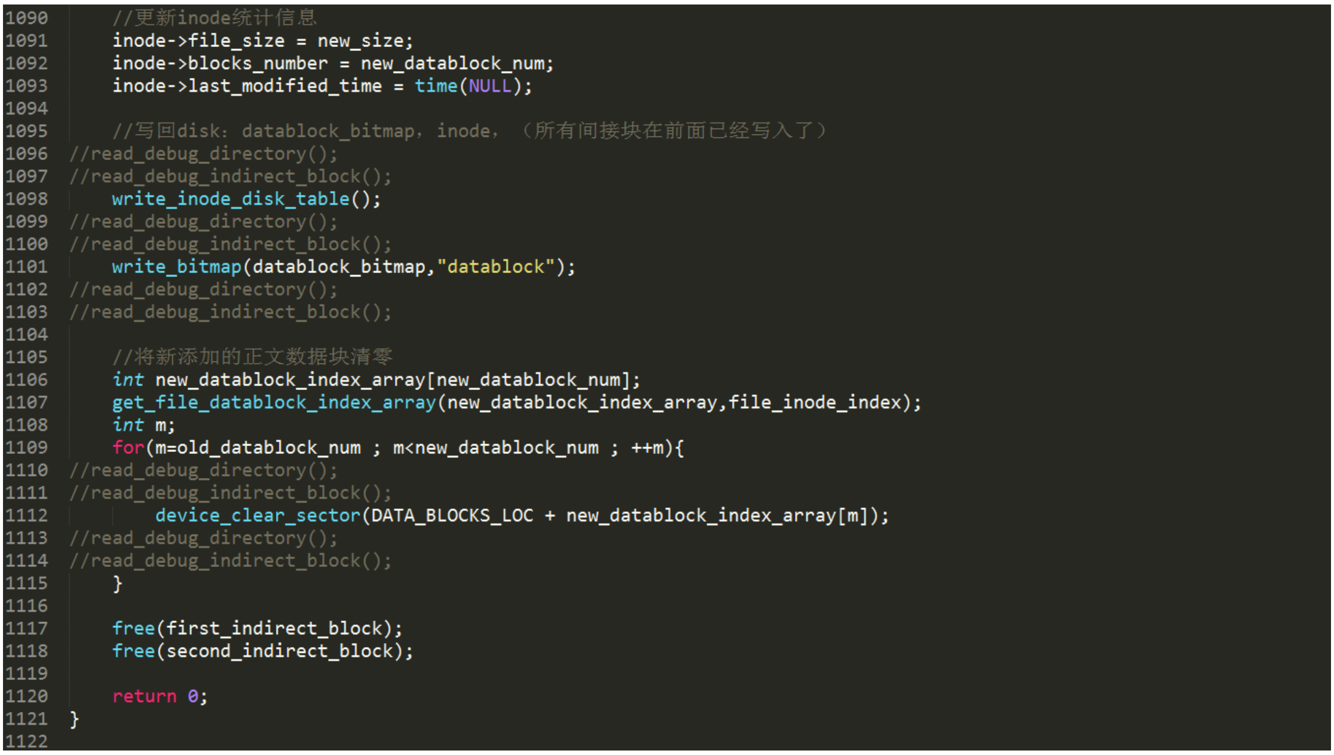
```
    int add_all = add_datablock_num + add_first_indirect_block_num + add_second_indirect_block_num;
    if( count_bitmap(datablock_bitmap) + add_all > DATA_BLOCKS_NUM   )
        return -ENOSPC;
    //////////////////////////////////////////
    //申请,注册文件数据的数据块（包括直接数据块、间接数据块）
    //申请,注册一级间接索引块
    //申请,注册二级间接索引块
    //////////////////////////////////////////

    int i;
    //申请文件数据的数据块（包括直接数据块、间接数据块）
    int new_datablock_index[add_datablock_num];
    for(i=0; i<add_datablock_num ; ++i){
        new_datablock_index[i] = apply_available_bit(datablock_bitmap);
        if(new_datablock_index[i] == -1){
            return -ENOSPC; // 可用空间不足
        }
        set_bitmap(datablock_bitmap, new_datablock_index[i] ,USED);
    }
    //申请一级间接索引块
    int new_first_indirect_block_index;
    for(i=0 ; i<add_first_indirect_block_num ; ++i ){
        new_first_indirect_block_index = apply_available_bit(datablock_bitmap);
        if(new_first_indirect_block_index == -1){
            return -ENOSPC;
        }
        set_bitmap(datablock_bitmap, new_first_indirect_block_index ,USED);
    }
    //申请二级间接索引块
    int new_second_indirect_block_index[add_second_indirect_block_num];
    for(i=0 ; i<add_second_indirect_block_num ; ++i){
        new_second_indirect_block_index[i] = apply_available_bit(datablock_bitmap);
        if(new_second_indirect_block_index[i] == -1){
            return -ENOSPC;
        }
        set_bitmap(datablock_bitmap, new_second_indirect_block_index[i] ,USED);
    }

```



```
    //////////////////////////////////////////
    //在结构体的 直接索引块数组 里添加 直接数据块
    //在结构体的 间接索引块号   里添加 一级间接索引块
    //在   一级间接索引块 里添加 二级间接索引块
    //在   二级间接索引块 里添加 间接数据块
    //////////////////////////////////////////

    //申请给间接块用的内存
    indirect_block_t *first_indirect_block =(indirect_block_t*)malloc(sizeof(indirect_block_t));
    if(first_indirect_block  == NULL)
        return -ENOMEM; //内存不足
    indirect_block_t *second_indirect_block=(indirect_block_t*)malloc(sizeof(indirect_block_t));
    if(second_indirect_block == NULL)
        return -ENOMEM; //内存不足

    int new_datablock_i=0,new_second_indirect_block_i=0; //新申请的数据块的index
    int j,k; //i:在直接块数组里的偏移, j: 二级间接索引块在一级间接索引块里的偏移, k: 间接数据块在二级间接索引块里的偏移
    //在结构体的 直接索引块数组 里添加 直接数据块
    for(i=old_datablock_num ; i<new_datablock_num && i<DIRECT_DATA_BLOCKS_NUM && new_datablock_i<add_datablock_num; ++i,++new_datablock_i){
        inode->direct_block[i] = new_datablock_index[new_datablock_i];
    }
    //在结构体的 间接索引块号   里添加 一级间接索引块
    if(add_first_indirect_block_num > 0)
        inode->indirect_block = new_first_indirect_block_index;

    //在   一级间接索引块 里添加 二级间接索引块
    if(add_first_indirect_block_num > 0){
//read_debug_directory();
//read_debug_indirect_block();
        device_clear_sector(DATA_BLOCKS_LOC + inode->indirect_block);
//read_debug_directory();
//read_debug_indirect_block();
    }
    device_read_sector(first_indirect_block,DATA_BLOCKS_LOC + inode->indirect_block);
    for(j=old_second_indirect_block_num ; j<new_second_indirect_block_num ; ++j,++new_second_indirect_block_i){
        first_indirect_block->indirect_block_table[j] = new_second_indirect_block_index[new_second_indirect_block_i];
    }
//read_debug_directory();
//read_debug_indirect_block();
```

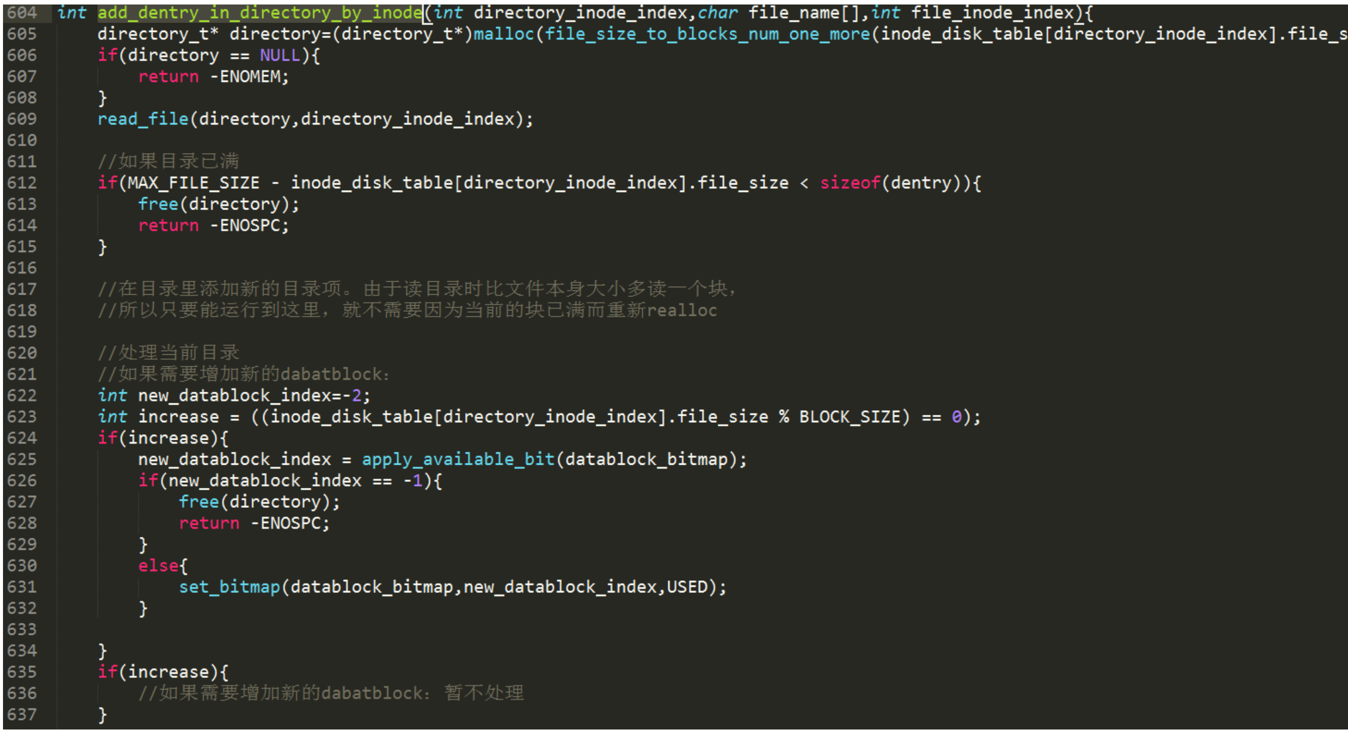


```
    device_write_sector(first_indirect_block,DATA_BLOCKS_LOC + inode->indirect_block);
//read_debug_directory();
//read_debug_indirect_block();

    //在    二级间接索引块  里添加  间接数据块
    //先填充上一个没填满的二级间接索引块
    int old_down_offset_in_first_indirect_block = old_indirect_datablock_num / (BLOCK_SIZE/sizeof(int));
    int old_offset_in_second_indirect_block = old_indirect_datablock_num % (BLOCK_SIZE/sizeof(int));
    device_read_sector(second_indirect_block,DATA_BLOCKS_LOC + first_indirect_block->indirect_block_table[ old_down_offset_in_first_indirect_blo
    for(k=old_offset_in_second_indirect_block ; k<BLOCK_SIZE/sizeof(int) && new_datablock_i<add_datablock_num ; ++k,++new_datablock_i){
        second_indirect_block->indirect_block_table[k] = new_datablock_index[new_datablock_i];
    }
//read_debug_directory();
//read_debug_indirect_block();
    device_write_sector(second_indirect_block,DATA_BLOCKS_LOC + first_indirect_block->indirect_block_table[ old_down_offset_in_first_indirect_bl
//read_debug_directory();
//read_debug_indirect_block();
    //再填充新的二级间接索引块
    for(j=old_second_indirect_block_num; j<new_second_indirect_block_num && new_datablock_i<add_datablock_num;++j){
//read_debug_directory();
//read_debug_indirect_block();
        device_clear_sector(DATA_BLOCKS_LOC + first_indirect_block->indirect_block_table[j]);
//read_debug_directory();
//read_debug_indirect_block();
        device_read_sector(second_indirect_block,DATA_BLOCKS_LOC + first_indirect_block->indirect_block_table[j]);
        for(k=0; k<BLOCK_SIZE/sizeof(int) && new_datablock_i<add_datablock_num ; ++k,++new_datablock_i){
            second_indirect_block->indirect_block_table[k] = new_datablock_index[new_datablock_i];
        }
//read_debug_directory();
//read_debug_indirect_block();
        device_write_sector(second_indirect_block,DATA_BLOCKS_LOC + first_indirect_block->indirect_block_table[j]);
//read_debug_directory();
//read_debug_indirect_block();
    }

```

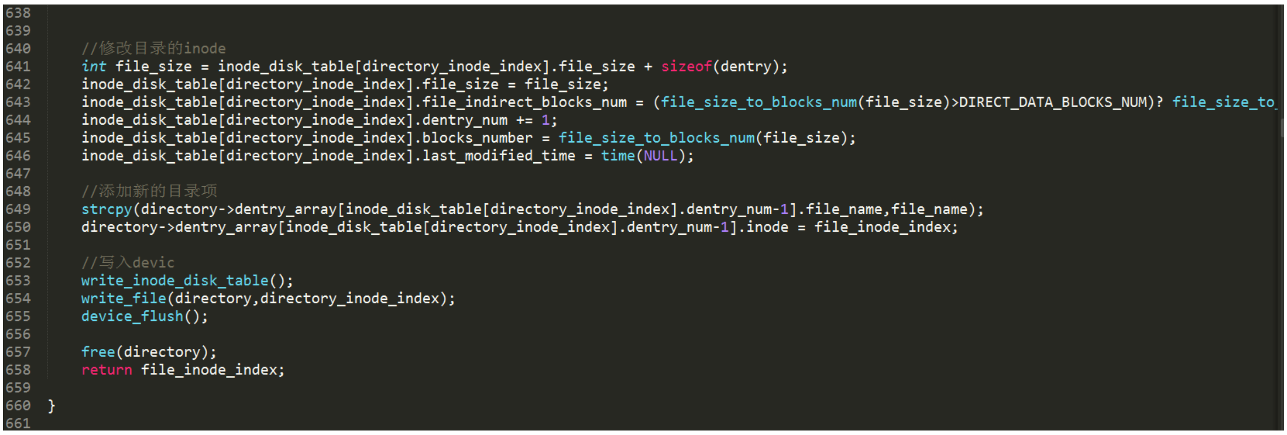


```
    //更新inode统计信息
    inode->file_size = new_size;
    inode->blocks_number = new_datablock_num;
    inode->last_modified_time = time(NULL);

    //写回disk：datablock_bitmap，inode，（所有间接块在前面已经写入了）
//read_debug_directory();
//read_debug_indirect_block();
    write_inode_disk_table();
//read_debug_directory();
//read_debug_indirect_block();
    write_bitmap(datablock_bitmap,"datablock");
//read_debug_directory();
//read_debug_indirect_block();

    //将新添加的正文数据块清零
    int new_datablock_index_array[new_datablock_num];
    get_file_datablock_index_array(new_datablock_index_array,file_inode_index);
    int m;
    for(m=old_datablock_num ; m<new_datablock_num ; ++m){
//read_debug_directory();
//read_debug_indirect_block();
        device_clear_sector(DATA_BLOCKS_LOC + new_datablock_index_array[m]);
//read_debug_directory();
//read_debug_indirect_block();
    }

    free(first_indirect_block);
    free(second_indirect_block);

    return 0;
}

```

3.在目录中添加一个目录项

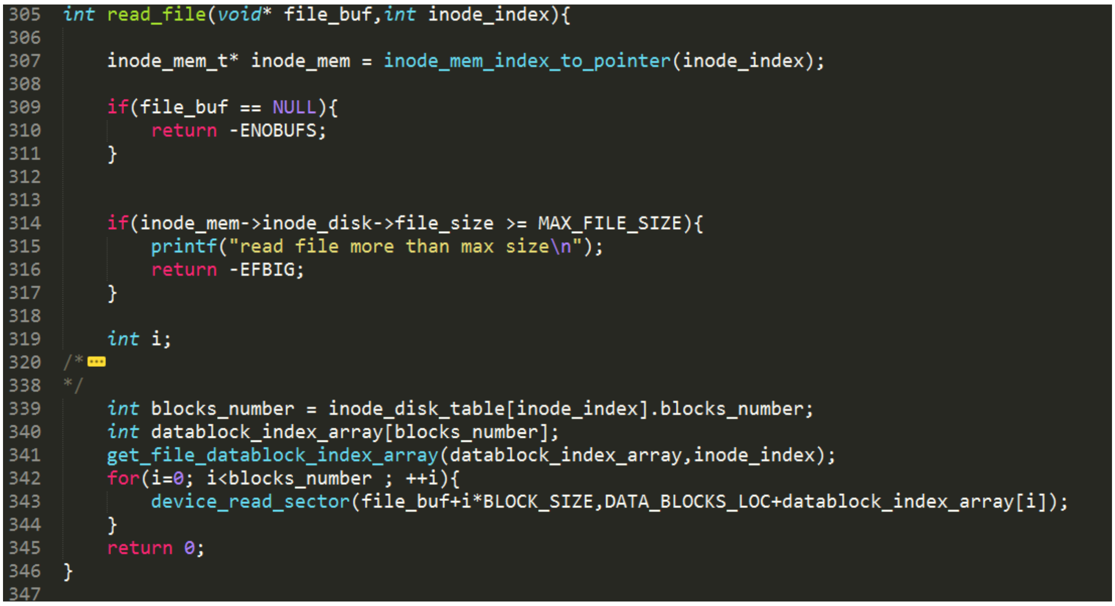


```c
int add_dentry_in_directory_by_inode(int directory_inode_index,char file_name[],int file_inode_index){
    directory_t* directory=(directory_t*)malloc(file_size_to_blocks_num_one_more(inode_disk_table[directory_inode_index].file_s
    if(directory == NULL){
        return -ENOMEM;
    }
    read_file(directory,directory_inode_index);

    //如果目录已满
    if(MAX_FILE_SIZE - inode_disk_table[directory_inode_index].file_size < sizeof(dentry)){
        free(directory);
        return -ENOSPC;
    }

    //在目录里添加新的目录项。由于读目录时比文件本身大小多读一个块，
    //所以只要能运行到这里，就不需要因为当前的块已满而重新realloc

    //处理当前目录
    //如果需要增加新的dabatblock:
    int new_datablock_index=-2;
    int increase = ((inode_disk_table[directory_inode_index].file_size % BLOCK_SIZE) == 0);
    if(increase){
        new_datablock_index = apply_available_bit(datablock_bitmap);
        if(new_datablock_index == -1){
            free(directory);
            return -ENOSPC;
        }
        else{
            set_bitmap(datablock_bitmap,new_datablock_index,USED);
        }

    }
    if(increase){
        //如果需要增加新的dabatblock: 暂不处理
    }
```



```c


    //修改目录的inode
    int file_size = inode_disk_table[directory_inode_index].file_size + sizeof(dentry);
    inode_disk_table[directory_inode_index].file_size = file_size;
    inode_disk_table[directory_inode_index].file_indirect_blocks_num = (file_size_to_blocks_num(file_size)>DIRECT_DATA_BLOCKS_NUM)? file_size_to_
    inode_disk_table[directory_inode_index].dentry_num += 1;
    inode_disk_table[directory_inode_index].blocks_number = file_size_to_blocks_num(file_size);
    inode_disk_table[directory_inode_index].last_modified_time = time(NULL);

    //添加新的目录项
    strcpy(directory->dentry_array[inode_disk_table[directory_inode_index].dentry_num-1].file_name,file_name);
    directory->dentry_array[inode_disk_table[directory_inode_index].dentry_num-1].inode = file_inode_index;

    //写入devic
    write_inode_disk_table();
    write_file(directory,directory_inode_index);
    device_flush();

    free(directory);
    return file_inode_index;

}

```

4.将一个由 inode 号索引的文件整个读进内存



```c
int read_file(void* file_buf,int inode_index){

    inode_mem_t* inode_mem = inode_mem_index_to_pointer(inode_index);

    if(file_buf == NULL){
        return -ENOBUFS;
    }


    if(inode_mem->inode_disk->file_size >= MAX_FILE_SIZE){
        printf("read file more than max size\n");
        return -EFBIG;
    }

    int i;
/*…
*/
    int blocks_number = inode_disk_table[inode_index].blocks_number;
    int datablock_index_array[blocks_number];
    get_file_datablock_index_array(datablock_index_array,inode_index);
    for(i=0; i<blocks_number ; ++i){
        device_read_sector(file_buf+i*BLOCK_SIZE,DATA_BLOCKS_LOC+datablock_index_array[i]);
    }
    return 0;
}

```

参考文献

[1] [单击此处键入参考文献内容]