lab5_report

练习题 1：实现位于 userland/servers/tmpfs/tmpfs.c 的 tfs_mknod 和 tfs_namex

- 关于`tfs_mknod`的实现：首先根据`mkdir`的值来确定inode的类型，然后调用`new_dent`函数创建inode对应dentry，再把它加入到`dir`中的`dentries`中
- 关于`tfs_namex`的实现：使用`while`循环逐个字符地对`name`进行处理，当识别到`'/'`后截取前面的目录名，首先调用`tfs_lookup`查找对应的dentry，若为空且`mkdir_p`为true，则创建缺失目录。最终在脱离`while`循环后得到的是去掉所有目录路径的文件名

练习题 2：实现位于 userland/servers/tmpfs/tmpfs.c 的 tfs_file_read 和 tfs_file_write 。提示：由于数据块的大小为PAGE_SIZE，因此读写可能会牵涉到多个页页面。 读取不能超过文件大小，而写入可能会增加文件大小（也可能需要创建新的数据块）。

- 关于`tfs_file_read`的实现：

  为了保证读取不超过文件大小，通过如下代码确定真实读取的`size`

  ```
  if(size + offset > inode->size){
      size = inode->size - offset;
  }
  ```

  `page_no`为文件所在的起始`page`，`page_off`为文件起始内容在`page_no`中的偏移，`page_end`为文件内容所在的最后一个`page`。之后我们在从`page_no`到`page_end`的范围内遍历`page`读取文件内容即可

  ```
  page_no = offset / PAGE_SIZE;
  page_off = offset % PAGE_SIZE;
  int page_end = (offset + size) / PAGE_SIZE;
  ```

- 关于`tfs_file_write`的是实现：

  由于写入可能会增加文件大小，通过如下代码修改最终的文件大小

  ```
  if(offset+size > inode->size){
      inode->size = offset+size;
  }
  ```

  `page_no`为写入内容所在的起始`page`，`page_off`为写入内容开始位置在`page_no`中的偏移，`page_end`为写入内容所在的最后一个`page`。之后我们在从`page_no`到`page_end`的范围内遍历`page`写入文件内容即可。

  page_max为文件内容所在的最后一个`page`，若`page_max < page_no`，我们需要将这之间缺失（`radix_get`得到为`NULL`）的page进行创建和补齐。同样的，我们需要将`page_no`和`page_off`之间缺失（`radix_get`得到为`NULL`）的page进行创建和补齐，然后才能正常写入内容。

```
int page_max = inode->size / PAGE_SIZE;
page_no = offset / PAGE_SIZE;
page_off = offset % PAGE_SIZE;
int page_end = (offset + size) / PAGE_SIZE;
```

练习题 3：实现位于 userland/servers/tmpfs/tmpfs.c 的 tfs_load_image 函数。需要通过之前实现的tmpfs函数进行目录和文件的创建，以及数据的读写。

调用`tfs_namex`过滤目录得到文件名，并一路创建路径上的缺失目录，然后创建文件名对应的文件/目录。接下来，再调用`tfs_file_write`写入文件内容

练习题 4：利用 userland/servers/tmpfs/tmpfs.c 中已经实现的函数，完成 在 userland/servers/tmpfs/tmpfs_ops.c 中的 fs_creat 、 tmpfs_unlink 和 tmpfs_mkdir 函数

- `fs_creat`：利用`tfs_namex(&dirat, &leaf, 1)`处理得到文件名和父目录，并创建路径中的缺失目录。在调用`tfs_creat`创建文件
- `tmpfs_unlink`：利用`tfs_namex`处理得到文件名和父目录,并调用`tfs_remove`删除文件
- `tmpfs_mkdir`：利用`tfs_namex(&dirat, &leaf, 1)`处理得到文件名和父目录，并创建路径中的缺失目录。在调用`tfs_creat`创建目录

练习题 5：实现在 userland/servers/shell/main.c 中定义的 getch ，该函数会每次从标准输入中获取字符，并实现在 userland/servers/shell/shell.c 中的 readline ，该函数会将按下回车键之前的输入内容存入内存缓冲区。

- getch:

```
char getch()
{
    int c;
    /* LAB 5 TODO BEGIN */
    c = cgetc();

    /* LAB 5 TODO END */

    return (char) c;
}
```

- readline：将`getch()`得到的字符写入内存缓冲区的buf中，并在识别到用户输入的`/n`后将退出循环体，返回`buf`。

  （补充：如果执行*make qemu-gdb+make gdb*，测试的时候要自己在命令行里面输入测试指令的话，要注意一下，在命令行里面输入的*enter*键可能会被识别为*\r*回车，而不是*\n*换行符。所以为了正常手动测试，得在*readline*里面把*\r*的处理也加进去。

练习题 6：根据在 userland/servers/shell/shell.c 中实现好的 bultin_cmd 函数，完成 shell中内置命令对应的 do_* 函数，需要支持的命令包括： ls [dir] 、 echo [string] 、 cat [filename] 和 top 。

- ls：主要是完成`fs_scan`。先调用`alloc_fd`生成`file_fd`,然后发送`FS_REQ_OPEN`的IPC请求打开文件，并将`file_fd`与文件系统维护的`fid`绑定。然后调用`getdents`，并处理得到的目录内容项，注意要过滤掉`.`这一项
- echo：简单的字符串处理
- cat：主要是完成`print_file_content`。先调用`alloc_fd`生成`file_fd`,然后发送`FS_REQ_OPEN`的IPC请求打开文件，并将`file_fd`与文件系统维护的`fid`绑定。然后发送`FS_REQ_READ`的IPC请求，读取文件内容

练习题 7：实现在 userland/servers/shell/shell.c 中定义的 run_cmd ，以通过输入文件名 来运行可执行文件，同时补全 do_complement 函数并修改 readline 函数，以支持按tab键自动补全根目录（ / ）下的文件名。

`run_cmd`中加入对`\t`的处理，即调用`do_complement`函数并在每次调用的时候`complement_time++`

先调用`alloc_fd`生成`file_fd`,然后发送`FS_REQ_OPEN`的IPC请求打开文件，并将`file_fd`与文件系统维护的`fid`绑定。然后调用`getdents`，并处理得到的目录内容项。根据`complement_time`来确定返回的文件名是哪一个

练习题 8：补全 userland/apps/lab5 目录下的 lab5_stdio.h 与 lab5_stdio.c 文件，以实现 fopen , fwrite , fread , fclose , fscanf , fprintf 五个函数，函数用法应与libc中一致，可 以参照 lab5_main.c 中测试代码。

补全的FILE结构体为

```
typedef struct FILE {
    /* LAB 5 TODO BEGIN */
    int fd;
    bool isWrite;

    /* LAB 5 TODO END */
} FILE;
```

- fopen：先调用`alloc_fd`生成`file_fd`,然后发送`FS_REQ_OPEN`的IPC请求打开文件，并将`file_fd`与文件系统维护的`fid`绑定。若打开失败，则发送`FS_REQ_CREAT`的IPC请求创建文件，然后再打开。
- fwrite 和fread：先发送`FS_REQ_OPEN`的IPC请求打开文件，接着再发送`FS_REQ_WRITE`和`FS_REQ_READ`的IPC请求进行读写操作
- fscanf：

  对变长参数采用如下方式处理：

```
va_list ap;
va_start(ap, fmt);
printf("%s\n", fmt);
int start = 0, i = 0, offset = 0;
while(i < strlen(fmt)){
    .......
}
va_end(ap);
```

调用fread读取文件内容到`rbuf`中，然后根据参数类型是`%s`还是`%d`来对字符串进行处理。需要注意的是，对于`%d`的整形，我们自定义了`str2int`函数实现字符串到int的转换：

```
void str2int(char *str, int *n){
    int tmp = 0;
    for(int i = 0; i < strlen(str); i++){
        tmp = tmp * 10 + (str[i] - '0');
    }

    *n = tmp;
}
```

- fprintf：变长参数处理同上。根据参数类型是`%s`还是`%d`来对参数进行处理，将内容记录在`wbuf`中，并最终调用`fwrite`写入文件。需要注意的是，对于`%d`的整形，我们自定义了`int2str`函数实现int到字符串的转换：

```
void int2str(int n, char *str){
    char buf[256];
    int i = 0, tmp = n;

    if(!str){
        return;
    }
    while(tmp){
        buf[i] = (char)(tmp % 10) + '0';
        tmp /= 10;
        i++;
    }
    int len = i;
    str[i] = '\0';
    while(i > 0){
        str[len - i] = buf[i - 1];
        i--;
    }
}
```

练习题 9：本练习需要实现 userland/server/fsm/main.c 中空缺的部分，使得用户程序将文件系统请求发送给FSM后，FSM根据访问路径向对应文件系统发起请求，并将结果返回给用户程序。

这里维护了`struct mount_point_info_node`来存储对应的文件系统信息。

```
struct mount_point_info_node {
    int fs_cap;
    char path[MAX_MOUNT_POINT_LEN + 1];
    int path_len;
    ipc_struct_t *_fs_ipc_struct;
    int refcnt;

    struct list_head node;
};
```

为了满足`ls`功能，我们需要处理`FS_REQ_OPEN`和`FS_REQ_GETDENTS64`请求

为了满足`cat`功能，我们需要处理`FS_REQ_OPEN`和`FS_REQ_READ`请求。

同时为了满足练习10，我们还要处理`FS_REQ_CREATE`请求

在处理`FS_REQ_OPEN`和`FS_REQ_CREATE`请求时，调用了自定义的`checkFakefs`函数确定对应的文件系统：

```
bool checkFakefs(char pathname[]){
    // printf("pathname is %s\n", pathname);
    bool isFakefs = false;
    char *path = pathname;
    char *fakefsPath = "/fakefs";
    while(path && *path != '\0'){
        if(*fakefsPath == *path){
            path++;
            fakefsPath++;
            if((!fakefsPath && !path) || (*fakefsPath == '\0' && *path == '\0'))
{
                // printf("in fakefs\n");
                return true;
            }
        }
        else{
            if((!fakefsPath || *fakefsPath == '\0') && path){
                // printf("in fakefs\n");
                return true;
            }
            break;
        }
    }
    // printf("in tmpfs\n");

    return false;
}
```

并调用`get_mount_point(path, strlen(path))`确定`mpinfo`，再向对应的文件系统发送请求。而在open中，还需要调用`fsm_set_mount_info_withfd`将`fd`与挂载的文件系统进行绑定。

在处理`FS_REQ_READ`和`FS_REQ_GETDENTS64`请求时，执行`mpinfo = fsm_get_mount_info_withfd(client_badge, fr->read.fd);`确定挂载的文件系统，然后再向对应的文件系统发送请求。

练习题 10：为减少文件操作过程中的IPC次数，可以对FSM的转发机制进行简化。本练习需要完 成 libchcore/src/fs/fsm.c 中空缺的部分，使得 fsm_read_file 和 fsm_write_file 函数先 利用ID为FS_REQ_GET_FS_CAP的请求通过FSM处理文件路径并获取对应文件系统的 Capability， 然后直接对相应文件系统发送文件操作请求

这里维护了`fs_cap_info_node`的结构体来存储对应的文件系统信息。

```
struct fs_cap_info_node {
    int fs_cap;
    ipc_struct_t *fs_ipc_struct;
    struct list_head node;
};
```

我们首先发送`FS_REQ_GET_FS_CAP`的请求，然后调用`int cap = ipc_get_msg_cap(ipc_msg_cap, 0)`得到对应文件系统的 Capability，接着`struct fs_cap_info_node *node = get_fs_cap_info(cap);`得到/绑定对应文件系统的相应信息。

然后就可以直接对相应的文件系统发送操作请求了，同样也是先发送`FS_REQ_OPEN`打开文件，再发送`FS_REQ_WRITE`和`FS_REQ_READ`进行读写

519021910594

陶昱丞