Lab Report 5

519021910685

Grizzy@sjtu.edu.cn

注：本次lab没有思考题，因此叙述较为简单

1

提供了mknod和namex的实现

mknod根据mkdir判断创建regular file还是directory，需要注意的是传入参数的len指的是name length

namex为一个递归函数，递归地解析路径名并且更新dirat和name

```
static int tfs_mknod(struct inode *dir, const char *name, size_t len, int mkdir)
{
  struct inode *inode;
  struct dentry *dent;

  BUG_ON(!name);

  if (len == 0) {
    WARN("mknod with len of 0");
    return -ENOENT;
  }
  /* LAB 5 TODO BEGIN */
  if(mkdir==true) {
    inode = new_dir();
  }
  else {
    inode = new_reg();
  }
  inode->size = 0;
  dent = new_dent(inode, name, len);
  htable_add(&(dir->dentries), dent->name.hash, &(dent->node));

  /* LAB 5 TODO END */

  return 0;
}


int tfs_namex(struct inode **dirat, const char **name, int mkdir_p)
{
  BUG_ON(dirat == NULL);
  BUG_ON(name == NULL);
  BUG_ON(*name == NULL);
```

```
34
35      char buff[MAX_FILENAME_LEN + 1];
36      int i;
37      struct dentry *dent;
38      int err;
39
40      if (**name == '/') {
41        *dirat = tmpfs_root;
42        // make sure `name` starts with actual name
43        while (**name && **name == '/')
44          ++(*name);
45      } else {
46        BUG_ON(*dirat == NULL);
47        BUG_ON((*dirat)->type != FS_DIR);
48      }
49
50      // make sure a child name exists
51      if (!**name)
52        return -EINVAL;
53
54      // `tfs_lookup` and `tfs_mkdir` are useful here
55
56      /* LAB 5 TODO BEGIN */
57
58      char *tmp = *name;
59      i = 0;
60      while(tmp[i]!='\0' && tmp[i]!='/')
61        i++;
62      strncpy(buff, tmp, i);
63      buff[i] = '\0';
64      dent = tfs_lookup(*dirat, buff, i);
65      // whether the dentry exists or not?
66      if(dent==NULL) {
67        // create intermediate directory entry
68        if(mkdir_p==true && tmp[i]=='/') {
69          tfs_mkdir(*dirat, buff, i);
70          dent = tfs_lookup(*dirat, buff, i);
71        }
72        else {
73          return -ENOENT;
74        }
75      }
76      // the named dentry is found or and intermediate dentry is created
77      // the last component and it does not end with '/'
78      if(tmp[i]=='\0') {
79        return 0;
80      }
81      // the last component and it ends with '/'
82      else if(tmp[i]=='/'&&tmp[i+1]=='\0') {
83        *dirat = dent->inode;
84        *name = tmp + i + 1;
85        return 0;
```

```
    }
    // intermediate path but not a dentry
    else if(dent->inode->type!=FS_DIR) {
      return -ENOTDIR;
    }
    // intermediate dentry
    else{
      *dirat = dent->inode;
      *name = tmp + i + 1;
      return tfs_namex(dirat, name, mkdir_p);
    }

    /* LAB 5 TODO END */

    /* we will never reach here? */
    return 0;
  }
```

2

tfs_file_read和tfs_file_write的实现

需要注意的是，都要判断一下文件的大小，分配空间对文件大小进行动态的调整，或者是读入少于size的数据

```
ssize_t tfs_file_read(struct inode * inode, off_t offset, char *buff,
          size_t size)
{
  BUG_ON(inode->type != FS_REG);
  BUG_ON(offset > inode->size);

  u64 page_no, page_off;
  u64 cur_off = offset;
  size_t to_read;
  void *page;

  /* LAB 5 TODO BEGIN */
  u64 page_current_count = (inode->size + PAGE_SIZE - 1) / PAGE_SIZE;
  u64 page_end_count  = (offset + size + PAGE_SIZE - 1) / PAGE_SIZE;
  page_no = offset / PAGE_SIZE;
  page_off = offset % PAGE_SIZE;
  size_t have_read = 0;
  size_t part;
  if(offset >= inode->size) {
    to_read = 0;
  } else if(offset + size > inode->size) {
    to_read = inode->size - offset;
  } else {
    to_read = size;
  }

```

```
27      for(u64 idx = page_no;idx < page_end_count;++idx) {
28        page = radix_get(&inode->data, idx);
29        if(idx==page_no) {
30          part = page_off + to_read > PAGE_SIZE ? PAGE_SIZE - page_off : to_read;
31          memcpy(buff + have_read, page + page_off, part);
32          have_read += part;
33        }
34        else {
35          part = to_read - have_read > PAGE_SIZE ? PAGE_SIZE : to_read - have_read;
36          memcpy(buff + have_read, page, part);
37          have_read += part;
38        }
39      }
40      /* LAB 5 TODO END */
41
42      return to_read;
43    }
44
45    ssize_t tfs_file_write(struct inode * inode, off_t offset, const char *data,
46                size_t size)
47    {
48      BUG_ON(inode->type != FS_REG);
49      BUG_ON(offset > inode->size);
50
51      u64 page_no, page_off;
52      u64 cur_off = offset;
53      size_t to_write;
54      void *page;
55
56      /* LAB 5 TODO BEGIN */
57      u64 page_current_count = (inode->size + PAGE_SIZE - 1) / PAGE_SIZE;
58      u64 page_end_count  = (offset + size + PAGE_SIZE - 1) / PAGE_SIZE;
59      inode->size = inode->size > offset + size ? inode->size : offset+size;
60      page_no = offset / PAGE_SIZE;
61      page_off = offset % PAGE_SIZE;
62      size_t written = 0;
63      if(page_end_count > page_current_count) {
64        for(u64 idx = page_current_count;idx < page_end_count;idx++) {
65          page = calloc(1, PAGE_SIZE);
66          radix_add(&inode->data, idx, page);
67        }
68      }
69      for(u64 idx = page_no;idx < page_end_count;++idx) {
70        page = radix_get(&inode->data, idx);
71        if(idx==page_no) {
72          to_write = page_off+size > PAGE_SIZE ? PAGE_SIZE - page_off : size;
73          memcpy(page + page_off, data + written, to_write);
74          written += to_write;
75        }
76        else {
77          to_write = size-written > PAGE_SIZE ? PAGE_SIZE : size-written;
78          memcpy(page, data + written, to_write);
```

```
        written += to_write;
      }
    }

    /* LAB 5 TODO END */

    return size;
  }
```

3

tfs_load_image的实现

```
int tfs_load_image(const char *start)
{
  struct cpio_file *f;
  struct inode *dirat;
  struct dentry *dent;
  const char *leaf;
  size_t len;
  int err;
  ssize_t write_count;

  BUG_ON(start == NULL);

  cpio_init_g_files();
  cpio_extract(start, "/");

  for (f = g_files.head.next; f; f = f->next) {
  /* LAB 5 TODO BEGIN */
  dirat = tmpfs_root;
  leaf = f->name;
  len = 0;
  int err = tfs_namex(&dirat, &leaf, 1);
  while(leaf[len]!='\0')
    len++;
  if(len==1 && leaf[0]=='.') {
    continue;
  }
  if(err == 0) {
    if(len==0) {
      // the last component ends with '/' and it is a directory
      continue;
    } else {
      // the last component should not be a directory
      dent = tfs_lookup(dirat, leaf, len);
      BUG_ON(dent==NULL);
      if(dent->inode->type==FS_DIR) return -EISDIR;
      size_t written = tfs_file_write(dent->inode, 0, f->data, f->header.c_filesize);
    }
```

```
    } else if(err == -ENOENT) {
      // the last component, which is a file, is missing
      tfs_creat(dirat, leaf, len);
      dent = tfs_lookup(dirat, leaf, len);
      size_t written = tfs_file_write(dent->inode, 0, f->data, f->header.c_filesize);
    } else {
      BUG_ON(err);
    }

    /* LAB 5 TODO END */
    }

    return 0;
  }
```

4

只需要根据已经实现的接口进行调用，即可完成本部分内容。

```
  int fs_creat(const char *path)
  {
    struct inode *dirat = NULL;
    const char *leaf = path;
    int err;

    BUG_ON(!path);
    BUG_ON(*path != '/');

    /* LAB 5 TODO BEGIN */
    err = tfs_namex(&dirat, &leaf, 1);
    if(err != -ENOENT) return -EEXIST;
    err = tfs_creat(dirat, leaf, strlen(leaf));
    /* LAB 5 TODO END */
    return 0;

  }

  int tmpfs_unlink(const char *path, int flags)
  {
    struct inode *dirat = NULL;
    const char *leaf = path;
    int err;

    BUG_ON(!path);
    BUG_ON(*path != '/');

    /* LAB 5 TODO BEGIN */
    err = tfs_namex(&dirat, &leaf, 1);
    if(err) return err;
    size_t len = 0;
```

```
  while(leaf[len]!='\0') len++;
  tfs_remove(dirat, leaf, len);
  /* LAB 5 TODO END */
  return err;
}

int tmpfs_mkdir(const char *path, mode_t mode)
{
  struct inode *dirat = NULL;
  const char *leaf = path;
  int err;

  BUG_ON(!path);
  BUG_ON(*path != '/');

  /* LAB 5 TODO BEGIN */
  err = tfs_namex(&dirat, &leaf, 1);
  if(err!=-ENOENT) return -EEXIST;
  size_t len = 0;
  while(leaf[len]!='\0') len++;
  err = tfs_mkdir(dirat, leaf, len);
  /* LAB 5 TODO END */
  return err;
}
```

5

调用chcore提供的系统调用完成getch函数

readline函数需要判读当前输入的是什么字符，并且将其实时呈现在console上。如果是tab则需要进行额外的自动补全，输入回车表示命令输入完成，执行对应的命令

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

  return (char) c;
}

char *readline(const char *prompt)
{
  static char buf[BUFLEN];

  int i = 0, j = 0;
  signed char c = 0;
  int ret = 0;
  char complement[BUFLEN];
```

```
  int complement_time = 0;

  if (prompt != NULL) {
    printf("%s", prompt);
  }

  while (1) {
    __chcore_sys_yield();
    c = getch();

  /* LAB 5 TODO BEGIN */
  /* Fill buf and handle tabs with do_complement(). */
  if(c=='\r' || c=='\n') {
    putc('\n');
    break;
  } else if(c=='\t') {
    putc(' ');
    complement_time++;
    do_complement(buf, complement, complement_time);
  } else {
    putc(c);
    buf[i] = c;
    i++;
  }

  /* LAB 5 TODO END */
  }

  return buf;
}
```

6

实现几个命令对应的函数，只需要构造相应的ipc_msg，进行ipc即可完成本部分内容

```
void print_file_content(char* path)
{

  /* LAB 5 TODO BEGIN */
  int fd = alloc_fd();
  int ret;
  struct ipc_msg *ipc_msg = ipc_create_msg(fs_ipc_struct_for_shell, sizeof(struct fs_request), 0);
  chcore_assert(ipc_msg);
  struct fs_request *fr = (struct fs_request*) ipc_get_msg_data(ipc_msg);
  fr->req = FS_REQ_OPEN;
  fr->open.new_fd = fd;
  strcpy(fr->open.pathname, path);
  // the `flags` and `mode` fields are useless
  // fr->open.flags = 0;
```

```
15    // fr->open.mode = 0;
16    ret = ipc_call(fs_ipc_struct_for_shell, ipc_msg);
17    ipc_destroy_msg(fs_ipc_struct_for_shell, ipc_msg);
18
19    char file_buf[BUFLEN];
20    ret = readfile(fd, file_buf, BUFLEN);
21    printf("%s", file_buf);
22
23    ipc_msg = ipc_create_msg(fs_ipc_struct_for_shell, sizeof(struct fs_request), 0);
24    chcore_assert(ipc_msg);
25    fr = (struct fs_request*) ipc_get_msg_data(ipc_msg);
26    fr->req = FS_REQ_CLOSE;
27    fr->close.fd = fd;
28    ret = ipc_call(fs_ipc_struct_for_shell, ipc_msg);
29    ipc_destroy_msg(fs_ipc_struct_for_shell, ipc_msg);
30
31    /* LAB 5 TODO END */
32
33  }
34
35  void fs_scan(char *path)
36  {
37
38    /* LAB 5 TODO BEGIN */
39    int fd = alloc_fd();
40    int ret;
41    struct ipc_msg *ipc_msg = ipc_create_msg(fs_ipc_struct_for_shell, sizeof(struct
    fs_request), 0);
42    chcore_assert(ipc_msg);
43    struct fs_request *fr = (struct fs_request*) ipc_get_msg_data(ipc_msg);
44    fr->req = FS_REQ_OPEN;
45    fr->open.new_fd = fd;
46    strcpy(fr->open.pathname, path);
47    // the `flags` and `mode` fields are useless
48    // fr->open.flags = 0;
49    // fr->open.mode = 0;
50    ret = ipc_call(fs_ipc_struct_for_shell, ipc_msg);
51    ipc_destroy_msg(fs_ipc_struct_for_shell, ipc_msg);
52
53    char name[BUFLEN];
54    char scan_buf[BUFLEN];
55    char tmp[BUFLEN];
56    int offset;
57    struct dirent *p;
58    ret = getdents(fd, scan_buf, BUFLEN);
59    int filled = 0;
60    int len = 0;
61    for (offset = 0; offset < ret; offset += p->d_reclen) {
62      p = (struct dirent *)(scan_buf + offset);
63      get_dent_name(p, tmp);
64      printf("%s ", tmp);
65    }
```

```
66
67    ipc_msg = ipc_create_msg(fs_ipc_struct_for_shell, sizeof(struct fs_request), 0);
68    chcore_assert(ipc_msg);
69    fr = (struct fs_request*) ipc_get_msg_data(ipc_msg);
70    fr->req = FS_REQ_CLOSE;
71    fr->close.fd = fd;
72    ret = ipc_call(fs_ipc_struct_for_shell, ipc_msg);
73    ipc_destroy_msg(fs_ipc_struct_for_shell, ipc_msg);
74
75    /* LAB 5 TODO END */
76  }
```

7

do_complement和run_cmd函数

```
1   int run_cmd(char *cmdline)
2   {
3     int cap = 0;
4     /* Hint: Function chcore_procm_spawn() could be used here. */
5     /* LAB 5 TODO BEGIN */
6     chcore_procm_spawn(cmdline, &cap);
7     /* LAB 5 TODO END */
8     return 0;
9   }
10
11  int do_complement(char *buf, char *complement, int complement_time)
12  {
13    int ret = 0, j = 0;
14    struct dirent *p;
15    char name[BUFLEN];
16    char scan_buf[BUFLEN];
17    int r = -1;
18    int offset;
19
20    /* LAB 5 TODO BEGIN */
21    int fd = alloc_fd();
22    const char path[] = "/";
23    struct ipc_msg *ipc_msg = ipc_create_msg(fs_ipc_struct_for_shell, sizeof(struct
    fs_request), 0);
24    chcore_assert(ipc_msg);
25    struct fs_request *fr = (struct fs_request*) ipc_get_msg_data(ipc_msg);
26    fr->req = FS_REQ_OPEN;
27    fr->open.new_fd = fd;
28    strcpy(fr->open.pathname, path);
29    // the `flags` and `mode` fields are useless
30    // fr->open.flags = 0;
31    // fr->open.mode = 0;
32    ret = ipc_call(fs_ipc_struct_for_shell, ipc_msg);
33    ipc_destroy_msg(fs_ipc_struct_for_shell, ipc_msg);
```

```
34
35      ret = getdents(fd, scan_buf, BUFLEN);
36      for (offset = 0; offset < ret; offset += p->d_reclen) {
37        p = (struct dirent *)(scan_buf + offset);
38        j++;
39        get_dent_name(p, complement);
40        if(j==complement_time) {
41          printf("%s", complement);
42        }
43      }
44
45      ipc_msg = ipc_create_msg(fs_ipc_struct_for_shell, sizeof(struct fs_request), 0);
46      chcore_assert(ipc_msg);
47      fr = (struct fs_request*) ipc_get_msg_data(ipc_msg);
48      fr->req = FS_REQ_CLOSE;
49      fr->close.fd = fd;
50      ret = ipc_call(fs_ipc_struct_for_shell, ipc_msg);
51      ipc_destroy_msg(fs_ipc_struct_for_shell, ipc_msg);
52
53      /* LAB 5 TODO END */
54
55      return r;
56    }
```

8

封装了fread、fwrite、fopen、fscanf、fprintf函数

FILE结构体的设计不需要维护buffer和pos信息，因为fd已经在文件系统底层映射并且维护了相关信息

这部分难点在于fprintf和fscanf函数解析format

下面给出部分实现，具体实现详见源码

```
1   int fprintf(FILE * f, const char * fmt, ...) {
2
3     /* LAB 5 TODO BEGIN */
4     va_list va;
5     va_start(va, fmt);
6     char buf[512];
7
8     int idx_fmt = 0, idx_buf = 0;
9     int len_fmt = strlen(fmt);
10    int val;
11    char *str_ptr;
12    while(idx_fmt < len_fmt) {
13      if(fmt[idx_fmt]=='%') {
14        idx_fmt++;
15        if(fmt[idx_fmt]=='d') {
16          val = va_arg(va, int);
```

```
          idx_fmt++;
          fprintf_handle_int(val, &idx_buf, buf);
        } else if(fmt[idx_fmt]=='s') {
          str_ptr = va_arg(va, char*);
          idx_fmt++;
          fprintf_handle_str(str_ptr, &idx_buf, buf);
        } else {
          return -1;
        }
      } else {
        buf[idx_buf] = fmt[idx_fmt];
        idx_fmt++;
        idx_buf++;
      }
    }
    fwrite(buf, strlen(buf), 1, f);

    /* LAB 5 TODO END */
      return 0;
  }

  int fscanf(FILE * f, const char * fmt, ...) {

    /* LAB 5 TODO BEGIN */
    va_list va;
    va_start(va, fmt);
    char buf[512];
    size_t size = fread(buf, 512, 1, f);

    int idx_fmt = 0, idx_buf = 0;
    int len_fmt = strlen(fmt);
    int *int_ptr;
    char *str_ptr;
    while(idx_fmt < len_fmt) {
      if(fmt[idx_fmt]=='%') {
        idx_fmt++;
        if(fmt[idx_fmt]=='d') {
          int_ptr = va_arg(va, int*);
          idx_fmt++;
          fscanf_handle_int(int_ptr, &idx_buf, buf);
        } else if(fmt[idx_fmt]=='s') {
          str_ptr = va_arg(va, char*);
          idx_fmt++;
          fscanf_handle_str(str_ptr, &idx_buf, buf);
        } else {
          return -1;
        }
      } else {
        idx_fmt++;
        idx_buf++;
      }
    }
```

```
69
70    /* LAB 5 TODO END */
71      return 0;
72  }
```

9

这部分需要注意，需要自己维护fd、mount path和mount info的映射关系，mount info中存有对应文件系统的ipc_struct，通过对应文件系统的ipc_struct发送ipc请求就可以将operation分发到不同的文件系统上去。

```
1   void fsm_server_dispatch(struct ipc_msg *ipc_msg, u64 client_badge)
2   {
3     int ret;
4     bool ret_with_cap = false;
5     struct fs_request *fr;
6     fr = (struct fs_request *)ipc_get_msg_data(ipc_msg);
7     struct mount_point_info_node *mpinfo = NULL;
8
9     /* You could add code here as you want.*/
10    /* LAB 5 TODO BEGIN */
11    int fd;
12    struct ipc_msg *ipc_msg_transfer;
13    struct fs_request *fr_transfer;
14
15    /* LAB 5 TODO END */
16
17    spinlock_lock(&fsmlock);
18
19    switch(fr->req) {
20      case FS_REQ_MOUNT:
21        ret = fsm_mount_fs(fr->mount.fs_path, fr->mount.mount_path); // path=(device_name),
    path2=(mount_point)
22        break;
23      case FS_REQ_UMOUNT:
24        ret = fsm_umount_fs(fr->mount.fs_path);
25        break;
26      case FS_REQ_GET_FS_CAP:
27        mpinfo = get_mount_point(fr->getfscap.pathname, strlen(fr->getfscap.pathname));
28        strip_path(mpinfo, fr->getfscap.pathname);
29        ipc_msg->cap_slot_number = 1;
30        ipc_set_msg_cap(ipc_msg, 0, mpinfo->fs_cap);
31        ret_with_cap = true;
32        break;
33
34      /* LAB 5 TODO BEGIN */
35      case FS_REQ_OPEN:
36        fd = fr->open.new_fd;
37        mpinfo = get_mount_point(fr->open.pathname, strlen(fr->open.pathname));
38        strip_path(mpinfo, fr->open.pathname);
```

```
39        ipc_msg_transfer = ipc_create_msg(mpinfo->_fs_ipc_struct, sizeof(struct fs_request),
    0);
40        fr_transfer = (struct fs_request*)ipc_get_msg_data(ipc_msg_transfer);
41        memcpy(fr_transfer, fr, sizeof(struct fs_request));
42        ret = ipc_call(mpinfo->_fs_ipc_struct, ipc_msg_transfer);
43        memcpy(ipc_get_msg_data(ipc_msg), ipc_get_msg_data(ipc_msg_transfer),
    ipc_msg_transfer->data_len);
44        ipc_destroy_msg(mpinfo->_fs_ipc_struct, ipc_msg_transfer);
45        fsm_set_mount_info_withfd(client_badge, fd, mpinfo);
46        break;
47      case FS_REQ_CLOSE:
48        ret = transfer_ipc_with_fd(fr->close.fd, ipc_msg, fr, client_badge);
49        break;
50      case FS_REQ_CREAT:
51        ret = transfer_ipc_with_pathname(fr->creat.pathname, ipc_msg, fr);
52        break;
53      case FS_REQ_MKDIR:
54        ret = transfer_ipc_with_pathname(fr->mkdir.pathname, ipc_msg, fr);
55        break;
56      case FS_REQ_RMDIR:
57        ret = transfer_ipc_with_pathname(fr->rmdir.pathname, ipc_msg, fr);
58        break;
59      case FS_REQ_UNLINK:
60        ret = transfer_ipc_with_pathname(fr->unlink.pathname, ipc_msg, fr);
61        break;
62      case FS_REQ_READ:
63        ret = transfer_ipc_with_fd(fr->read.fd, ipc_msg, fr, client_badge);
64        break;
65      case FS_REQ_WRITE:
66        ret = transfer_ipc_with_fd(fr->write.fd, ipc_msg, fr, client_badge);
67        break;
68      case FS_REQ_GET_SIZE:
69        ret = transfer_ipc_with_pathname(fr->getsize.pathname, ipc_msg, fr);
70        break;
71      case FS_REQ_LSEEK:
72        ret = transfer_ipc_with_fd(fr->lseek.fd, ipc_msg, fr, client_badge);
73        break;
74      case FS_REQ_GETDENTS64:
75        ret = transfer_ipc_with_fd(fr->getdents64.fd, ipc_msg, fr, client_badge);
76        break;
77
78      /* LAB 5 TODO END */
79
80      default:
81        printf("[Error] Strange FS Server request number %d\n", fr->req);
82        ret = -EINVAL;
83      break;
84
85    }
86
87    spinlock_unlock(&fsmlock);
88
```

```
    if(ret_with_cap) {
      ipc_return_with_cap(ipc_msg, ret);
    } else {
      ipc_return(ipc_msg, ret);
    }
  }
```

10

根据FS_REQ_GET_FS_CAP获取对应文件系统的cap，然后直接想对应文件系统发送ipc请求

```
nt fsm_write_file(const char* path, char* buf, unsigned long size) {
        if (!fsm_ipc_struct) {
                connect_fsm_server();
        }
        int ret = 0;

        /* LAB 5 TODO BEGIN */
        u64 cap;
        int fd;
        struct fs_cap_info_node *fs_cap_info_node;
        struct ipc_struct *fsx_ipc_struct;
        struct ipc_msg *fsm_ipc_msg, *fsx_ipc_msg;
        struct fs_request *fsm_fr_request, *fsx_fr_request;

        fsm_ipc_msg = ipc_create_msg(fsm_ipc_struct, sizeof(struct fs_request), 0);
        fsm_fr_request = (struct fs_request*)ipc_get_msg_data(fsm_ipc_msg);
        fsm_fr_request->req = FS_REQ_GET_FS_CAP;
        memcpy(fsm_fr_request->getfscap.pathname, path, size);
        ret = ipc_call(fsm_ipc_struct, fsm_ipc_msg);
        fsm_fr_request = (struct fs_request*)ipc_get_msg_data(fsm_ipc_msg);
        cap = ipc_get_msg_cap(fsm_ipc_msg, 0);
        ipc_destroy_msg(fsm_ipc_struct, fsm_ipc_msg);

        fs_cap_info_node = get_fs_cap_info(cap);
        fsx_ipc_struct = fs_cap_info_node->fs_ipc_struct;

        fd = alloc_fd();
        fsx_ipc_msg = ipc_create_msg(fsx_ipc_struct, sizeof(struct fs_request), 0);
        fsx_fr_request = (struct fs_request*)ipc_get_msg_data(fsx_ipc_msg);
        fsx_fr_request->req = FS_REQ_OPEN;
        fsx_fr_request->open.new_fd = fd;
        strcpy(fsx_fr_request->open.pathname, fsm_fr_request->getfscap.pathname);
        ret = ipc_call(fsx_ipc_struct, fsx_ipc_msg);
        ipc_destroy_msg(fsx_ipc_struct, fsx_ipc_msg);

        fsx_ipc_msg = ipc_create_msg(fsx_ipc_struct, sizeof(struct fs_request), 0);
        fsx_fr_request = (struct fs_request*)ipc_get_msg_data(fsx_ipc_msg);
        fsx_fr_request->req = FS_REQ_WRITE;
        fsx_fr_request->write.fd = fd;
```

```
        fsx_fr_request->write.count = size;
        memcpy((void *)fsx_fr_request + sizeof(struct fs_request), buf, size);
        ret = ipc_call(fsx_ipc_struct, fsx_ipc_msg);
        ipc_destroy_msg(fsx_ipc_struct, fsx_ipc_msg);

        fsx_ipc_msg = ipc_create_msg(fsx_ipc_struct, sizeof(struct fs_request), 0);
        fsx_fr_request = (struct fs_request*)ipc_get_msg_data(fsx_ipc_msg);
        fsx_fr_request->req = FS_REQ_CLOSE;
        fsx_fr_request->close.fd = fd;
        ret = ipc_call(fsx_ipc_struct, fsx_ipc_msg);
        ipc_destroy_msg(fsx_ipc_struct, fsx_ipc_msg);

        /* LAB 5 TODO END */

        return ret;
}

/* Read content from the file at `path`. */
int fsm_read_file(const char* path, char* buf, unsigned long size) {

        if (!fsm_ipc_struct) {
                connect_fsm_server();
        }
        int ret = 0;

        /* LAB 5 TODO BEGIN */
        u64 cap;
        int fd;
        struct fs_cap_info_node *fs_cap_info_node;
        struct ipc_struct *fsx_ipc_struct;
        struct ipc_msg *fsm_ipc_msg, *fsx_ipc_msg;
        struct fs_request *fsm_fr_request, *fsx_fr_request;

        fsm_ipc_msg = ipc_create_msg(fsm_ipc_struct, sizeof(struct fs_request), 0);
        fsm_fr_request = (struct fs_request*)ipc_get_msg_data(fsm_ipc_msg);
        fsm_fr_request->req = FS_REQ_GET_FS_CAP;
        memcpy(fsm_fr_request->getfscap.pathname, path, size);
        ret = ipc_call(fsm_ipc_struct, fsm_ipc_msg);
        fsm_fr_request = (struct fs_request*)ipc_get_msg_data(fsm_ipc_msg);
        cap = ipc_get_msg_cap(fsm_ipc_msg, 0);
        ipc_destroy_msg(fsm_ipc_struct, fsm_ipc_msg);

        fs_cap_info_node = get_fs_cap_info(cap);
        fsx_ipc_struct = fs_cap_info_node->fs_ipc_struct;

        fd = alloc_fd();
        fsx_ipc_msg = ipc_create_msg(fsx_ipc_struct, sizeof(struct fs_request), 0);
        fsx_fr_request = (struct fs_request*)ipc_get_msg_data(fsx_ipc_msg);
        fsx_fr_request->req = FS_REQ_OPEN;
        fsx_fr_request->open.new_fd = fd;
        strcpy(fsx_fr_request->open.pathname, fsm_fr_request->getfscap.pathname);
        ret = ipc_call(fsx_ipc_struct, fsx_ipc_msg);
```

```
92          ipc_destroy_msg(fsx_ipc_struct, fsx_ipc_msg);
93
94          fsx_ipc_msg = ipc_create_msg(fsx_ipc_struct, sizeof(struct fs_request), 0);
95          fsx_fr_request = (struct fs_request*)ipc_get_msg_data(fsx_ipc_msg);
96          fsx_fr_request->req = FS_REQ_READ;
97          fsx_fr_request->read.fd = fd;
98          fsx_fr_request->read.count = size;
99          ret = ipc_call(fsx_ipc_struct, fsx_ipc_msg);
100         memcpy(buf, ipc_get_msg_data(fsx_ipc_msg), ret);
101         ipc_destroy_msg(fsx_ipc_struct, fsx_ipc_msg);
102
103         fsx_ipc_msg = ipc_create_msg(fsx_ipc_struct, sizeof(struct fs_request), 0);
104         fsx_fr_request = (struct fs_request*)ipc_get_msg_data(fsx_ipc_msg);
105         fsx_fr_request->req = FS_REQ_CLOSE;
106         fsx_fr_request->close.fd = fd;
107         ret = ipc_call(fsx_ipc_struct, fsx_ipc_msg);
108         ipc_destroy_msg(fsx_ipc_struct, fsx_ipc_msg);
109
110         /* LAB 5 TODO END */
111
112         return ret;
113 }
```