BUAA OS Lab5-2 课上测试

目录

BUAA OS Lab5-2 课上测试

回到顶部

一、exam部分

（一）题目

在exam中，我们需要新增三种文件的打开方式：O_APPEND，O_ALONE，O_CREAT。

（1）O_APPEND

我们原来的文件系统在打开文件时，指针为0，这就导致了写的时候会覆盖文件原有内容。

如一个文件./motd原有内容为：

```
This is /motd, the message of the day.
Welcome to the MOS kernel, now with a file system!
```

那么在经过如下操作后

```
int fdnum = open("/motd", O_RDWR);
 fwritef(fdnum, "test append");
```

文件的内容就变成了

```
test appendtd, the message of the day.
Welcome to the MOS kernel, now with a file system!
```

我们需要增加一种文件打开类型O_APPEND，**使得文件打开后的偏移指针为文件大小**，这样开始写就不会覆盖原来的文件内容了。

（2）O_ALONE

我们原来的操作系统中，经过fork得到的子进程会和父进程共享描述符Fd，因此，一方读文件的时候，另一方的Fd的偏移指针也会改变（因为这个Fd是共享映射的）。

如一个文件./motd原有内容为：

```
This is a NEW message of the day!
```

经过如下操作：

```
int r, fdnum, n;
    char buf[200];
    fdnum = open("/newmotd", O_RDWR | O_ALONE);
    if ((r = fork()) == 0) {
        n = read(fdnum, buf, 5);
        writef("[child] buffer is \'%s\'\n", buf);
    } else {
        n = read(fdnum, buf, 5);
        writef("[father] buffer is \'%s\'\n", buf);
    }
```

得到的结果为：

```
[father] buffer is 'This '
[child] buffer is 'is a '
```

我们需要新增一种文件打开类型O_ALONE，**使得以此方式打开文件的父进程与子进程不再共享这一文件的文件描述符的偏移指针。**修改后的输出应该如下：

```
[father] buffer is 'This '
[child] buffer is 'This '
```

（3）O_CREAT

（为什么不叫O_CREATE？？？）

我们原来的操作系统中，如果用户进程打开不存在的进程，会返回错误E_NOT_FOUND。

我们需要新增一种文件打开类型O_CREAT，使得以此方式打开文件，**如果文件不存在，会创建一个空文件并正常打开。**

（二）分析与实现

首先，我们需要加入上述三种类型的宏定义（其中，O_CREAT原本就有了）。具体操作为在user/lib.h中加入。

（1）O_APPEND

善良的助教在题目中“明示”我们，“使得文件打开后的偏移指针为文件大小”，以及“仅需修改 user/file.c 中的 open 函数”。

首先我们需要明确，文件的偏移指针是啥。文件的偏移指针其实就是用户空间中的文件描述符Fd的成员fd_offset。它表示当前正在操作的文件内容的位置。

```
struct Fd {
    u_int fd_dev_id;
    u_int fd_offset;
    u_int fd_omode;
};
```

那么，用户空间的描述符Fd又是什么时候得到的呢？就在打开文件时（user/file.c的open），文件系统的服务进程就会把ffd（里面有个Fd）共享映射给用户进程。当然，这些已经在我们原来的open中就实现了，我们在用户空间已经得到了Fd。

因此，我们要做的就是在open中，判断打开模式，如果是O_APPEND，就把Fd的fd_offset初始化成文件的大小。

```
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;

```

```
    r = fd_alloc(&fd);
    if (r < 0) return r;
    r = fsipc_open(path, mode, fd);
    if (r < 0) return r;
    va = fd2data(fd);
    ffd = (struct Filefd*)fd;
    fileid = ffd->f_fileid;
    size = ffd->f_file.f_size;

    /* -------------- 修改部分 ---------------- */
    if ((mode & O_APPEND) != 0) {
        fd->fd_offset = size;
    }
    /* ---------------------------------- */

    for (i = 0; i < size; i = i + BY2BLK) {
        r = fsipc_map(fileid, i, va + i);
        if (r < 0) return r;
    }
    return fd2num(fd);
}
```

需要注意的是，文件打开类型的判断 `(mode & O_APPEND) != 0` 中的括号以及 != 0（不是==1）

（2）O_ALONE

首先，要使得子进程的Fd的偏移指针不继承父进程的偏移，容易想到在fork中子进程需要对O_ALONE的Fd的偏移指针清0.

但是此时有一个问题，就是文件系统的服务进程在把ffd映射给用户进程时，含有标志位PTE_LIBRARY，因此父进程的Fd的映射含有PTE_LIBRARY，而我们在fork中，**对页面进行COW保护时，对于有PTE_LIBRARY的页面，不会进行COW保护。**这就意味着，Fd对于父子进程是共享的，没有COW保护，这就导致子进程修改Fd时，父进程的Fd也发生了变化。

当然，善良的助教又给出了提升“**修改文件描述符页面进行映射时的 PTE_LIBRARY 标志位**”，也就是说，在服务进程映射ffd给用户进程时，**如果是O_ALONE的，就不加PTE_LIBRARY**，那么这个Fd就会被COW保护，从而父子进程就得到了各自的Fd，子进程也就能肆意修改了。

那么，服务进程映射ffd给用户进程在哪儿呢？答案就是serve_open（因为在打开文件的一开始，服务进程就要把ffd映射给用户进程）

于是，我们需要修改这个映射的过程，对于O_ALONE的，不加PTE_LIBRARY。

```
void
serve_open(u_int envid, struct Fsreq_open *rq)
{
    writef("serve_open %08x %x 0x%x\n", envid, (int)rq->req_path, rq->req_omode);

    u_char path[MAXPATHLEN];
    struct File *f;
    struct Filefd *ff;
    int fileid;
    int r;
    struct Open *o;
    user_bcopy(rq->req_path, path, MAXPATHLEN);
    path[MAXPATHLEN - 1] = 0;
    if ((r = open_alloc(&o)) < 0) {
        user_panic("open_alloc failed: %d, invalid path: %s", r, path);
        ipc_send(envid, r, 0, 0);
    }
    fileid = r;
    if ((r = file_open((char *)path, &f)) < 0) {
    //  user_panic("file_open failed: %d, invalid path: %s", r, path);
        ipc_send(envid, r, 0, 0);
        return ;
    }
    o->o_file = f;
    ff = (struct Filefd *)o->o_ff;
    ff->f_file = *f;
    ff->f_fileid = o->o_fileid;
    o->o_mode = rq->req_omode;
```

```
    ff->f_fd.fd_omode = o->o_mode;
    ff->f_fd.fd_dev_id = devfile.dev_id;

    /* -------------- 修改部分 ---------------- */
    if ((rq->req_omode & O_ALONE) != 0) ipc_send(envid, 0, (u_int)o->o_ff, PTE_V |
    else ipc_send(envid, 0, (u_int)o->o_ff, PTE_V | PTE_R | PTE_LIBRARY);
    /* ------------------------------------- */
}
```

修改完PTE_LIBRARY的映射后，子进程就可以自己修改Fd的偏移指针了。

```
int fork() {
    /* ... */
    if (id == 0) {
        // 子进程 清空对应Fd的偏移指针
        struct Fd* fd;
        for (i = 0; i < MAXFD; i++) {
            fd = (struct Fd*)num2fd(i);
            if ((fd->fd_omode & O_ALONE) != 0) {
                fd->fd_offset = 0;
            }
        }
        return 0;
    }
}
```

(3) O_CREAT

用户进程打开一个不存在的文件时，原本会返回错误，我们需要修改为创建一个新的空文件并正常打开。对于创建空文件，善良的助教已经给出了提示“需要调用 fs.c 中的 file_create 函数”。

那么，我们需要在哪里修改呢？这个函数应该是一个和open有关的函数，同时它需要调用file_create来创建一个空文件并正常打开。根据open的流程一步步走，可以发现在serve_open中，调用file_open打开不存在的文件时就会发生错误。看来，serve_open就是我们需要修改的地方了。

为什么不是更为底层的file_open呢？因为只有在O_CREAT的情况下才会创建空文件，而file_open只是简单的根据path寻找File，并不会判断文件打开类型，因此修改的函数为serve_open

于是，在serve_open中，如果file_create找不到文件，且为O_CREAT时，我们就用file_create创建一个空文件，然后按正常流程继续就可以了。

```
void
serve_open(u_int envid, struct Fsreq_open *rq)
{
    writef("serve_open %08x %x 0x%x\n", envid, (int)rq->req_path, rq->req_omode);

    u_char path[MAXPATHLEN];
    struct File *f;
    struct Filefd *ff;
    int fileid;
    int r;
    struct Open *o;
    user_bcopy(rq->req_path, path, MAXPATHLEN);
    path[MAXPATHLEN - 1] = 0;
    if ((r = open_alloc(&o)) < 0) {
        user_panic("open_alloc failed: %d, invalid path: %s", r, path);
        ipc_send(envid, r, 0, 0);
    }
    fileid = r;
     /* -------------- 修改部分 ---------------- */
    if ((r = file_open((char *)path, &f)) < 0) {
    //  user_panic("file_open failed: %d, invalid path: %s", r, path);
        if ((rq->req_omode & O_CREAT) != 0) {
            r = file_create((char*)path, &f);
            if (r < 0) user_panic("cannot create file");
        }
        else { // 非O_CREAT 返回
            ipc_send(envid, r, 0, 0);
            return ;
```

```
        }
    }
    /* ---------------------------------- */
    o->o_file = f;
    ff = (struct Filefd *)o->o_ff;
    ff->f_file = *f;
    ff->f_fileid = o->o_fileid;
    o->o_mode = rq->req_omode;
    ff->f_fd.fd_omode = o->o_mode;
    ff->f_fd.fd_dev_id = devfile.dev_id;
    if ((rq->req_omode & O_ALONE) != 0) ipc_send(envid, 0, (u_int)o->o_ff, PTE_V |
    else ipc_send(envid, 0, (u_int)o->o_ff, PTE_V | PTE_R | PTE_LIBRARY);
}
```

目录
导航

回到顶部

二、Extra部分

（一）题目

我们需要在user/file.c中实现函数

```
int list_dir(const char *path, char* ans)
```

该函数接受以 path 为绝对路径的文件夹，将此文件夹的所有文件的文件名以 空格(space) 为间隔保存在字符串 ans 中。函数执行成功时返回 0 ，目录不存在时返回 -1 。

其中，文件夹内文件的遍历顺序与dir_look_up相同。

实现步骤：

1. 在 user/lib.h 中添加函数声明 int list_dir(const char* path, char* ans); 。
2. 回顾课下实现 fs/fs.c 中的 dir_lookup 函数，模仿此函数列出一个目录下的所有文件的文件名。
3. 在文件系统服务进程中增加新的 Fsreq 及相关服务，通过IPC机制将结果回传给用户进程。

（二）分析

好久没见过这么短的Extra题目了~

虽然题目比较短，但是我们还需要实现其他的辅助函数来完成整个功能。

（1）获取文件夹内的文件名

```
int
dir_lookup(struct File *dir, char *name, struct File **file)
{
    int r;
    u_int i, j, nblock;
    void *blk;
    struct File *f;
    nblock = dir->f_size / BY2BLK;
    for (i = 0; i < nblock; i++) {
        r = file_get_block(dir, i, &blk);
        if (r < 0) return r;
        for (j = 0; j < FILE2BLK; j++) {
            f =(struct File*)(blk + j * sizeof(struct File));
            if (strcmp(f->f_name, name) == 0) {
                *file = f;
                return 0;
            }
        }
    }
        // If we find the target file, set the result to *file and set f_dir field
    return -E_NOT_FOUND;
}
```

首先，dir_look_up是在dir中寻找name的文件。这与我们题目中传递的参数是path好像不太一样，不过没关系，我们有**walk_path函数来直接根据path找到文件（dir也是文件）。**

dir_look_up中，首先计算出文件一共占了多少个Block（nblock），然后依次通过file_get_block访问这些Block，需要注意的是，dir的Block中，存的都是文件控制块File，因此再对每个Block遍历，就可以得到每个Block内的每个文件控制块File的信息了。

同样，我们在获取文件名时，同样可以采取这种**二重遍历**的方法。

于是，我们可以在fs/file.c中写出一个获取文件名的函数。（**注意，这个函数是在服务进程空间中的**）

```
int fs_list_dir(const char *path, char* ans);
```

这个函数根据path把读到的文件名写入ans中（为了整体结构的完整性，这里先给出函数的声明与作用）

（2）用户进程与服务进程的映射

首先，题目中要求的list_dir是在user/file.c中实现的，也就是这个函数是在用户空间中。而我们前面提到的fs_list_dir则是文件系统的服务进程中的，因此必然需要通过ipc来实现用户进程和服务进程的交互。

这还不简单，用户进程把path、ans传给服务进程，服务进程直接调用上面那个函数不就行了吗?

但是，我们需要特别注意，**用户进程和服务进程的内存空间是两个独立的空间**，也就是用户进程的ans指针传到服务进程中就成了废指针，服务进程无法根据这个ans指针直接访问用户进程的ans。

因此，**我们只能通过页面共享映射的方法来把服务进程中得到的结果映射给用户进程。**

于是，这就涉及到用户进程和服务进程的交互了。

通过课下的学习，我们知道，用户进程根据需要创建出一个请求Req，通过fsipc发送给服务进程（其实就是共享Req的页面），然后服务进程进行一通操作，把结果再通过ipc映射给用户进程。

根据以上的分析，我们可以得到以下步骤：

1. **用户进程调用list_dir（user/file.c）**
2. **list_dir调用fsipc_xxx来把需求Req发送给服务进程（user/fsipc.c）**
3. **服务进程收到请求，开始服务serve_xxx（fs/serve.c）**
4. **serve_xxx中，调用fs_list_dir得到了结果，再把结果通过ipc映射给用户进程**
5. **用户进程的fsipc收到结果**

接下来我们开始根据以上步骤来进行具体实现。

（三）具体实现

（1）list_dir

```
int list_dir(const char *path, char* ans) {
    return fsipc_exam(path, ans);
}
```

（这里我把用户进程收到结果后的一些操作，如结果复制到ans，放在了fsipc_exam中，叫exam是因为课上脑子不太清楚...）

（2）fsipc_exam

首先我们需要构造相应的请求结构体：Fsreq_exam（emm，为了统一，后面的都叫exam了...不要在意这些细节）。

我们还需要增加新的请求类型宏定义FSREQ_EXAM（在include/fs.h中）

```
struct Fsreq_exam {
    u_char path[MAXPATHLEN];
}
#define FSREQ_EXAM 8
```

然后在fsipc_exam中初始化该结构体，再通过ipc发送给服务进程。

```
int fsipc_exam(const char *path, char* ans) {
    struct Fsreq_exam *req;
    u_int perm;
    /* 构造Req */
```

```
    req = (struct Fsreq_exam*)fsipcbuf;
    user_bcopy(path, req->req_path, MAXPATHLEN);
    /* 用来接收页面映射的buf */
    char tmp[2*BY2PG];
    char *buf = tmp;
    buf = buf + BY2PG;
    buf = ROUNDDOWN(buf, BY2PG);
    syscall_mem_unmap(0, buf);
    /* 发送Req */
    int r = fsipc(FSREQ_EXAM, req, (u_int)buf, &perm); // buf就是接收映射的虚地址
    /* 把结果复制到ans */
    user_bcopy(buf, ans, strlen(buf));
    return r;
}
```

在这个函数中，我们需要开出一个页面buf来接收服务进程返回的结果的映射。有两点需要注意：一是最好我们能**把结果映射到这个buf整个页面的最开始**，这样就可以直接复制buf到ans；二是**需要通过syscall_mem_unmap解除buf原来的映射**（题目中也有相关提示）。

为了使结果映射到buf的最开始，我用了一个非常不优雅的写法，开一个2个页面大小的大数组，然后再页对齐，从而得到了buf。~~虽然这个写法丑，但是很直观嘛（其实是lab2没学好）~~。

buf就是用来接收页面映射的，在通过ipc得到结果后，把buf的内容复制到ans就可以了（**此时的ans指针是在用户空间中**，因此可以直接复制过去）

（3）serve_exam

现在我们就来到了服务进程中。

服务进程在接受到相应请求后，就经过分发会来到这个函数，进行一通操作（fs_list_dir）得到结果，并把结果映射给用户进程。

```
void serve_exam(u_int envid, struct Fsreq_exam *rq) {
    u_char path[MAXPATHLEN];
    /* 同上，开出一个页面大小的buf */
    char tmp[2*BY2PG];
    char* buf = tmp + BY2PG;
    buf = ROUNDDOWN(buf, BY2PG);
    /* 通过fs_list_dir得到结果并写入buf */
    user_bcopy(rq->req_path, path, MAXPATHLEN);
    int r = fs_list_dir(path, buf);
    /* 把返回值传递给用户进程 并把结果buf映射给用户进程 */
    ipc_send(envid, r, buf, PTE_V | PTE_R | PTE_LIBRARY);
}
```

其中的buf的取法依然是一个非常不优雅的取法。

当然，不要忘了在serve中根据请求类型进行分发。

```
void
serve(void)
{
    u_int req, whom, perm;

    for (;;) {
        perm = 0;

        req = ipc_recv(&whom, REQVA, &perm);

        // All requests must contain an argument page
        if (!(perm & PTE_V)) {
            writef("Invalid request from %08x: no argument page\n", whom);
            continue; // just leave it hanging, waiting for the next request.
        }

        switch (req) {
            /* ... */
            /* --------- 增加新的请求分发 --------- */
            case FSREQ_EXAM:
                serve_exam(whom, (struct Fsreq_exam *)REQVA);

```

```
                break;
            /* ------------------------------- */
            default:
                writef("Invalid request code %d from %08x\n", whom, req);
                break;
        }

        syscall_mem_unmap(0, REQVA);
    }
}
```

(4) fs_list_dir

这个函数只需要模仿dir_look_up进行二重遍历即可。但是需要先使用walk_path来找到path对应的文件夹。

```
int fs_list_dir(const char *path, char* ans) {
    void* blk;
    struct File* dir;
    struct File* f;
    int i,j;
    int r;
    r = walk_path(path, 0, &dir, 0);
    if (r < 0) return -1;
    int nblock = dir->f_size / BY2BLK;
    /* 二重遍历 获得文件名 */
    for (i = 0; i < nblock; i++) {
        r = file_get_block(dir, i, &blk);
        if (r < 0) return r;
        for (j = 0; j < FILE2BLK; j++) {
            f = (struct File*) (blk + j * sizeof(struct File));
            if (*(f->f_name) != '\0') {
                user_bcopy(f->f_name, ans, strlen(f->f_name));
                ans = ans + strlen(f->f_name);
                *ans = ' ';
                ans++;
            }
        }
    }
    return 0;
}
```

至此，我们就完成了Extra部分的任务了。

笔者在课上因为页面映射的问题比较纠结，最终采用了“暴力且丑陋”的写法，相信一定有更优雅的写法~

上述exam和Extra的思路可能还有不足之处，有不到位的地方请大家谅解并指正（比如那个丑陋的页面映射），谢谢！

回到顶部

补充

经过其他同学的启发，可以直接把ans放在Req中，反正Req已经是用户进程和服务进程共享的了，服务进程直接把结果写进Req的ans就可以了。这样就可以避免额外的页面映射了（这个方法太牛了！）

Boooooomb
粉丝 - 1 关注 - 0
+加关注

« 上一篇： BUAA OS Lab5-2分析
» 下一篇： BUAA-OO 第四单元 & 课程总结

posted @ 2022-06-09 17:34 Boooooomb 阅读(123) 评论(0) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部

发表评论

目录
导航

编辑 预览

支持 Markdown

自动补全

提交评论 退出 订阅评论 我的博客

[Ctrl+Enter快捷键提交]