JOS 实验六实验记录

作者：卓达城
指导老师：邵志远
单位：华中科技大学集群网络与服务计算实验室

在开始实验之前先把相关 lab5 的代码和 lab6 合并

lab6 代码量和逻辑没有 lab5 多，编程技巧的运用也没有 lab4 多，也没有 lab2 和 lab3 涉及底层的东西多，如果前面的实验做得足够完美，应该是很简单。

但是世界上是没有完美的东西的。。

lab6 的调试工作可以用痛苦来形容，因为 lab6 的涉及前面五个实验，只要前面有一点点错，往往要花一两天才能查出来。

我的天啊~~~~~~~

做实验之前先来改正一下前面的错误(这些错误找起来真的是呕心沥血啊~~)：

syscall.c

```
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a
{
        int kkk;
        // Call the function corresponding to the 'syscallno' pa
        // Return any appropriate return value.
        // LAB 3: Your code here.
//      cprintf("syscallno : %d\n", syscallno);
        switch(syscallno){
                case SYS_cputs:
                        sys_cputs((char*)a1,a2);
                        return 0;
                case SYS_cgetc:
//i just want to fuck myself.
//wo cao wo kao wo fuck!
//this bug waste me 1 day to fix it.
                        return sys_cgetc();
//                      return 0;
```

serv.c

```
void
serve_dirty(envid_t envid, struct Fsreq_dirty *rq)
{
        struct OpenFile *o;
        int r;

        if (debug)
                cprintf("serve_dirty %08x %08x %08x\n", envi

        // Mark the page containing the requested file offse
        // Returns 0 on success, < 0 on error.

        // LAB 5: Your code here.
//      panic("serve_dirty not implemented");
        if(( r = openfile_lookup(envid, rq->req_fileid, &o))
                ipc_send(envid, r, 0, 0);
                return;
        }
        if(( r = file_dirty(o->o_file, rq->req_offset)) < 0)
                ipc_send(envid, r, 0, 0);
                return;
        }
        ipc_send(envid, 0, 0, 0);
        return;
}
```

pmap.c

这个 bug 我足足用了两天的时间来修正，两天，两天，两天。。。。。

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *

{
        // Fill this function in
        pte_t *pte;
        pte = pgdir_walk(pgdir,va,1);
//      cprintf("pern:%x\n",perm);
//      cprintf("-------->\n");
        if( pte == NULL){
                return -E_NO_MEM;
        } else {
                pp -> pp_ref ++;
                if((*pte & PTE_P) !=0){
                        page_remove(pgdir,va);
                }
                *pte = page2pa(pp) |PTE_P |perm;
//              cprintf("-------->\n");
//just for fucking!!
//this bug waste me two days for deleting.
//zhuodc@qq.com fucking !!!!
//              tlb_invalidate(pgdir, va);
//              return 0;
        }
        tlb_invalidate(pgdir, va);
        return 0;
}
```

功能增强：serv.c

```
// Open the file
if ((r = file_open(path, &f)) < 0) {
//      panic("file_open fail !! r: %d\n",r );
        if (debug)
                cprintf("file_open failed: %e", r);
        if((rq -> req_omode & O_CREAT) != 0 ){
                r = file_create(path, &f);
                if( r < 0){
                        panic("file_create fail!!\n");
                }
        }else{
                goto out;
        }
}
```

增加新建文件的功能。

做 lab6 之前要先明白 lab6 到底要做什么？lab6 要做的就是父子进程共享文件、管道还有 shell。

第一步共享文件：

具体点描述就是父进程开了一个文件，然后 fork 一个子进程，然后子进程改文件的东西，父进程读文件的时候，文件的内容跟子进程改变之后的是一样的。

回顾 lab4 中的 copy on write 机制：

当父进程或者子进程要写它们共有的东西的时候，系统会新开一页，然后把原来那页的内容复制过去，这样做能使父子进程完全相互独立，就是说子进程开始之后，父进程不能影响子进程（如果子进程不愿意的话）。

现在我们要做的事情就是如果父进程有打开文件的话，父子进程共享这个文件，无论父进程还是子进程修改这个文件，父子进程读这个文件的时候都会是改变之后的文件（仅限于文件所在内存页是这样，其它还是相互独立的）。

第二步管道：

说白了就是通过文件的共享特性实现进程之间的通信。

第三部 shell：

就是利用管道来实现一些类似 echo hello | cat 这样的命令。

第一部分：共享文件

EX1

修改 fork.c

```
duppage(envid_t envid, unsigned pn)
{
        int r;
        void *addr;
        pte_t pte;

        // LAB 4: Your code here.
//      panic("duppage not implemented");
        //panic("duppage not implemented");
        pte = vpt[pn];
        addr = (void *)(pn * PGSIZE);
        if ((pte & PTE_P) == 0 || (pte & PTE_U) == 0)
                panic("duppage: pte unpresent or non-user\n");
//      if ((pte & PTE_W) == 0 && (pte & PTE_COW) == 0) {
//              if ((r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_U)) < 0)
//                      return r;
//      }
        if( (pte & PTE_SHARE) != 0){
        //      cprintf("000000000000000000000000!!!!!!!!!!!!!!!\n");
//              cprintf("------>fork:addr:0x%x\n",addr);
                if((r = sys_page_map(0, addr, envid, addr, PTE_USER) < 0)){
                        panic("duppage sys_page_map fail!!\n");
                        return r;
                }

        }else if ((pte & PTE_W) == 0 && (pte & PTE_COW) == 0) {
                if ((r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_U)) < 0)
                        return r;
        } else {
                if ((r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_U | PTE_COW)) < 0)
                        return r;
                if ((r = sys_page_map(0, addr, 0, addr, PTE_P | PTE_U | PTE_COW)) < 0)
                        return r;
        }
        return 0;
}
```

如果是 PTE_SHARE 类型的话，映射到共同的物理页。

EX2

```
        pte_t * vpt_e = (pte_t *)vpt;
        pte_t * vpd_e = (pte_t *)vpd;
        unsigned int addr;
        addr = 0;
        for( i = 0 ; i * PTSIZE < UTOP; i++){
                if( (vpd_e[i] & PTE_P) != 0){
                        for( j = i * 1024; j < i * 1024 + 1024; j++){
                                if( (vpt_e[j] & PTE_P) != 0){
                                        if( (vpt_e[j] & PTE_SHARE) != 0){
                                                addr = i * PTSIZE + (j % 1024)  * PGSIZE;
                                                if( addr == UXSTACKTOP - PGSIZE){
                                                        continue;
                                                }
                                                if( ( r = sys_page_map(0, (void *)addr, child, (void *)addr, PTE_USER)) < 0){
                                                        return r;
                                                }
                                                cprintf("addr: 0x%x\n", addr);
                                        }
                                }
                        }
                }
        }
```

把父进程打开的文件映射到子进程里面。这里主要作用是用来作为输入和输出文件的。

EX3

修改文件服务器进程，使所有打开的文件都具有 PTE_SHARE 属性

函数 void serve_map(envid_t envid, struct Fsreq_map *rq)

```
}
if(( o -> o_mode & O_ACCMODE) == O_RDONLY)
        perm = PTE_P | PTE_U | PTE_SHARE;
else
        perm =  PTE_P | PTE_U | PTE_SHARE | PTE_W;
ipc_send(envid, 0, blk, perm);
```

函数：void serve_open(envid_t envid, struct Fsreq_open *rq)

```
        cprintf("sending success, page %08x\n", (uintptr_t)
ipc_send(envid, 0, o->o_fd, PTE_P|PTE_U|PTE_W|PTE_SHARE);
return;
```

这里好像 jos 的作者已经给出了。

EX4

请见下文

第二部分 管道

由于我是把实验做完了再开始写文档的，所以解决冲突的代码也在里面了。

先说下大概原理，这里很难讲清楚，所以最好是看实验要求文档，哪里写得很清楚（锻炼锻炼英文吧）。

流程大概如下：

父进程通过 pipe 建立两个文件，这两个文件跟其它的不一样，我们用 a，b 表示文件句柄（fd），ay，by 表示句柄映射的数据页。如果按照之前的文件系统他们会独立映射，但是现在不是，现在的情况是 ay = by，就是数据页是相同的。

子进程建立，复制共享父进程的文件，所以 a，b 句柄相同，然后他们映射的页也相同。

现在是 pageref(a) = 2, pageref(b) = 2, pageref(ay) = 4

如果 pageref(a) = 2,pageref(ay) = 2，那就表明 b 已经关闭了。我们为了让进程不会老是在等待，所以如果 b 关闭了，就会返回 0，用 a 来读的进程就会知道，不再一直等待。

但是用这样的方法来判断会由于进程切换而导致错误，具体为什么可以自己想一想，同时强烈建议看实验要求（就是 mit 那份英文的文档），写的非常详细透彻。

在 pipe.c 里面我们要完成 3 个函数，修改 2 个函数

static int _pipeisclosed(struct Fd *fd, struct Pipe *p)

```
// everybody left is what fd is.  So the other
// the pipe is closed.
int run_count;
int flag;
while(1){
        flag = 0;
        run_count = env -> env_runs;
        if( pageref(fd) == pageref(p) ){
                flag = 1;
        }
        if( run_count == env -> env_runs && flag == 1){
                return 1;
        }
        if( run_count == env -> env_runs && flag == 0){
                return 0;
        }
        if( run_count != env -> env_runs){
        //      cprintf("---------avoid interrupt---------\n");
        }
}
panic("_pipeisclosed not implemented");
return 0;
```

这里通过统计进程的运行次数来达到原子操作的效果。

为了达到这个效果，我们会在每一次运行进程的时候把 env_runs++，这样，如果在进程切换的时候执行上述函数，上述函数就会进入循环继续执行。

我们还需要改代码:（env.c env_run 函数）

```
        // LAB 3: Your code here.
//      cprintf("------------------>\n");
        curenv = e;
        curenv -> env_runs ++;
        lcr3(curenv->env_cr3);
//      cprintf("run the user programme!!\n");

//      print_trapframe(&curenv->env_tf);
//      cprintf("666666\n");
        env_pop_tf(&(curenv -> env_tf));
       // panic("env_run not yet implemented");
```

static ssize_t piperead(struct Fd *fd, void *vbuf, size_t n, off_t offset)

```
        struct Pipe *p;
        int i;
        p =  (struct Pipe *)fd2data(fd);
//      if( _pipeisclosed(fd, p) == 1){
        //      panic("piperead : _pipeisclosed\n");
//              return 0;
//      }
//      if( p -> p_rpos > p -> p_wpos){
//              panic("piperead : no contend to read\n");
//      }
        while( p-> p_rpos >= p-> p_wpos){
                if(_pipeisclosed(fd, p) == 1){
                        return 0;
                }
                sys_yield();
        }
        for( i = 0; i < n && (p -> p_rpos < p -> p_wpos); i++){
        //      if( p -> p_rpos >= p -> p_wpos){
        //              if( _pipeisclosed(fd, p) == 1){
        //                      return 0;
        //              }
        //              sys_yield();
        //      }
                ((char *)vbuf)[i] = ((char *)p -> p_buf)[p -> p_rpos % PIPEBUFSIZ];
        //      cprintf("child %d : %c\n", i, ((char )((char *)vbuf)[i]));
                p -> p_rpos ++;
        }
        return i;
//      panic("piperead not implemented");
//      return -E_INVAL;
```

这个函数我一开始没有按照作者的意思去写，结果测试用例可以过，但是 shell 的时候不能过，调了很久，后来发现是调用它的函数决定了它必须按照作者的意思去写。请看 jos 作者的代码注释。

static ssize_t pipewrite(struct Fd *fd, const void *vbuf, size_t n, off_t offset)

```
        int i;
        struct Pipe * p;
        p = (struct Pipe *)fd2data(fd);
//      if( _pipeisclosed(fd, p) == 1){
//              panic("pipewrite : pipe is close\n");
//      }
//      if( p -> p_rpos > p -> p_wpos){
//              panic("pipewrite : fail!!\n");
//      }
        for( i = 0; i < n; i++){
                while( p -> p_wpos - p -> p_rpos >= PIPEBUFSIZ){
                        if(_pipeisclosed(fd, p) == 1){
                                return 0;
                        }
                        sys_yield();
                }
                ((char *)p -> p_buf)[p -> p_wpos % PIPEBUFSIZ] = ((char *)vbuf)[i];
        //      p -> p_wpos ++;
        //      cprintf("parent %d :%c\n", i, ((char *)p -> p_buf)[p -> p_wpos % PIPEBUFSIZ]);
                p -> p_wpos ++;
        }
        return i;
//      return -E_INVAL;
```

同上

要修改的函数

static int pipeclose(struct Fd *fd)

```
static int
pipeclose(struct Fd *fd)
{
        sys_page_unmap(0, fd);
        return sys_page_unmap(0, fd2data(fd));
}
```

为了解决竞争问题，必须先调用 sys_page_unmap 函数。

要修改的函数

所在文件：fd.c

int dup(int oldfdnum, int newfdnum)

```
//      if ((r = sys_page_map(0, oldfd, 0, newfd, vpt[VPN(oldfd)] & PTE_USER)) < 0)
//              goto err;
        if (vpd[PDX(ova)]) {
                for (i = 0; i < PTSIZE; i += PGSIZE) {
                        pte = vpt[VPN(ova + i)];
                        if (pte&PTE_P) {
                                // should be no error here -- pd is already allocated
                                if ((r = sys_page_map(0, ova + i, 0, nva + i, pte & PTE_USER)) < 0)
                                        goto err;
                        }
                }
        }
    if ((r = sys_page_map(0, oldfd, 0, newfd, vpt[VPN(oldfd)] & PTE_USER)) < 0)
            goto err;
```

同样是为了解决竞争问题。

第三部分：shell

EX8

先添加一个键盘中断

添加一个中断要做三步：

trap.c

```
extern void timer();
extern void system_call();
extern void kbd_int();
```

idt_init 函数

```
SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, timer, 0);
SETGATE(idt[IRQ_OFFSET + IRQ_KBD], 0, GD_KT, kbd_int, 0);
```

trapentry.S

```
TRAPHANDLER_NOEC(timer, IRQ_OFFSET + IRQ_TIMER)
TRAPHANDLER_NOEC(kbd_int, IRQ_OFFSET + IRQ_KBD)
```

至于中断的原理前面已经讲得很多了，这里就不多说了。

EX9

sh.c

按照 jos 的设计方案，所有在 shell 运行的程序的输入都是 fd[0],输出文件都是 fd[1]。

下面是要添加的代码：

```
case '<':
        cprintf("@@@@@@@@@@
        // Input redirectio
        // Grab the filenam
        if (gettoken(0, &t)
                cprintf("syntax error: < not followed by word\n");
                exit();
        }
        // Open 't' for reading as file descriptor 0
        // (which environments use as standard input).
        // We can't open a file onto a particular descriptor,
        // so open the file as 'fd',
        // then check whether 'fd' is 0.
        // If not, dup 'fd' onto file descriptor 0,
        // then close the original 'fd'.

        // LAB 5: Your code here.
//      panic("< redirection not implemented");
        printf("---------------------------------<<\n");
        fd = open(t, O_RDWR);
        if(fd < 0){
                panic("< open fail!!\n");
        }
        if( fd == 0){
                panic("run cmd case < error!!\n");
        }
        r = dup(fd, 0);
        if( r < 0){
                panic("< dup fail!!\n");
        }
        r = close(fd);
        if( r < 0){
                panic("< close fail!!\n");
        }
        break;
```

把任意一个 fd 复制到 fd[0]，shell 将会把它作为 shell 运行的程序的输入文件。

```
case '>':
        // Output redirection
        // Grab the filename from the argument list
        cprintf("@@@@@@@@@@@@@%c@@@@@@@@@@\n",c);
        if (gettoken(0, &t) != 'w') {
                cprintf("syntax error: > not followed by word\n");
                exit();
        }
        // Open 't' for writing as file descriptor 1
        // (which environments use as standard output).
        // We can't open a file onto a particular descriptor,
        // so open the file as 'fd',
        // then check whether 'fd' is 1.
        // If not, dup 'fd' onto file descriptor 1,
        // then close the original 'fd'.

        // LAB 5: Your code here.
        panic("> redirection not implemented");
        cprintf("name : %s\n", t);
        fd = open(t, O_RDWR | O_CREAT);
        if( fd < 0){
                panic("case > open fail!!\n");
        }
        if( fd == 1){
                panic("run cmd case > error!!\n");
        }
        cprintf("case > open success!! %d\n", fd);
        cprintf("---------->\n");
        r = dup(fd, 1);
        cprintf("dup r : %d\n",r);
        if( r < 0){
                panic(" case > dup fail!!\n");
        }
        printf("case > dup success!!\n");
        close(fd);
        break;
```

把任意一个 fd 复制到 fd[1],将会作为 shell 运行的程序的输出文件。

```
case '|':
        cprintf("@@@@@@@@@@@@%c@@@@@@@@@@\n",c);
        // Pipe
        // Set up pipe redirection.

        // Allocate a pipe by calling 'pipe(p)'.
        // Like the Unix version of pipe() (man 2 pipe),
        // this function allocates two file descriptors;
        // data written onto 'p[1]' can be read from 'p[0]'.
        // Then fork.
        // The child runs the right side of the pipe:
        //      Use dup() to duplicate the read end of the pipe
        //      (p[0]) onto file descriptor 0 (standard input).
        //      Then close the pipe (both p[0] and p[1]).
        //      (The read end will still be open, as file
        //      descriptor 0.)
        //      Then 'goto again', to parse the rest of the
        //      command line as a new command.
        // The parent runs the left side of the pipe:
        //      Set 'pipe_child' to the child env ID.
        //      dup() the write end of the pipe onto
        //      file descriptor 1 (standard output).
        //      Then close the pipe.
        //      Then 'goto runit', to execute this piece of
        //      the pipeline.

        // LAB 5: Your code here.
        panic("| not implemented");
        pipe(p);
        pipe_child = fork();
        if( pipe_child == 0){
                dup(p[0], 0);
        //      pipeclose(INDEX2FD(p[0]));
        //      pipeclose(INDEX2FD(p[1]));
                close(p[0]);
                close(p[1]);
                goto again;
        } else{
```

```
                dup(p[1], 1);
        //      pipeclose(INDEX2FD(p[0]));
        //      pipeclose(INDEX2FD(p[1]));
                close(p[0]);
                close(p[1]);
                goto runit;
        }

        break;
```

这个就是管道的实现
把第一个运行的程序的输出作为第二个要运行的程序的输入。
这就是管道的实现原理。

至此，实验完成。