Lab6

> 班级：222111
>
> 学号：22373340
>
> 姓名：詹佳博

思考题

Thinking 6.1

```
#include <stdlib.h>
#include <unistd.h>

int fildes[2];
char buf[100];
int status;

int main()
{
    status = pipe(fildes);
    if (status == -1) {
        printf("error\n");
    }

    switch (fork()) {
        case -1:
            break;

        case 0:     /* 子进程 - 作为管道的读者 */
            close(fildes[0]);
            write(fildes[1], "Hello world\n", 12);
            close(fildes[1]);
            exit(EXIT_SUCCESS);

        default:    /* 父进程 - 作为管道的写者 */
            close(fildes[1]);
            read(fildes[0], buf, 100);
            printf("father-process read:%s", buf);
            close(fildes[0]);
            exit(EXIT_SUCCESS);

    }
}
```

Thinking 6.2

- 修改前：先将文件描述符(f0)进行复制(map到新地址，f1)，再将文件内容进行复制。这样就有可能 `pageref(fd)` 比 `pageref(pipe)` 先更新，调用 `dup` 后 `pageref(fd)` 会比 `pageref(pipe)` 先加1。

- 在复制文件描述符页面（如通过进程fork）后，如果在增加pipe的引用计数之前发生了时钟中断或其他形式的调度，可能会引入一个条件。这种条件可能导致另一个进程在调用`pipeisclosed`函数时，由于`pageref(fd[0])`（读端文件描述符的页面引用计数）**暂时等于**`pageref(pipe)`（pipe本身的页面引用计数），而错误地认为pipe的**读/写端**已经关闭。

Thinking 6.3

- 系统调用一定**是**原子操作。因为在进行系统调用的时候，系统陷入内核，就会关闭时钟中断。
- ```
  LEAF(msyscall)
      // Just use 'syscall' instruction and return.

      /* Exercise 4.1: Your code here. */
      syscall
      jr  ra

  END(msyscall)
  /* ---------------------------- */
  .text
  LEAF(env_pop_tf)
  .set reorder
  .set at
      mtc0    a1, CP0_ENTRYHI
      move    sp, a0
      RESET_KCLOCK
      j       ret_from_exception
  END(env_pop_tf)
  /* ---------------------------- */
  FEXPORT(ret_from_exception)

      move    a0, sp
      addiu   sp, sp, -32
      jal     do_signal
      nop
      addiu   sp, sp, 32

      RESTORE_ALL
      eret
  /* ---------------------------- */
  .macro RESET_KCLOCK
      li  t0, TIMER_INTERVAL
      /* Exercise 3.11: Your code here. */
      mtc0    zero, CP0_COUNT
      mtc0    t0, CP0_COMPARE

  .endm
  ```

Thinking 6.4

- 可以解决。若调换fd和pipe在close中的unmap顺序，使得fd引用次数的 -1 先于pipe，在两个unmap的间隙，`pageref(pipe)>pageref(fd)`仍成立，即使此时发生中断，也不会影响判断管道是否关闭的正确性。
  ```

- `dup`也会出现同样的问题，有可能有`page_ref(fd) == page_ref(pipe)`这样的情况。正确方法是先对`pipe`进行map，再对`fd`进行map即可。

Thinking 6.5

- 在`user/lib/files.c`文件中，`open`函数的实现是通过调用同文件夹下的`fsipc_open`函数来完成的。`fsipc_open`函数负责通过进程间通信（IPC）机制向文件系统服务进程发送打开文件的请求，并等待服务进程返回的文件描述符。服务进程中的`serve_open`函数会接收到这个请求，然后调用`file_open`（或类似的底层函数）来实际打开文件。一旦文件成功打开，`serve_open`函数会通过IPC将文件描述符发送回用户进程，从而实现了用户进程与文件系统服务进程之间的文件描述符共享。
- 代码如下：

```
//env.c
static void load_icode(struct Env *e, const void *binary, size_t size) {
    /* Step 1: Use 'elf_from' to parse an ELF header from 'binary'. */
    const Elf32_Ehdr *ehdr = elf_from(binary, size);
    if (!ehdr) {
        panic("bad elf at %x", binary);
    }

    /* Step 2: Load the segments using 'ELF_FOREACH_PHDR_OFF' and
'elf_load_seg'.
     * As a loader, we just care about loadable segments, so parse only
program headers here.
     */
    size_t ph_off;
    ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
        Elf32_Phdr *ph = (Elf32_Phdr *)(binary + ph_off);
        if (ph->p_type == PT_LOAD) {
            // 'elf_load_seg' is defined in lib/elfloader.c
            // 'load_icode_mapper' defines the way in which a page in this
segment
            // should be mapped.
            panic_on(elf_load_seg(ph, binary + ph->p_offset,
load_icode_mapper, e));
        }
    }

    /* Step 3: Set 'e->env_tf.cp0_epc' to 'ehdr->e_entry'. */
    /* Exercise 3.6: Your code here. */
    e->env_tf.cp0_epc = ehdr->e_entry;

}
//elfloader.c
int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void
*data) {
    u_long va = ph->p_vaddr;
    size_t bin_size = ph->p_filesz;
    size_t sgsize = ph->p_memsz;
    u_int perm = PTE_V;
    if (ph->p_flags & PF_W) {
        perm |= PTE_D;
    }
```

```
    int r;
    size_t i;
    u_long offset = va - ROUNDDOWN(va, PAGE_SIZE);
    if (offset != 0) {
        if ((r = map_page(data, va, offset, perm, bin,
                  MIN(bin_size, PAGE_SIZE - offset))) != 0) {
            return r;
        }
    }

    /* Step 1: load all content of bin into memory. */
    for (i = offset ? MIN(bin_size, PAGE_SIZE - offset) : 0; i < bin_size; i
+= PAGE_SIZE) {
        if ((r = map_page(data, va + i, 0, perm, bin + i, MIN(bin_size - i,
PAGE_SIZE))) !=
            0) {
            return r;
        }
    }

    /* Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`. */
    while (i < sgsize) {
        if ((r = map_page(data, va + i, 0, perm, NULL, MIN(sgsize - i,
PAGE_SIZE))) != 0) {
            return r;
        }
        i += PAGE_SIZE;
    }
    return 0;
}
```

- 在程序加载的过程中，当`elf_load_seg`函数处理到`.bss`段时，它不会从文件中读取数据，而是直接调用`map_page`函数来将相应的虚拟地址空间映射到物理内存页。`map_page`函数内部会进一步调用`load_icode_mapper`来执行实际的页面映射操作。
    - 由于`.bss`段的特性（未初始化的静态数据区），该函数会将新映射的页面内容初始化为零。最终，通过调用`page_insert`函数，这些页面会被添加到页表中，并设置适当的权限。整个过程中，`.bss`段的内容不需要从任何文件中加载，而是直接通过映射和初始化来准备。

Thinking 6.6

- ```
  // user/init.c
  // stdin should be 0, because no file descriptors are open yet
      if ((r = opencons()) != 0) {
          user_panic("opencons: %d", r);
      }
      // stdout
      if ((r = dup(0, 1)) < 0) {
          user_panic("dup: %d", r);
      }
  ```

- 在shell进程初始化的时候，由我们规定文件描述符0是标准输入而文件描述符1是标准输出。
  ```

Thinking 6.7

```
// sh.c
for (;;) {
        if (interactive) {
            printf("\n$ ");
        }
        readline(buf, sizeof buf);

        if (buf[0] == '#') {
            continue;
        }
        if (echocmds) {
            printf("# %s\n", buf);
        }
        if ((r = fork()) < 0) {
            user_panic("fork: %d", r);
        }
        if (r == 0) {
            runcmd(buf);
            exit();
        } else {
            wait(r);
        }
    }
```

- 除了`echocmds`和注释两种情况外，都需要`fork`一个子shell来处理输入的命令。也就是外部指令。
- cd使用次数比较多，如果每次生成多个fork子进程，会影响效率。

Thinking 6.8

```
[00002803] pipecreate
spawn ls.b: 0x2803 spawn 0x3805
spawn cat.b: 0x3004 spawn 0x4006
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00004006] destroying 00004006
[00004006] free env 00004006
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...
```

- 总共`spawn`了两次，分别是由最初被`fork`出的`2803`进程`spawn`出了`3805`进程，以及`3004`(被2803进程在`parsecmd`时`fork`得到)进程`spawn`出了`4006`进程。
- 观察到了四次进程销毁。
    - 2803进程：由主shell进程fork出来的子shell进程，用于解析并执行当前命令；

- 3004进程：由2803进程fork出来的子进程，用于解析并执行管道右端的命令；
- 3805进程：由2803进程spawn出来的子进程，用于执行管道左边的命令；
- 4006进程：由3004进程spawn出来的子进程，用于执行管道右边的命令；

难点分析

1. Exercise 6.1 的return n；要写在user_panic前面，不然总是会执行到user_panic而报错。
2. 对`spawn`函数的理解与填写，这要结合包括Lab1，Lab3，Lab4，Lab5等实验的知识点如`fork`，ELF文件的加载等进行理解；

实验体会

本次Lab难度较小，除了spawn函数需要勾连起之前很多Lab的回忆。但是也是终于完成os实验了啊啊啊啊啊啊，实验成绩虽然不尽人意，但是还是收获到了很多知识！