

# Operating System Labs Project 1

## Project 1a: A Unix Shell

### Part 1 基本架构:

在一个无限循环中打印参数 `mysh>`

然后通过 `fgets` 读入命令，并通过 `execvp` 和 `fork` 来执行

在读入命令之后通过一层层的字符串处理剖析命令类型已经是否合法性是否涉及了后台处理，批处理以及重定向。

### Part2 命令分类:

为了对不同命令进行正确处理，程序对命令分成 `built-in` 命令和非 `built-in`，其中将 `exit`, `pwd`, `cd` 和 `wait` 分类为 `built-in` 命令，对于其他命令则定义为非 `built-in` 命令。

二者的处理方式分别被放在 `exe_built_in_commands()` 函数和 `exe_command_with_child_process()` 函数中。

对于 `built-in` 命令，有如下规则：

- ① 均不需要创建 `fork` 执行（因为不涉及 `execvp`）
- ② 对于 `cd` 命令，参数至多为 2 个，对于其他命令，参数仅能为 1 个
- ③ 对于 `cd` 命令，第二个参数必须为合法路径
- ④ 对于所有 `built-in` 命令，如果调用系统函数时出错，也会报错（如 `chdir`）
- ⑤ 全部通过手动系统调用执行，而不是 `execvp`

而对于非 `built-in` 命令，规则如下：

- ① 均需要 `fork()` 创建的子程序来执行
- ② 需要检测是否涉及重定向
- ③ 需要检测是否是后台处理模式
- ④ 均通过 `execvp` 来执行，其参数存储在 `shell` 数组中

### Part 3 命令字符串处理:

我们约定：对于合法的命令，至多只有一个“>”符号，若为 0 个，则不存在重定向，否则我们认为这是一条重定向命令。也就是说，当得到一条指令时，我们首先统计其>符号的数量，在源码中通过函数 `count` 实现（自行撰写的）。

- ① 若 `count > 1`: 报错
- ② 若 `count == 1`:

我们认为这是一条重定向命令，令 `index` 等于>符号的索引，将>前的所有命令标识为 `shell`，之后的标识为 `file`，然后通过 `strtok` 来分割。

- a) 若 `file` 的段数超过 2，那么报错

- b) 若 file 的段数等于 2, 那么第二段必须是"&", 来表示是后台 job
- c) 若 file 的段数等于 1, 那么认为这一段是文件名
  - 在 b,c 合法的情况下, 进行 close 标准输出流, 然后 open 文件名来重定向, 如果 open 失败, 那么报错
- ③ 若 count == 0, 那么我们认为这不是一条重定向, 采用 strtok 来分割字符串, 全部存储在 shell 数组中

至此我们解决了重定向问题, 对 shell 数组的内容进行讨论:

- ① 如果 shell 的段数是 0, 那么说明是空白符, 无视并通过 continue 重置
- ② 如果 shell 的段数大于 0, 首先检测最后一段是否为"&", 若是, 那么标识 flag = 1, 否则为 0, 然后执行 rc = fork(), 如果 flag = 1, 那么我们认为这是一个 background job, 在 rc > 0 的时候不执行 wait, 否则我们调用 waitpid(rc, NULL) 来等待。而当输入"wait"命令时, 则调用 wait 来等待全部子进程结束。

在合法的情况下, 我们通过在(rc==0)的子进程部分调用 execvp 来执行 shell 数组的内容即可达到效果, 并根据返回值来得到是否执行成功。

#### Part4 命令行参数-批处理模式

对于批处理模式, 我们要做的首先是加入命令行参数, 根据参数个数来决定模式:

- ① argc = 1, 那么是正常模式, 从 stdin 来输入
- ② argc = 2, 则是批处理模式
  - a) 若打开 argv[1] 成功, 那么从 argv[1] 文件流来输入
  - b) 否则报错
- ③ 若 argc > 2 报错

批处理只改变输入的来源, 并且不打印"mysh"和打印命令本身, 因此并不需要改动其他部分, 只需要对命令行参数的个数进行讨论即可。

#### Project 1b: xv6 System Call

函数思想本身比较简单:

- ① 创建一个全局变量 times\_of\_read
- ② 在每次 read 被调用的时候执行 times\_of\_read++
- ③ 每当调用 getreadcount 时返回 times\_of\_read

所以主要流程有:

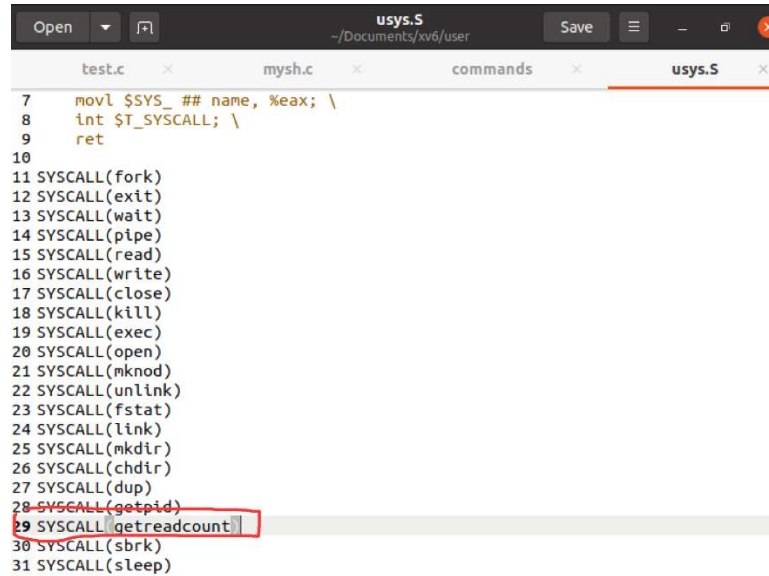
- ① 如何注册一个系统调用
- ② 如何实现计数
- ③ 如何解决限制并发的的问题

#### Part1 如何注册

- ① 首先在 user.h 和 defs.h 中定义函数:

int getreadcount(void);

- ② 进入/user/usy.S 中，在函数最后添加



```
Open  usys.S  Save  ~/Documents/xv6/user
test.c  mysh.c  commands  usys.S
7  movl $SYS_ ## name, %eax; \
8  int $T_SYSCALL; \
9  ret
10
11 SYSCALL(fork)
12 SYSCALL(exit)
13 SYSCALL(wait)
14 SYSCALL(pipe)
15 SYSCALL(read)
16 SYSCALL(write)
17 SYSCALL(close)
18 SYSCALL(kill)
19 SYSCALL(exec)
20 SYSCALL(open)
21 SYSCALL(mknod)
22 SYSCALL(unlink)
23 SYSCALL(fstat)
24 SYSCALL(link)
25 SYSCALL(mkdir)
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL getreadcount
30 SYSCALL(sbrk)
31 SYSCALL(sleep)
```

- ③ 在 sys\_call.h 后添加：

#define SYS\_getreadcount 22

- ④ 在 syscall.c 中添加

extern int sys\_getreadcount(void);  
并在 static int (\*syscall[])(void) 中添加  
[SYS\_getreadcount] sys\_getreadcount;

## Part2 如何实现计数

- ① 在 sysproc.c 中增加全局变量 int times\_of\_read
- ② 在 syscall.c 中标注 extern int sys\_getreadcount(void);
- ③ 在 sysfile.c 中添加 extern int times\_of\_read;
- ④ 在 sysfile.c 中撰写函数：
  - a) 在 sys\_read() 中添加 times\_of\_read++;
  - b) 新增函数 int getreadcount(){  
return times\_of\_read;  
}

## Part3 如何限制并发

在 Part2 的基础上

- ① 在 sysproc.c 中新增头文件 spinlock.h 来 include
- ② 在 sysproc.c 定义结构体 struct spinlock lock;
- ③ 在 sysfile.c 中引用 extern struct spinlock lock;
- ④ 在两处 times\_of\_read 的地方前后分别添加：  
acquire(&lock);  
release(&lock);

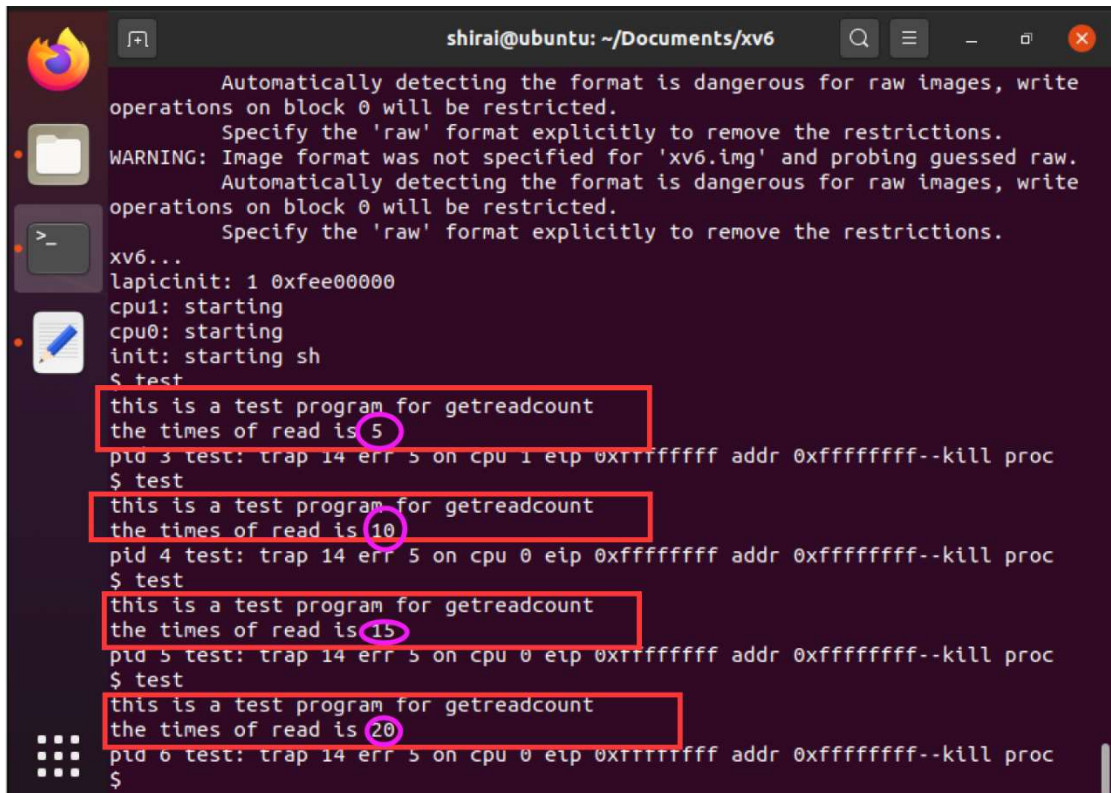
这样便可以限制在并发进程中，times\_of\_read 不会出现值的二义性  
这样便完成了该函数的注册

## Part4 简单的测试

在/user 文件夹下撰写一个 test.c, 其内容为简单的打印 getreadcount()的值:

```
#include "types.h"
#include "stat.h"
#include "user.h"
int main(void)
{
    printf("1", "This is a test program for getreadcount\n");
    printf("1", "The times of read is %d\n", getreadcount());
    return 0;
}
```

然后将其添加到 makefile 的编译文件中, 执行 qemu, 其结果如下



```
shirai@ubuntu: ~/Documents/xv6
Automatically detecting the format is dangerous for raw images, write
operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'xv6.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write
operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
xv6...
lapicinit: 1 0xfee00000
cpu1: starting
cpu0: starting
init: starting sh
$ test
this is a test program for getreadcount
the times of read is 5
pid 3 test: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc
$ test
this is a test program for getreadcount
the times of read is 10
pid 4 test: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc
$ test
this is a test program for getreadcount
the times of read is 15
pid 5 test: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc
$ test
this is a test program for getreadcount
the times of read is 20
pid 6 test: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc
$
```

可以发现, 每次执行 test 之后, times\_of\_read 的值就加 5, 所以可以得知在执行过程中使用了 5 次 read。

### 实验总结:

对于 a 部分, 个人感觉像是字符串处理的算法题+对 shell 的系统调用学习。在了解 shell 的系统调用之后, 整个程序就变成了对字符串处理的分类讨论: 如何处理空格, 如何处理换行, 在包含>的情况下检测重定向, 根据参数个数分类讨论等等, 是对失去 C++string 之后的字符练习, 学到了很多, 同时结合系统调用, 对 os 的原理有更多理解

对于 b 部分, 参考了较多博客, 函数本身的功能并不难, 但是要考虑并发的

情况下就需要对系统原理有一定了解，阅读核心源码会对后续试验有较大帮助。此外还学习了一些 `make` 的基础，是收获颇丰的一次实验。