

lab2 report

实验目的

- 学习如何使用Linux系统调用：实现一个简单的 shell
- 学习如何添加Linux系统调用：实现一个简单的 top

实验环境

- OS: Ubuntu 20.04.4 LTS
- Linux内核版本: 4.9.263

实验内容

shell

“

设置优先级为：分隔符(;) < 管道符(|) < 重定向符(>>)

内置指令

由于子进程无法修改父进程的参数，所以改变当前 shell 的参数（如 source 命令、exit 命令、kill 命令）基本都是由shell内建命令实现的。

在本实验中我们实现三个简单的内置指令 `cd`，`kill`，`exit`。我们设计 `exec_builtin()` 函数来处理内置指令，核心代码如下：

```
1 char old_path[MAX_BUF_SIZE]; // 保存上一次的工作目录
2 int exec_builtin(int argc, char**argv, int *fd) {
3     if(argc == 0)
4         return 0;
5     if (strcmp(argv[0], "cd") == 0) {
6         // "cd -" 切换到上一次的工作目录
7         if (strcmp(argv[1], "-") == 0) {
8             char tmp[MAX_BUF_SIZE];
9             getcwd(tmp, MAX_BUF_SIZE);
10            chdir(old_path);
11            strcpy(old_path, tmp);
```

```

12         return 0;
13     }
14     else {
15         getcwd(old_path, MAX_BUF_SIZE);
16         // chdir, 改变当前工作目录
17         return chdir(argv[1]);
18     }
19 }
20 else if (strcmp(argv[0], "kill") == 0) {
21     // kill, 向进程发送信号
22     if (argc == 2)
23         return kill(atoi(argv[1]), SIGTERM);
24     else if (argc == 3)
25         return kill(atoi(argv[1]), atoi(argv[2]));
26     else
27         return -1;
28 }
29 else if (strcmp(argv[0], "exit") == 0){
30     // exit, 退出程序
31     exit(0);
32 }
33 else {
34     // 不是内置指令时
35     return -1;
36 }
37 }

```

- ❶ cd 指令使用系统调用 `chdir()` 实现，直接将目标路径（可以是相对路径或绝对路径）作为参数传入即可。但不支持 `cd -` 的命令，所以需要开一个数组保存上一个工作目录（需要用到系统调用 `getcwd()`），接收到 `cd -` 指令时返回上一个工作目录。
- ❷ kill 指令使用系统调用 `kill()` 实现，将进程编号和信号量传入即可。
- ❸ exit 指令使用系统调用 `exit()` 实现。

重定向符

当指令中出现 `>`，`>>`，`<` 这三个重定向符时，需要将指令的标准输入或标准输出重定向为文件输入或输出。

我们设计 `process_redirect()` 函数处理重定向符，设置指令的输入输出，核心代码如下：

```

1 int process_redirect(int argc, char** argv, int *fd) {
2     /* 默认输入输出到命令行，即输入STDIN_FILENO, 输出STDOUT_FILENO */
3     fd[READ_END] = STDIN_FILENO;
4     fd[WRITE_END] = STDOUT_FILENO;

```

```

5     int i = 0, j = 0;
6     while(i < argc) {
7         int tfd;
8         if(strcmp(argv[i], ">") == 0) {
9             //打开输出文件从头写入
10            tfd = open(argv[i+1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
11            if(tfd < 0) {
12                printf("open '%s' error: %s\n", argv[i+1], strerror(errno));
13            }
14            else {
15                //输出重定向
16                fd[WRITE_END] = tfd;
17            }
18            i += 2;
19        }
20        else if(strcmp(argv[i], ">>") == 0) {
21            //打开输出文件追加写入
22            tfd = open(argv[i+1], O_WRONLY | O_CREAT | O_APPEND, 0666);
23            if(tfd < 0) {
24                printf("open '%s' error: %s\n", argv[i+1], strerror(errno));
25            }
26            else {
27                //输出重定向
28                fd[WRITE_END] = tfd;
29            }
30            i += 2;
31        }
32        else if(strcmp(argv[i], "<") == 0) {
33            //读输入文件
34            tfd = open(argv[i+1], O_RDONLY);
35            if(tfd < 0) {
36                printf("open '%s' error: %s\n", argv[i+1], strerror(errno));
37            }
38            else {
39                //输入重定向
40                fd[READ_END] = tfd;
41            }
42            i += 2;
43        }
44        else {
45            argv[j++] = argv[i++];
46        }
47    }
48    argv[j] = NULL;
49    return j;    // 新的argc

```

50 | }

检测重定向符种类，判断重定向类型，重定向完成后返回去除了重定向符的实际命令。

重定向的实现：使用系统调用 `open()` 打开或创建文件，设置只写或只读，重头写入或追加写入，然后进行重定向。

返回实际指令：没碰到重定向符时，直接复制过去，碰到重定向符后需要 `i+=2` 跳过两个参数，分别是重定向符和目标文件。

“

这样写是支持把重定向符写在命令开头的，跳过后可以继续往后执行。

管道

在含有管道的命令中，管道分隔的各个指令是并行的，所以内置指令也在子进程中运行，因此我们设计 `execute()` 函数，用于执行内置指令或普通指令。核心代码如下：

```
1 int execute(int argc, char** argv) {
2     int fd[2];
3     // 默认输入输出到命令行，即输入STDIN_FILENO，输出STDOUT_FILENO
4     fd[READ_END] = STDIN_FILENO;
5     fd[WRITE_END] = STDOUT_FILENO;
6     // 处理重定向符，如果不做本部分内容，请注释掉process_redirect的调用
7     argc = process_redirect(argc, argv, fd);
8     if(exec_builtin(argc, argv, fd) == 0) {
9         exit(0);
10    }
11    // 将标准输入输出STDIN_FILENO和STDOUT_FILENO修改为fd对应的文件
12    dup2(fd[READ_END], STDIN_FILENO);
13    dup2(fd[WRITE_END], STDOUT_FILENO);
14    // execvp, 执行命令
15    execvp(argv[0], argv);
16    exit(1);
17 }
```

先处理重定向符，然后使用系统调用 `execvp()` 执行指令。

然后我们设计一个函数 `run_command()` 用来执行可能带有 `|`、`>>` 等重定向符的一个或多个指令（以分号分隔），核心代码如下：

```
1 // 运行一个包含管道或重定向符的命令
2 int run_command(char *cmdline) {
3     /* 由管道操作符'|'分割的命令行各个部分，每个部分是一条命令 */
4     /* 拆解命令行 */
5     char *commands[128];
```

```

6   int cmd_count = split_string(cmdline, "|", commands);
7   if(cmd_count == 0)
8       return 0;
9   else if(cmd_count == 1) {        // 没有管道的单一命令
10      char *argv[MAX_CMD_ARG_NUM];
11      int argc;
12      int fd[2];
13      // 处理参数, 分出命令名和参数
14      argc = split_string(commands[0], " ", argv);
15      if(exec_builtin(argc, argv, fd) == 0)
16          return 0;
17      // 创建子进程, 运行命令, 等待命令运行结束
18      int pid = fork();
19      if(pid == 0)
20          execute(argc, argv);
21      else    wait(NULL);
22  }
23  else { // 多管道, 包含一个
24      int read_fd;    // 上一个管道的读端口 (出口)
25      for(int i = 0; i < cmd_count; i++) {
26          int pipefd[2];
27          /* 创建管道, n条命令只需要n-1个管道 */
28          if(i != cmd_count - 1) {
29              int ret = pipe(pipefd);
30              if(ret < 0) {
31                  printf("pipe error!\n");
32                  continue;
33              }
34          }
35          int pid = fork();
36          if(pid == 0) {
37              /* 除了最后一条命令外, 都将标准输出重定向到当前管道入口 */
38              if(i != cmd_count - 1) {
39                  dup2(pipefd[WRITE_END], STDOUT_FILENO);
40                  close(pipefd[WRITE_END]);
41              }
42              /* 除了第一条命令外, 都将标准输入重定向到上一个管道出口 */
43              if(i != 0) {
44                  dup2(read_fd, STDIN_FILENO);
45                  close(read_fd);
46              }
47              char *argv[MAX_CMD_ARG_NUM];
48              int argc = split_string(commands[i], " ", argv);
49              execute(argc, argv);
50              exit(255);

```

```

51     }
52     /* 父进程除了第一条命令，都需要关闭当前命令用完的上一个管道读端口
53     * 父进程除了最后一条命令，都需要保存当前命令的管道读端口
54     * 记得关闭父进程没用的管道写端口 */
55     if(i != 0) {
56         close(read_fd);
57     }
58     if(i != cmd_count - 1) {
59         read_fd = pipefd[READ_END];
60         close(pipefd[WRITE_END]);
61     }
62 }
63 // 等待所有子进程结束
64 while (wait(NULL) > 0);
65 }
66 }

```

先以 `|` 为分隔符拆解命令行，如果不存在管道符，那么直接作为普通命令执行：如果是内置指令，就在父进程执行，如果是其他指令，就 `fork()` 一个子进程执行前面封装好的 `execute()` 函数。

如果存在管道符，那么需要根据管道符的数量多开子进程，子进程间并行。使用系统调用 `dup2()` 来重定向输入输出，除了最后一条命令外，都将标准输出重定向到当前管道入口，除了第一条命令外，都将标准输入重定向到上一个管道出口。然后调用 `execute()` 执行。

最后等待所有子进程结束，命令运行结束。

多命令

由 `;` 分隔的多个子命令，和在多行中依次运行这些子命令效果相同。

```

1 // 多命令执行
2 cmd_count = split_string(cmdline, ";", commands);
3 for (int i = 0; i < cmd_count; i++) {
4     run_command(commands[i]);
5 }

```

直接依次运行即可。

top

注册

内核的汇编代码会在 `syscalls_64.h` 中查找调用号。为便于添加系统调用，x86平台提供了一个专门用来注册系统调用的文件 `syscall_64.tbl`。在编译时，脚本 `syscalltbl.sh` 会被运行，将上述 `syscall_64.tbl` 文件中登记过的系统调用都生成到前面的 `syscalls_64.h` 文件中。因此我们需要修改 `syscall_64.tbl`。

打开 `linux-4.9.263/arch/x86/entry/syscalls/syscall_64.tbl`，加入我们要注册的系统调用名字和编号即可，代码如下：

```
1 332 common mytop sys_mytop
```

声明

打开 `linux-4.9.263/include/linux/syscalls.h`，里面是对于系统调用函数原型的定义，在最后面加上我们创建的新的系统调用函数原型。

代码如下：

```
1 asmlinkage long sys_mytop(int __user * num, pid_t __user * pid, char __user * name,
    int __user * state, u64 __user * runtime);
```

“

如果传入了用户空间的地址，需要加入 `__user` 宏来说明。

我们要传入 5 个参数，都是用户空间的地址，存储函数的返回值。第一个是进程个数，第二个是 pid 数组，第三个是进程名的数组，第四个是进程状态的数组，第五个是总运行时间的数组。

“

进程名数组实际上是二维数组，我们用一维数组实现，固定第二维为进程名的最大长度 $15 + 1 = 16$ ，每存储一个进程名就 `name += 16`，偏移到下一个存储地址。

实现

在 `linux-4.9.264/kernel/sys.c` 代码的最后添加你自己的函数定义，用 `SYSCALL_DEFINE()` 宏实现。

代码如下：

```
1 SYSCALL_DEFINE5(mytop, int __user *, num, pid_t __user *, pid, char __user *, name,
2   int __user *, state, u64 __user *, runtime)
3 {
4     struct task_struct* task;
5     int count = 0;
6     pid_t pid_temp;
7     char name_temp[16];
8     int state_temp;
```

```

8      u64 runtime_temp;
9      printk("[Syscall] mytop\n");
10     printk("[StdID] PB21111715\n");
11     for_each_process(task){
12         pid_temp = task->pid;
13         strcpy(name_temp, task->comm);
14         state_temp = task->state;
15         runtime_temp = task->se.sum_exec_runtime;
16         if (copy_to_user(pid++, &pid_temp, sizeof(pid_t)))
17             return -1;
18         if (copy_to_user(name, name_temp, sizeof(char)*16))
19             return -1;
20         if (copy_to_user(state++, &state_temp, sizeof(int)))
21             return -1;
22         if (copy_to_user(runtime++, &runtime_temp, sizeof(u64)))
23             return -1;
24         name += 16;
25         count++;
26     }
27     if (copy_to_user(num, &count, sizeof(int)))
28         return -1;
29     return 0;
30 }

```

调用 `printk()` 函数输出系统调用名字和我的学号；调用 `for_each_process()` 函数遍历所有进程，将需要的信息 copy 到用户空间，然后返回。

测试

用 C 语言编写测试代码，实现类似 `top` 指令的功能，定时调用上面写的系统调用 `mytop`，然后计算各进程 cpu 占用率，并按降序排列，打印出前 20 个进程。

代码如下：

```

1  #define MAX 1000
2  int main(int argc, char *argv[]) {
3      int num = 0;
4      __pid_t pid[MAX];
5      char name[MAX][16];
6      int state[MAX];
7      __U64_TYPE runtime[MAX];
8      int num_old = 0;    // 上一次的进程数
9      __pid_t pid_old[MAX];
10     __U64_TYPE runtime_old[MAX];    // 保存旧进程的 pid 和 runtime
11     double cpu_usage[MAX];

```



```

12     int table[MAX]; // 按占用率排序后的映射表
13     int interval;
14     if (argc == 1)
15         interval = 1;
16     else
17         interval = atoi(argv[1]);
18     syscall(332, &num_old, pid_old, name, state, runtime_old);
19     while(1) {
20         system("clear");
21         syscall(332, &num, pid, name, state, runtime);
22         for (int i = 0; i < num; i++) {
23             int j;
24             for (j = 0; j < num_old; j++) {
25                 if (pid[i] == pid_old[j]) { // 保证同一进程
26                     cpu_usage[i] = (runtime[i] - runtime_old[j]) / (10000000.0 *
interval);
27                     break;
28                 }
29             }
30             if (j >= num_old)
31                 cpu_usage[i] = runtime[i] / (10000000.0 * interval); // 新进程
32             table[i] = i;
33         }
34         num_old = num; // 更新老进程
35         for (int i = 0; i < num; i++) {
36             pid_old[i] = pid[i];
37             runtime_old[i] = runtime[i];
38         }
39         // 排序获得占用率前20, 第一个元素是占用率最大的
40         printf("PID\t \tNAME\t \tSTATE\t \tCPU_USAGE(%) \tRUNTIME(ns) \n");
41         for (int i = 0; i < 20; i++) {
42             for (int j = i + 1; j < num; j++) {
43                 if (cpu_usage[table[i]] < cpu_usage[table[j]]) {
44                     int tmp = table[i];
45                     table[i] = table[j];
46                     table[j] = tmp;
47                 }
48             }
49             printf("%-8d \t%-16s %-8d \t%-8lf \t%-8lu \n", pid[table[i]],
name[table[i]], !state[table[i]], cpu_usage[table[i]], runtime[table[i]]);
50         }
51         sleep(interval);
52     }
53     return 0;
54 }

```

通过编号调用 `mytop()` 获得所有进程的相关信息，然后分别计算各进程的 `cpu` 占用率(实际运行时间 / 刷新间隔)，再通过交换排序(选择排序)获得占用率前 20，最后打印输出。

“

定义映射表 `table[]`，不用实际交换进程信息所在位置，交换下标即可，可以减少时间开销。

“

注意：计算时要保证是同一个进程，因为下标会变

实验结果

测试 shell

编译

```

问题 输出 调试控制台 终端 端口
● ningli@liano:~/oslab/lab2$ gcc -o shell shell.c
○ ningli@liano:~/oslab/lab2$

```

运行

```

问题 输出 调试控制台 终端 端口
● ningli@liano:~/oslab/lab2$ gcc -o shell shell.c
○ ningli@liano:~/oslab/lab2$ ./shell
shell:/home/ningli/oslab/lab2 ->

```

测试

单命令、单管道、重定向符

```

问题 输出 调试控制台 终端 端口
● ningli@liano:~/oslab/lab2$ gcc -o shell shell.c
● ningli@liano:~/oslab/lab2$ ./shell
shell:/home/ningli/oslab/lab2 -> ls -la
.  ..  get_ps_info.c  lab2.pdf  lab2.sh  mytop  out  qemu.sh  shell  shell.c  .vscode
shell:/home/ningli/oslab/lab2 -> ps aux | wc -l
319
shell:/home/ningli/oslab/lab2 -> ps aux > out
shell:/home/ningli/oslab/lab2 -> cd ..
shell:/home/ningli/oslab -> cd ..
shell:/home/ningli/oslab/lab2 -> exit
○ ningli@liano:~/oslab/lab2$

```

多命令、多管道

```

问题  输出  调试控制台  终端  端口
ningli@liano:~/oslab/lab2$ ./shell
shell:/home/ningli/oslab/lab2-> ps aux | grep ningli | wc -l
94
shell:/home/ningli/oslab/lab2 -> echo hello > a > b ; cd .. ; pwd ; cd ./lab2 ; pwd
/home/ningli/oslab
shell:/home/ningli/oslab/lab2 -> cat a
hello
shell:/home/ningli/oslab/lab2 -> ps aux | grep ningli > out | wc -l ; exit
0
ningli@liano:~/oslab/lab2$

```

kill

默认参数

```

ningli      3862  0.0  0.1 4951692 14968 ?        Sl   12:20   0:00 /home/ningli/.vscode-server/extensions/ms-vscode.cpptoc
root        4063  0.0  0.0      0      0 ?        I    12:21   0:00 [kworker/u256:0-events_unbound]
root        4768  0.0  0.0      0      0 ?        I    12:27   0:00 [kworker/u256:1-events_power_efficient]
ningli      4823  0.0  0.0  19524  5224 pts/10    Ss   12:28   0:00 /bin/bash --init-file /home/ningli/.vscode-server/bin/7
ningli      4887  0.0  0.0  16716   564 ?        S    12:28   0:00 sleep 180
ningli      4888  0.0  0.0  16716   584 ?        S    12:28   0:00 sleep 180
root        4973  0.0  0.0      0      0 ?        I    12:29   0:00 [kworker/3:0-cgroup_destroy]
ningli      5005  5.1  1.4 1260440 121024 ?        Sl   12:29   0:00 evince /home/ningli/oslab/lab2/lab2.pdf
ningli      5013  0.0  0.0  156060  5312 ?        Sl   12:29   0:00 /usr/libexec/evince
ningli      5026  0.0  0.0   2500   512 pts/10    S+   12:29   0:00 ./shell
ningli      5042  0.0  0.0  20144  3300 pts/10    R+   12:29   0:00 ps aux
shell:/home/ningli/oslab/lab2 -> kill 5005
shell:/home/ningli/oslab/lab2 ->

```

强制结束

```

ningli      4633  0.0  0.0  16716   576 ?        S    12:25   0:00 sleep 180
ningli      4634  0.0  0.0  16716   584 ?        S    12:25   0:00 sleep 180
root        4768  0.0  0.0      0      0 ?        I    12:27   0:00 [kworker/u256:1-events_power_efficient]
ningli      4808  1.9  1.6 1105864 135392 ?        Sl   12:27   0:00 evince /home/ningli/oslab/lab2/lab2.pdf
ningli      4814  0.0  0.0  156060  5332 ?        Sl   12:27   0:00 /usr/libexec/evince
ningli      4823  0.0  0.0  19524  5220 pts/10    Ss   12:28   0:00 /bin/bash --init-file /home/ningli/.vscode-server/bin/7
ningli      4840  0.0  0.0   2500   512 pts/10    S+   12:28   0:00 ./shell
ningli      4845  0.0  0.0   2616   596 ?        S    12:28   0:00 /bin/sh -c "/home/ningli/.vscode-server/bin/7
ningli      4846  0.0  0.0  18132  3216 ?        S    12:28   0:00 /bin/bash /home/ningli/.vscode-server/bin/7
ningli      4850  0.0  0.0  16716   580 ?        S    12:28   0:00 sleep 1
ningli      4851  0.0  0.0  20144  3312 pts/10    R+   12:28   0:00 ps aux
shell:/home/ningli/oslab/lab2 -> kill 4808 9
shell:/home/ningli/oslab/lab2 ->

```

测试 mytop

编译

为了方便测试，我写了一个编译代码和 linux 内核的脚本，代码如下：

```

1  #!/bin/bash
2  gcc -static -o mytop get_ps_info.c
3  sudo cp mytop ~/oslab/busybox-1.32.1/_install
4  cd ~/oslab/busybox-1.32.1/_install
5  find . -print0 | cpio --null -ov --format=newc | gzip -9 > ~/oslab/initramfs-busybox-
x64.cpio.gz
6  cd ~/oslab/linux-4.9.263
7  make -j $((`nproc`-1))

```

运行结果如下：

```

ningli@liano: ~/oslab/lab2
ningli@liano:~/oslab/lab2$ sh lab2.sh
[sudo] ningli 的密码:
./bin
./bin/mktemp
./bin/true
./bin/netstat
./bin/mkdir
./bin/grep
./bin/nice
./bin/uname

```

运行

还有一个启动 qemu 的脚本，代码如下：

```

1  #!/bin/bash
2  qemu-system-x86_64 -kernel ~/oslab/linux-4.9.263/arch/x86_64/boot/bzImage -initrd
~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr root=/dev/ram init=/init"

```

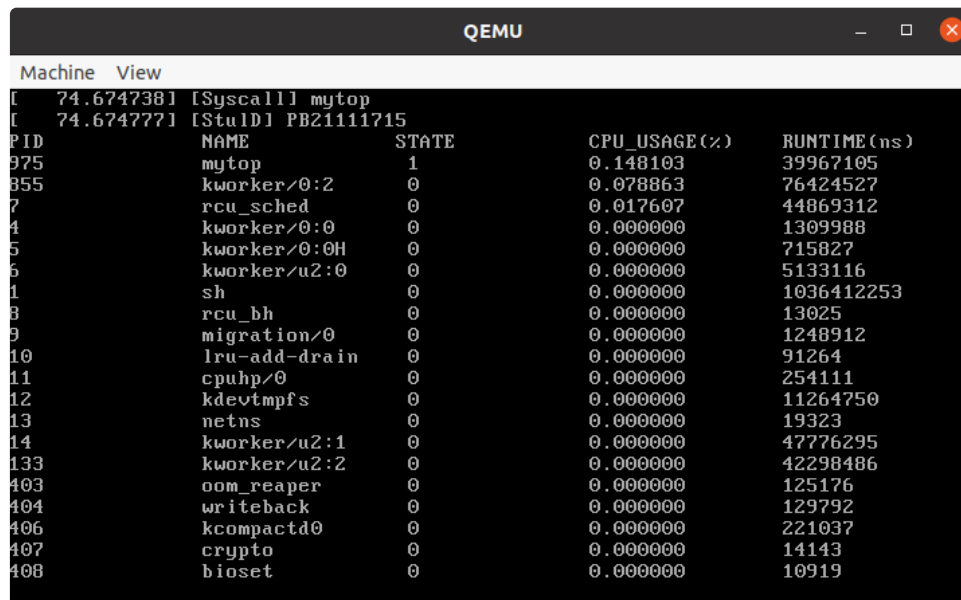
运行结果如下：

```

This boot took 1.55 seconds
/bin/sh: can't access tty; job control turned off
/ # [ 1.760713] tsc: Refined TSC clocksource calibration: 2304.555 MHz
[ 1.760992] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x21380432
081, max_idle_ns: 440795280577 ns
[ 1.933155] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input3
[ 2.770247] clocksource: Switched to clocksource tsc
CHK include/generated/uapi/linux/version.h
CHK include/generated/utsrelease.h
CHK scripts/mod/devicetable-offsets.h
CHK include/generated/timeconst.h
CHK include/generated/bounds.h
CHK include/generated/asm-offsets.h
CALL scripts/checksyscalls.sh
CHK include/generated/compile.h
Building modules, stage 2.
MODPOST 18 modules
Kernel: arch/x86/boot/bzImage is ready (#7)
ningli@liano:~/oslab/lab2$ sh qemu.sh

```

测试



```
Machine View
[ 74.674738] [Syscall] mytop
[ 74.674777] [StdIn] PBZ1111715
PID      NAME          STATE      CPU_USAGE(%)  RUNTIME(ns)
975      mytop          1          0.148103      39967105
855      kworker/0:2    0          0.078863      76424527
7        rcu_sched      0          0.017607      44869312
4        kworker/0:0    0          0.000000      1309988
5        kworker/0:0H   0          0.000000      715827
6        kworker/u2:0   0          0.000000      5133116
1        sh             0          0.000000      1036412253
8        rcu_bh         0          0.000000      13025
9        migration/0    0          0.000000      1248912
10       lru-add-drain  0          0.000000      91264
11       cpuhp/0        0          0.000000      254111
12       kdevtmpfs      0          0.000000      11264750
13       netns          0          0.000000      19323
14       kworker/u2:1   0          0.000000      47776295
133      kworker/u2:2   0          0.000000      42298486
403      oom_reaper     0          0.000000      125176
404      writeback      0          0.000000      129792
406      kcompactd0     0          0.000000      221037
407      crypto         0          0.000000      14143
408      bioset         0          0.000000      10919
```

实验总结

收获

- 学会了系统调用的使用和 shell 的原理
- 学会了系统调用的注册和实现

建议

难度适中，无建议