

# 实验二 添加 Linux 系统调用

## 一、实验目的

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

## 二、实验环境

OS: Ubuntu 18.04  
Linux 内核版本: 4.9.263

## 三、实验步骤

### 3.1 编写系统调用实现一个 Linux ps

#### 3.1.1 注册系统调用

打开 linux-4.9.263/arch/x86/entry/syscalls/syscall\_64.tbl, 在文件中添加系统调用注册

syscall\_64.tbl

~/oslab/linux-4.9.263/arch/x86/entry/syscalls

保存(S)

316	common	renameat2	sys_renameat2
317	common	seccomp	sys_seccomp
318	common	getrandom	sys_getrandom
319	common	memfd_create	sys_memfd_create
320	common	kexec_file_load	sys_kexec_file_load
321	common	bpf	sys_bpf
322	64	execveat	sys_execveat/ptregs
323	common	userfaultfd	sys_userfaultfd
324	common	membarrier	sys_membarrier
325	common	mlock2	sys_mlock2
326	common	copy_file_range	sys_copy_file_range
327	64	preadv2	sys_preadv2
328	64	pwritev2	sys_pwritev2
329	common	pkey_mprotect	sys_pkey_mprotect
330	common	pkey_alloc	sys_pkey_alloc
331	common	pkey_free	sys_pkey_free
332	common	ps_counter	sys_ps_counter
333	common	ps_info	sys_ps_info

#  
# x32-specific system call numbers start at 512 to avoid cache impact  
# for native 64-bit operation.  
#  
512 x32 rt\_sigaction compat\_sys\_rt\_sigaction  
513 x32 rt\_sigreturn sys32\_x32\_rt\_sigreturn  
514 x32 ioctl compat\_sys\_ioctl  
515 x32 readv compat\_sys\_readv

纯文本 制表符宽度: 8 第 341 行, 第 55 列 插入

#### 3.1.2 定义函数原型

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, 在最后面加上我们要创建的新系统调用函数原型, 格式为 `asmlinkage long sys_xxx(...)` , 注意如果传入了用户空间的地址, 需要加入 `__user` 宏来说明。

```
syscalls.h
~/oslab/linux-4.9.263/include/linux

asmlinkage long sys_kcmp(ptid_t ptid1, ptid_t ptid2, int type,
                        unsigned long idx1, unsigned long idx2);
asmlinkage long sys_finit_module(int fd, const char __user *uargs, int flags);
asmlinkage long sys_seccomp(unsigned int op, unsigned int flags,
                        const char __user *uargs);
asmlinkage long sys_getrandom(char __user *buf, size_t count,
                        unsigned int flags);
asmlinkage long sys_bpf(int cmd, union bpf_attr *attr, unsigned int size);

asmlinkage long sys_execveat(int dfd, const char __user *filename,
                        const char __user *const __user *argv,
                        const char __user *const __user *envp, int flags);

asmlinkage long sys_membarrier(int cmd, int flags);
asmlinkage long sys_copy_file_range(int fd_in, loff_t __user *off_in,
                        int fd_out, loff_t __user *off_out,
                        size_t len, unsigned int flags);

asmlinkage long sys_mlock2(unsigned long start, size_t len, int flags);

asmlinkage long sys_pkey_mprotect(unsigned long start, size_t len,
                        unsigned long prot, int pkey);
asmlinkage long sys_pkey_alloc(unsigned long flags, unsigned long init_val);
asmlinkage long sys_pkey_free(int pkey);
asmlinkage long sys_ps_counter(int __user * num);
asmlinkage long sys_ps_info(int __user * num, int __user * pid, long long int __user * time);
#endif

C/C++/ObjC 头文件 制表符宽度: 8 第 907 行, 第 2 列 插入
```

### 3.1.3 实现函数

linux-4.9.264/kernel/sys.c 代码的最后添加你自己的函数

在获取进程运行的时间信息时,会遇到两个变量,分别是结构体中的 utime 和 stime。查阅资料得知,stime 是系统时间,即进程在内核模式下花费的时间,而 utime 是在用户模式下花费的时间.这些值取决于该特定过程的安排.没有为其更新定义此类间隔.随着各个模式中的时间花费的变化,它们会快速更新。所以进程运行的总时间应该是 stime+utime.

需要注意的是,通过系统调用获取的信息需要使用 copy\_to\_user()函数复制到用户空间的变量中,才可以在用户空间访问其内容。在具体实现的过程中,我使用一个整型变量存储进程数量,分别使用一个数组储存 PID 和进程运行时间,需要注意不同类型指针的正确使用,这在实际的编程实现中给我带来了不小的麻烦和困扰。

```
sys.c
~/oslab/linux-4.9.263/kernel

printf("[syscall] ps_counter\n"),
for_each_process(task){
    counter++;
}
copy_to_user(num, &counter, sizeof(int));
return 0;
}
SYSCALL_DEFINE3(ps_info, int __user *, num, int __user *, pid, long long int __user *, time){
    struct task_struct* task;
    int counter = 0;
    int i = 0;
    int pid_temp[100];
    long long int time_temp[100];
    printf("[syscall] ps_info\n");
    for_each_process(task)
    {
        pid_temp[counter] = task -> pid;
        time_temp[counter] = task -> utime + task -> stime;
        copy_to_user(&pid[counter], &pid_temp[counter], sizeof(int));
        copy_to_user(&time[counter], &time_temp[counter], sizeof(long long int));
        counter++;
    }
    copy_to_user(num, &counter, sizeof(int));
    return 0;
}
#endif /* CONFIG_COMPAT */

C 制表符宽度: 8 第 2478 行, 第 56 列 插入
```

### 3.1.4 编写测试程序

编写程序 ps\_info.c 用以测试

```
打开(O)  ps_info.c 保存(S)
#include<unistd.h>
#include<sys/syscall.h>
#include<stdio.h>
int main(void)
{
    int num;
    int i=0;
    int pid[100];
    long long int time[100];
    syscall(333, &num,pid,time);
    printf("process number is %d\n",num);
    printf("PID      TIME\n");
    for(i=0;i<num;i++)
    {
        printf("%-8d%lld\n",pid[i],time[i]);
    }
    return 0;
}
```

### 3.1.4 编译运行程序

按照文档步骤依次执行，得到运行结果

```
QEMU
[ 2.325023] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input3
[ 3.027895] clocksource: Switched to clocksource tsc

/ # ./ps_info
[ 11.578355] [Syscall] ps_info
process number is 50
PID      TIME
1         1141
2         5
3         3
4         1
5         0
6         29
7         11
8         0
9         1
10        0
11        0
12        15
13        0
14        66
310       2
403       0
404       0
```

## 3.2 熟悉 Linux 下系统调用

根据提供的文档和代码框架，可以初步实现单条命令的运行，并支持提示符输出当前路径；支持两条命令之间的管道，以及 cd 和 exit 内置指令；支持多条命令间的管道。

### 3.2.1 思路分析

#### 3.2.1.1 单条命令

内置命令：shell 主进程中执行

外部命令：fork 一个新进程，并使用 exec 系函数完成

#### 3.2.1.2 单管道的实现

内置命令：fork 一个新进程并处理

标记 a|b，流程为：

1. 父进程创建管道，这个管道父进程和所有子进程共享；

2. Fork 进程 a，并把 a 的标准输出改为输出到管道的写端，执行（含内置命令）
3. Fork 进程 b，并把 b 的标准输入改为从管道的读端读取，执行（含内置命令）
4. 等待所有子进程执行结束

### 3.2.1.3 多条命令间管道的实现

1. 创建管道，n 条命令只需要 n-1 条管道，所以有一次循环是不需要创建管道的
2. 除了最后一条命令外，都将标准输出重定向到当前管道写端
3. 除了第一条命令外，都将标准输入重定向到上一个管道的读端
4. 等待所有子进程执行结束

### 3.2.2 使用的系统调用 API

在编写的过程中使用了如下 API：

```
int chdir(const char *path);
```

chdir 是 C 语言中的一个系统调用函数（同 cd），用于改变当前工作目录，其参数为 Path 目标目录，可以是绝对目录或相对目录。返回值：成功返回 0，失败返回 -1

```
int execvp(const char* file, const char* argv[]);
```

- (1) 第一个参数是要运行的文件，会在环境变量 PATH 中查找 file，并执行。
- (2) 第二个参数，是一个参数列表，如同在 shell 中调用程序一样，参数列表为 0, 1, 2, 3……
- (3) argv 列表最后一个必须是 NULL。
- (4) 失败会返回 -1，成功无返回值，但是，失败会在当前进程运行，执行成功后，直接结束当前进程，可以在子进程中运行。

```
char *getcwd( char *buffer, int maxlen );
```

功能：获取当前工作目录

参数说明：getcwd()会将当前工作目录的绝对路径复制到参数 buffer 所指的内存空间中,参数 maxlen 为 buffer 的空间大小。

返回值：成功则返回当前工作目录，失败返回 **FALSE**。

```
int pipe(int fd[2]);
```

pipe 函数定义中的 fd 参数是一个大小为 2 的一个数组类型的指针。该函数成功时返回 0，并将一对打开的文件描述符值填入 fd 参数指向的数组。失败时返回 -1 并设置 errno。

通过 pipe 函数创建的这两个文件描述符 fd[0] 和 fd[1] 分别构成管道的两端，往 fd[1] 写入的数据可以从 fd[0] 读出。并且 fd[1] 一端只能进行写操作，fd[0] 一端只能进行读操作，不能反过来使用。要实现双向数据传输，可以使用两个管道。

```
int dup2(int oldfd, int newfd);
```

若参数 newfd 已经被程序使用，则系统就会将 newfd 所指的文件关闭，若 newfd 等于 oldfd，则返回 newfd，而不关闭 newfd 所指的文件。dup2 所复制的文件描述符与原来的文件描述符共享各种文件状态。共享所有的锁定，读写位置和各项权限或 flags 等。

返回值：

若 dup2 调用成功则返回新的文件描述符，出错则返回-1.

### 3.2.3shell 实现

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <string.h>
5.  #include <sys/wait.h>
6.  #include <sys/types.h>
7.
8.  #define MAX_CMDLINE_LENGTH  1024    /* max cmdline length in a line*/
9.  #define MAX_BUF_SIZE        4096    /* max buffer size */
10. #define MAX_CMD_ARG_NUM     32      /* max number of single command args */
11. #define WRITE_END 1          // pipe write end
12. #define READ_END 0          // pipe read end
13.
14. int split_string(char* string, char *sep, char** string_clips) {
15.
16.     char string_dup[MAX_BUF_SIZE];
17.     string_clips[0] = strtok(string, sep);
18.     int clip_num=0;
19.
20.     do {
21.         char *head, *tail;
22.         head = string_clips[clip_num];
23.         tail = head + strlen(string_clips[clip_num]) - 1;
24.         while(*head == ' ' && head != tail)
25.             head ++;
26.         while(*tail == ' ' && tail != head)
27.             tail --;
28.         *(tail + 1) = '\0';
29.         string_clips[clip_num] = head;
30.         clip_num ++;
31.     }while(string_clips[clip_num]=strtok(NULL, sep));
32.     return clip_num;
33. }
34.
35. /*
36.  执行内置命令
37.  arguments:
38.      argc: 命令的参数个数
39.      argv: 依次代表每个参数，注意第一个参数就是要执行的命令，
40.      若执行"ls a b c"命令，则 argc=4, argv={"ls", "a", "b", "c"}
41.  return:
```

```
42.         int, 若执行成功返回 0, 否则返回值非零
43.     */
44.     int exec_builtin(int argc, char**argv) {
45.         if(argc == 0) {
46.             return 0;
47.         }
48.         /* TODO: 添加和实现内置指令 */
49.
50.         if (strcmp(argv[0], "cd") == 0)
51.         {
52.             if(argc > 2)
53.             {
54.                 printf("ERROR!\n");
55.                 return 0;
56.             }
57.             if(chdir(argv[1]) == -1)    //更改当前工作目录。 参数: Path 目标目录,
                                         可以是绝对目录或相对目录。 返回值: 成功返回 0 , 失败返回-1
58.             {
59.                 printf("Destination directory does not exist\n");
60.                 return 0;
61.             }
62.         }
63.         else
64.         {
65.             if (strcmp(argv[0], "pwd") == 0)
66.             {
67.                 return 0;
68.             }
69.             else
70.             {
71.                 if (strcmp(argv[0], "exit") == 0)
72.                 {
73.                     exit(0);
74.                 }
75.                 else
76.                 { // 不是内置指令时
77.                     return -1;
78.                 }
79.             }
80.         }
81.     }
82.
83.     /*
84.     在本进程中执行, 且执行完毕后结束进程。

```

```

85.         arguments:
86.             argc: 命令的参数个数
87.             argv: 依次代表每个参数，注意第一个参数就是要执行的命令，
88.                 若执行"ls a b c"命令，则 argc=4, argv={"ls", "a", "b", "c"}
89.         return:
90.             int, 若执行成功则不会返回（进程直接结束），否则返回非零
91.     */
92.     int execute(int argc, char** argv) {
93.         if(exec_builtin(argc, argv) == 0) {
94.             exit(0);
95.         }
96.         /* TODO:运行命令 */
97.         pid_t pid;
98.         pid = fork();
99.         if(pid == 0)
100.        {
101.            if(execvp(argv[0], argv) == -1)
102.            { //execvp 有两个参数：要运行的程序名和那个程序的命令行参数。
103.                exit(-1);
104.            }
105.            exit(0);
106.        }
107.        else
108.        {
109.            if(pid > 0)
110.            {
111.                wait(NULL);
112.                return 0;
113.            }
114.        }
115.        return -1;
116.    }
117.
118.     int main() {
119.         /* 输入的命令行 */
120.         char cmdline[MAX_CMDLINE_LENGTH];
121.
122.         /* 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 */
123.         char *commands[128];
124.         int cmd_count;
125.         while (1) {
126.             /* TODO:增加打印当前目录，格式类似"shell:/home/oslab ->", 你需要改下面的 printf */
127.             char current_working_directory[256];

```

```

128.         getcwd(current_working_directory, 256);
129.         printf("shell: %s -> ", current_working_directory);
130.         fflush(stdout);
131.
132.         fgets(cmdline, 256, stdin);
133.         strtok(cmdline, "\n");
134.
135.         /* 拆解命令行 */
136.         cmd_count = split_string(cmdline, "|", commands);
137.
138.         if(cmd_count == 0) {
139.             continue;
140.         } else if(cmd_count == 1) {      // 没有管道的单一命令
141.             char *argv[MAX_CMD_ARG_NUM];
142.             /* TODO:处理参数，分出命令名和参数 */
143.             int argc = split_string(commands[0], " ", argv);
144.
145.             /* 在没有管道时，内建命令直接在主进程中完成，外部命令通过创建子进程完
成 */
146.             if(exec_builtin(argc, argv) == 0) {
147.                 continue;
148.             }
149.             /* TODO:创建子进程，运行命令，等待命令运行结束 */
150.             pid_t pid;
151.             pid = fork();
152.             if(pid == 0)
153.             {
154.                 if(execute(argc, argv) != 0)
155.                 {
156.                     exit(-1);
157.                 }
158.             }
159.             else
160.             {
161.                 if(pid > 0)
162.                 {
163.                     wait(NULL);
164.                     exit(0);
165.                 }
166.             }
167.
168.         } else if(cmd_count == 2) {      // 两个命令间的管道
169.             int pipefd[2];
170.             int ret = pipe(pipefd);

```



```

171.         if(ret < 0) {
172.             printf("pipe error!\n");
173.             continue;
174.         }
175.         // 子进程 1
176.         //fork 进程 a,并把 a 的标准输出改为输出到管道的写端,执行
177.         int pid = fork();
178.         if(pid == 0) {
179.             /*TODO:子进程 1 将标准输出重定向到管道,注意这里数组的下标被挖空
了要补全*/
180.             close(pipefd[READ_END]);
181.             dup2(pipefd[WRITE_END], STDOUT_FILENO);
182.             close(pipefd[WRITE_END]);
183.             /*
184.              在使用管道时,为了可以并发运行,所以内建命令也在子进程中运行
185.              因此我们用了一个封装好的 execute 函数
186.              */
187.             char *argv[MAX_CMD_ARG_NUM];
188.             int argc = split_string(commands[0], " ", argv);
189.             execute(argc, argv);
190.             exit(255);
191.         }
192.         // 因为在 shell 的设计中,管道是并发执行的,所以我们不在每个子进程结束
193.         后才运行下一个
194.         // 而是直接创建下一个子进程
195.         // 子进程 2
196.         //fork 进程 b,并把 b 的标准输入改为从管道的读端读取,执行
197.         pid = fork();
198.         if(pid == 0) {
199.             /* TODO:子进程 2 将标准输入重定向到管道,注意这里数组的下标被挖空
了要补全 */
200.             close(pipefd[WRITE_END]);
201.             dup2(pipefd[READ_END], STDIN_FILENO);
202.             close(pipefd[READ_END]);
203.
204.             char *argv[MAX_CMD_ARG_NUM];
205.             /* TODO:处理参数,分出命令名和参数,并使用 execute 运行
206.              * 在使用管道时,为了可以并发运行,所以内建命令也在子进程中运行
207.              * 因此我们用了一个封装好的 execute 函数
208.              */
209.             int argc = split_string(commands[1], " ", argv);
210.             execute(argc, argv);
211.             exit(255);

```

```

212.         }
213.         close(pipefd[WRITE_END]);
214.         close(pipefd[READ_END]);
215.         while (wait(NULL) > 0);
216.
217.     } else {    // 三个以上的命令
218.         int read_fd;    // 上一个管道的读端口（出口）
219.         for(int i = 0; i < cmd_count; i++) {
220.             int pipefd[2];
221.             /* TODO:创建管道，n 条命令只需要 n-1 个管道，所以有一次循环中是不
用创建管道的*/
222.             if(i < cmd_count - 1)
223.             {
224.                 int ret = pipe(pipefd);
225.                 if(ret < 0)
226.                 {
227.                     printf("pipe error!\n");
228.                     continue;
229.                 }
230.             }
231.             int pid = fork();
232.             if(pid == 0) {
233.                 /* TODO: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
*/
234.                 if(i < cmd_count - 1)
235.                 {
236.                     dup2(pipefd[WRITE_END], STDOUT_FILENO);
237.                 }
238.
239.                 /* TODO:除了第一条命令外，都将标准输入重定向到上一个管道入
口 */
240.                 if(i > 0)
241.                 {
242.                     dup2(read_fd, STDIN_FILENO);
243.                 }
244.
245.                 /* TODO:处理参数，分出命令名和参数，并使用 execute 运行
246.                 * 在使用管道时，为了可以并发运行，所以内建命令也在子进程中运
行
247.                 * 因此我们用了一个封装好的 execute 函数*/
248.                 char *argv[MAX_CMD_ARG_NUM];
249.                 int argc = split_string(commands[i], " ", argv);
250.                 execute(argc, argv);
251.                 exit(255);

```

```

252.         }
253.         /* 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
254.         * 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
255.         * 记得关闭父进程没用的管道写端口
256.         */
257.         if(pid > 0)
258.         {
259.
260.             if(i > 0) //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
261.             {
262.                 close(read_fd);
263.             }
264.             if(i < cmd_count - 1) //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
265.             {
266.                 read_fd=pipefd[0];
267.             }
268.             close(pipefd[WRITE_END]); //记得关闭父进程没用的管道
           写端口
269.
270.         }
271.         // 因为在 shell 的设计中，管道是并发执行的，所以我们不在每个子进程
           结束后才运行下一个
272.         // 而是直接创建下一个子进程
273.     }
274.     // TODO:等待所有子进程结束
275.     while (wait(NULL) > 0);
276. }
277. }
278. }

```

### 3.2.4 实现结果

```
elonwu@ubuntu: ~/oslab
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
elonwu@ubuntu:~/oslab$ ./shell
shell: /home/elonwu/oslab -> date
2021年 05月 07日 星期五 02:28:57 PDT
shell: /home/elonwu/oslab -> cd ../
shell: /home/elonwu -> cd oslab
shell: /home/elonwu/oslab -> cd linux-4.9.263
shell: /home/elonwu/oslab/linux-4.9.263 -> ls
arch          firmware  lib       README     usr
block         fs        MAINTAINERS REPORTING-BUGS virt
certs         include  Makefile  samples    vmlinux
COPYING       init     mm        scripts    vmlinux-gdb.py
CREDITS       ipc      modules.builtin security    vmlinux.o
crypto        Kbuild   modules.order sound       wget-log
Documentation Kconfig  Module.symvers System.map
drivers       kernel   net       tools
shell: /home/elonwu/oslab/linux-4.9.263 -> ls | grep k
block
kernel
Makefile
shell: /home/elonwu/oslab/linux-4.9.263 -> ls | grep k | grep M
Makefile
shell: /home/elonwu/oslab/linux-4.9.263 -> exit
elonwu@ubuntu:~/oslab$
```

#### 四、实验步骤

通过本次实验，我初步掌握了系统调用的添加，并且熟悉了 linux 下的系统调用，对于课堂上老师教授的知识有了更深层次的理解。实验题目的难度层层递进，有基础操作的考核，也有所学知识综合，难易结合，既有复习又有思考，让所学在实践中得以运用，加深了我对操作系统知识的理解。希望今后实验可以保持本次实验中详细实验指导描述的优点，辅助完成每项试验内容。