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

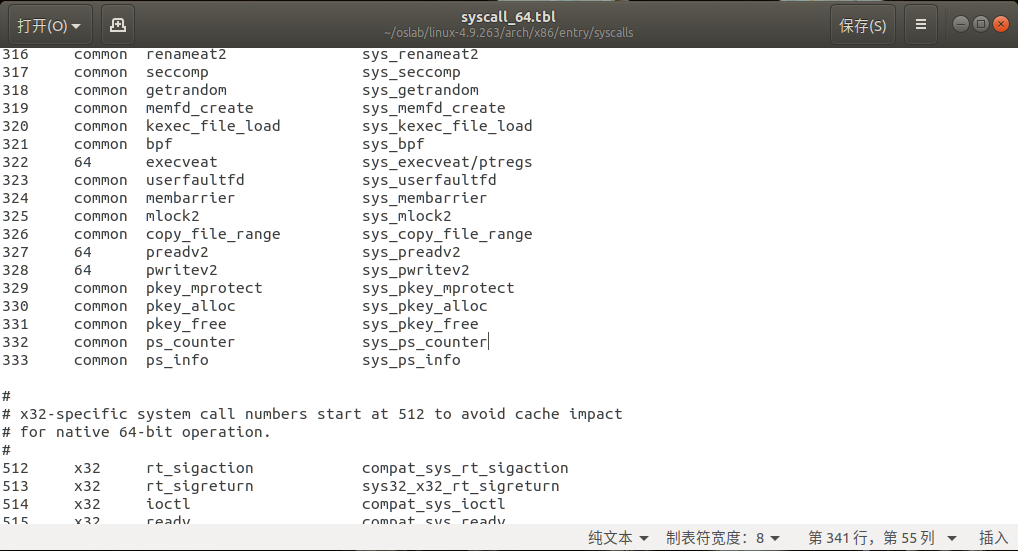
Linux内核版本: 4.9.263

三、实验步骤

3.1编写系统调用实现一个Linux ps

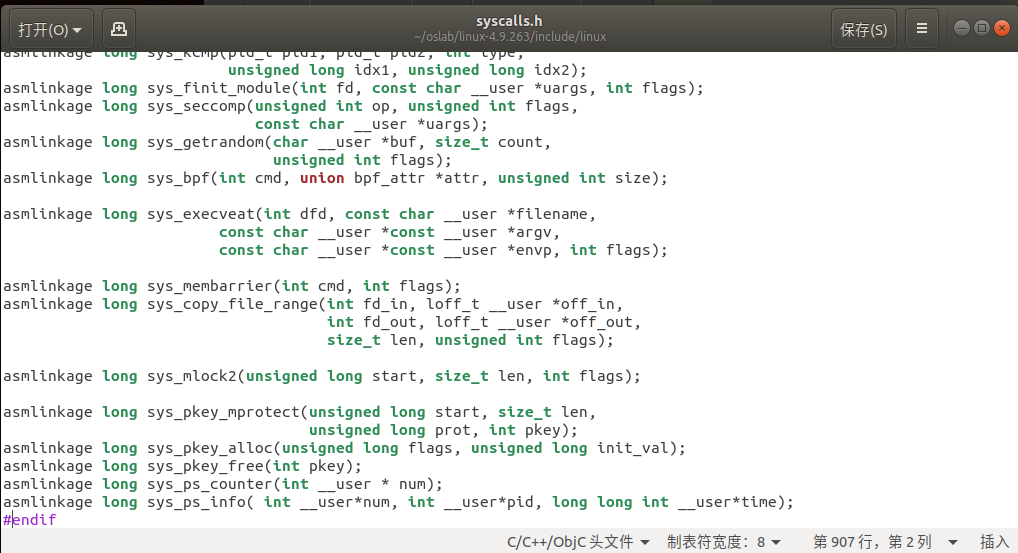
3.1.1注册系统调用

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



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。

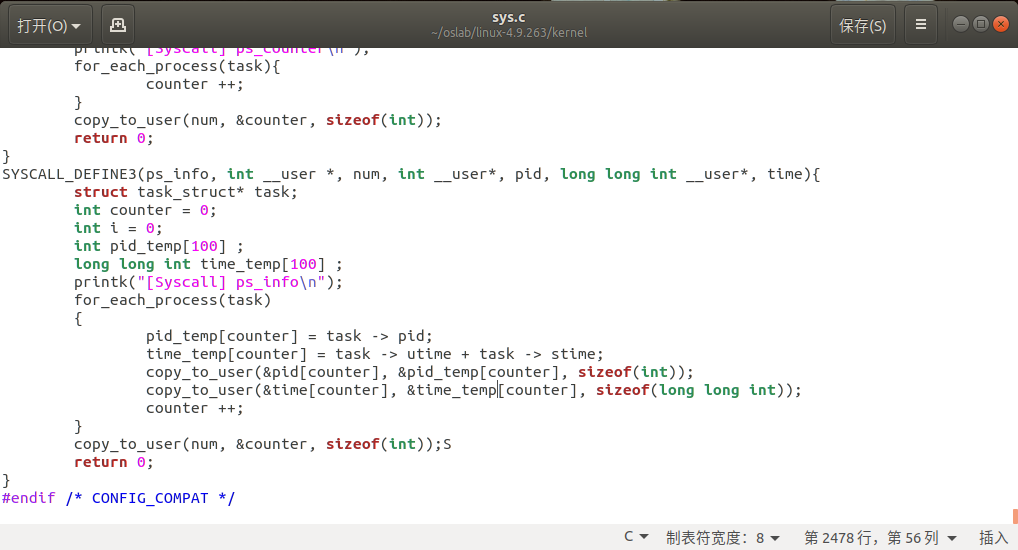


3.1.3实现函数

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

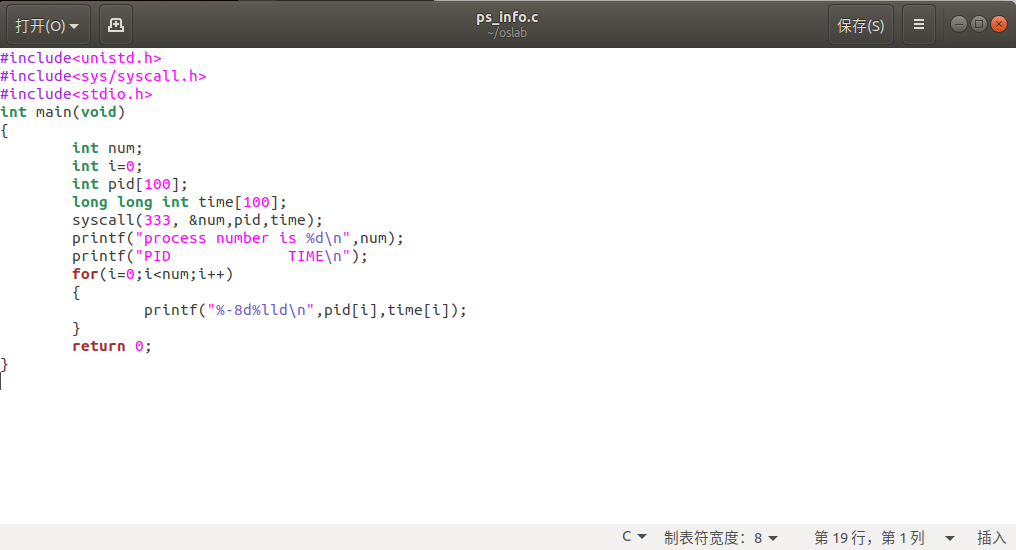
在获取进程运行的时间信息时，会遇到两个变量，分别是结构体中的utime和stime。查阅资料得知，stime是系统时间,即进程在内核模式下花费的时间,而utime是在用户模式下花费的时间.这些值取决于该特定过程的安排.没有为其更新定义此类间隔.随着各个模式中的时间花费的变化,它们会快速更新。所以进程运行的总时间应该是stime+utime.

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，需要注意不同数据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。



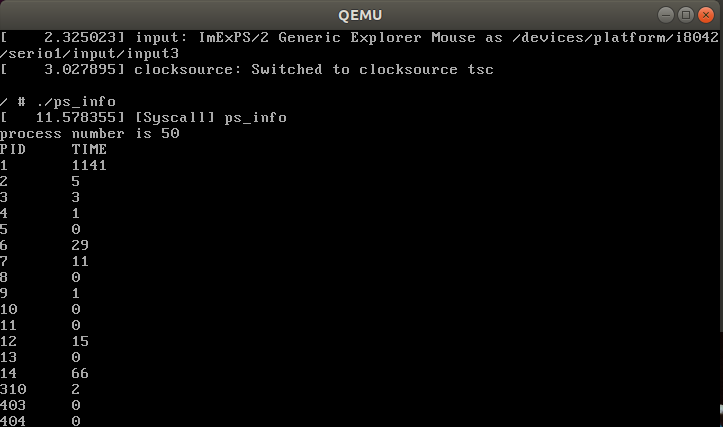
3.1.4编写测试程序

编写程序ps\_info.c用以测试



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



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.

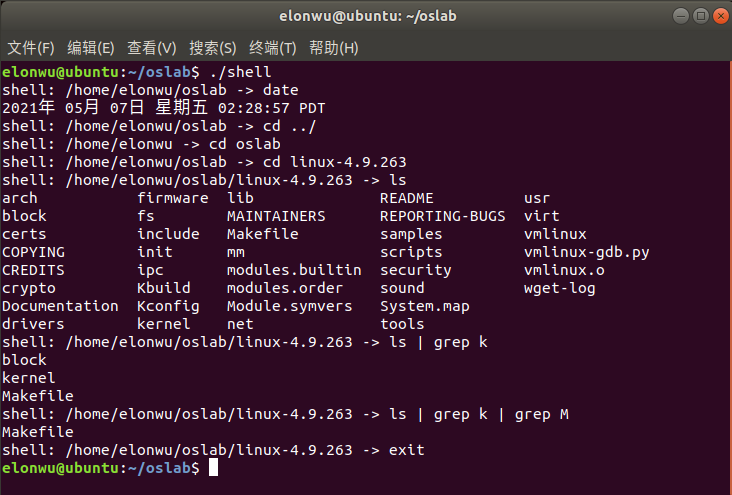
返回值：

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. #define MAX\_CMDLINE\_LENGTH  1024    /\* max cmdline length in a line\*/
8. #define MAX\_BUF\_SIZE        4096    /\* max buffer size \*/
9. #define MAX\_CMD\_ARG\_NUM     32      /\* max number of single command args \*/
10. #define WRITE\_END 1     // pipe write end
11. #define READ\_END 0      // pipe read end
12. int split\_string(char\* string, char \*sep, char\*\* string\_clips) {
14. char string\_dup[MAX\_BUF\_SIZE];
15. string\_clips[0] = strtok(string, sep);
16. int clip\_num=0;
18. do {
19. char \*head, \*tail;
20. head = string\_clips[clip\_num];
21. tail = head + strlen(string\_clips[clip\_num]) - 1;
22. while(\*head == ' ' && head != tail)
23. head ++;
24. while(\*tail == ' ' && tail != head)
25. tail --;
26. \*(tail + 1) = '\0';
27. string\_clips[clip\_num] = head;
28. clip\_num ++;
29. }while(string\_clips[clip\_num]=strtok(NULL, sep));
30. return clip\_num;
31. }
32. /\*
33. 执行内置命令
34. arguments:
35. argc: 命令的参数个数
36. argv: 依次代表每个参数，注意第一个参数就是要执行的命令，
37. 若执行"ls a b c"命令，则argc=4, argv={"ls", "a", "b", "c"}
38. return:
39. int, 若执行成功返回0，否则返回值非零
40. \*/
41. int exec\_builtin(int argc, char\*\*argv) {
42. if(argc == 0) {
43. return 0;
44. }
45. /\* TODO: 添加和实现内置指令 \*/
46. if (strcmp(argv[0], "cd") == 0)
47. {
48. if(argc > 2)
49. {
50. printf("ERROR!\n");
51. return 0;
52. }
53. 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
54. {
55. printf("Destination directory does not exist\n");
56. return 0;
57. }
58. }
59. else
60. {
61. if (strcmp(argv[0], "pwd") == 0)
62. {
63. return 0;
64. }
65. else
66. {
67. if (strcmp(argv[0], "exit") == 0)
68. {
69. exit(0);
70. }
71. else
72. {// 不是内置指令时
73. return -1;
74. }
75. }
76. }
77. }
78. /\*
79. 在本进程中执行，且执行完毕后结束进程。
80. arguments:
81. argc: 命令的参数个数
82. argv: 依次代表每个参数，注意第一个参数就是要执行的命令，
83. 若执行"ls a b c"命令，则argc=4, argv={"ls", "a", "b", "c"}
84. return:
85. int, 若执行成功则不会返回（进程直接结束），否则返回非零
86. \*/
87. int execute(int argc, char\*\* argv) {
88. if(exec\_builtin(argc, argv) == 0) {
89. exit(0);
90. }
91. /\* TODO:运行命令 \*/
92. pid\_t pid;
93. pid = fork();
94. if(pid == 0)
95. {
96. if(execvp(argv[0], argv) == -1)
97. {  //execvp有两个参数：要运行的程序名和那个程序的命令行参数。
98. exit(-1);
99. }
100. exit(0);
101. }
102. else
103. {
104. if(pid > 0)
105. {
106. wait(NULL);
107. return 0;
108. }
109. }
110. return -1;
111. }
112. int main() {
113. /\* 输入的命令行 \*/
114. char cmdline[MAX\_CMDLINE\_LENGTH];
115. /\* 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 \*/
116. char \*commands[128];
117. int cmd\_count;
118. while (1) {
119. /\* TODO:增加打印当前目录，格式类似"shell:/home/oslab ->"，你需要改下面的printf \*/
120. char current\_working\_directory[256];
121. getcwd(current\_working\_directory, 256);
122. printf("shell: %s -> ", current\_working\_directory);
123. fflush(stdout);
124. fgets(cmdline, 256, stdin);
125. strtok(cmdline, "\n");
126. /\* 拆解命令行 \*/
127. cmd\_count = split\_string(cmdline, "|", commands);
128. if(cmd\_count == 0) {
129. continue;
130. } else if(cmd\_count == 1) {     // 没有管道的单一命令
131. char \*argv[MAX\_CMD\_ARG\_NUM];
132. /\* TODO:处理参数，分出命令名和参数 \*/
133. int argc = split\_string(commands[0], " ", argv);
134. /\* 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 \*/
135. if(exec\_builtin(argc, argv) == 0) {
136. continue;
137. }
138. /\* TODO:创建子进程，运行命令，等待命令运行结束 \*/
139. pid\_t pid;
140. pid = fork();
141. if(pid == 0)
142. {
143. if(execute(argc, argv) != 0)
144. {
145. exit(-1);
146. }
147. }
148. else
149. {
150. if(pid > 0)
151. {
152. wait(NULL);
153. exit(0);
154. }
155. }
156. } else if(cmd\_count == 2) {     // 两个命令间的管道
157. int pipefd[2];
158. int ret = pipe(pipefd);
159. if(ret < 0) {
160. printf("pipe error!\n");
161. continue;
162. }
163. // 子进程1
164. //fork进程a,并把a的标准输出改为输出到管道的写端，执行
165. int pid = fork();
166. if(pid == 0) {
167. /\*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\*/
168. close(pipefd[READ\_END]);
169. dup2(pipefd[WRITE\_END], STDOUT\_FILENO);
170. close(pipefd[WRITE\_END]);
171. /\*
172. 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
173. 因此我们用了一个封装好的execute函数
174. \*/
175. char \*argv[MAX\_CMD\_ARG\_NUM];
176. int argc = split\_string(commands[0], " ", argv);
177. execute(argc, argv);
178. exit(255);
180. }
181. // 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
182. // 而是直接创建下一个子进程
183. // 子进程2
184. //fork进程b，并把b的标准输入改为从管道的读端读取，执行
185. pid = fork();
186. if(pid == 0) {
187. /\* 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 \*/
188. close(pipefd[WRITE\_END]);
189. dup2(pipefd[READ\_END], STDIN\_FILENO);
190. close(pipefd[READ\_END]);
191. char \*argv[MAX\_CMD\_ARG\_NUM];
192. /\* TODO:处理参数，分出命令名和参数，并使用execute运行
193. \* 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
194. \* 因此我们用了一个封装好的execute函数
195. \*/
196. int argc = split\_string(commands[1], " ", argv);
197. execute(argc, argv);
198. exit(255);
199. }
200. close(pipefd[WRITE\_END]);
201. close(pipefd[READ\_END]);
202. while (wait(NULL) > 0);
203. } else {    // 三个以上的命令
204. int read\_fd;    // 上一个管道的读端口（出口）
205. for(int i = 0; i < cmd\_count; i++) {
206. int pipefd[2];
207. /\* 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\*/
208. if(i < cmd\_count - 1)
209. {
210. int ret = pipe(pipefd);
211. if(ret < 0)
212. {
213. printf("pipe error!\n");
214. continue;
215. }
216. }
217. int pid = fork();
218. if(pid == 0) {
219. /\* 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\*/
220. if(i < cmd\_count - 1)
221. {
222. dup2(pipefd[WRITE\_END], STDOUT\_FILENO);
223. }
224. /\* 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 \*/
225. if(i > 0)
226. {
227. dup2(read\_fd, STDIN\_FILENO);
228. }
229. /\* TODO:处理参数，分出命令名和参数，并使用execute运行
230. \* 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
231. \* 因此我们用了一个封装好的execute函数\*/
232. char \*argv[MAX\_CMD\_ARG\_NUM];
233. int argc = split\_string(commands[i], " ", argv);
234. execute(argc, argv);
235. exit(255);
236. }
237. /\* 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
238. \* 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
239. \* 记得关闭父进程没用的管道写端口
240. \*/
241. if(pid > 0)
242. {
243. 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
244. {
245. close(read\_fd);
246. }
247. 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
248. {
249. read\_fd=pipefd[0];
250. }
251. close(pipefd[WRITE\_END]);  //记得关闭父进程没用的管道写端口
252. }
253. // 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
254. // 而是直接创建下一个子进程
255. }
256. // TODO:等待所有子进程结束
257. while (wait(NULL) > 0);
258. }
259. }
260. }

3.2.4实现结果



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，也有所学知识的综合，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。