**execve系统调用分析**

2015年01月04日 18:15:27 [剑圣风暴](https://me.csdn.net/fukai555) 阅读数：1751

    在Linux平台下，我们一般都是在命令行下键入"./hello"来运行一个当前目录下的hello应用程序（"./"指定当前目录）。虽然看似很简单，但这么小小的一个操作其实涉及到了很多的知识。比如：shell是如何将hello调入内存的？hello在运行前shell执行了哪些操作？hello的父进程又是哪个？回答这些问题之前我们先来看下面的一个例子：  
这个例子包含两个程序，第一个test\_hello程序可以看成是hello的父进程，因为在test\_hello中调用了fork函数生成一个子进程，并在子进程中调用execve函数执行hello程序。最后父进程等待子进程的退出，并打印出子进程的退出码。第二个hello程序简单的将main函数的参数和环境变量打印出来。

*test\_hello.c代码：*

1. #include <stdio.h>
2. #include <stdlib.h>
4. char \* argv[]={ "arg1","arg2", NULL };
5. char \* envp[] = { "PATH=/home", NULL };
7. int main()
8. {
9. int pid;
10. int i;
11. if(!(pid=fork()))
12. {
13. execve("hello",argv,envp);
14. }
15. else
16. printf("child pid %d\n\n",pid);
18. while (1)
19. if (pid == wait(&i))
20. break;
21. printf("\n\rchild %d died with code %04x\n\r",pid,i);
22. return 0;
23. }

*hello.c代码：*

1. #include <stdio.h>
3. int main(int argc, char \*\*argv, char \*\*envp)
4. {
5. int i;
6. printf("hello program start\n");
7. for(i=0;i<argc;i++)
8. printf("argv: %s\n",argv[i]);
9. printf("arge: %s\n",envp[0]);
11. printf("hello program end\n");
12. return 0x05;
13. }

在同一个目录下将上面两个程序进行编译链接，然后运行"./test\_hello"，最后打印结果如下：

fk@E430:~/gcc$ ./test\_hello

child pid 3690

hello program start

argv: arg1

argv: arg2

arge: PATH=/home

hello program end

child 3690 died with code 0500

    从运行的结果可以看出，首先test\_hello父进程先运行打印处子进程的pid，然后子进程通过执行execve系统调用将hello程序装载进来并运行，打印出execve系统调用向hello程序传递的参数和环境变量，最后hello退出后父进程也将终止wait并退出。上面程序其实就是一个说明shell运行机制的简单例子（实际的shell远比这复杂），为什么这么说呢？首先可以把test\_hello看成是"shell"，然后把hello看成是我们的"命令"（ls， echo 类似的命令），而hello的执行依靠的是test\_hello，并且hello的参数也是由test\_hello来传递，这与我们平常用的shell道理是一致的（比如: ls -l /etc/passwd）,只不过我们平常用的shell没有显示的调用而已（/bin/sh）。  
    所以简单来说，shell执行一个命令就是这么一个过程：首先fork出一个子进程，然后在子进程中execve这个命令程序，并传递命令程序的参数和环境变量，最后在父进程中wait子进程退出。  
  
    下面再通过分析linux0.11的execve系统调用来进一步说明一个程序的执行过程。首先来看一下linux0.11的init进程中关于shell启动的代码部分：

1. static char \* argv[] = { "-/bin/sh",NULL };
2. static char \* envp[] = { "HOME=/usr/root", NULL };
4. void init(void)
5. {
6. */\*... 省略前面代码...\*/*
8. while (1) {
9. if ((pid=fork())<0) {
10. printf("Fork failed in init\r\n");
11. continue;
12. }
13. if (!pid) {
14. close(0);close(1);close(2);
15. setsid();
16. (void) open("/dev/tty0",O\_RDWR,0);
17. (void) dup(0);
18. (void) dup(0);
19. \_exit(execve("/bin/sh",argv,envp));
20. }
21. while (1)
22. if (pid == wait(&i))
23. break;
24. printf("\n\rchild %d died with code %04x\n\r",pid,i);
25. sync();
26. }
28. \_exit(0); */\* NOTE! \_exit, not exit() \*/*
29. }

    上面的init函数其实就是在所谓的init进程中执行，在该函数中可以看到包含一个while循环，在循环中首先fork出一个子进程，然后在子进程中执行shell终端程序(/bin/sh)，父进程一直等待子进程退出。当子进程退出后（shell中执行exit命令），父进程将break等待，然后又重新fork子进程并运行shell程序，父进程又等待，如此形成一个循环（似乎只有强制关机才可“退出”）。  
    上述代码我们只关注execve("/bin/sh",argv,envp)函数，其它的先不管。按照最开始的分析，执行这条语句将启动/bin/sh程序，并给该程序传递argv参数和envp环境变量。注意sh也只是一个正常的具有main函数的可执行程序而已，只不过它的功能与我们平常写的程序稍有不同，它会读取用户在命令行输入的程序名以及相应参数，然后fork出一个子进程去execve这个程序，子程序执行完成后又回到命令行等待输入，最后一直循环这个过程（似乎跟init函数有点类似，不过概念完全不同）。所以execve所做的工作就是将所要运行的程序调入内存并运行，准确的说应该是将fork出来的子进程替换成所要运行的程序。那么这个替换又是如何进行的呢？下面具体分析execve这个系统调用。  
    首先execve函数将调用system\_call.s文件中的sys\_execve函数，该函数如下所示：

sys\_execve:

lea EIP(%esp),%eax

pushl %eax

call do\_execve

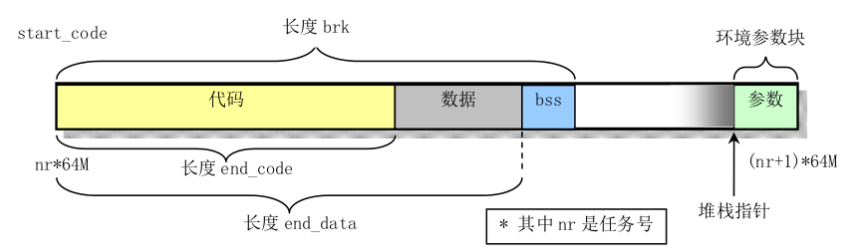
addl $4,%esp

ret

    可以看到sys\_execve内部其实又调用的是exec.c文件中的do\_execve函数，但是注意在调用do\_execve函数前的pushl %eax语句，这条语句其实是给do\_execve函数压入参数，根据C语言参数传递规则，do\_execve函数的第一个参数即为eax寄存器的值，而eax寄存器中存放的内容为子进程进行系统调用时压入子进程内核态堆栈的eip指针，也即系统调用的[返回地址](https://www.baidu.com/s?wd=%E8%BF%94%E5%9B%9E%E5%9C%B0%E5%9D%80&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)，所以这里可以猜想do\_execve函数内部可能会对该eip指针进行修改。  
    接下来再来看一下do\_execve函数声明：  
int do\_execve(unsigned long \* eip,long tmp,char \* filename, char \*\* argv, char \*\* envp)  
    我们已经知道了第一个参数的由来以及含义，第二个参数其实是sys\_execve函数的返回地址（call sys\_execve指令自动产生），这里并没有用，接下来的3个参数即为C程序中执行execve系统调用所传递进来的3个参数，他们分别是：可执行程序的**全路径名称**，参数以及环境变量。  
    在具体研究do\_execve函数前，再先大概了解一下gcc1.3版本编译出来的a.out格式的可执行文件结构。linux0.11支持的可执行文件格式为a.out（现在采用的是ELF文件格式），每个二进制的执行文件的头部数据都从下面数据结构开始：

1. struct exec {
2. unsigned long a\_magic; */\* Use macros N\_MAGIC, etc for access \*/*
3. unsigned a\_text; */\* length of text, in bytes \*/*
4. unsigned a\_data; */\* length of data, in bytes \*/*
5. unsigned a\_bss; */\* length of uninitialized data area for file, in bytes \*/*
6. unsigned a\_syms; */\* length of symbol table data in file, in bytes \*/*
7. unsigned a\_entry; */\* start address \*/*
8. unsigned a\_trsize; */\* length of relocation info for text, in bytes \*/*
9. unsigned a\_drsize; */\* length of relocation info for data, in bytes \*/*
10. };

    这个结构就基本上描述了该可执行程序的必要信息，比如：代码段，数据段，bss段的长度，应用程序的入口地址a\_entry等。所以可以想象，如果将子进程的eip替换为a\_entry，就应该可以运行该可执行文件了，当然还必须要设置**堆栈指针**以及**进程描述符**。因此do\_execve函数完成的功能应该包含上述的这些操作（设置堆栈，设置进程描述符，替换eip），其中对于堆栈的操作稍微复杂点。堆栈操作的过程其实是将可执行程序的参数和环境变量复制到64M线性空间的末尾128KB处（128KB足够放很多参数了），并且为这些参数和环境变量创建对应的指针表(因为要访问这些参数，肯定要定义指向这些参数的指针了)，最终的堆栈指针p将指向指针表的第一个元素（因为在进入main函数时需要从堆栈中弹出3个参数：1个整型参数argc和2个字符串指针参数argv，envp，所以p应指向整型参数argc）。最终进程的64M线性空间的布局如下所示：



    从do\_execve函数的源码可以看到，如我们想象的那样，do\_execve函数最后修改了子进程的eip和esp，这样函数返回后将执行可执行程序的内容。

1. eip[0] = ex.a\_entry; */\* eip, magic happens :-) \*/*
2. eip[3] = p; */\* stack pointer \*/*

    至此，关于do\_execve函数的功能介绍到这里，总结一下：对于一个a.out格式的可执行文件，首先获取该文件的inode节点指针，并判断该文件的属性以及执行权限，然后获取可执行文件的a.out头结构指针ex，根据ex结构判断可执行文件是否符合标准，然后对子进程的进程描述符的相关字段进行赋值，最后修改子进程内核态堆栈中eip和esp并返回。返回后，fork出来的子进程将被可执行程序替代并运行。其中关于执行脚本文件相关的代码没有分析（#!），因为这部分代码仅在执行文件为shell脚本的时候可用。  
  
  
附do\_execve函数源码:

1. int do\_execve(unsigned long \* eip,long tmp,char \* filename,
2. char \*\* argv, char \*\* envp)
3. {
4. struct m\_inode \* inode;
5. struct buffer\_head \* bh;
6. struct exec ex;
7. unsigned long page[MAX\_ARG\_PAGES];
8. int i,argc,envc;
9. int e\_uid, e\_gid;
10. int retval;
11. int sh\_bang = 0;
12. unsigned long p=PAGE\_SIZE\*MAX\_ARG\_PAGES-4;
14. if ((0xffff & eip[1]) != 0x000f)
15. panic("execve called from supervisor mode");
16. for (i=0 ; i<MAX\_ARG\_PAGES ; i++) */\* clear page-table \*/*
17. page[i]=0;
18. if (!(inode=namei(filename))) */\* get executables inode \*/*
19. return -ENOENT;
20. argc = count(argv);
21. envc = count(envp);
23. restart\_interp:
24. if (!S\_ISREG(inode->i\_mode)) { */\* must be regular file \*/*
25. retval = -EACCES;
26. goto exec\_error2;
27. }
28. i = inode->i\_mode;
29. e\_uid = (i & S\_ISUID) ? inode->i\_uid : current->euid;
30. e\_gid = (i & S\_ISGID) ? inode->i\_gid : current->egid;
31. if (current->euid == inode->i\_uid)
32. i >>= 6;
33. else if (current->egid == inode->i\_gid)
34. i >>= 3;
35. if (!(i & 1) &&
36. !((inode->i\_mode & 0111) && suser())) {
37. retval = -ENOEXEC;
38. goto exec\_error2;
39. }
40. if (!(bh = bread(inode->i\_dev,inode->i\_zone[0]))) {
41. retval = -EACCES;
42. goto exec\_error2;
43. }
44. ex = \*((struct exec \*) bh->b\_data); */\* read exec-header \*/*
45. if ((bh->b\_data[0] == '#') && (bh->b\_data[1] == '!') && (!sh\_bang)) {
46. */\**
47. *\* This section does the #! interpretation.*
48. *\* Sorta complicated, but hopefully it will work. -TYT*
49. *\*/*
51. char buf[1023], \*cp, \*interp, \*i\_name, \*i\_arg;
52. unsigned long old\_fs;
54. strncpy(buf, bh->b\_data+2, 1022);
55. brelse(bh);
56. iput(inode);
57. buf[1022] = '\0';
58. if ((cp = strchr(buf, '\n'))) {
59. \*cp = '\0';
60. for (cp = buf; (\*cp == ' ') || (\*cp == '\t'); cp++);
61. }
62. if (!cp || \*cp == '\0') {
63. retval = -ENOEXEC; */\* No interpreter name found \*/*
64. goto exec\_error1;
65. }
66. interp = i\_name = cp;
67. i\_arg = 0;
68. for ( ; \*cp && (\*cp != ' ') && (\*cp != '\t'); cp++) {
69. if (\*cp == '/')
70. i\_name = cp+1;
71. }
72. if (\*cp) {
73. \*cp++ = '\0';
74. i\_arg = cp;
75. }
76. */\**
77. *\* OK, we've parsed out the interpreter name and*
78. *\* (optional) argument.*
79. *\*/*
80. if (sh\_bang++ == 0) {
81. p = copy\_strings(envc, envp, page, p, 0);
82. p = copy\_strings(--argc, argv+1, page, p, 0);
83. }
84. */\**
85. *\* Splice in (1) the interpreter's name for argv[0]*
86. *\* (2) (optional) argument to interpreter*
87. *\* (3) filename of shell script*
88. *\**
89. *\* This is done in reverse order, because of how the*
90. *\* user environment and arguments are stored.*
91. *\*/*
92. p = copy\_strings(1, &filename, page, p, 1);
93. argc++;
94. if (i\_arg) {
95. p = copy\_strings(1, &i\_arg, page, p, 2);
96. argc++;
97. }
98. p = copy\_strings(1, &i\_name, page, p, 2);
99. argc++;
100. if (!p) {
101. retval = -ENOMEM;
102. goto exec\_error1;
103. }
104. */\**
105. *\* OK, now restart the process with the interpreter's inode.*
106. *\*/*
107. old\_fs = get\_fs();
108. set\_fs(get\_ds());
109. if (!(inode=namei(interp))) { */\* get executables inode \*/*
110. set\_fs(old\_fs);
111. retval = -ENOENT;
112. goto exec\_error1;
113. }
114. set\_fs(old\_fs);
115. goto restart\_interp;
116. }
117. brelse(bh);
118. if (N\_MAGIC(ex) != ZMAGIC || ex.a\_trsize || ex.a\_drsize ||
119. ex.a\_text+ex.a\_data+ex.a\_bss>0x3000000 ||
120. inode->i\_size < ex.a\_text+ex.a\_data+ex.a\_syms+N\_TXTOFF(ex)) {
121. retval = -ENOEXEC;
122. goto exec\_error2;
123. }
124. if (N\_TXTOFF(ex) != BLOCK\_SIZE) {
125. printk("%s: N\_TXTOFF != BLOCK\_SIZE. See a.out.h.", filename);
126. retval = -ENOEXEC;
127. goto exec\_error2;
128. }
129. if (!sh\_bang) {
130. p = copy\_strings(envc,envp,page,p,0);
131. p = copy\_strings(argc,argv,page,p,0);
132. if (!p) {
133. retval = -ENOMEM;
134. goto exec\_error2;
135. }
136. }
137. */\* OK, This is the point of no return \*/*
138. if (current->executable)
139. iput(current->executable);
140. current->executable = inode;
141. for (i=0 ; i<32 ; i++)
142. current->sigaction[i].sa\_handler = NULL;
143. for (i=0 ; i<NR\_OPEN ; i++)
144. if ((current->close\_on\_exec>>i)&1)
145. sys\_close(i);
146. current->close\_on\_exec = 0;
147. free\_page\_tables(get\_base(current->ldt[1]),get\_limit(0x0f));
148. free\_page\_tables(get\_base(current->ldt[2]),get\_limit(0x17));
149. if (last\_task\_used\_math == current)
150. last\_task\_used\_math = NULL;
151. current->used\_math = 0;
152. p += change\_ldt(ex.a\_text,page)-MAX\_ARG\_PAGES\*PAGE\_SIZE;
153. p = (unsigned long) create\_tables((char \*)p,argc,envc);
154. current->brk = ex.a\_bss +
155. (current->end\_data = ex.a\_data +
156. (current->end\_code = ex.a\_text));
157. current->start\_stack = p & 0xfffff000;
158. current->euid = e\_uid;
159. current->egid = e\_gid;
160. i = ex.a\_text+ex.a\_data;
161. while (i&0xfff)
162. put\_fs\_byte(0,(char \*) (i++));
163. eip[0] = ex.a\_entry; */\* eip, magic happens :-) \*/*
164. eip[3] = p; */\* stack pointer \*/*
165. return 0;
166. exec\_error2:
167. iput(inode);
168. exec\_error1:
169. for (i=0 ; i<MAX\_ARG\_PAGES ; i++)
170. free\_page(page[i]);
171. return(retval);
172. }