apue看到第八章，对exec函数族的理解一直都很混乱，总觉得不对劲儿，其实不能理解的先暂时跳过，看到后面，再结合实例也就慢慢的理解了。

以下内容转自：http://www.cppblog.com/prayer/archive/2009/04/15/80077.html

也许有不少读者从本系列文章一推出就开始读，一直到这里还有一个很大的疑惑：既然所有新进程都是由fork产生的，而且由fork产生的子进程和父进程几乎完全一样，那岂不是意味着系统中所有的进程都应该一模一样了吗？而且，就我们的常识来说，当我们执行一个程序的时候，新产生的进程的内容应就是程序的内容才对。是我们理解错了吗？显然不是，要解决这些疑惑，就必须提到我们下面要介绍的exec系统调用。

1.10.1 简介

说是exec系统调用，实际上在Linux中，并不存在一个exec()的函数形式，exec指的是一组函数，一共有6个，分别是：

```
#include <unistd.h> int execl(const char *path, const char *arg, ...); int execlp(const char *file, const char *arg, ...); int execle(const char *path, const char *arg, ..., char *const envp[]); int execv(const char *path, char *const argv[]); int execvp(const char *file, char *const argv[]); int execve(const char *path, char *const argv[], char *const envp[]);
```

其中只有execve是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。

exec函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何Linux下可执行的脚本文件。

与一般情况不同，**exec函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段，数据段和堆栈等都已经被新的内容取代，**只留下进程ID等一些表面上的信息仍保持原样，颇有些神似"三十六计"中的"金蝉脱壳"。看上去还是旧的躯壳，却已经注入了新的灵魂。**只有调用失败了，它们才会返回一个-1，从原程序的调用点接着往下执行。**

现在我们应该明白了，Linux下是如何执行新程序的，每当有进程认为自己不能为系统和拥护做出任何贡献了，他就可以发挥最后一点余热，调用任何一个exec，让自己以新的面貌重生；或者，更普遍的情况是，如果一个进程想执行另一个程序，它就可以fork出一个新进程，然后调用任何一个exec，这样看起来就好像通过执行应用程序而产生了一个新进程一样。

事实上第二种情况被应用得如此普遍，以至于Linux专门为其作了优化，我们已经知道，fork会将调用进程的所有内容原封不动的拷贝到新产生的子进程中去，这些拷贝的动作很消耗时间，而如果fork完之后我们马上就调用exec，这些辛辛苦苦拷贝来的东西又会被立刻抹掉，这看起来非常不划算，于是人们设计了一种"写时拷贝（copy-on-write）"技术，使得fork结

束后并不立刻复制父进程的内容，而是到了真正实用的时候才复制，这样如果下一条语句是exec，它就不会白白作无用功了，也就提高了效率。

1.10.2 稍稍深入

上面6条函数看起来似乎很复杂，但实际上无论是作用还是用法都非常相似，只有很微小的差别。在学习它们之前，先来了解一下我们习以为常的main函数。

下面这个main函数的形式可能有些出乎我们的意料：

```
int main(int argc, char *argv[], char *envp[])
```

它可能与绝大多数教科书上描述的都不一样，但实际上，这才是main函数真正完整的形式。

参数argc指出了运行该程序时命令行参数的个数，数组argv存放了所有的命令行参数，数组envp存放了所有的环境变量。环境变量指的是一组值，从用户登录后就一直存在，很多应用程序需要依靠它来确定系统的一些细节，我们最常见的环境变量是PATH，它指出了应到哪里去搜索应用程序，如/bin；HOME也是比较常见的环境变量，它指出了我们在系统中的个人目录。环境变量一般以字符串"XXX=xxx"的形式存在，XXX表示变量名，xxx表示变量的值。

值得一提的是，argv数组和envp数组存放的都是指向字符串的指针，这两个数组都以一个NULL元素表示数组的结尾。

我们可以通过以下这个程序来观看传到argc、argv和envp里的都是什么东西：

```
/* main.c */
int main(int argc, char *argv[], char *envp[]) {
printf("\n### ARGC ###\n%d\n", argc);
printf("\n### ARGV ###\n");
while(*argv) printf("%s\n", *(argv++));
printf("\n### ENVP ###\n");
while(*envp) printf("%s\n", *(envp++));
return 0; }
```

编译它：

```
$ cc main.c -o
main
```

运行时，我们故意加几个没有任何作用的命令行参数：

```
$ ./main -xx
000 ###
ARGC ### 3
### ARGV
### ./main -
xx 000 ###
ENVP ###
PWD=/home/
lei
REMOTEHOST
=dt.laser.com
HOSTNAME=l
ocalhost.local
domain
QTDIR=/usr/li
b/qt-2.3.1
LESSOPEN=|/
usr/bin/lesspi
pe.sh %s
KDEDIR=/usr
USER=lei
LS_COLORS=
MACHTYPE=i
386-redhat-
linux-gnu
MAIL=/var/sp
ool/mail/lei
INPUTRC=/et
c/inputrc
LANG=en_US
LOGNAME=le
i SHLVL=1
SHELL=/bin/b
ash
HOSTTYPE=i3
86
OSTYPE=linux
-gnu
HISTSIZE=100
0 TERM=ansi
HOME=/hom
e/lei
PATH=/usr/lo
cal/bin:/bin:/u
```

```
sr/bin:/usr/X1
1R6/bin:/hom
e/lei/bin
_=./main
```

我们看到，程序将"./main"作为第1个命令行参数，所以我们一共有3个命令行参数。这可能与大家平时习惯的说法有些不同，小心不要搞错了。
现在回过头来看一下exec函数族，先把注意力集中在execve上：

```
int
execve(const
char *path,
char *const
argv[], char
*const envp[]);
```

对比一下main函数的完整形式，看出问题了吗？是的，这两个函数里的argv和envp是完全一一对应的关系。execve第1个参数path是被执行应用程序的完整路径，第2个参数argv就是传给被执行应用程序的命令行参数，第3个参数envp是传给被执行应用程序的环境变量。
留心看一下这6个函数还可以发现，前3个函数都是以execl开头的，后3个都是以execv开头的，它们的区别在于，execv开头的函数是以"char *argv[]"这样的形式传递命令行参数，而execl开头的函数采用了我们更容易习惯的方式，把参数一个一个列出来，然后以一个NULL表示结束。这里的NULL的作用和argv数组里的NULL作用是一样的。
在全部6个函数中，只有execle和execve使用了char *envp[]传递环境变量，其它的4个函数都没有这个参数，这并不意味着它们不传递环境变量，这4个函数将把默认的环境变量不做任何修改地传给被执行的应用程序。而execle和execve会用指定的环境变量去替代默认的那些。
还有2个以p结尾的函数execlp和execvp，咋看起来，它们和execl与execv的差别很小，事实也确是如此，除execlp和execvp之外的4个函数都要求，它们的第1个参数path必须是一个完整的路径，如"/bin/ls"；而execlp和execvp的第1个参数file可以简单到仅仅是一个文件名，如"ls"，这两个函数可以自动到环境变量PATH制定的目录里去寻找。

1.10.3 实战

知识介绍得差不多了，接下来我们看看实际的应用：

```
/* exec.c */
#include
<unistd.h>
main() { char
*envp[]=
{"PATH=/tmp
", "USER=lei",
"STATUS=test
ing", NULL};
```

```
char *argv_execv[]={"echo", "excuted by execv", NULL};
char *argv_execvp[]={"echo", "executed by execvp", NULL}; char *argv_execve[]={"env", NULL};
if(fork()==0)
if(execl("/bin/echo", "echo", "executed by execl", NULL)<0)
perror("Err on execl");
if(fork()==0)
if(execlp("echo", "echo", "executed by execlp", NULL)<0)
perror("Err on execlp");
if(fork()==0)
if(execle("/usr/bin/env", "env", NULL, envp)<0)
perror("Err on execle");
if(fork()==0)
if(execv("/bin/echo", argv_execv)<0)
perror("Err on execv");
if(fork()==0)
if(execvp("echo", argv_execvp)<0)
perror("Err on execvp");
if(fork()==0)
```

```
if(fork()==0)
if(execve("/usr
/bin/env",
argv_execve,
envp)<0)
perror("Err on
execve"); }
```

程序里调用了2个Linux常用的系统命令，echo和env。echo会把后面跟的命令行参数原封不动的打印出来，env用来列出所有环境变量。
由于各个子进程执行的顺序无法控制，所以有可能出现一个比较混乱的输出--各子进程打印的结果交杂在一起，而不是严格按照程序中列出的次序。
编译并运行：

```
$ cc exec.c -o
exec $ ./exec
executed by
execl
PATH=/tmp
USER=lei
STATUS=testi
ng executed
by execlp
excuted by
execv
executed by
execvp
PATH=/tmp
USER=lei
STATUS=testi
ng
```

果然不出所料，execle输出的结果跑到了execlp前面。
大家在平时的编程中，如果用到了exec函数族，一定记得要加错误判断语句。因为与其他系统调用比起来，exec很容易受伤，被执行文件的位置，权限等很多因素都能导致该调用的失败。最常见的错误是：

1. 找不到文件或路径，此时errno被设置为ENOENT；
2. 数组argv和envp忘记用NULL结束，此时errno被设置为EFAULT；
3. 没有对要执行文件的运行权限，此时errno被设置为EACCES。