尋找最初的初衷

迷失在LINUX的小書僮

Nov 27 Tue 2012 17:35

Linux下進程的建立 並附Linux exec函數族

[此篇文章並非原創若有冒犯敬請來信告知]

kogeiman

我們都知道，進程就是正在執行的程序。而在Linux中，可以使用一個進程來創建另外一個進程。這樣的話，Linux的進程的組織結構其實有點像Linux目錄樹，是個層次結構的,可以使用pstree命令來查看。在最上面是init程序的執行進程。它是所有進程的老祖宗。Linux提供了兩個函數來創建進程。

1.fork()

fork()提供了創建進程的基本操作，可以說它是Linux系統多任務的基礎。該函數在unistd.h庫中聲明。

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>
```

```
int main()
{
	printf("創建進程前\n" );
	pid_tpid = fork();
	if(!pid ){
		printf("我是子進程喲,我的PID是：%d\n" ,getpid() );
	}elseif( pid>0 ){
		printf("我是父進程，我的PID是：%d，我的子進程PID是：%d\n",getpid(),pid );
	}else{
		printf("創建進程失敗了喲\n" );
		exit(1);
	}
	return1;
}
```

在調用fork()之前，只有一個進程，但是fork()之後，將產生一個該進程的子進程，該子進程完全複製父進程，此時父子兩個進程同時運行。在fork()的時候，如果返回的是0，則說明該進程是子進程。如果返回大於0則說明是父進程。如果小於0（其實是-1），則說明創建進程失敗了。

每個進程都有一個唯一標示符，即PID，可以使用getpid()來獲取。父進程返回的pid其實是子進程的pid。

貌似這樣看，fork()之後也沒有什麼作用。其實不然，如果fork()之後跟其他linux功能使用，還是用處很大的。比如我們可以在父子進程中通過通信協議來通信，就可以協同完成一些任務了。

2.exec系列函數

如果只有fork()，肯定是不完美的，因爲fork()只能參數一個父進程的副本。而exec系列函數則可以幫助我們建立一個全新的新進程。

```
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[]);
```

以上函數在unistd.h聲明。

下面我們以execl()函數爲例：

```
#include <stdio.h>

#include <unistd.h>

int main()

execl("/bin/ls","ls","-l",NULL);

printf("如果execl執行失敗，這個就會打印出來了\n");

return1;
```

}該程序運行到execle()時，載入ls程序，並且覆蓋當前程序的空間。這樣就參數了一個新的進程，但是注意，這個新進程的PID跟載入它的進程是一樣的。

3.fork()和exec()一起調用

fork()可以創建子進程，但是子進程只是父進程的副本。我們可以利用exec()函數在子進程來重新載入一個全新的進程。下面看一個兩個函數聯用的列子。

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

pid_tpid = fork();

switch(pid )

{

case0:

printf("子進程\n");

execl("/bin/ls","ls","-l",NULL);

case-1:

printf("fork失敗了\n");

exit(1);

default:
```

```
wait(NULL);

printf("完成了喲！\n");

exit(0);

}
```

首先，fork建立子進程，然後在子進程中使用execl()產生一個ls程序的進程。而父進程則調用wait()來等待，直到子進程調用結束。

附錄：
說是exec系統調用，實際上在Linux中，並不存在一個exec()的函數形式，exec指的是一組函數，一共有6個，分別是：

```
#include<unistd.h>
extern char **environ;
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是真正意義上的系統調用，其它都是在此基礎上經過包裝的庫函數。

exec函數族的作用是根據指定的文件名找到可執行文件，並用它來取代調用進程的內容，換句話說，就是在調用進程內部執行一個可執行文件。這裏的可執行文件既可以是二進制文件，也可以是任何Linux下可執行的腳本文件。

與一般情況不同，exec函數族的函數執行成功後不會返回，因爲調用進程的實體，包括代碼段，數據段和堆棧等都已經被新的內容取代，只留下進程ID等一些表面上的信息仍保持原樣，頗有些神似”三十六計”中的”金蟬脫殼”。看上去還是舊的軀殼，卻已經注入了新的靈魂。只有調用失敗了，它們纔會返回一個-1，從原程序的調用點接着往下執行。

現在我們應該明白了，Linux下是如何執行新程序的，每當有進程認爲自己不能爲系統和擁護做出任何貢獻了，他就可以發揮最後一點餘熱，調用任何一個exec，讓自己以新的面貌重生；或者，更普遍的情況是，如果一個進程想執行另一個程序，它就可以fork出一個新進程，然後調用任何一個exec，這樣看起來就好像通過執行應用程序而產生了一個新進程一樣。

事實上第二種情況被應用得如此普遍，以至於Linux專門爲其作了優化，我們已經知道，fork會將調用進程的所有內容原封不動的拷貝到新產生的子進程中去，這些拷貝的動作很消耗時間，而如果fork完之後我們馬上就調用exec，這些辛辛苦苦拷貝來的東西又會被立刻抹掉，這看起來非常不划算，於是人們設計了一種”寫時拷貝（copy-on-write）”技術，使得fork結束後並不立刻複製父進程的內容，而是到了真正實用的時候才複製，這樣如果下一條語句是exec，它就不會白白作無用功了，也就提高了效率。

返回值
如果執行成功則函數不會返回，執行失敗則直接返回-1，失敗原因存於errno 中。

大家在平時的編程中，如果用到了exec函數族，一定記得要加錯誤判斷語句。因爲與其他系統調用比起來，exec很容易受傷，被執行文件的位置，權限等很多因素都能導致該調用的失敗。最常見的錯誤是：
1.找不到文件或路徑，此時errno被設置爲ENOENT；
2.數組argv和envp忘記用NULL結束，此時errno被設置爲EFAULT；
3.沒有對要執行文件的運行權限，此時errno被設置爲EACCES。

l表示以參數列表的形式調用

v表示以參數數組的方式調用

e表示可傳遞環境變量

p表示PATH中搜索執行的文件，如果給出的不是絕對路徑就會去PATH搜索相應名字的文件，如PATH沒有設置，則會默認在/bin,/usr/bin下搜索。

另：調用時參數必須以NULL結束。原進程打開的文件描述符是不會在exec中關閉的，除非用fcntl設置它們的“執行時關閉標誌（close on exec）”而原進程打開的目錄流都將在新進程中關閉。

例子：

```
#include <unistd.h>
int main(int argc, char *argv[])
{
char *envp[]={“PATH=/tmp”, “USER=lei”, “STATUS=testing”, 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(execve("/usr/bin/env", argv_execve, envp)<0)
perror("Err on execve");
}
}
```

其他參考：

Linux--exec函數族及system函數

http://blog.csdn.net/cnctloveyu/article/details/4129520

http://blog.chinaunix.net/space.php?uid=20384806&do=blog&cuid=392843

linux下exec系列函數使用範例

http://hi.baidu.com/colin719/blog/item/f6ea44e782e1152fb938205c.html

Linux--exec函數族以及管道技術

http://user.qzone.qq.com/119994997/blog/1236688022

全站分類：進修深造
個人分類：linux
此分類上一篇： bzero & memset
此分類下一篇： 平行處理
上一篇： bzero & memset
下一篇： 平行處理
歷史上的今天
2012: return與exit()的差別
2012: 平行處理
2012: bzero & memset
2012: 360度全景攝影機
2012: 讀懂函式庫的 man page

▲TOP

引用列表 (0)

http://ryan0988.pixnet.net/blog/trackback/31cb2593a0/67029604 點我複製

留言列表 (0)