第八章 进程控制

```
mail ~ # whereis init
init: /usr/src/linux-3.6.11-gentoo/init /usr/src/linux/init /sbin/init
```
源码处

简单看了一下，以我现在功力，还无法吸收它，先做如是状，待以后来处理。

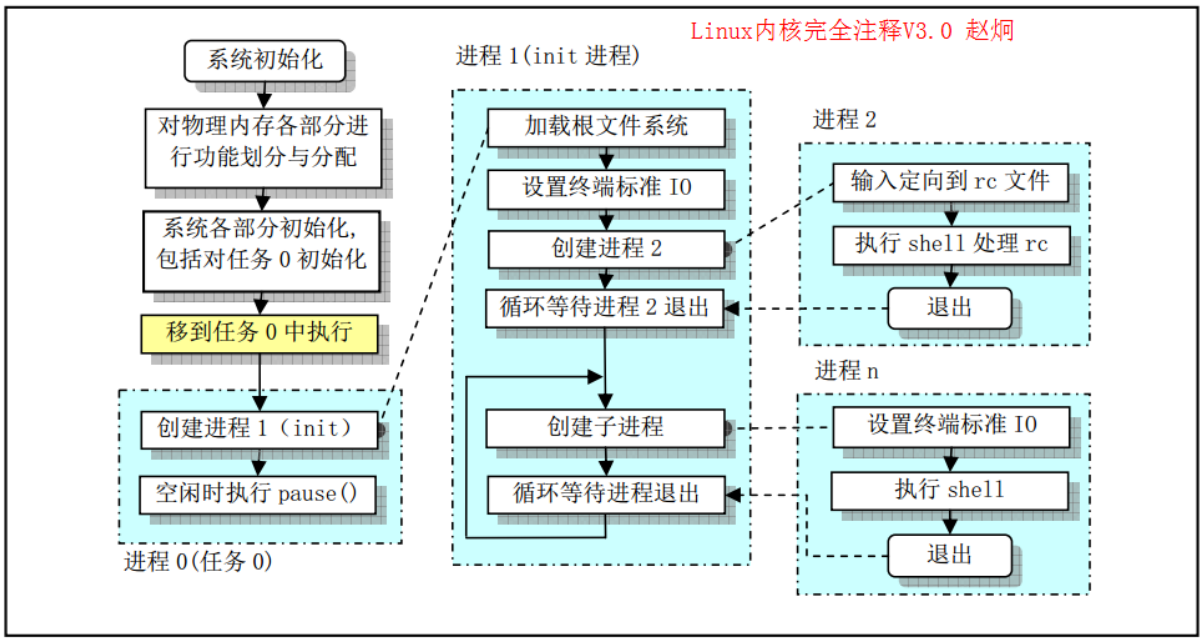


图 7-2 内核初始化程序流程示意图

基本进程控制原语：创建新进程——>fork；执行新程序——>exec；等待程序终止——>wait；终止程序——>exit。

突发奇想的比喻：

fork，就如同生孩子的过程，孩子生出来后，两个对象出声，生母喊孩子(名字)生出来了，孩子喊(哇啊哇啊……)；

exec，就如同孩子的成长，想让孩子成为艺术家，就给他艺术家的路线，想成为科学家，就让他走科学家路线，这样到最后，他的名字、性别、肤色都不会变。

当然，exec也可以比喻成转行，总之就是一个学习方向的开端。

一个比较有趣的题，问：下面程序各自会输出几个x?

I、第一种情况：

```c
#include <stdio.h>
int main(int argc, const char *argv[]) {
    int i;
    for(i=0; i<2; i++){
        fork();
        printf("x");
    }
    return 0;
}
```

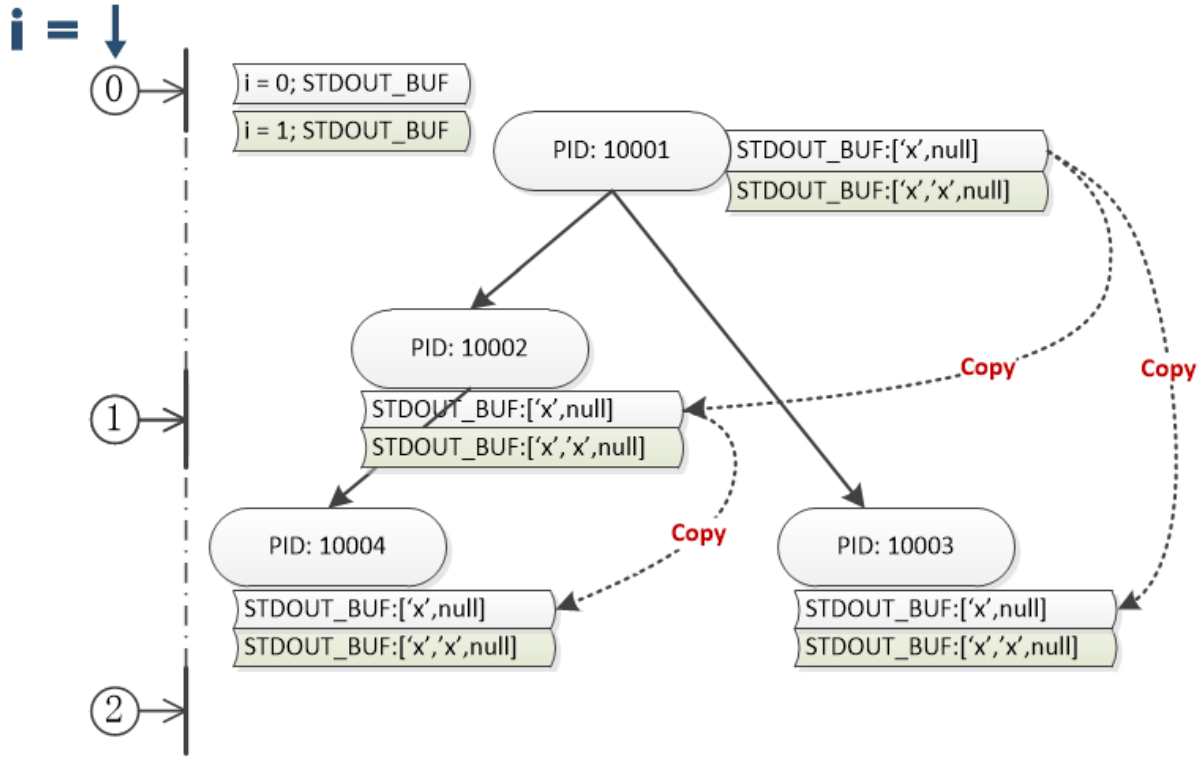
II、换种情况呢?

```c
#include <stdio.h>
int main(int argc, const char *argv[]) {
    int i;
    printf("x");
    for(i=0; i<2; i++){
        fork();
    }
    return 0;
}
```

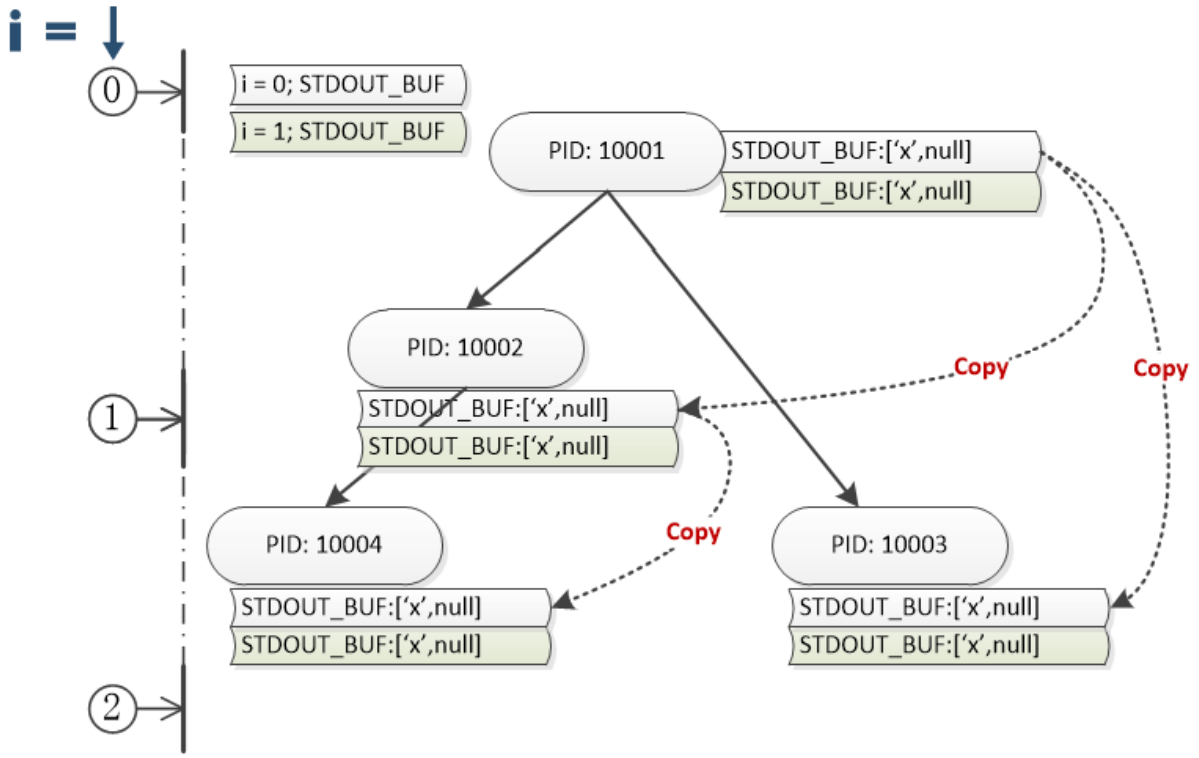
III、或这样呢?

```c
#include <stdio.h>
int main(int argc, const char *argv[]) {
    int i;
    printf("x\n");
    for(i=0; i<2; i++){
        fork();
    }
    return 0;
}
```

图解上述题：
涉及AUPE 5.4节内容；
解题I：



解题II：


i =
0
1
2
i = 0; STDOUT_BUF
i = 1; STDOUT_BUF
PID: 10001
STDOUT_BUF:[‘x’,null]
STDOUT_BUF:[‘x’,null]
PID: 10002
STDOUT_BUF:[‘x’,null]
STDOUT_BUF:[‘x’,null]
PID: 10004
STDOUT_BUF:[‘x’,null]
STDOUT_BUF:[‘x’,null]
PID: 10003
STDOUT_BUF:[‘x’,null]
STDOUT_BUF:[‘x’,null]
Copy
Copy
Copy

解题III：

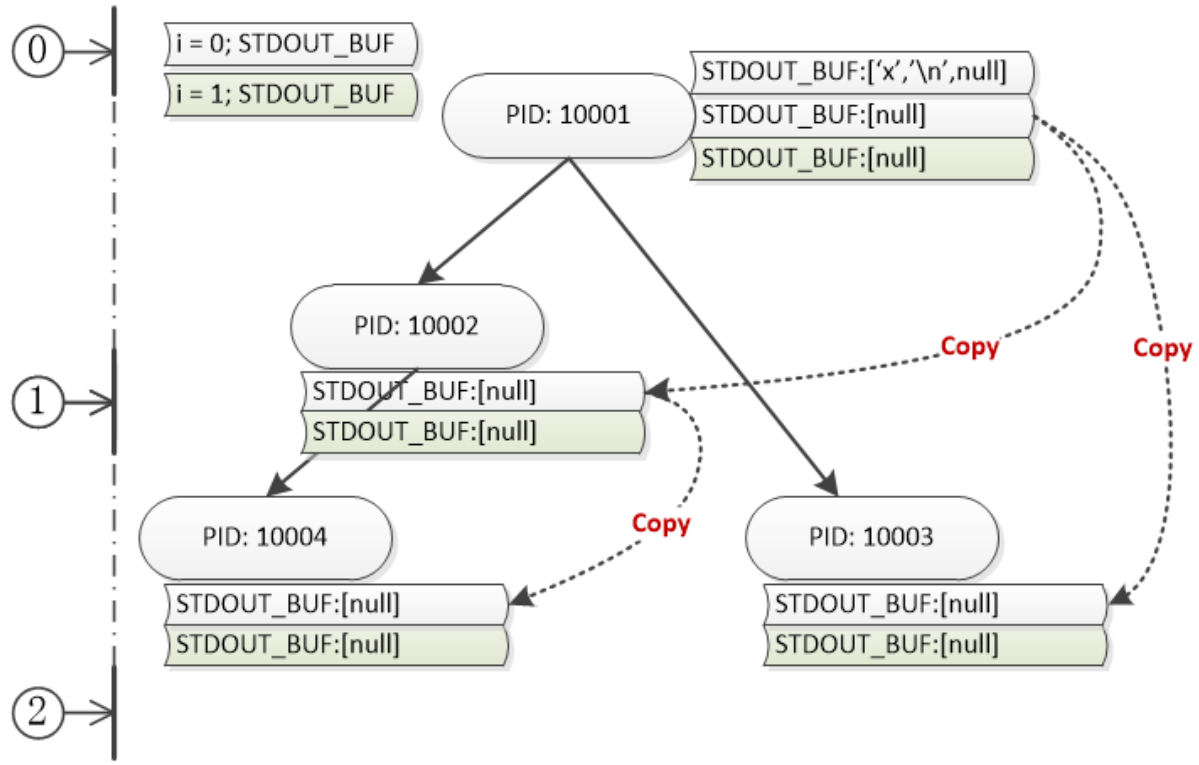

0
1
2
i = 0; STDOUT_BUF
i = 1; STDOUT_BUF
PID: 10001
STDOUT_BUF:[‘x’,’\n’,null]
STDOUT_BUF:[null]
STDOUT_BUF:[null]
PID: 10002
STDOUT_BUF:[null]
STDOUT_BUF:[null]
PID: 10004
STDOUT_BUF:[null]
STDOUT_BUF:[null]
PID: 10003
STDOUT_BUF:[null]
STDOUT_BUF:[null]
Copy
Copy
Copy

表8-6　6个exec函数之间的区别

函　数	*pathname*	*filename*	参数表	*argv*[]	environ	*envp*[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
名字中的字母		p	l	v		e

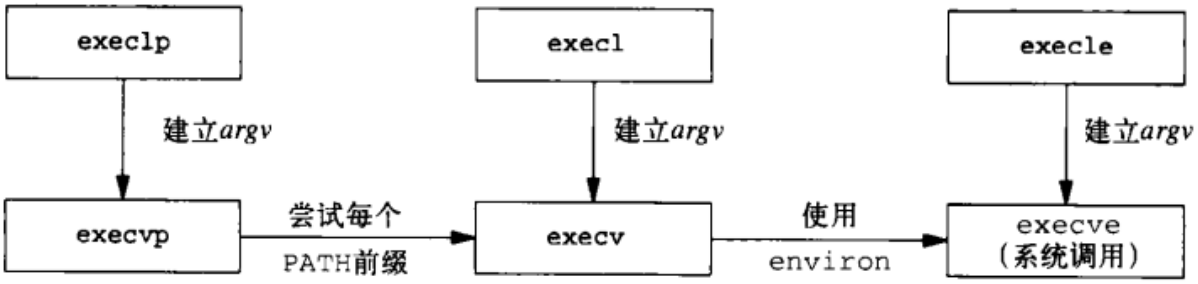


图8-2　6个exec函数之间的关系

下面这段程序，能让人更好的理解exec；

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, const char *argv[]) {
    printf("Start!\n");
    printf("%d\n", execl("/bin/echo", "echo", "executed by execl", NULL));
    printf("End!\n");
    return 0;
}
```

序。当进程调用一种exec函数时，该进程执行的程序完全替换为新程序，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用一个全新的程序替换了当前进程的正文、数据、堆和栈段。

脚本解释器：下面列出3种有各自特色语法的脚本，以加深对脚本及脚本解释器的理解；

```
for (( i=0; i<=$#; i++ ))
do
    eval echo '$'$i;
done
/opt/cpp/interpreter_example.sh
```

```
mail cpp # chmod +x interpreter_example.sh
mail cpp # ./interpreter_example.sh xx yy
./interpreter_example.sh
xx
yy
```

```
#!/usr/bin/php
<?php
    print_r($argv);
?>
/opt/cpp/interpreter_example.php
```

```
mail cpp # chmod +x interpreter_example.php
mail cpp # ./interpreter_example.php xx yy
Array
(
    [0] => ./interpreter_example.php
    [1] => xx
    [2] => yy
)
```

```
#!/usr/bin/python2.7
import sys
for arg in sys.argv:
    print arg;
/opt/cpp/interpreter_example.py
```

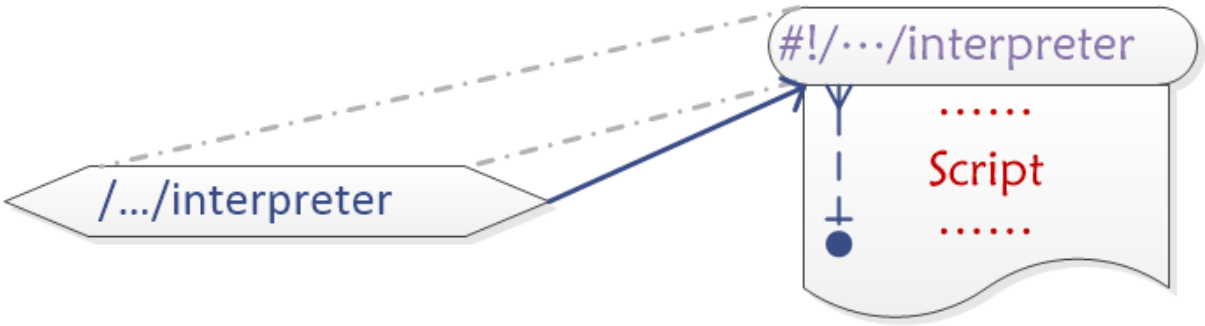
```
mail cpp # chmod +x interpreter_example.py
mail cpp # ./interpreter_example.py xx yy
./interpreter_example.py
xx
yy
```

问：上图中的shell脚本为什么不需要指定解释器?

解：

(3) 解释器脚本使我们可以使用除/bin/sh以外的其他shell来编写shell脚本。当execlp找到一个非机器可执行的可执行文件时，它总是调用/bin/sh来解释执行该文件。但是，使用解释器脚本，则可编写成：

```
#!/bin/csh
```



"#!/.../interpreter"声明文本红色部分的脚本由解释器"/.../interpreter"来解释。

把上面php脚本部分变一下，如下：

第一次扩展：

```
1 <?php
2     print_r($argv);
3 ?>
/opt/cpp/interpreter_example.php.2
```

```
mail cpp # /usr/bin/php interpreter_example.php.2 xx yy
Array
(
    [0] => interpreter_example.php.2
    [1] => xx
    [2] => yy
)
```

第二次扩展：

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, const char *argv[]) {
    execl("/usr/bin/php", "look_at_me", "/opt/cpp/interpreter_example.php.2", "xx", "yy", NULL);
    return 0;
}
```

/opt/cpp/exec_script_example.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=001/7(14%)]

```php
<?php
    print_r($argv);
    sleep(100000);
?>
```

稍作修改

/opt/cpp/interpreter_example.php.2

```
mail cpp # gcc exec_script_example.c -o exec_script_example
mail cpp # ./exec_script_example &
[1] 24167
mail cpp # Array
(
    [0] => /opt/cpp/interpreter_example.php.2
    [1] => xx
    [2] => yy
)

mail cpp # ps aux|grep -v grep|grep 'look_at_me'
root     24167  0.5  0.4  84748  8776 pts/0    S    17:19   0:00 look_at_me /opt/cpp/interpreter_example.php.2 xx yy
```

http://man7.org/linux/man-pages/man2/execve.2.html

Interpreter scripts

An interpreter script is a text file that has execute permission enabled and whose first line is of the form:

    #! interpreter [optional-arg]

The *interpreter* must be a valid pathname for an executable which is not itself a script. If the *filename* argument of **execve**() specifies an interpreter script, then *interpreter* will be invoked with the following arguments:

    interpreter [optional-arg] filename arg...

where *arg...* is the series of words pointed to by the *argv* argument of **execve**(), starting at *argv*[1].

For portable use, *optional-arg* should either be absent, or be specified as a single word (i.e., it should not contain white space); see NOTES below.

在该程序中先调用execle，它要求一个路径名和一个特定的环境。下一个调用的是execlp，它用一个文件名，并将调用者的环境传送给新程序。execlp在这里能够工作的原因是因为目录/home/sar/bin是当前路径前缀之一。注意，我们将第一个参数（新程序中的argv[0]）设置为路径名的文件名分量。某些shell将此参数设置为完整的路径名。这只是一个惯例。我们可将argv[0]设置为任何字符串。当login命令执行shell时就是这样做的。在执行shell之前，login在argv[0]之前加一个/作为前缀，这向shell指明它是作为登录shell被调用的。登录shell将执行启动配置文件（start-up profile）命令，而非登录shell则不会执行这些命令。

程序清单8-8中要执行两次的程序echoall示于程序清单8-9中。这是一个很普通的程序，它回送其所有命令行参数及全部环境表。

第三次扩展：

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    execv(argv[1], (argv+2));
    return 0;
}
```

/opt/cpp/exec_script_example_2.c [FORMAT=unix:utf-8]

```
mail cpp # gcc exec_script_example_2.c -o exec_script_example_2
mail cpp # ./exec_script_example_2 /usr/bin/php look_at_me /opt/cpp/interpreter_example.php.2 xx yy &
[1] 28627
mail cpp # Array
(
    [0] => /opt/cpp/interpreter_example.php.2
    [1] => xx
    [2] => yy
)

mail cpp # ps aux|grep -v grep|grep look_at_me
root     28627  1.3  0.4  84748  8776 pts/0    S    17:48   0:00 look_at_me /opt/cpp/interpreter_example.php.2 xx yy
```

关于用户id，第四章笔记有做详细演示。

```
user    www www;
worker_processes    8;
```

nginx.conf

```
; Unix user/group of processes
; Note: The user is mandatory. If
;       will be used.
user = www
group = www
/etc/php/fpm-php5.4/php-fpm.conf
```

习题：

8-1、linux上vfork中的exit并没关闭标准输出流，所以为达到该题要求，添加红框中的fclose来声明它；

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int glob = 6;
int main(int argc, const char *argv[]) {
    int var;
    pid_t pid;
    var = 88;
    printf("Befor vfork!\n");
    if (0 > (pid = vfork())) {
        printf("Error by vfork!\n");
        return 1;
    } else if (0 == pid) {
        glob++;
        var++;
        fclose(stdout);
        exit(0);
    }
    fprintf( stderr, "printf result: %d\n", printf("PID:%d, glob:%d, var:%d\n", pid, glob, var));
    return 0;
}
```

stdout已关闭，所以这里选择stderr做输出

两次输出结果：

fclose被注释掉

```
mail cpp # gcc 8-1.c -o 8-1
mail cpp # ./8-1
Befor vfork!
PID:14282, glob:7, var:89
printf result: 26
```

fclose被启用

```
mail cpp # gcc 8-1.c -o 8-1
mail cpp # ./8-1
Befor vfork!
printf result: -1
```

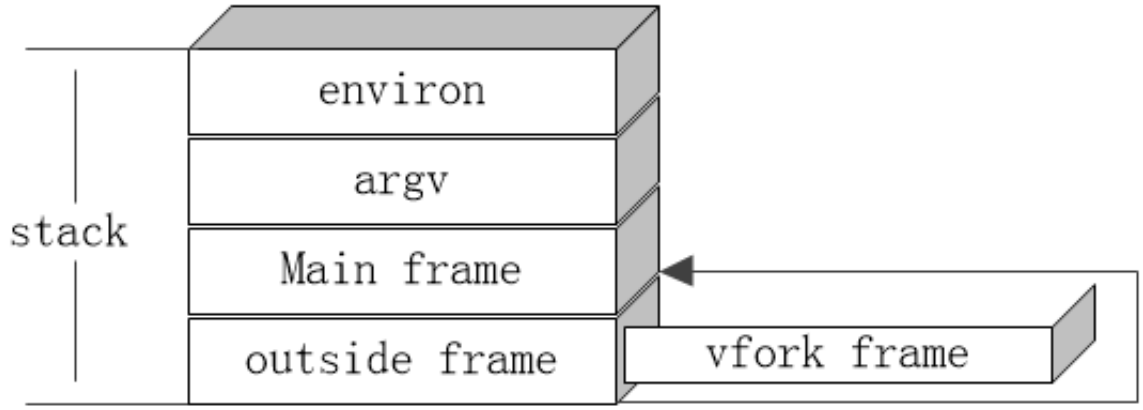
回顾：

对一个进程预定义了三个流，并且这三个流可以自动地被进程使用，它们是：标准输入、标准输出和标准出错。这些流引用的文件与3.2节中提到的文件描述符STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO所引用的文件相同。

这三个标准I/O流通过预定义文件指针stdin、stdout和stderr加以引用。这三个文件指针同样定义在头文件<stdio.h>中。

当一个进程正常终止时（直接调用exit函数，或从main函数返回），则所有带未写缓冲数据的标准I/O流都会被冲洗，所有打开的标准I/O流都会被关闭。 5.4节最后一段

8-2、函数会等vfork结束后，再返回。



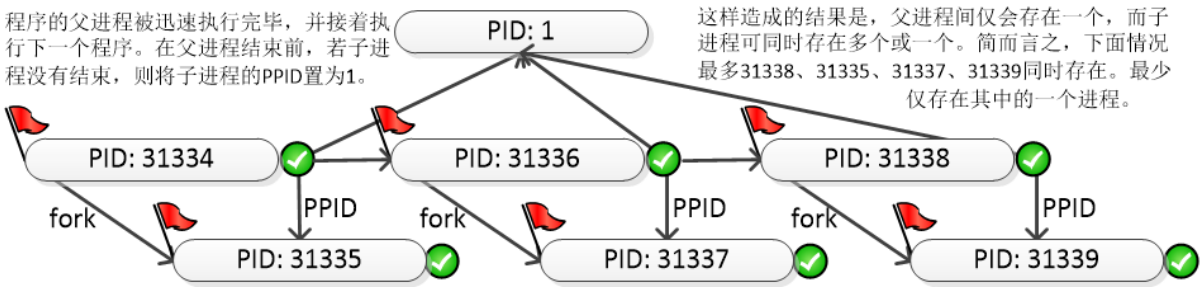
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void pr_exit(int status) {
    if (WIFEXITED(status)) {
        printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("abnormal termination, signal number = %d%s\n", WTERMSIG(status), WCOREDUMP(status) ? " (core file generated)" : "");
    } else if (WIFSTOPPED(status)) {
        printf("child stopped, signal number = %d\n", WSTOPSIG(status));
    }
}

pid_t pid;
int status;
int outside() {
    if (0 > (pid = vfork())) {
        printf("Error by vfork!\n");
        _exit(1);
    } else if (0 == pid) {
        printf("print by child!\n");
        sleep(2);
        exit(0);
    }
    printf("print by funcation!\n");
    return 0;
}
```

```c
int main(int argc, const char *argv[]) {
    printf("Befor vfork!\n");
    printf("%d\n", outside());
    if (waitpid( pid, &status, 0) != pid) {
        printf("Error by wait!\n");
    }
    printf("print by main!\n");
    pr_exit(status);
    return 0;
}
```

/opt/cpp/8-2.c [FORMAT=unix:utf-8] [TYPE=C]

```
mail cpp # gcc 8-2.c -o 8-2
mail cpp # ./8-2
Befor vfork!
print by child!
print by funcation!
0
print by main!
normal termination, exit status = 0
```

8-3、题中命令间分号的作用为->(当前一个命令执行完毕后(不论返回值等于多少)，才执行分号后面的命令)；输出该结果的过程如下图所示；使子进程先输出时，该问题不存在。




```c
#include <stdio.h>
#include <fcntl.h>          // For open
#include <sys/types.h>      // pid_t;

static void charatatime(char *str) {
    char *ptr;
    int c;
    setbuf(stdout, NULL);
    for (ptr = str; 0 != (c = *ptr++); ) {
        usleep(10000);
        putc(c, stdout);
    }
}
```

使字符串输出时间变长，从而产生能被内核调度器再次调入执行的机会

```c
int main(int argc, const char *argv[]) {
    printf("%d Begin!\n", getpid());
    pid_t pid;
    int fd;
    char path[BUFSIZ], flag = 'p';
    sprintf( path, "./pid_%d.flag", getpid());
    fd = open( path, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    unlink( path);
    pwrite( fd, &flag, sizeof(flag), 0);
    if (0 > (pid = fork())) {
        printf("error by fork!\n");
    } else if (0 == pid) {
        pread( fd, &flag, sizeof(flag), 0);
        while('c' != flag) {
            pread( fd, &flag, sizeof(flag), 0);
        }使用原子读写避免过程中另一个进程更改其偏移量
        charatatime("output from child\n");
        flag = 'p';
        pwrite( fd, &flag, sizeof(flag), 0);
        printf("%d Over!\n", getpid());
    } else {
        pread( fd, &flag, sizeof(flag), 0);
        while('p' != flag) {
            pread( fd, &flag, sizeof(flag), 0);
        }
        charatatime("output from parent\n");
        flag = 'c';
        pwrite( fd, &flag, sizeof(flag), 0);
    }
    return 0;
}
/opt/cpp/8-3.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=
```

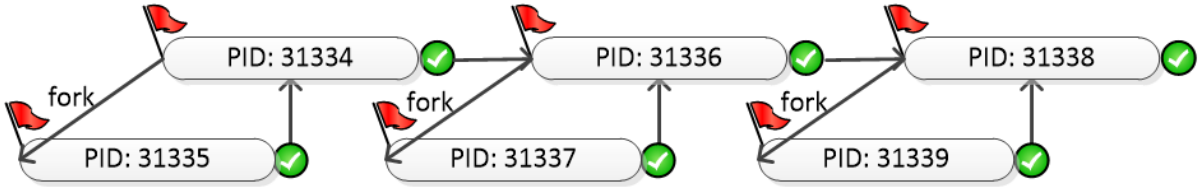
控制父进程与子进程谁先输出

```
mail cpp # gcc 8-3.c -o 8-3
mail cpp # ./8-3; ./8-3; ./8-3;
31334 Begin!
output from parent
31336 Begin!
oouuttppuutt  ffrroomm  cphairledn
31335 Over!
t          与习题8-3输出类似
31338 Begin!
ooutuptutp ufrto m fcrhiolmd
31337 Over!
 parent
mail cpp # output from child
31339 Over!
```

```
 18      int fd;  改为使子进程先输出
 19      char path[BUFSIZ], flag = 'c';
/opt/cpp/8-3.c [FORMAT=unix:utf-8] [TYPE=C]
```

```
mail cpp # gcc 8-3.c -o 8-3
mail cpp # ./8-3; ./8-3; ./8-3;
31508 Begin!
output from child
31509 Over!子进程先输出的效果
output from parent
31510 Begin!
output from child
31511 Over!
output from parent
31512 Begin!
output from child
31513 Over!
output from parent
```

父子进程同时存在，但每时每刻，都将只有一个进程输出



8-4、显示全路径；

argv[1]和argv[2]已右移了两个位置。注意，内核取execl调用中的*pathname*而非第一个参数（testinterp），因为一般而言，*pathname*包含了比第一个参数更多的信息。 □

```
mail cpp # cat testinterp
#!/opt/cpp/echoarg foo
mail cpp # cat echoarg.c
#include <stdio.h>

int main(int argc, const char *argv[]) {
    int i;
    for ( i=0; NULL != argv[i]; i++) {
        printf("argv[%d] -> %s\n", i, argv[i]);
    }
    return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, const char *argv[]) {
    pid_t pid;
    if (0 > (pid = fork())) {
        printf("error by fork!\n");
    } else if (0 == pid) {
        if (0 > execlp("testinterp", "xxtestinterp", "myarg1", "MY ARG2", (char *) 0)) {
            printf("error by execlp!\n");
        }
    }
    if (0 > waitpid( pid, NULL, 0)) {
        printf("waitpid error!\n");
    }
    return 0;
}
```

方便区分

```
mail cpp # ln -s /opt/cpp/testinterp /usr/bin/testinterp
mail cpp # ./8-4
argv[0] -> /opt/cpp/echoarg
argv[1] -> foo
argv[2] -> /usr/bin/testinterp
argv[3] -> myarg1
argv[4] -> MY ARG2
```

8-5、标准中没有直接获取“保存的设置用户id”的函数，具体获得方法，参见第四章笔记中的reader.cpp程序，程序中的euid变量保存的就是“保存的设置用户id”；

(4) 当执行完过滤器操作后，man调用setuid (*euid*)，其中*euid*是用户man的数值用户ID (man调用geteuid，得到用户man的用户ID，然后将其保存起来)。因为setuid的参数等于保存的设置用户ID，所以这种调用是许可的（这就是为什么需要保存的设置用户ID的原因）。现在得到

8-6、上面8-3中的程序，由子进程先执行时，就会产生僵尸进程；

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, const char *argv[]) {
    pid_t pid;
    char cmd[BUFSIZ] = "ps aux | grep Z | grep -v grep";
    if (0 > (pid = fork())) {
        printf("error by fork!\n");
    } else if (0 == pid) {
        printf("Bye bye!\n");
        return 0;
    }
    // if (0 > waitpid( pid, NULL, 0)) {
    //     printf("waitpid error!\n");
    // }
    sleep(1);
    system(cmd);
    return 0;
}
```

后面的不再执行，立即退出

避免僵尸进程出现的方法之一

等待子进程结束（仅仅为了测试）

/opt/cpp/8-6.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [R

```
mail cpp # gcc -g 8-6.c -o 8-6
mail cpp # ./8-6
Bye bye!
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      6448  0.0  0.0      0     0 pts/1    Z+   03:04   0:00 [8-6] <defunct>
```

8-7、

```
tongue aupe # grep -r __dirstream /usr/include/*
/usr/include/dirent.h:typedef struct __dirstream DIR;
```

```
struct __dirstream
  {
    int fd;         /* File descriptor.  */

    __libc_lock_define (, lock) /* Mutex lock for this structure.  */

    size_t allocation;      /* Space allocated for the block.  */
    size_t size;        /* Total valid data in the block.  */
    size_t offset;      /* Current offset into the block.  */

    off_t filepos;      /* Position of next entry to read.  */

    /* Directory block.  */
    char data[0] __attribute__ ((aligned (__alignof__ (void*))));
  };
```

~/glibc-2.17/sysdeps/posix/dirstream.h [FORMAT=unix:utf-8] [TYPE=CPP]

```c
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <fcntl.h>

int main(int argc, const char *argv[]) {
    DIR *pDir = NULL;
    pDir = opendir("/");
    printf( "FD_CLOEXEC by opendir: %d\n", fcntl( dirfd(pDir), F_GETFD));
    // (gdb) p (int) (DIR *) *pDir
    printf( "FD_CLOEXEC by open: %d\n", fcntl( open("/", O_RDONLY), F_GETFD));
    return 0;
}
```

```
tongue aupe # gcc -g 8-7.c -o 8-7
tongue aupe # ./8-7
FD_CLOEXEC by opendir: 1
FD_CLOEXEC by open: 0
```