4. 进程控制

本节重点：

- 学习进程创建,fork/vfork
- 学习到进程终止,认识$?
- 学习到进程等待
- 学习到进程程序替换
- 微型shell，重新认识shell运行原理

1. 进程创建

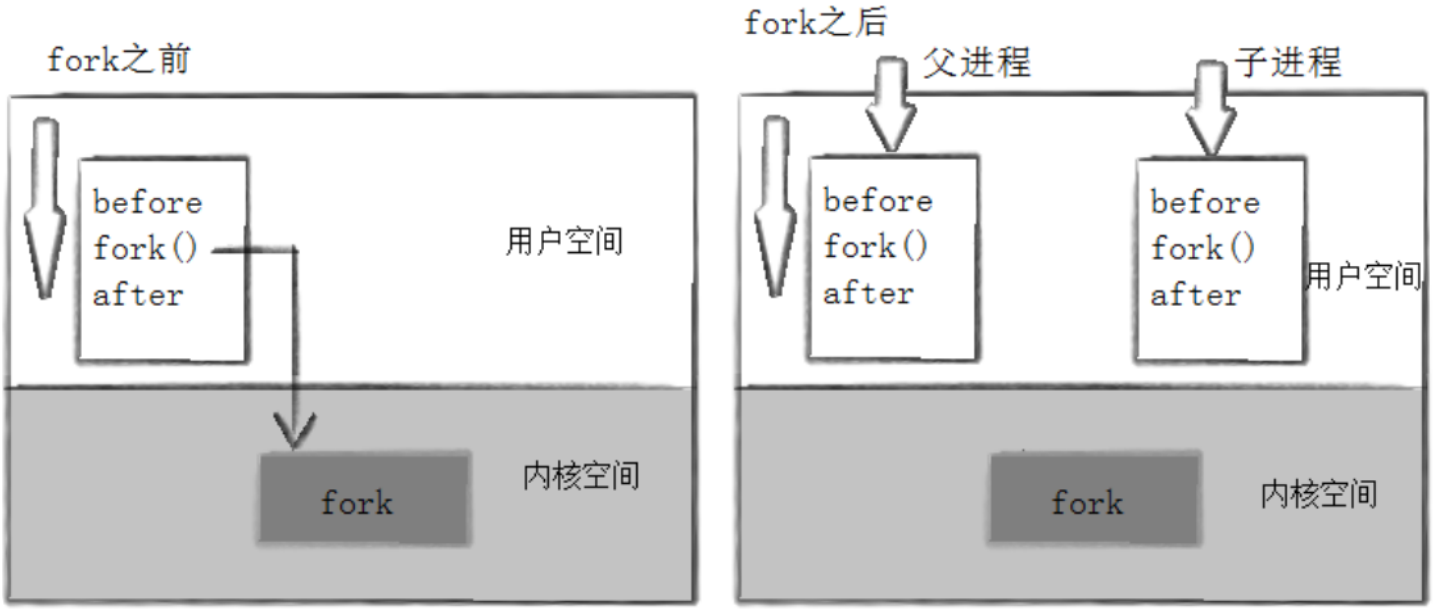
1-1 fork函数初识

在linux中fork函数是非常重要的函数，它从已存在进程中创建一个新进程。新进程为子进程，而原进程为父进程。

```
#include <unistd.h>
pid_t fork(void);
返回值：自进程中返回0，父进程返回子进程id，出错返回-1
```

进程调用fork，当控制转移到内核中的fork代码后，内核做：

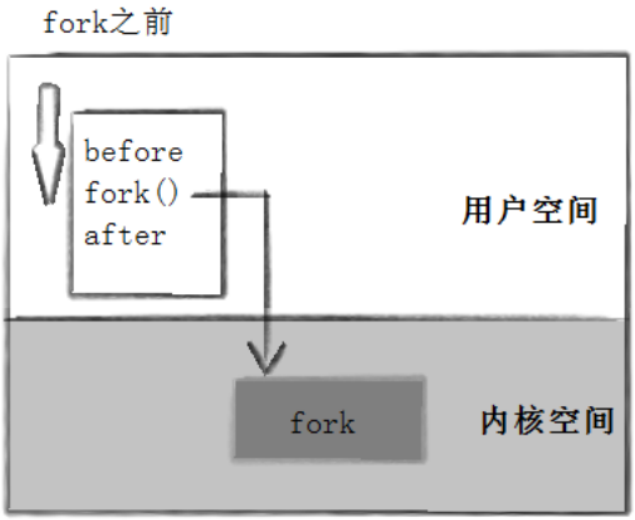
> - 分配新的内存块和内核数据结构给子进程
> - 将父进程部分数据结构内容拷贝至子进程
> - 添加子进程到系统进程列表当中
> - fork返回，开始调度器调度

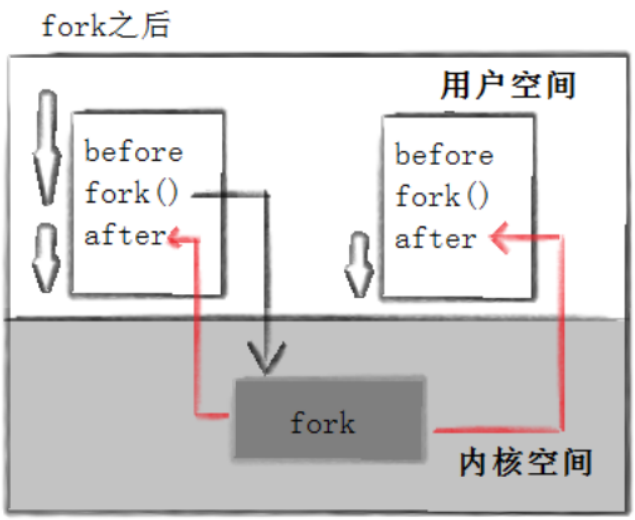


当一个进程调用fork之后，就有两个二进制代码相同的进程。而且它们都运行到相同的地方。但每个进程都将可以开始它们自己的旅程，看如下程序。

```
1  int main( void )
2  {
3     pid_t pid;
4
5     printf("Before: pid is %d\n", getpid());
6     if ( (pid=fork()) == -1 )perror("fork()"),exit(1);
7     printf("After:pid is %d, fork return %d\n", getpid(), pid);
8          sleep(1);
9     return 0;
10 }
11
12 运行结果:
13 [root@localhost linux]# ./a.out
14 Before: pid is 43676
15 After:pid is 43676, fork return 43677
16 After:pid is 43677, fork return 0
```

这里看到了三行输出，一行before，两行after。进程43676先打印before消息，然后它有打印after。另一个after消息有43677打印的。注意到进程43677没有打印before，为什么呢？如下图所示





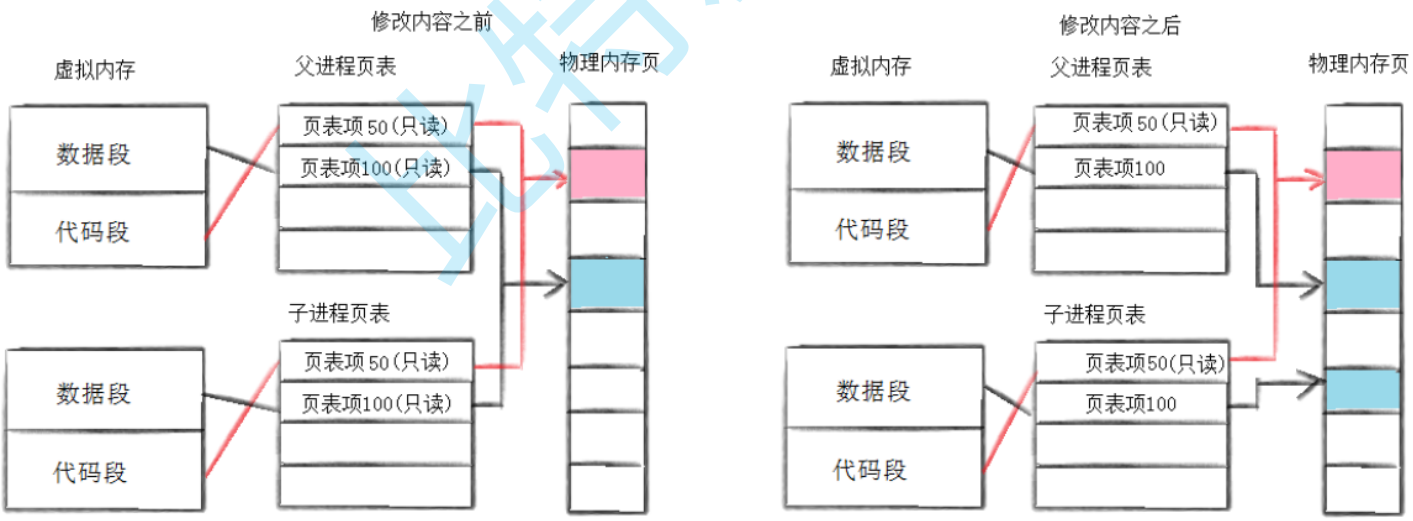
所以，fork之前父进程独立执行，fork之后，父子两个执行流分别执行。注意，fork之后，谁先执行完全由调度器决定。

1-2 fork函数返回值

- 子进程返回0，
- 父进程返回的是子进程的pid。

1-3 写时拷贝

通常，父子代码共享，父子再不写入时，数据也是共享的，当任意一方试图写入，便以写时拷贝的方式各自一份副本。具体见下图:



因为有写时拷贝技术的存在,所以父子进程得以彻底分离离！完成了进程独立性的技术保证!

写时拷贝,是一种延时申请技术,可以提高整机内存的使用率

1-4 fork常规用法

- 一个父进程希望复制自己，使父子进程同时执行不同的代码段。例如，父进程等待客户端请求，生成子进程来处理请求。

- 一个进程要执行一个不同的程序。例如子进程从fork返回后，调用exec函数。

1-5 fork调用失败的原因

- 系统中有太多的进程
- 实际用户的进程数超过了限制

2. 进程终止

进程终止的本质是释放系统资源，就是释放进程申请的相关内核数据结构和对应的数据和代码。

2-1 进程退出场景

- 代码运行完毕，结果正确
- 代码运行完毕，结果不正确
- 代码异常终止

2-2 进程常见退出方法

正常终止（可以通过 `echo $?` 查看进程退出码）：

1. 从main返回
2. 调用exit
3. _exit

异常退出：

- ctrl + c，信号终止

2-2-1 退出码

退出码（退出状态）可以告诉我们最后一次执行的命令的状态。在命令结束以后，我们可以知道命令是成功完成的还是以错误结束的。其基本思想是，程序返回退出代码 `0` 时表示执行成功，没有问题。代码 `1` 或 `0` 以外的任何代码都被视为不成功。

Linux Shell 中的主要退出码：

退出码	解释
0	命令成功执行
1	通用错误代码
2	命令（或参数）使用不当
126	权限被拒绝（或）无法执行
127	未找到命令，或 PATH 错误
128+n	命令被信号从外部终止，或遇到致命错误
130	通过 Ctrl+C 或 SIGINT 终止（终止代码 2 或键盘中断）
143	通过 SIGTERM 终止（默认终止）
255/*	退出码超过了 0-255 的范围，因此重新计算（LCTT 译注：超过 255 后，用退出取模）

- 退出码 `0` 表示命令执行无误，这是完成命令的理想状态。
- 退出码 `1` 我们也可以将其解释为 “不被允许的操作” 。例如在没有 sudo 权限的情况下使用 yum；再例如除以 `0` 等操作也会返回错误码 `1`，对应的命令为 `let a=1/0`
- `130`（`SIGINT` 或 `^C`）和 `143`（`SIGTERM`）等终止信号是非常典型的，它们属于 `128+n` 信号，其中 `n` 代表终止码。
- 可以使用strerror函数来获取退出码对应的描述。

2-3-2 _exit函数

```
#include <unistd.h>
void _exit(int status);
参数：status 定义了进程的终止状态，父进程通过wait来获取该值
```

- 说明：虽然status是int，但是仅有低8位可以被父进程所用。所以_exit(-1)时，在终端执行$?发现返回值是255。

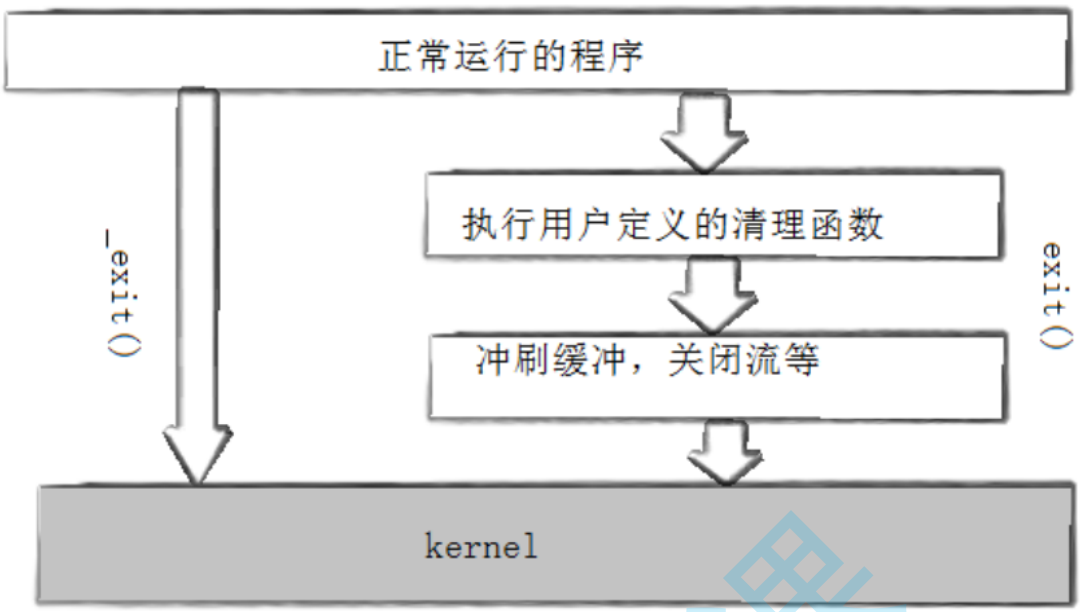
2-3-3 exit函数

```
#include <unistd.h>
void exit(int status);
```

exit最后也会调用_exit, 但在调用_exit之前，还做了其他工作：

1. 执行用户通过 atexit或on_exit定义的清理函数。

2. 关闭所有打开的流，所有的缓存数据均被写入
3. 调用_exit



实例：

```
1  int main()
2  {
3      printf("hello");
4      exit(0);
5  }
6  运行结果:
7  [root@localhost linux]# ./a.out
8  hello[root@localhost linux]#
9
10 int main()
11 {
12     printf("hello");
13     _exit(0);
14 }
15 运行结果:
16 [root@localhost linux]# ./a.out
17 [root@localhost linux]#
```

2-3-4 return退出

return是一种更常见的退出进程方法。执行return n等同于执行exit(n),因为调用main的运行时函数会将main的返回值当做 exit的参数。

3. 进程等待

3-1 进程等待必要性

- 之前讲过，子进程退出，父进程如果不管不顾，就可能造成‘僵尸进程’的问题，进而造成内存泄漏。
- 另外，进程一旦变成僵尸状态，那就刀枪不入，“杀人不眨眼”的kill -9 也无能为力，因为谁也没有办法杀死一个已经死去的进程。
- 最后，父进程派给子进程的任务完成的如何，我们需要知道。如，子进程运行完成，结果对还是不对，或者是否正常退出。
- 父进程通过进程等待的方式，回收子进程资源，获取子进程退出信息

3-2 进程等待的方法

3-2-1 wait方法

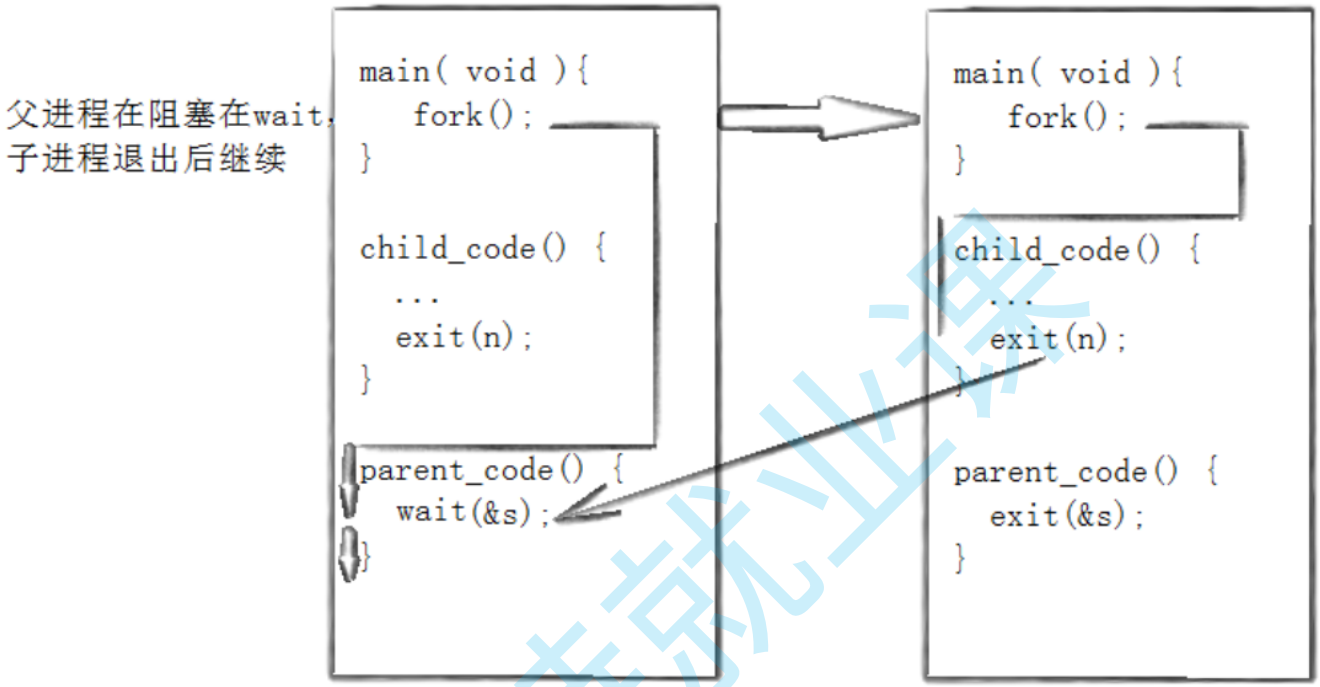
```
#include<sys/types.h>
#include<sys/wait.h>

pid_t wait(int* status);

返回值:
        成功返回被等待进程pid，失败返回-1。
参数:
        输出型参数，获取子进程退出状态,不关心则可以设置成为NULL
```

3-2-2 waitpid方法

```
pid_ t waitpid(pid_t pid, int *status, int options);
返回值:
        当正常返回的时候waitpid返回收集到的子进程的进程ID;
        如果设置了选项WNOHANG,而调用中waitpid发现没有已退出的子进程可收集,则返回0;
        如果调用中出错,则返回-1,这时errno会被设置成相应的值以指示错误所在;
参数:
        pid:
                Pid=-1,等待任一个子进程。与wait等效。
                Pid>0.等待其进程ID与pid相等的子进程。
        status: 输出型参数
                WIFEXITED(status): 若为正常终止子进程返回的状态，则为真。（查看进程是否是正常退出）
                WEXITSTATUS(status): 若WIFEXITED非零，提取子进程退出码。（查看进程的退出码）
        options:默认为0，表示阻塞等待
```

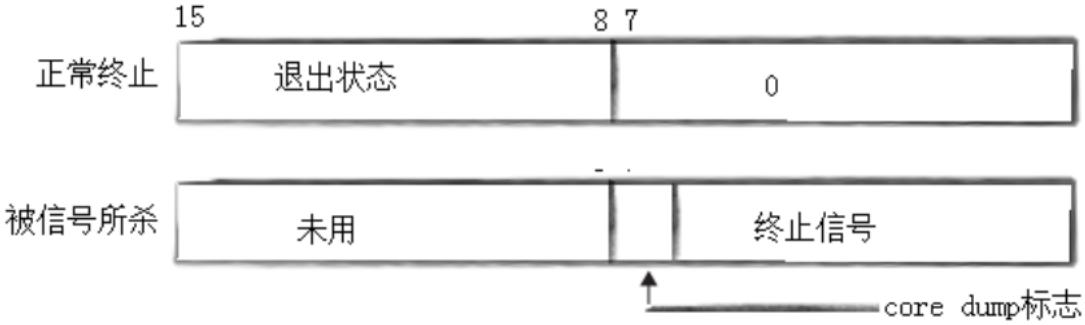
```
14                  WNOHANG：若pid指定的子进程没有结束，则waitpid()函数返回0，不予以等待。若正常结束，则返回该子进程的ID。
```

- 如果子进程已经退出，调用wait/waitpid时，wait/waitpid会立即返回，并且释放资源，获得子进程退出信息。
- 如果在任意时刻调用wait/waitpid，子进程存在且正常运行，则进程可能阻塞。
- 如果不存在该子进程，则立即出错返回。



3-2-3 获取子进程status

- wait和waitpid，都有一个status参数，该参数是一个输出型参数，由操作系统填充。
- 如果传递NULL，表示不关心子进程的退出状态信息。
- 否则，操作系统会根据该参数，将子进程的退出信息反馈给父进程。
- status不能简单的当作整形来看待，可以当作位图来看待，具体细节如下图（只研究status低16比特位）：




```
1  测试代码：
```

```c
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main( void )
{
    pid_t pid;

    if ( (pid=fork()) == -1 )
        perror("fork"),exit(1);

    if ( pid == 0 ){
        sleep(20);
        exit(10);
    } else {
        int st;
        int ret = wait(&st);

        if ( ret > 0 && ( st & 0X7F ) == 0 ){ // 正常退出
            printf("child exit code:%d\n", (st>>8)&0XFF);
        } else if( ret > 0 ) {  // 异常退出
            printf("sig code : %d\n", st&0X7F );
        }
    }
}

测试结果:
# ./a.out #等20秒退出
child exit code:10
# ./a.out #在其他终端kill掉
sig code : 9
```


3-2-4 阻塞与非阻塞等待

- 进程的阻塞等待方式:

```c
int main()
{
        pid_t pid;
        pid = fork();
        if(pid < 0){
            printf("%s fork error\n",__FUNCTION__);
            return 1;
```

```
        } else if( pid == 0 ){ //child
            printf("child is run, pid is : %d\n",getpid());
            sleep(5);
            exit(257);
        } else{
            int status = 0;
            pid_t ret = waitpid(-1, &status, 0);//阻塞式等待，等待5S
            printf("this is test for wait\n");
            if( WIFEXITED(status) && ret == pid ){
                        printf("wait child 5s success, child return code is
:%d.\n",WEXITSTATUS(status));
                }else{
                        printf("wait child failed, return.\n");
                        return 1;
                }
        }
        return 0;
}

运行结果:
[root@localhost linux]# ./a.out
child is run, pid is : 45110
this is test for wait
wait child 5s success, child return code is :1.
```

- 进程的非阻塞等待方式:

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

typedef void (*handler_t)();       // 函数指针类型
std::vector<handler_t> handlers;   // 函数指针数组

void fun_one() {
    printf("这是一个临时任务1\n");
}
void fun_two() {
    printf("这是一个临时任务2\n");
}
void Load() {
    handlers.push_back(fun_one);
    handlers.push_back(fun_two);
```

```
}
void handler() {
    if (handlers.empty())
        Load();
    for (auto iter : handlers)
        iter();
}

int main() {
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        printf("%s fork error\n", __FUNCTION__);
        return 1;
    } else if (pid == 0) {  // child
        printf("child is run, pid is : %d\n", getpid());
        sleep(5);
        exit(1);
    } else {
        int status = 0;
        pid_t ret = 0;
        do {
            ret = waitpid(-1, &status, WNOHANG);  // 非阻塞式等待
            if (ret == 0) {
                printf("child is running\n");
            }
            handler();
        } while (ret == 0);

        if (WIFEXITED(status) && ret == pid) {
            printf("wait child 5s success, child return code is :%d.\n",
                   WEXITSTATUS(status));
        } else {
            printf("wait child failed, return.\n");
            return 1;
        }
    }
    return 0;
}
```

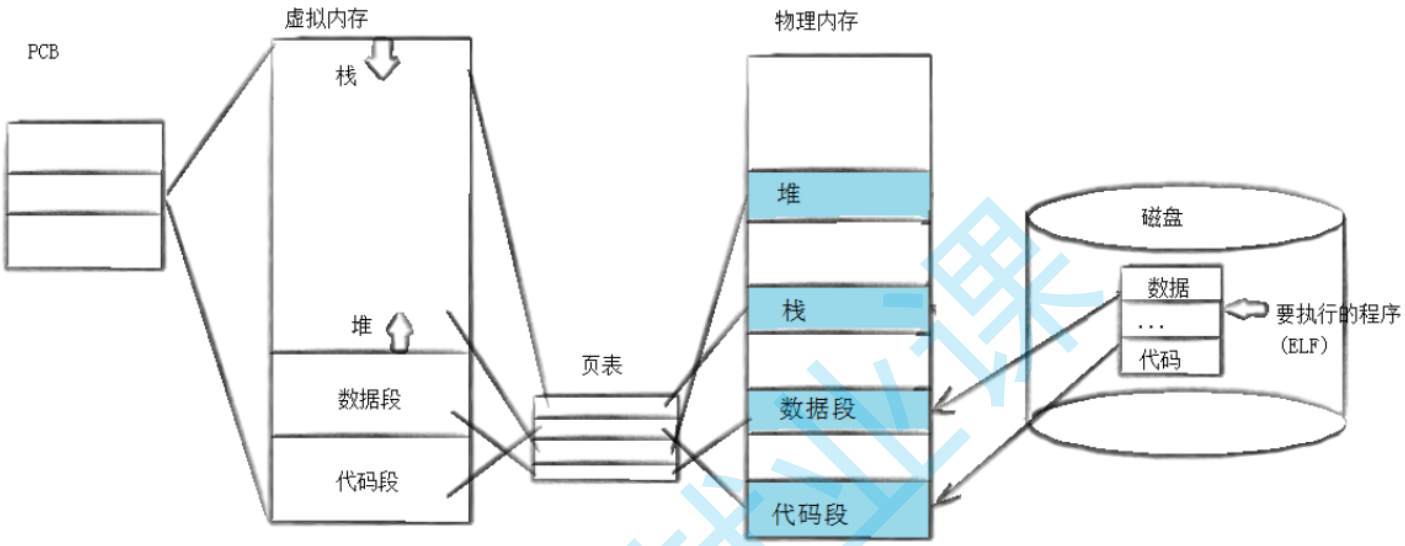

4. 进程程序替换

`fork()` 之后,父子各自执行父进程代码的一部分如果子进程就想执行一个全新的程序呢？进程的程序替换来完成这个功能！

程序替换是通过特定的接口，加载磁盘上的一个全新的程序(代码和数据)，加载到调用进程的地址空间中!

4-1 替换原理

用fork创建子进程后执行的是和父进程相同的程序(但有可能执行不同的代码分支),子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时,该进程的用户空间代码和数据完全被新程序替换,从新程序的启动例程开始执行。调用exec并不创建新进程,所以调用exec前后该进程的id并未改变。



4-2 替换函数

其实有六种以exec开头的函数,统称exec函数:

```c
#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[]);
```

4-2-1 函数解释

- 这些函数如果调用成功则加载新的程序从启动代码开始执行,不再返回。
- 如果调用出错则返回-1
- 所以exec函数只有出错的返回值而没有成功的返回值。

4-2-2 命名理解

这些函数原型看起来很容易混,但只要掌握了规律就很好记。

- l(list) : 表示参数采用列表
- v(vector) : 参数用数组
- p(path) : 有p自动搜索环境变量PATH
- e(env) : 表示自己维护环境变量

函数名	参数格式	是否带路径	是否使用当前环境变量
execl	列表	不是	是
execlp	列表	是	是
execle	列表	不是	不是，须自己组装环境变量
execv	数组	不是	是
execvp	数组	是	是
execve	数组	不是	不是，须自己组装环境变量

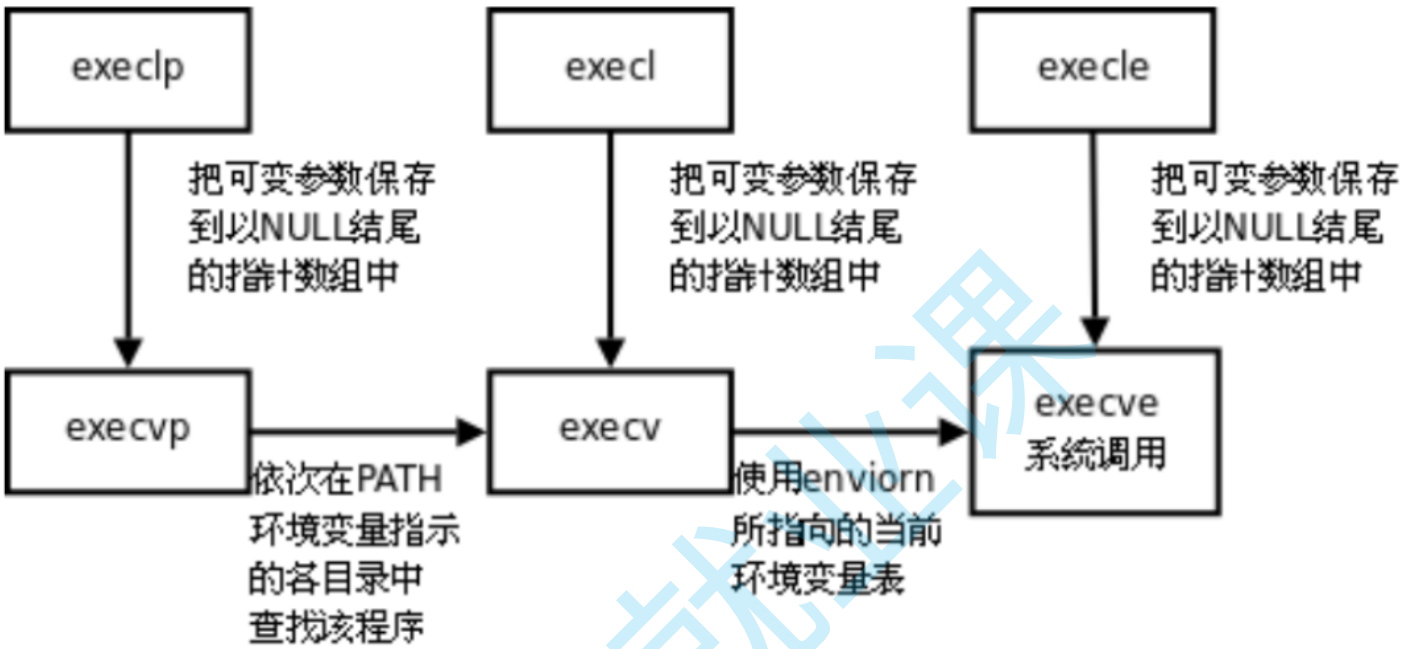
exec调用举例如下:

```c
#include <unistd.h>

int main()
{
    char *const argv[] = {"ps", "-ef", NULL};
    char *const envp[] = {"PATH=/bin:/usr/bin", "TERM=console", NULL};

    execl("/bin/ps", "ps", "-ef", NULL);

    // 带p的，可以使用环境变量PATH，无需写全路径
    execlp("ps", "ps", "-ef", NULL);

    // 带e的，需要自己组装环境变量
    execle("ps", "ps", "-ef", NULL, envp);

    execv("/bin/ps", argv);

    // 带p的，可以使用环境变量PATH，无需写全路径
    execvp("ps", argv);

    // 带e的，需要自己组装环境变量
    execve("/bin/ps", argv, envp);

```

```
24      exit(0);
25  }
```

事实上,只有execve是真正的系统调用,其它五个函数最终都调用execve,所以execve在man手册 第2节,其它函数在man手册第3节。这些函数之间的关系如下图所示。

下图exec函数簇 一个完整的例子:



> 📌 注意：
>
> - 课堂上，我们还有结合时间，演示环境变量如何传入的问题

5. 自主Shell命令行解释器

5-1 目标

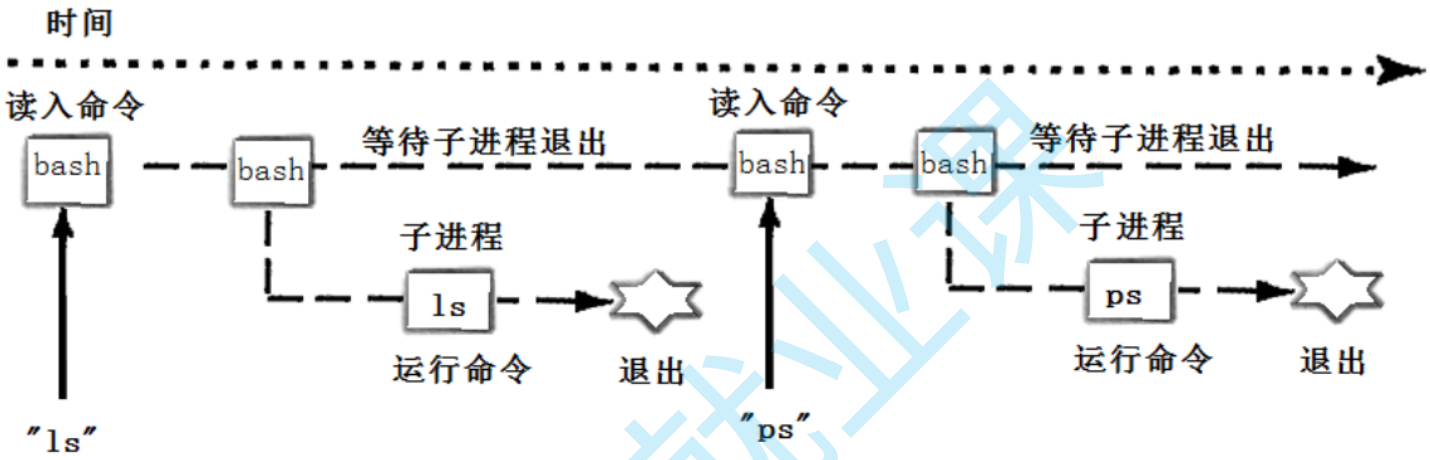
- 要能处理普通命令
- 要能处理内建命令
- 要能帮助我们理解内建命令/本地变量/环境变量这些概念
- 要能帮助我们理解shell的允许原理

5-2 实现原理

考虑下面这个与shell典型的互动：

```
[root@localhost epoll]# ls
client.cpp  readme.md  server.cpp  utility.h
[root@localhost epoll]# ps
  PID TTY          TIME CMD
 3451 pts/0    00:00:00 bash
 3514 pts/0    00:00:00 ps
```

用下图的时间轴来表示事件的发生次序。其中时间从左向右。shell由标识为sh的方块代表，它随着时间的流逝从左向右移动。shell从用户读入字符串"ls"。shell建立一个新的进程，然后在那个进程中运行ls程序并等待那个进程结束。



然后shell读取新的一行输入，建立一个新的进程，在这个进程中运行程序 并等待这个进程结束。

所以要写一个shell，需要循环以下过程:

1. 获取命令行
2. 解析命令行
3. 建立一个子进程（fork）
4. 替换子进程（execvp）
5. 父进程等待子进程退出（wait）

根据这些思路，和我们前面的学的技术，就可以自己来实现一个shell了。

5-3 源码

实现代码:

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <string>
```

```cpp
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <ctype.h>

using namespace std;

const int basesize = 1024;
const int argvnum = 64;
const int envnum = 64;
// 全局的命令行参数表
char *gargv[argvnum];
int gargc = 0;

// 全局的变量
int lastcode = 0;

// 我的系统的环境变量
char *genv[envnum];

// 全局的当前shell工作路径
char pwd[basesize];
char pwdenv[basesize];

// "    "file.txt
#define TrimSpace(pos) do{\
    while(isspace(*pos)){\
        pos++;\
    }\
}while(0)

string GetUserName()
{
    string name = getenv("USER");
    return name.empty() ? "None" : name;
}

string GetHostName()
{
    string hostname = getenv("HOSTNAME");
    return hostname.empty() ? "None" : hostname;
}

string GetPwd()
{
    if(nullptr == getcwd(pwd, sizeof(pwd))) return "None";
    snprintf(pwdenv, sizeof(pwdenv),"PWD=%s", pwd);
```

```
53      putenv(pwdenv); // PWD=XXX
54      return pwd;
55
56      //string pwd = getenv("PWD");
57      //return pwd.empty() ? "None" : pwd;
58  }
59
60  string LastDir()
61  {
62      string curr = GetPwd();
63      if(curr == "/" || curr == "None") return curr;
64      // /home/whb/XXX
65      size_t pos = curr.rfind("/");
66      if(pos == std::string::npos) return curr;
67      return curr.substr(pos+1);
68  }
69
70  string MakeCommandLine()
71  {
72      // [whb@bite-alicloud myshell]$
73      char command_line[basesize];
74      snprintf(command_line, basesize, "[%s@%s %s]# ",\
75              GetUserName().c_str(), GetHostName().c_str(), LastDir().c_str());
76      return command_line;
77  }
78
79  void PrintCommandLine() // 1. 命令行提示符
80  {
81      printf("%s", MakeCommandLine().c_str());
82      fflush(stdout);
83  }
84
85  bool GetCommandLine(char command_buffer[], int size)   // 2. 获取用户命令
86  {
87      // 我们认为：我们要将用户输入的命令行，当做一个完整的字符串
88      // "ls -a -l -n"
89      char *result = fgets(command_buffer, size, stdin);
90      if(!result)
91      {
92          return false;
93      }
94      command_buffer[strlen(command_buffer)-1] = 0;
95      if(strlen(command_buffer) == 0) return false;
96      return true;
97  }
98
99  void ParseCommandLine(char command_buffer[], int len) // 3. 分析命令
```

```
100 {
101     (void)len;
102     memset(gargv, 0, sizeof(gargv));
103     gargc = 0;
104 
105     // "ls -a -l -n"
106     const char *sep = " ";
107     gargv[gargc++] = strtok(command_buffer, sep);
108     // =是刻意写的
109     while((bool)(gargv[gargc++] = strtok(nullptr, sep)));
110     gargc--;
111 }
112 
113 void debug()
114 {
115     printf("argc: %d\n", gargc);
116     for(int i = 0; gargv[i]; i++)
117     {
118         printf("argv[%d]: %s\n", i, gargv[i]);
119     }
120 }
121 // 在shell中
122 // 有些命令，必须由子进程来执行
123 // 有些命令，不能由子进程执行，要由shell自己执行 --- 内建命令 built command
124 bool ExecuteCommand()   // 4. 执行命令
125 {
126     // 让子进程进行执行
127     pid_t id = fork();
128     if(id < 0) return false;
129     if(id == 0)
130     {
131         //子进程
132         // 1. 执行命令
133         execvpe(gargv[0], gargv, genv);
134         // 2. 退出
135         exit(1);
136     }
137     int status = 0;
138     pid_t rid = waitpid(id, &status, 0);
139     if(rid > 0)
140     {
141         if(WIFEXITED(status))
142         {
143             lastcode = WEXITSTATUS(status);
144         }
145         else
146         {
```

```
147             lastcode = 100;
148         }
149         return true;
150     }
151     return false;
152 }
153 
154 void AddEnv(const char *item)
155 {
156     int index = 0;
157     while(genv[index])
158     {
159         index++;
160     }
161 
162     genv[index] = (char*)malloc(strlen(item)+1);
163     strncpy(genv[index], item, strlen(item)+1);
164     genv[++index] = nullptr;
165 }
166 // shell自己执行命令，本质是shell调用自己的函数
167 bool CheckAndExecBuiltCommand()
168 {
169     if(strcmp(gargv[0], "cd") == 0)
170     {
171         // 内建命令
172         if(gargc == 2)
173         {
174             chdir(gargv[1]);
175             lastcode = 0;
176         }
177         else
178         {
179             lastcode = 1;
180         }
181         return true;
182     }
183     else if(strcmp(gargv[0], "export") == 0)
184     {
185         // export也是内建命令
186         if(gargc == 2)
187         {
188             AddEnv(gargv[1]);
189             lastcode = 0;
190         }
191         else
192         {
193             lastcode = 2;
```

```
194         }
195         return true;
196     }
197     else if(strcmp(gargv[0], "env") == 0)
198     {
199         for(int i = 0; genv[i]; i++)
200         {
201             printf("%s\n", genv[i]);
202         }
203         lastcode = 0;
204         return true;
205     }
206     else if(strcmp(gargv[0], "echo") == 0)
207     {
208         if(gargc == 2)
209         {
210             // echo $?
211             // echo $PATH
212             // echo hello
213             if(gargv[1][0] == '$')
214             {
215                 if(gargv[1][1] == '?')
216                 {
217                     printf("%d\n", lastcode);
218                     lastcode = 0;
219                 }
220             }
221             else
222             {
223                printf("%s\n", gargv[1]);
224                 lastcode = 0;
225             }
226         }
227         else
228         {
229             lastcode = 3;
230         }
231         return true;
232     }
233     return false;
234 }
235
236 // 作为一个shell，获取环境变量应该从系统的配置来
237 // 我们今天就直接从父shell中获取环境变量
238 void InitEnv()
239 {
240     extern char **environ;
```

```
241     int index = 0;
242     while(environ[index])
243     {
244         genv[index] = (char*)malloc(strlen(environ[index])+1);
245         strncpy(genv[index], environ[index], strlen(environ[index])+1);
246         index++;
247     }
248     genv[index] = nullptr;
249 }
250
251 int main()
252 {
253     InitEnv();
254     char command_buffer[basesize];
255     while(true)
256     {
257         PrintCommandLine(); // 1. 命令行提示符
258         // command_buffer -> output
259         if( !GetCommandLine(command_buffer, basesize) )   // 2. 获取用户命令
260         {
261             continue;
262         }
263         //printf("%s\n", command_buffer);
264         ParseCommandLine(command_buffer, strlen(command_buffer)); // 3. 分析命令
265
266         if ( CheckAndExecBuiltCommand() )
267         {
268             continue;
269         }
270
271         ExecuteCommand();    // 4. 执行命令
272     }
273     return 0;
274 }
275
```

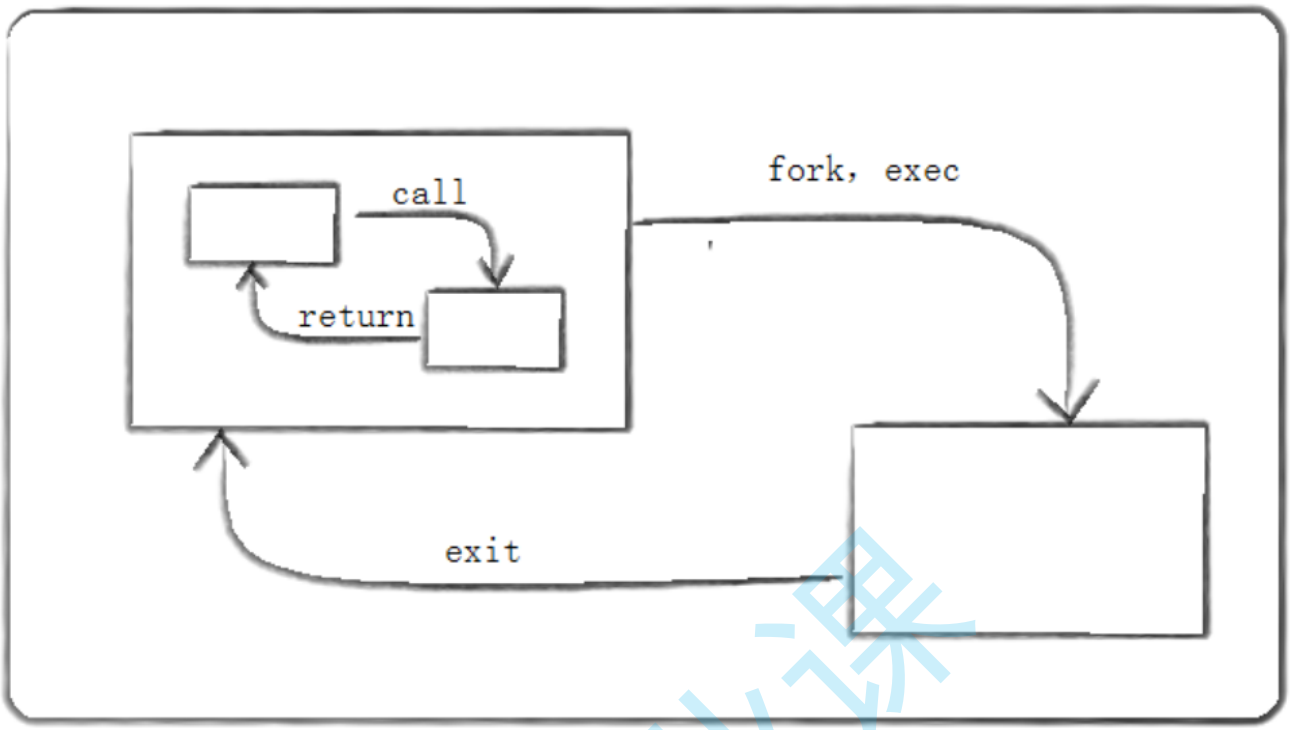

5-4 总结

在继续学习新知识前，我们来思考函数和进程之间的相似性

exec/exit就像call/return

一个C程序有很多函数组成。一个函数可以调用另外一个函数，同时传递给它一些参数。被调用的函数执行一定的操作，然后返回一个值。每个函数都有他的局部变量，不同的函数通过call/return系统进行通信。

这种通过参数和返回值在拥有私有数据的函数间通信的模式是结构化程序设计的基础。Linux鼓励将这种应用于程序之内的模式扩展到程序之间。如下图



一个C程序可以fork/exec另一个程序，并传给它一些参数。这个被调用的程序执行一定的操作，然后通过exit(n)来返回值。调用它的进程可以通过wait（&ret）来获取exit的返回值。