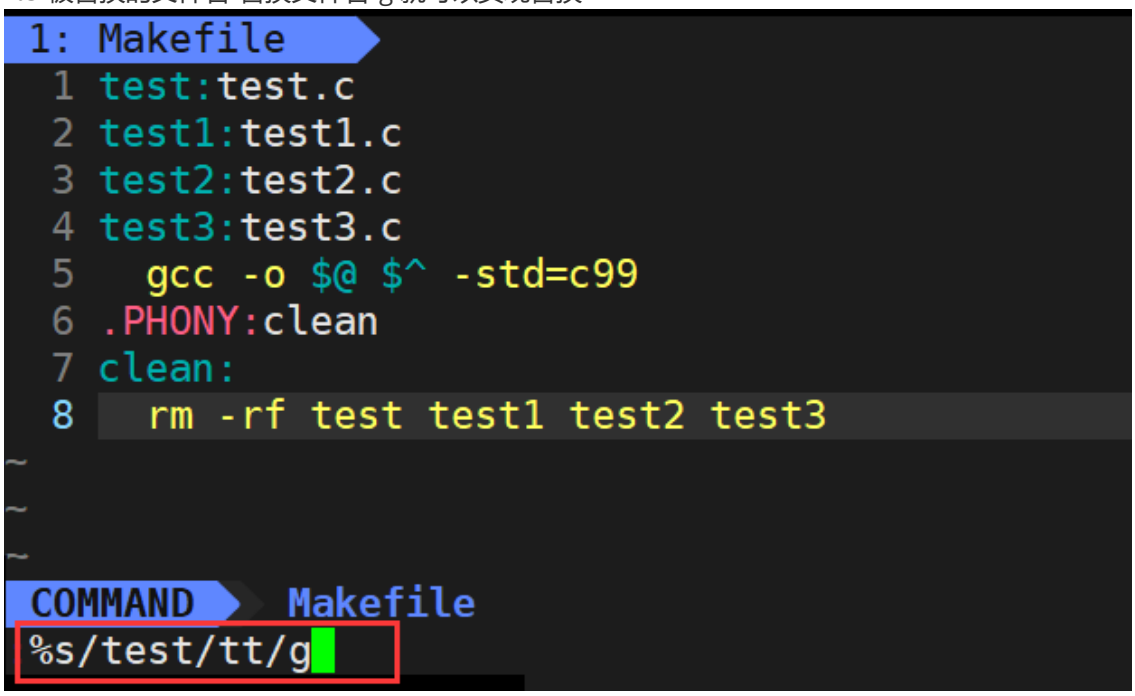
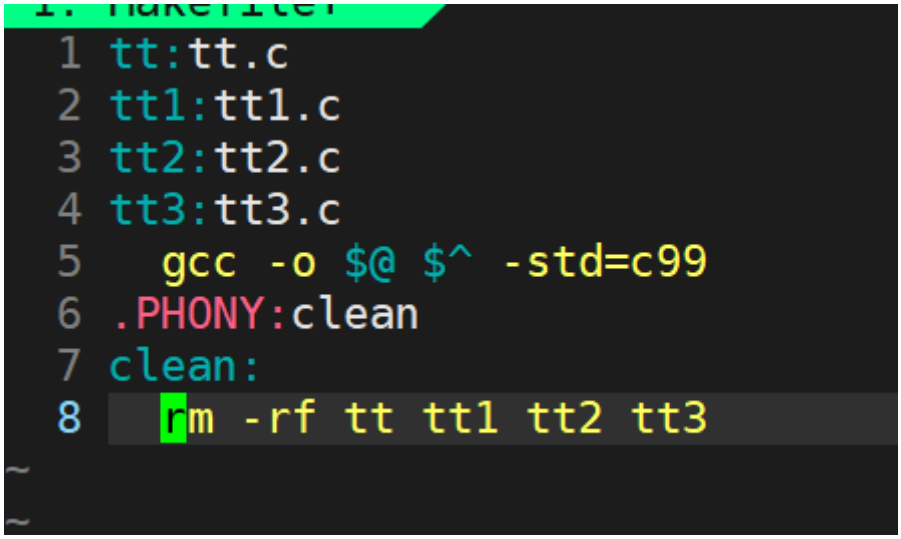
@TOC

补充几个知识:

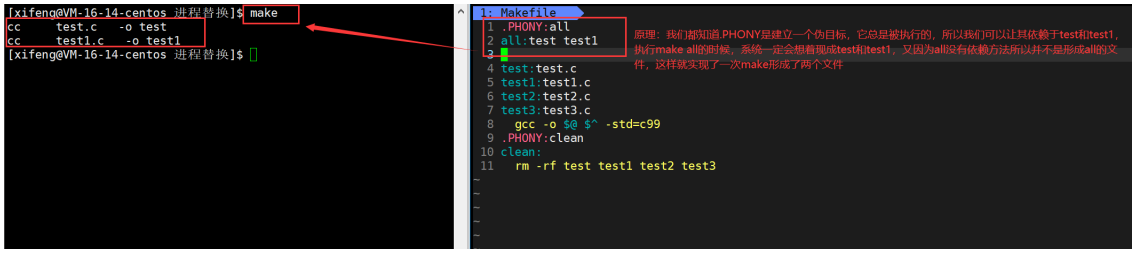
1. %s/被替换的文件名/替换文件名/g 就可以实现替换



替换之后所有的test都被替换成了tt



2. **如何在Makefile中一次形成两个可执行?**

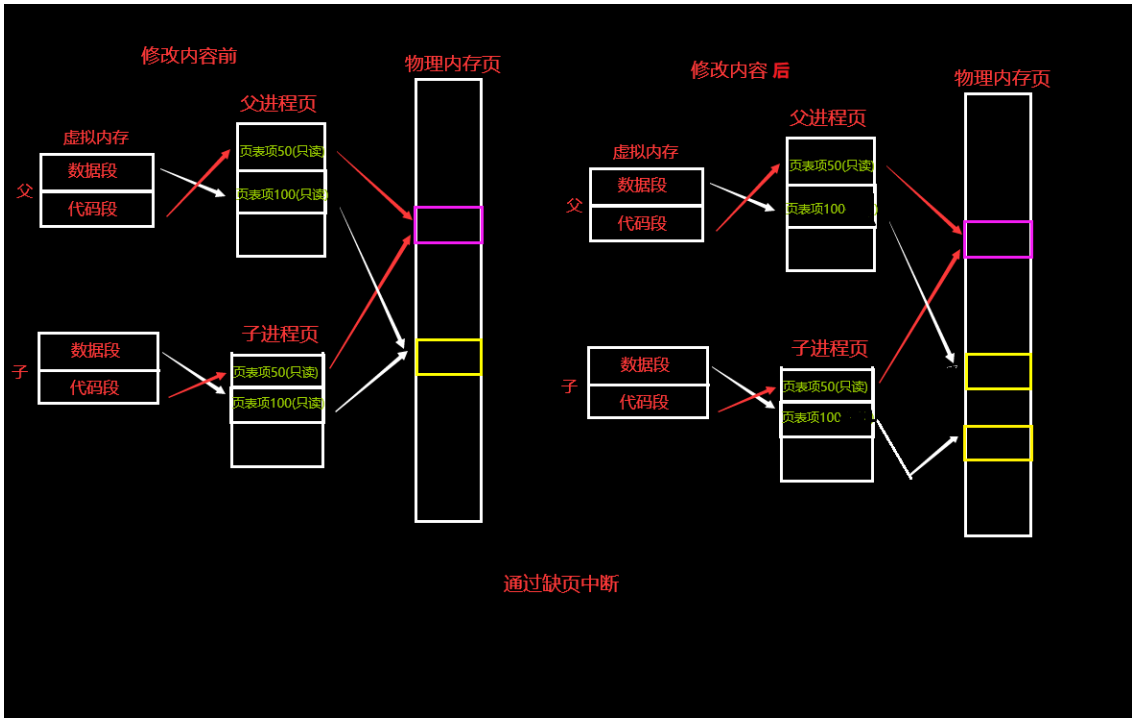


形成多个可执行同理，只需要把想生成的可执行跟在all后面即可

进程控制

一、fork函数初识(见之前)

- 写实拷贝



二、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：父进程来等待客户端的请求，生成子进程来处理请求
- 一个进程要执行不同的程序，例如：子进程从fork返回后，调用exec函数

三、fork调用失败的原因

- 系统中有太多的进程
- 实际用户的进程超过了限制

创建进程是需要很大成本的

进程终止

一、进程退出场景

- 代码运行完毕，结果正确
- 代码运行完毕，结果错误
- 代码异常终止(一般程序崩溃后退出码也就没有意义了)

为什么main总会return 0？意义在哪?

- main函数的return值是进程的退出码
    - 查看退出码：**echo $?**（输出最近一次进程退出时的退出码）
    - 一般用0表示：success
    - !0：failed（错误）之所以用!0表示，是因为相比于程序错误，我们更想知道的是为什么程序会错误，所以我们可以通过结束时的退出码，来查看进程退出的原因，所以可以用具体的数字来代表一种退出原因

- **strerror() :可以查看退出码所对应的内容头文件string.h**

```
[xifeng@VM-16-14-centos lesson11]$ vim Makefile
[xifeng@VM-16-14-centos lesson11]$ make
gcc -o test test.c -std=c99
[xifeng@VM-16-14-centos lesson11]$ ./test
0 :Success
1 :Operation not permitted
2 :No such file or directory
3 :No such process
4 :Interrupted system call
5 :Input/output error
6 :No such device or address
7 :Argument list too long
8 :Exec format error
9 :Bad file descriptor
10 :No child processes
```

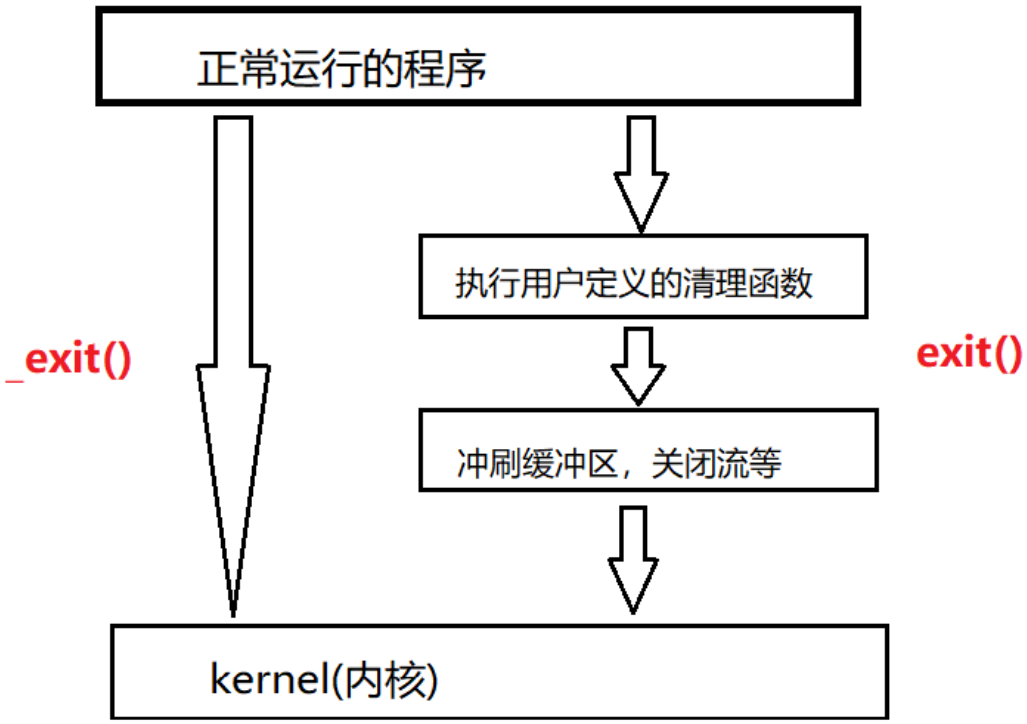
进程退出的方式

1. main函数return，代表进程退出，非main函数return叫做函数返回
2. **exit()**：进程终止，头文件**stdlib**，在**任意地方调用都代表终止进程，参数是退出码**

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    printf("hello word!");
    sleep(4);//数据被暂存到输出缓冲区中
    exit(EXIT_SUCCESS);
    //return 0;
    //都能够看到hello word被打印，说明刷新了缓冲区
    //其原因就是main return or exit 本身就会要求系统，进行缓冲区刷新!
}
```

3. **_exit()**：终止进程，头文件**unistd**，强制终止进程，不要进行进程的后续收尾工作，比如刷新缓冲区(指的是**用户级缓冲区**)

```
[xifeng@VM-16-14-centos lesson11]$ cat test1.c | head -8
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
  printf("hello word");
  _exit(12);
}
[xifeng@VM-16-14-centos lesson11]$ ./test1
[xifeng@VM-16-14-centos lesson11]$ echo $?
12
[xifeng@VM-16-14-centos lesson11]$
```



进程退出，OS层面做了什么?

系统层面，少了一个进程：free PCB，free mm_struct, free页表和各种映射关系

进程等待

- **进程为什么要等待?**

  父进程fork之后，可能有这样一种情况：父进程需要子进程完成某种任务，这样父进程就需要知道子进程完成的情况，所以一般需要父进程通过wait/waitpid等待子进程退出，这种现象就叫做**进程等待**

- **为什么要父进程等待?**

  1. 通过获取子进程退出信息，能够得知子进程执行结果
  2. 可以保证：时序问题，子进程先退出，父进程后退出
  3. 进程退出的时候，会先进入僵尸状态，会造成内存泄露的问题，需要通过父进程wait，释放该子进程占用的资源

进程等待的方法

wait使用方法

-
```
NAME
       wait, waitpid, waitid - wait for process to change state

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

       pid_t wait(int *status);

       pid_t waitpid(pid_t pid, int *status, int options);
```

- 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
- 例子：

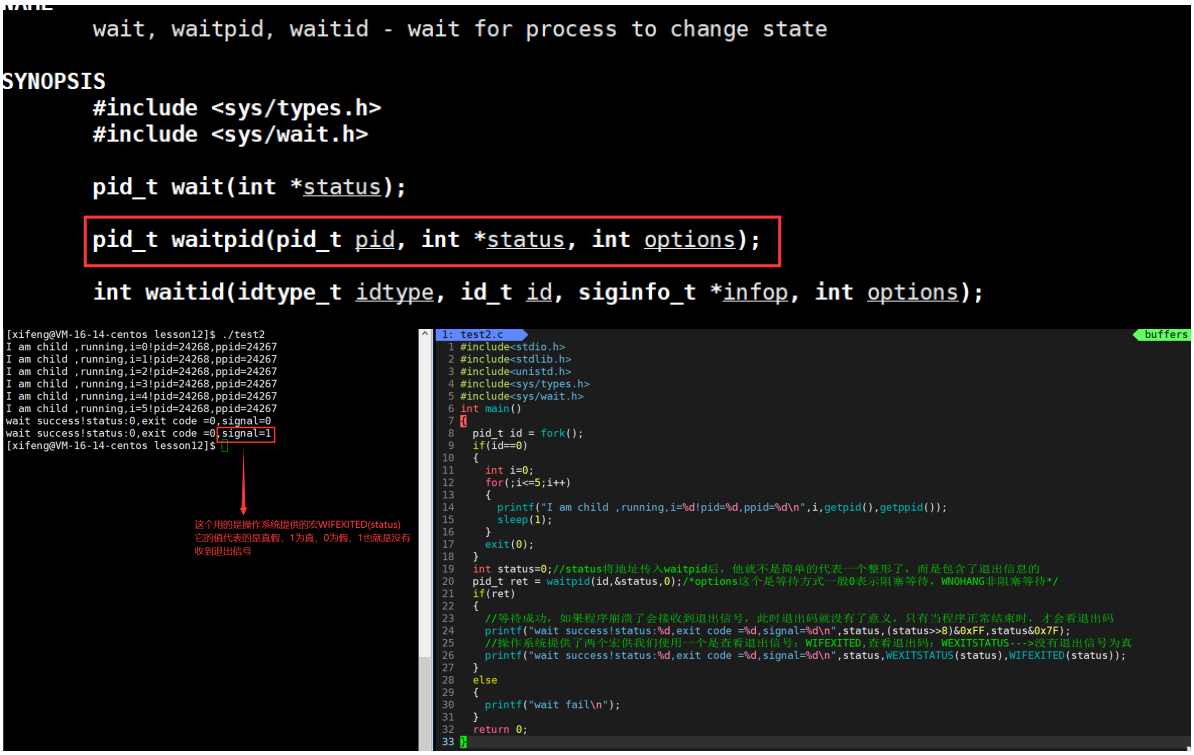
```c
#include<sys/wait.h>
int main()
{
  pid_t id=fork();
  if(id==0)
  {
    //child
    int i=5;
    while(i)
    {
      printf("i:%d,child running,PID:%d\n",i,getpid());
      i--;
      sleep(1);
    }
    //让子进程运行完直接退出
    exit(0);
  }
  //父进程
  sleep(10);                  //先休息10秒然后再开始等待
  printf("father begin\n"); //可以看到僵尸进程被回收
  pid_t ret=wait(NULL);
  if(ret>0)
  printf("ret:%d\n",ret);
  else
    printf("father wait faied!\n");
  sleep(10);
```

```
i:5,child running,PID:15151
i:4,child running,PID:15151
i:3,child running,PID:15151
i:2,child running,PID:15151
i:1,child running,PID:15151
father begin
ret:15151
```

```
15150 15151 15150 18040 pts/0    15150 S+    1001   0:00 ./test
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 S+    1001   0:00 ./test
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 S+    1001   0:00 ./test
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 S+    1001   0:00 ./test
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 S+    1001   0:00 ./test
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 Z+    1001   0:00 [test] <defunct>
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 Z+    1001   0:00 [test] <defunct>
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 Z+    1001   0:00 [test] <defunct>
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 Z+    1001   0:00 [test] <defunct>
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
15150 15151 15150 18040 pts/0    15150 Z+    1001   0:00 [test] <defunct>
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
##################
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
18040 15150 15150 18040 pts/0    15150 S+    1001   0:00 ./test
```

可以观察到一开始有两个进程同时运行，之后有一个进程变成了僵尸进程，又过了一段时间这个僵尸进程也结束了，这个就可以说明首先wait是可以回收僵尸进程的，同时wait的返回值如果正常返回就是等待那个进程的pid，如果异常返回就是返回-1

waitpid使用方法



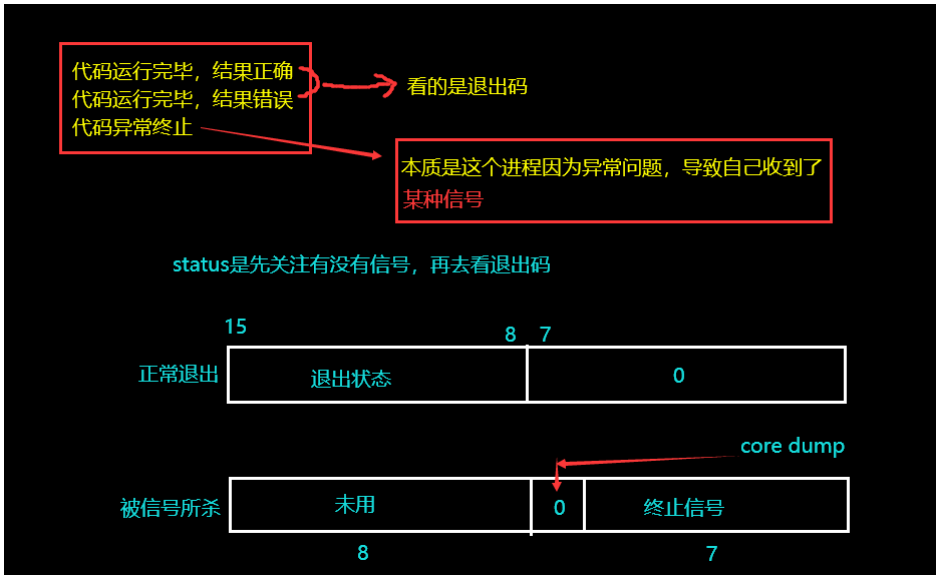
- 头文件与wait相同：**pid_ t waitpid(pid_t pid, int *status, int options);**
- 返回值：
  1. 当正常返回的时候waitpid返回收集到的子进程的进程ID
  2. 如果设置了选项WNOHANG,而调用中waitpid发现没有已退出的子进程可收集,则返回0
  3. 如果调用中出错,则返回-1,这时errno会被设置成相应的值以指示错误所在
- 参数：
  1. pid：
     - Pid=-1,等待任一个子进程。与wait等效。
     - Pid>0.等待其进程ID与pid相等的子进程
  2. status(输出型参数):
     - 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）
     - 例子：(一定要让父进程通过status得到进程执行的结果)

```
 9    if(id==0)
10    {
11      int i=5;
12      while(i)
13      {
14        printf("i:%d,child running,PID:%d\n",i,getpid());
15        i--;
16      }
17      exit(10);
18    }
19    printf("father begin\n");
20    int status=0;
21      pid_t ret=waitpid(id,&status,0);
22    if(ret>0)
23    printf("ret:%d ,status:%d\n",ret,status);
24    else
25      printf("father wait faied!\n");
```

父进程拿到的status的值与子进程 如何退出 强相关！！！

```
:!./test2
i:5,child running,PID:23762
i:4,child running,PID:23762
i:3,child running,PID:23762
i:2,child running,PID:23762
i:1,child running,PID:23762
father begin
ret:23762 ,status:2560
```

这里的status就不是简单的一个整型了，我们知道它有32个比特位，但是判断代码运行结果：**只使用低16个比特位！**高16比特位暂时不考虑(如果没有收到信号代表代码是运行完成的，没有异常终止)；次低8位代表的就是进程退出的退出码，前7位就是进程的终止信号，如果程序没有异常终止那么就为0



- 用**status**来获取进程的退出码或者退出信号

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
  pid_t id=fork();
  if(id==0)
  {
    int i=5;
    while(i)
    {
      printf("i:%d,child running,PID:%d\n",i,getpid());
      i--;
      sleep(1);
    }
    exit(10);
  }
  printf("father begin\n");
  int status=0;
  pid_t ret=waitpid(id,&status,0);
  if(ret>0)
  printf("ret:%d ,status exit code:%d,status exit signal:%d\n",ret,(status>>8)&0xFF,status&0x7F);
  else
    printf("father wait faied!\n");
  return 0;
}
```

注意这个退出码，如果退出信号为0，退出码才会有意义

获取退出码是次低8位，所以直接右移跟1111 1111按位与

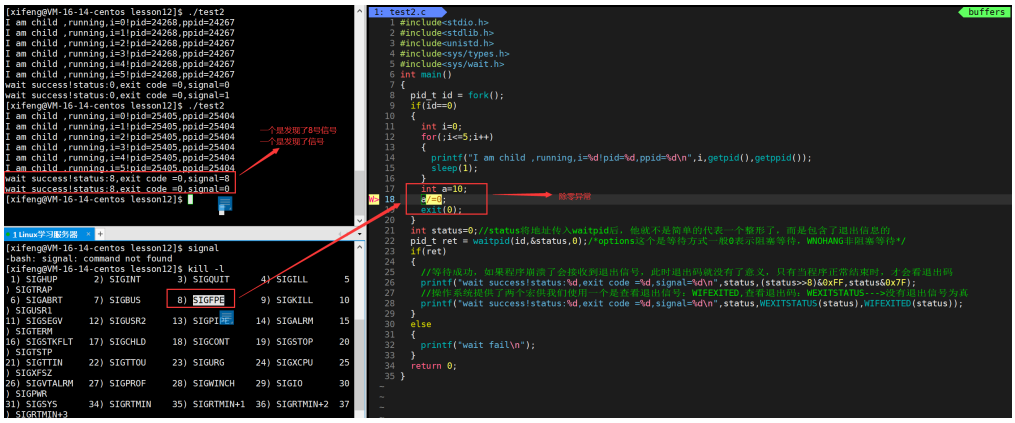
退出信号是低7位所以不需要右移直接跟 0111 1111按位与

```
[xifeng@VM-16-14-centos lesson11]$ ./test2
father begin
i:5,child running,PID:26168
i:4,child running,PID:26168
i:3,child running,PID:26168
i:2,child running,PID:26168
i:1,child running,PID:26168
ret:26168 ,status exit code:10,status exit signal:0
```

结束时的退出码

进程正常结束

下面这个例子是程序被异常终止，也就是接受到了信号



- **bash**是命令行启动的所有进程的父进程，它一定是通过wait的方式得到子进程的退出结果，所以我们能看到echo $?能够查到子进程的退出码
- 操作系统给我们提供了两个宏，可以就是让我们每次查看退出码的时候都进行位操作，一个是**WIFEXITED**如果没有收到退出信号就为真，就可以用**WEXITSTATUS**来获取退出码

```c
  int status=0;
  pid_t ret=waitpid(id,&status,0);
  if(ret>0)
  {
    if(WIFEXITED(status))
      printf("exit code:%d",WEXITSTATUS(status));
//  printf("ret:%d ,status exit code:%d,status exit signal:%d\n",ret,(status>>8)&0xFF,status&0x7F);
  else
    printf("father wait faied!\n");
```

3. options:

- options的值为0：默认行为，阻塞等待；设置为WNOHANG：设置等待方式为非阻塞
- 阻塞的本质：其实就是进程的PCB被放入了等待队列，并将进程的状态改为S状态
- 返回的本质：进程的PCB从等待队列拿到R队列，从而被CPU调度
- 非阻塞：调度一个接口cpu可以正常返回，cpu不断重复调度父进程就是不断重复waitpid的过程
- 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PID。

```
[xifeng@VM-16-14-centos lesson12]$ ./test3
father do thing!
I am child ,running,i=0!pid=32437,ppid=32436
father do thing!
I am child ,running,i=1!pid=32437,ppid=32436
father do thing!
I am child ,running,i=2!pid=32437,ppid=32436
father do thing!
I am child ,running,i=3!pid=32437,ppid=32436
father do thing!
I am child ,running,i=4!pid=32437,ppid=32436
father do thing!
I am child ,running,i=5!pid=32437,ppid=32436
father do thing!
wait success!status:0,exit code =0,signal=0
wait success!status:0,exit code =0,signal=1
wait success!status:0,exit code =0,signal=0
wait success!status:0,exit code =0,signal=1
[xifeng@VM-16-14-centos lesson12]$
```

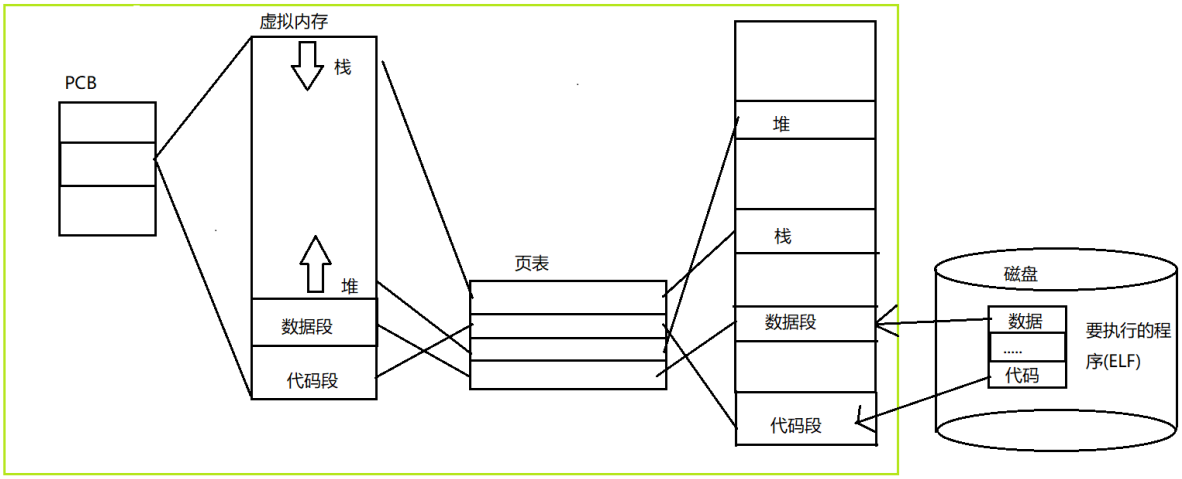
test3.c

```c
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
  pid_t id = fork();
  if(id==0)
  {
    int i=0;
    for(;i<=5;i++)
    {
      printf("I am child ,running,i=%d!pid=%d,ppid=%d\n",i,getpid(),getppid());
      sleep(1);
    }
  }
  int status=0;
  while(1)
  {
    pid_t ret = waitpid(id,&status,WNOHANG);/*options这个是等待方式一般0表示阻塞等待，WNOHANG非阻塞等待*/
    if(ret==0)
    {
      //子进程还没有结束
      //父进程可以做自己的事情
      printf("father do thing!\n");
      sleep(1);
    }
    else if(ret)
    {
      //等待成功打印出对应的信息
      printf("wait success!status:%d,exit code =%d,signal=%d\n",status,(status>>8)&0xFF,status&0x7F);
      printf("wait success!status:%d,exit code =%d,signal=%d\n",status,WEXITSTATUS(status),WIFEXITED(status));
      break;
    }
    else
    {
      //等待失败可以打印出失败的原因，失败就没有必要再等待了
      perror("wait fail\n");
      break;
    }
  } //下面就是return 0;了
```

```c
  int status=0;
//  pid_t ret=waitpid(id,&status,0);
  while(1)
  {
    pid_t ret=waitpid(id,&status,WNOHANG);
    if(ret==0)
    {
       //子进程没有退出，但是waitpid等待成功，需要父进程重复等待
       //非阻断等待可以让父进程去做一些别的事情
       printf("father no thing\n");
    }
    else if(ret>0)
    {
      //子进程退出，waitpid得到了相应的返回值，同时就没有必要继续循环了
       printf("ret:%d ,status exit code:%d,status exit signal:%d\n",ret,(status>>8)&0xFF,status&0x7F);
       break;
    }
    else
    {
      //等待失败，也没有必要再继续等待
      perror("waitipd");
      break;
    }
    sleep(1);
  }
```

非阻断需要重复的去调用

进程程序替换



进程替换从上图来看，就是延用了之前进程的壳子，只是将新的程序的代码和数据给替换了进去

进程不变，仅仅替换当前进程的代码和数据的技术叫做进程的程序替换

- **为什么要进行程序替换?**

  想让子进程执行一个全新的程序

- **什么是程序替换?（原理？）**

  用新进程的代码和数据替换掉原来的代码和数据，其他都不变。

程序替换的本质：就是把**程序的进程代码和数据加载到特定的进程上下文中**(加载就需要加载器，它的底层原理可以理解成**exec*程序替换函数**(*可以理解成系列),进程程序替换会更改代码区的代码，这也会发生**写时拷贝**

- **进程的程序替换使用----怎么办？ ----阶段一**

  1. 现象：就是替换之后后续的代码不会执行，如果是子进程发生了程序替换，那么只会改变子程序的代码和数据不会影响父进程

```
[xifeng@VM-16-14-centos lesson11]$ cat test.c | head -20
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
  printf("I am a process!  pid:%d\n",getpid());
  //只要进程的程序替换成功，就不会执行后续的代码，意味着exec*系列的函数，成功的时候，不需要返回值
  //只要exec*返回了，那么一定是调用失败了！！
  execl("/usr/bin/ls","ls","-a","-l",NULL);
  exit(1);
  printf("hello\n");
  printf("hello\n");
  printf("hello\n");
  printf("hello\n");
  printf("hello\n");
  printf("hello\n");
  return 0;
}
[xifeng@VM-16-14-centos lesson11]$ ./test
I am a process!  pid:9426
total 48
drwxrwxr-x 2 xifeng xifeng 4096 Mar 25 16:53 .
drwxrwxr-x 6 xifeng xifeng 4096 Mar 21 13:44 ..
-rw-rw-r-- 1 xifeng xifeng   82 Mar 25 16:51 Makefile
-rwxrwxr-x 1 xifeng xifeng 8512 Mar 25 16:53 test
-rwxrwxr-x 1 xifeng xifeng 8720 Mar 25 11:58 test2
-rw-rw-r-- 1 xifeng xifeng 1504 Mar 25 11:58 test2.c
-rw-rw-r-- 1 xifeng xifeng 1453 Mar 25 16:53 test.c
-rw-r--r-- 1 xifeng xifeng 3155 Mar 22 21:36 程序地址空间.md
[xifeng@VM-16-14-centos lesson11]$
```

并没有被执行，调用失败可以用echo $?来查看退出码

  2. exec* 如果有返回值那么就是调用失败

- **阶段二**

  各个程序替换函数的基本使用

替换函数

```
EXEC(3)                    Linux Programmer's Manual                    EXEC(3)

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #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 execvpe(const char *file, char *const argv[],
                   char *const envp[]);
```

```
NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

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

命名理解

- l(list)：表示参数采用列表形式

```
execl("/usr/bin/ls"/*要执行谁*/,"ls","-l","-a"/*怎么执行*/,NULL);//传递结束一定要以NULL结尾
```

- v(vector):参数用数组

```
char* argv[]={"ls","-a","-l","-n",NULL};
execv("/usr/bin/ls",argv);
```

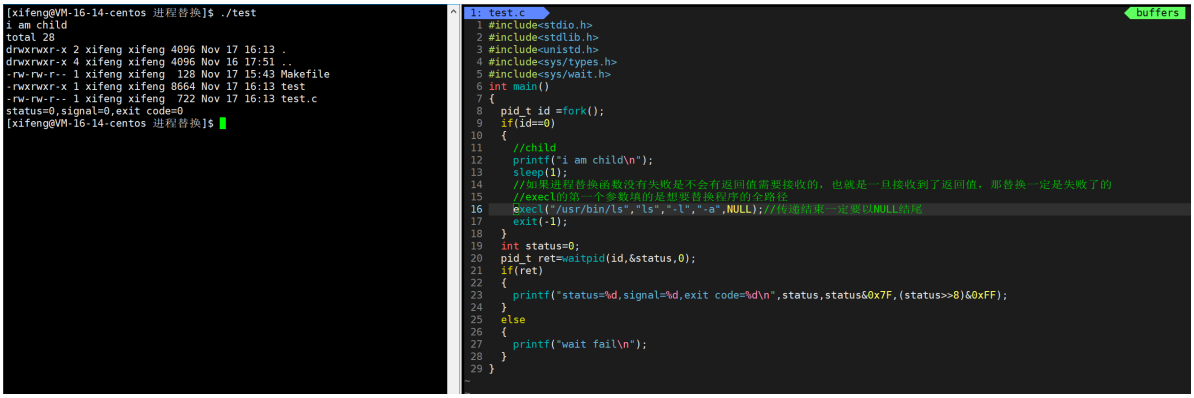
- p(path):有p自动搜索环境变量PATH

```
execlp("ls"/*要执行谁*/,"ls"/*怎么执行，两个ls意义不同*/,"-a","-l",NULL);
```

- e(env)：表示自己维护环境变量(通俗点就是可以自定义环境变量)

execl函数

```
int execl(count char*path,count char *arg,...);
//path 就是要执行的文件全路径一定要是路径+文件名
// ... 意思是可变参数列表，要执行的目标程序在命令行上怎么执行这里就怎么一个一个传递进去()
//必须以NULL作为传入的参数的结束
```



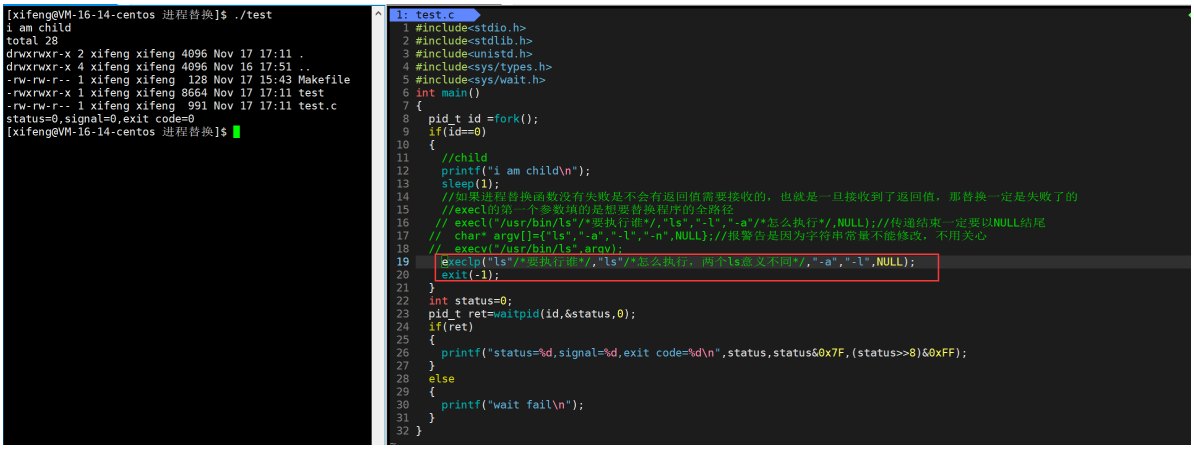
execv函数

```
int execv(const char* path,char *const argv[]);
//第一个参数还是传全路径
//第二个参数传的是一个数组，在命令行上怎么执行就将其存在数组中，然后传进去，要以NULL结尾
```

与execl除了传参的不同，其他没有区别

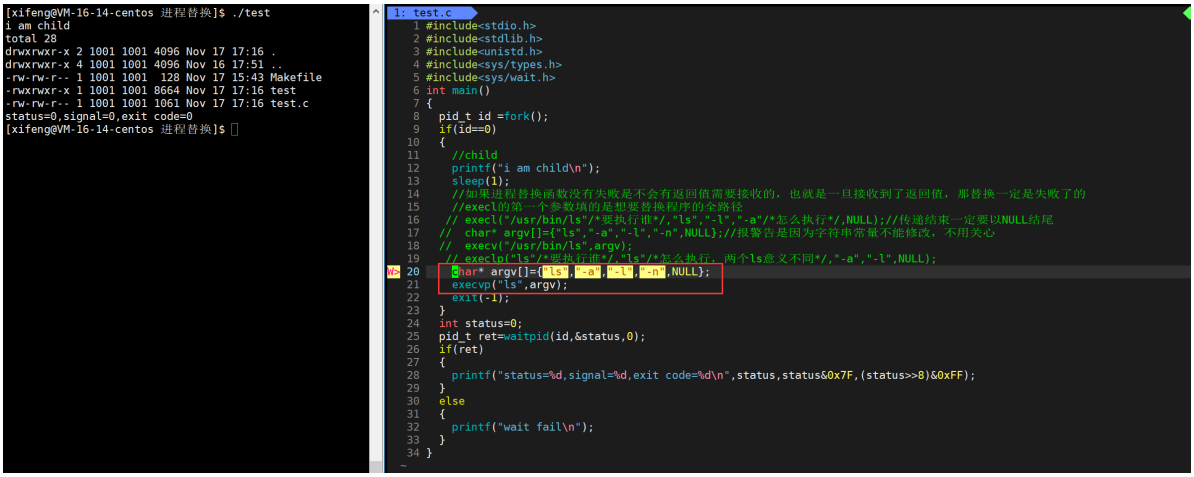
execlp函数

```
int execlp(const char* file,const char* arg,...);
//第一个参数是文件名，需要在环境变量当中才行，会自动去找寻地址
//后面的参数跟execl相同
```



execvp函数

```
int execvp(char* file,char *const argv[]);
//不用带路径会自动去环境变量中找，且通过数组传参
```



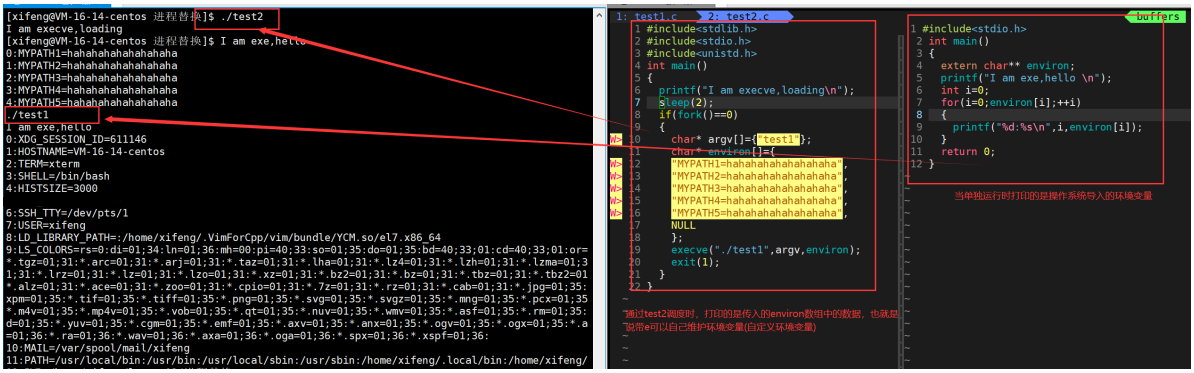
execle函数

```
int execle(const char* path,const char *arg,...,char *const envp[]);
```

execve函数---->只有这一个是系统调用，其他都是库函数

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

用法同execle，唯一的区别就是它传的是数组



execvpe函数

```
int execvpe(const char *file, char *const argv[],char *const envp[]);
```

跟上面的没有什么区别就是多了个p，可以参照之前的来看

为什么会有这么多接口?

是为了满足不同的应用场景