



Linux操作系统

授课教师：刘二小 副教授

单位：杭州电子科技大学

通信工程学院

第八讲 静态库、动态库及进程

库是在链接阶段和相应的.o目标文件形成的可执行文件，从本质上来说是一种可执行代码的二进制格式，可以被载入内存中执行。库分静态库和动态库两种。

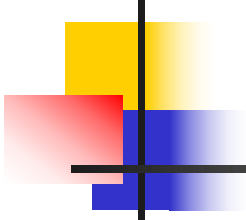
- 静态函数库(.a)
- 动态函数库(.so)

- 静态库(*.a@Linux,*.lib@Wins)
- 动态库(*.so@Linux,*.dll@Wins)

库文件

↓
链接





(一)函数库

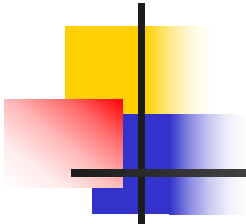
1. 静态函数库

- 这类库的名字一般是libxxx.a；利用静态函数库编译成的文件占用比较大空间，因为整个函数库的所有数据都会被整合进目标代码中
- 特点：
 - 编译后的执行程序不需要外部的函数库支持，因为所有使用的函数都已经被编译进去了。
 - 缺点:如果静态函数库改变了，程序必须重新编译。



2. 动态函数库

- 这类库的名字一般是libxxx.so，相对于静态函数库，动态函数库在编译的时候并没有被编译进目标代码中，程序执行到相关函数时才调用该函数库里的相应函数；
- 特点：动态函数库所产生的可执行文件比较小。由于函数库没有被整合进程序，而是程序运行时动态的申请并调用，所以程序的运行环境中必须提供相应的库。**动态函数库的改变并不影响程序**，所以动态函数库的升级/更新比较方便。



- 当使用静态库时，连接器会找出程序所需的函数，然后将它们复制到执行文件，由于这种复制是完整的，所以一旦链接成功，静态库在**不存在的情况下**可执行文件能够正常执行。

- 动态库截然不同，动态库会在执行程序内留下一个标记指明当程序执行时必须载入的库文件，所以当执行文件执行时才动态加载库文件，因此使用动态库必然会节省空间。

- Linux下进行连接的缺省操作是首先链接动态库，即，如果同时存在相同库名的静态库和动态库，不特别约定的话，默认将与动态库相链接。



静态库

•静态库的创建

- (1) 编写源代码
- (2) 源代码分别编译成.o文件
- (3) 利用ar命令把所有的.o文件打包，生成一个静态库
- (4) 使用静态库（只需要把头文件和库文件拷贝给需要的程序即可）



静态库

•静态库的创建

ar命令选项及说明

- r 将目标文件加入到静态库中
- t 显示静态库中的文件
- a 将目标文件追加到静态库文件现有文件之后
- b 将目标文件追加到静态库文件现有文件之前
- d 从指定的静态库中删除目标文件
- x 从指定的静态库中提取目标文件
- p 把静态库文件中指定的文件输出到标准输出
- q 快速地把文件追加到静态库中



静态库

例：

(1) 编写sort.c 和bank.h文件

bank.h如下

```
struct student
{
    int id;
    char name[10];
    float score;
};
```

sort.c如下

```
#include "bank.h" //sort.c和bank.h位于同一目录中
void sortaz(struct student stu[],int n)
{
    int i,j;
    struct student t;
    for (i=0;i<n-1;i++)
        for (j=0;j<n-1;j++)
            if(stu[j].score>stu[j+1].score)
            { t=stu[j];
              stu[j]=stu[j+1];
              stu[j+1]=t;
            }
}
```


静态库

例：

(2) 将sort.c文件生成sort.o文件

- gcc -c sort.c

(3) 创建静态库并将目标文件加入到库中

- ar -r 目标库文件名称 目标文件列表 (多个目标文件的的话用空格分隔)

- ar -r lib**math**.a sort.o

```
dong.out  jing.out  math.c  m.o  sort.o
lex@lex-virtual-machine:~/demo/lesson8$ ar -r libmath.a sort.o
ar: creating libmath.a
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h      jing1.out  libA.a      main.c      m.o         sort.o
dong.out   jing.out  libmath.a   m.c         sort.c      test1.c
```

静态库使用的两种方法：

(1) 参数法

格式：gcc 主程序 -l 静态库名(去掉lib和.a) -L 静态库存放位置

(2) 直接法

格式：gcc 主程序 静态库全名

例：输入5个学生的基本信息，按成绩从低到高进行排序(main.c)

```
#include <stdio.h>
#include "bank.h"
main()
{
    struct student stu1[3];
    int i;
    for (i=0;i<3;i++)
    {
        printf("input %dth student id,i+1);
        scanf("%d",&stu1[i].id);
        printf("input %dth student name,i+1);
        scanf("%s",&stu1[i].name);
        printf("input %dth student score,i+1);
        scanf("%f",&stu1[i].score);
    }
    sortaz(stu1,3);
    for(i=0;i<3;i++)
        printf("name=%s,score=%f\n",stu1[i].name,stu1[i].score);
}
```

链接libmath.a静态库，编译main文件

(1) 参数法： `gcc main.c -o main -l math -L .`

- 静态库全名为libmath.a,使用前将前缀lib和后缀.a去掉后即为math,静态库存放在当前目录下，所以用.表示。
- -L：加载库文件路径；
- -l:指明库文件名称；其中-l math可以连写在一起“-lmath”；
- 执行./main后即可输出结果

(2) 直接法： `gcc main.c libmath.a`或者 `gcc main.c -o main libmath.a`

运行结果:

```
|  
main.c: In function 'main':  
main.c:12:17: warning: format '%s' expects argument of type 'char *', but  
argument 2 has type 'char (*)[10]' [-Wformat=]  
12 |         scanf("%s",&stu1[i].name);  
    |         ~^ ~~~~~  
    |         | |  
    |         | char (*)[10]  
    |         char *  
main.c:16:5: warning: implicit declaration of function 'sortaz' [-Wimplicit-  
function-declaration]  
16 |         sortaz(stu1,5);  
    |         ~~~~~  
lex@lex-virtual-machine:~/demo/lesson8$ ls  
bank.h libmath.a main main.c sort.c sort.o  
lex@lex-virtual-machine:~/demo/lesson8$ ./main  
input 1th student id:20  
input 1th student name:lex1  
input 1th student score:82  
input 2th student id:23  
input 2th student name:lex2  
input 2th student score:93  
input 3th student id:34  
input 3th student name:lex3  
input 3th student score:95  
input 4th student id:0  
input 4th student name:lex4  
input 4th student score:86.5  
input 5th student id:17  
input 5th student name:lex5  
input 5th student score:72  
name=lex5,score=72.000000  
name=lex1,score=82.000000  
name=lex4,score=86.500000  
name=lex2,score=93.000000  
name=lex3,score=95.000000
```

(1)编译文件,生成动态库libmath.so

```
gcc -shared -fPIC sort.c -o libmath.so
```

-fPIC: (Position-Independent Code) 表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

动态库又称共享库，编译时链接动态库，但不加载目标代码，只有在运行时才加载相关的目标代码（所调用的库函数）到内存，进程结束时自动释放其所占内存空间

(2)使用libmath.so

gcc main.c -l math -L .

```
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h  libmath.so  main  main.c  sort.c  sort.o
lex@lex-virtual-machine:~/demo/lesson8$ ./main
./main: error while loading shared libraries: libmath.so: cannot open shared object file: No such file or directory
lex@lex-virtual-machine:~/demo/lesson8$
```

当执行./main时，出现上述错误，则因为Linux动态加载器找不到libmath.so文件。因为动态库是在程序运行阶段进行链接的，一般情况加载器会自动在/lib目录下搜寻动态库进行链接，所以需要将libmath.so移动到/lib目录下即可。

```
./main: error while loading shared libraries: libmath.so: cannot o
o such file or directory
lex@lex-virtual-machine:~/demo/lesson8$ mv libmath.so /lib
mv: cannot move 'libmath.so' to '/lib/libmath.so': Permission deni
lex@lex-virtual-machine:~/demo/lesson8$ ./main
./main: error while loading shared libraries: libmath.so: cannot o
o such file or directory
lex@lex-virtual-machine:~/demo/lesson8$ sudo mv libmath.so /lib
lex@lex-virtual-machine:~/demo/lesson8$ ./main
input 1th student id:230
input 1th student name:23
input 1th student score:34
input 2th student id:34
input 2th student name:243
input 2th student score:24
input 3th student id:45
input 3th student name:45
input 3th student score:325
input 4th student id:656
input 4th student name:56
input 4th student score:46
input 5th student id:22345
input 5th student name:6345
input 5th student score:656
name=243,score=24.000000
name=23,score=34.000000
name=56,score=46.000000
name=45,score=325.000000
name=6345,score=656.000000
```

静态库与动态库的区别、测试与验证

- 静态库在程序编译时会被链接到目标代码中，程序运行时将不再需要该静态库。编译之后程序文件比较大，但隔离性好！
- 动态库在程序编译时并不会被链接到目标代码中，而是在程序运行时才被载入，因此在程序运行时还需要动态库存在。编译后的程序文件相对较小，多个应用程序可以使用同一个动态库，启动多个应用程序时，只需要将动态库加载到内存一次即可。

	区别	创建方法	使用方法	例子
静态库	(1)编译时链接到目标代码； (2)程序运行时不需要该静态库； (3)编译之后程序文件比较大，隔离性好；	ar -r 目标库文件名称 (用到该库的)目标文件列表 例： ar -r libmath.a sort.o	参数法： gcc 主程序 -l 静态库名 (去掉lib和.a) -L 静态库存放位置 直接法： gcc 主程序 静态库全名	参数法： gcc main.c -o main -l math -L . 直接法： gcc main.c -o main libmath.a
动态库	(1)编译时不会链接到目标代码； (2)程序运行时运行时才载入动态库； (3)编译之后程序文件相对较小； (4)多个程序使用同一动态库时，只需加载到内存一次；	gcc -shared -fPIC *.c -o lib***.so 例： (1)先生成.o,后生成.so: ➤ gcc -c -fPIC sort.c ➤ gcc -shared sort.o -o libmath.so (2)或者使用合并命令： gcc -shared -fPIC sort.c -o libmath.so	参数法： gcc 主程序 -l 动态库名 (去掉lib和.so) -L 动态库存放位置 (/lib, 当前目录无效，即时当前目录存在.so，并且这里指定当前目录后也是无效的) 直接法： gcc 主程序 动态库全名 (可加绝对路径)	参数法： gcc main.c -o main -l math -L /lib 直接法： gcc main.c -o main ./libmath.so(加绝对路径即可)

静态库与动态库的区别、测试与验证

➤ 测试验证1：参数法默认加载库类型测试

```
390 history 4
lex@lex-virtual-machine:~/demo/lesson8$ history 10
382 sudo mv libmath.so /lib
383 ls
384 rm main
385 ls
386 gcc main.c -lmath -L /lib -o main1
387 ls
388 gcc main.c -static -lmath -L /lib -o main2
```

```
lex@lex-virtual-machine:~/demo/lesson8$ ls -il
total 1016
1055592 -rw-rw-r-- 1 lex lex    68 4月 10 09:54 bank.h
1055575 -rwxrwxr-x 1 lex lex  16832 4月 10 13:28 main1
1055580 -rwxrwxr-x 1 lex lex 1002312 4月 10 13:28 main2
```

当静态库和动态库处于统一目录/lib中，利用参数法加载和使用库文件时，默认链接的是动态库文件，如果需要指定静态库，则需加选项-static关键字；

静态库与动态库的区别、测试与验证

➤ 测试验证2：静态库更新后是否需要重新生成可执行文件？

```
//#test1.c
```

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

```
main()
```

```
{
```

```
int a,b,c;
```

```
a=5;b=15;
```

```
c=max(a,b);
```

```
printf("the bigger one is %d\n",c);
```

```
}
```

```
//#m.c
```

```
int max(int a,int b)
```

```
{return a>b?a:b;}
```

(1)生成静态库 libA.a

ar -r libA.a m.o

(2)链接静态库生成可执行文件jing.out并执行；

```
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h  jing.out  libA.a  main.c  m.c  m.o  sort.c  sort.o  test1.c
lex@lex-virtual-machine:~/demo/lesson8$ ./jing.out
the bigger one is 15
lex@lex-virtual-machine:~/demo/lesson8$
```

(3)改变静态库文件内容如下，然后重新生成库文件libA.a，但是不重新生成可执行文件！

```
//#m.c
```

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

```
int max(int a,int b)
```

```
{
```

```
printf("test staticlib!\n");
```

```
return a>b?a:b;
```

```
}
```

```
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h  jing.out  libA.a  main.c  m.c  m.o  sort.c  sort.o  test1.c
lex@lex-virtual-machine:~/demo/lesson8$ rm libA.a
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h  jing.out  main.c  m.c  m.o  sort.c  sort.o  test1.c
lex@lex-virtual-machine:~/demo/lesson8$ gcc m.c -c
lex@lex-virtual-machine:~/demo/lesson8$ ar -r libA.a m.o
ar: creating libA.a
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h  jing.out  libA.a  main.c  m.c  m.o  sort.c  sort.o  test1.c
lex@lex-virtual-machine:~/demo/lesson8$ ./jing.out
the bigger one is 15
lex@lex-virtual-machine:~/demo/lesson8$ gcc test1.c -LA -L. -o jing1.out
test1.c:3:1: warning: return type defaults to 'int' [-Wimplicit-int]
   3 | main()
     | ^~~~~
test1.c: In function 'main':
test1.c:7:3: warning: implicit declaration of function 'max' [-Wimplicit-function-declaration]
   7 | c=max(a,b);
     | ^~~~~
lex@lex-virtual-machine:~/demo/lesson8$ ./jing1.out
test staticlib
the bigger one is 15
```

结论：如果不重新生成可执行文件，结果并没有更新；重新进行编译链接后生成新的jing1.out可执行文件，结果是更新的！

静态库与动态库的区别、测试与验证

- 测试验证2：动态库更新后是否需要重新生成可执行文件？

```
//#m.c
#include <stdio.h>
int max(int a,int b)
{
    //printf("test staticlib!\n");
    return a>b?a:b;
}
```

```
//#m.c
#include <stdio.h>
int max(int a,int b)
{
    printf("test staticlib!\n");
    return a>b?a:b;
}
```

```
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h  jing1.out  jing.out  libA.a  main.c  m.c  sort.c  sort.o  test1.c
lex@lex-virtual-machine:~/demo/lesson8$ gcc -c -fPIC m.c
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h  jing1.out  jing.out  libA.a  main.c  m.c  m.o  sort.c  sort.o  test1.c
lex@lex-virtual-machine:~/demo/lesson8$ gcc -shared m.o -o libA.so
lex@lex-virtual-machine:~/demo/lesson8$ ls
bank.h  jing1.out  libA.a  libA.so  main.c  m.c  m.o  sort.c  sort.o  test1.c
lex@lex-virtual-machine:~/demo/lesson8$ sudo mv libA.so /lib
[sudo] password for lex:
lex@lex-virtual-machine:~/demo/lesson8$ gcc test1.c -LA -L /lib -o dong.out
test1.c:3:1: warning: return type defaults to 'int' [-Wimplicit-int]
    3 | main()
      | ^~~~~~
test1.c: In function 'main':
test1.c:7:3: warning: implicit declaration of function 'max' [-Wimplicit-function-declaration]
    7 | c=max(a,b);
      | ^~~~~
lex@lex-virtual-machine:~/demo/lesson8$ ./dong.out
the bigger one is 15
```

```
lex@lex-virtual-machine:~/demo/lesson8$ gcc -c -fPIC m.c
lex@lex-virtual-machine:~/demo/lesson8$ gcc -shared m.o -o libA.so
lex@lex-virtual-machine:~/demo/lesson8$ sudo mv libA.so /lib
lex@lex-virtual-machine:~/demo/lesson8$ ./dong.out
test staticlib
the bigger one is 15
```

结论：对于动态库而言，库文件更新后不要重新链接编译生成可执行文件；

(二)进程及进程通信

(1)进程与PCB

- 在编写完代码并运行起来时，在磁盘中会形成一个可执行文件，双击这个可执行文件时（程序时），这个程序会加载到内存中，而这个时候不能把它叫做程序了，应该叫做进程。只要把程序（运行起来）加载到内存中，就称之为进程。
- 当一个程序加载到内存中，操作系统要为刚刚加载到内存的程序创建一个结构体（PCB，进程控制块），进程信息被放在这个结构体中（PCB），可以理解为PCB是进程的属性的集合。
- 在Linux操作系统下的PCB是：task_struct
- task_struct是Linux内核的一种数据结构，它会被装载到RAM(内存)里并且包含着进程的信息。



(二)进程及进程通信

(2)进程状态 ps -lA|more

•F 进程标志, 4为超级用户

•S 进程状态

•PID 进程号

•PPID 父进程号

•C CPU使用资源百分比

•PRI priority优先级

•NI Nice值

•ADDR 核心功能, 该进程在内存的哪一部分, 如果运行的进程, 则为'-';

•SZ 用掉的内存大小

•WCHAN 当前进程是否运行, '-'表示正在执行

•TTY 登录者的终端位置

•TIME 用掉的CPU的时间

•CMD 执行的指令

```
lex@lex-virtual-machine:~/Desktop$ ps -lA|more
F S  UID      PID      PPID  C  PRI  NI  ADDR  SZ  WCHAN  TTY      TIME  CMD
4 S   0         1         0  0   80   0  - 42598  -    ?       ?      00:00:09 systemd
1 S   0         2         0  0   80   0  -    0  -    ?       ?      00:00:00 kthreadd
1 I   0         3         2  0   60  -20  -    0  -    ?       ?      00:00:00 rcu_gp
1 I   0         4         2  0   60  -20  -    0  -    ?       ?      00:00:00 rcu_par_gp
1 I   0         5         2  0   60  -20  -    0  -    ?       ?      00:00:00 slub_flushwq
1 I   0         6         2  0   60  -20  -    0  -    ?       ?      00:00:00 netns
1 I   0         8         2  0   60  -20  -    0  -    ?       ?      00:00:00 kworker/0:0H-events_highpri
1 I   0        10         2  0   60  -20  -    0  -    ?       ?      00:00:00 mm_percpu_wq
1 S   0        11         2  0   80   0  -    0  -    ?       ?      00:00:00 rcu_tasks_rude_
1 S   0        12         2  0   80   0  -    0  -    ?       ?      00:00:00 rcu_tasks_trace
1 S   0        13         2  0   80   0  -    0  -    ?       ?      00:00:00 ksoftirqd/0
1 I   0        14         2  0   80   0  -    0  -    ?       ?      00:00:02 rcu_sched
1 S   0        15         2  0  -40  -    0  -    ?       ?      00:00:00 migration/0
1 S   0        16         2  0   9   -    0  -    ?       ?      00:00:00 idle_inject/0
1 I   0        17         2  0   80   0  -    0  -    ?       ?      00:00:04 kworker/0:1-rcu_par_gp
1 S   0        18         2  0   80   0  -    0  -    ?       ?      00:00:00 cpuhp/0
1 S   0        19         2  0   80   0  -    0  -    ?       ?      00:00:00 cpuhp/1
1 S   0        20         2  0   9   -    0  -    ?       ?      00:00:00 idle_inject/1
1 S   0        21         2  0  -40  -    0  -    ?       ?      00:00:00 migration/1
```

■ D 无法中断的休眠状态 (通常 IO 的进程);

■ R 正在运行可中在队列中可过行的;

■ S 处于休眠状态;

■ T 停止或被追踪;

■ X 死掉的进程 (基本很少见);

■ Z 僵尸进程;

■ < 优先级高的进程

■ N 优先级较低的进程

■ L 有些页被锁进内存;

■ s 进程的领导者 (在它之下有子进程);

■ l 多进程的 (使用 CLONE_THREAD, 类似 NPTL pthreads);

■ + 位于后台的进程组;

(二)进程及进程通信

(3)父子进程

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    while(1)
    {
        printf("LEX:pid:%d,ppid:%d\n",getpid(),getppid());
        sleep(1);
    }
    return 0;
}
```

```
lex@lex-virtual-machine:~/demo/lesson8$ ./pro_p_solo
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
Lex:pid:13429,ppid:3709
```

只有一个进程在运行

如何再创建一个进程呢？—fork函数可以创建一个子进程

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    pid_t id=fork();//创建子进程
    while(1)
    {
        if(id==0)
        {
            printf("I am process...child---pid:%d,ppid:%d\n",getpid(),getppid());
            sleep(1);
        }
        else if(id>0)
        {
            printf("I am process..father---pid:%d,ppid:%d\n",getpid(),getppid());
            sleep(1);
        }
    }
    return 0;
}
```

```
lex@lex-virtual-machine:~/Desktop$ ps -lA|grep "135"
1 I   0   13503      2  0  80   0 -    0 -    ?      00:00:00 kworker/u
1 I   0   13504      2  0  80   0 -    0 -    ?      00:00:00 kworker/1
1 I   0   13516      2  0  80   0 -    0 -    ?      00:00:00 kworker/0
0 S  1000  13527    3709  0  80   0 -   624 hrtime pts/0    00:00:00 pros_fork
1 S  1000  13528    13527  0  80   0 -   624 hrtime pts/0    00:00:00 pros_fork
0 R  1000  13570    4023  0  80   0 -  3622 -    pts/1    00:00:00 ps
0 S  1000  13571    4023  0  80   0 -  3028 pipe_r pts/1    00:00:00 grep
```

◆ 什么是fork函数：

——在调用fork函数之前，只有一个进程(父进程)，当这个进程调用fork函数之后，fork函数会复制一个进程(子进程)，区别是PID不同，它们的关系是父子关系。

◆ fork函数会返回两次值：

——给父进程返回子进程的pid。

——给子进程返回0。

——失败时，在父进程中返回-1，不创建子进程，并且errno被设置。

(二)进程及进程通信

(4)进程状态-创建运行态进程R

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    while(1);
    return 0;
}
```

```
lex@lex-virtual-machine:~/Desktop$ ps aux| grep 'pro_R'
lex      3992 99.6  0.0   2364    580 pts/0    R+   21:15   3:30 ./pro_R
lex      4035  0.0  0.0   12116    724 pts/1    S+   21:18   0:00 grep --color=auto pro_R
```



(二)进程及进程通信

(4)进程状态-创建休眠态进程S

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    while(1)
        sleep(10);
    return 0;
}
```

```
lex@lex-virtual-machine:~/Desktop$ ps aux| grep 'pro_S'
lex      4075  0.0  0.0   2364   512 pts/0    S+   21:22   0:00 ./pro_S
lex      4077  0.0  0.0  12116   660 pts/1    S+   21:22   0:00 grep --color=auto pro_S
```

S状态是浅度睡眠，随时可以被唤醒，也可以被杀掉。

(二)进程及进程通信

(5)进程状态-X死亡状态和Z僵尸状态

僵尸状态：一个处于僵尸状态的进程，会等待它的父进程或操作系统对它的信息进行读取，之后才会被释放。
死亡状态：进程被操作系统释放了或者自己退出了。

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    pid_t id=fork();
    int count=5;
    while(1)
    {
        if(id==0)
        {
            while(count) {
                printf("i am process..child---pid:%d,ppid:%d\n,count: %d",getpid(),getppid(),--count);
                sleep(1); }
            printf("child quit....\n");
            exit(1);
        }
        else if(id>0)
        {
            printf("i am process..father---pid:%d,ppid:%d\n",getpid(),getppid());
            sleep(1);
        }
    }
    return 0;
}
```

```
lex@lex-virtual-machine:~/Desktop$ ps -aux|grep 'pro_sz'
lex      14520  0.0  0.0   2496   512 pts/0    S+   15:17   0:00 ./pro_sz
lex      14521  0.0  0.0      0      0 pts/0    Z+   15:17   0:00 [pro_sz] <defunct>
lex      14523  0.0  0.0  12116   720 pts/1    +   15:18   0:00 grep --color=auto pro_sz
```

僵尸进程

可以用wait方法和waitpid方法避免僵尸状态