# Linux 进程编程

**Andew Huang <bluedrum@163.com>**

课程内容:

- 进程的概念
- 进程的内存布局
- Linux下进程的控制
  - 创建新进程
  - 进程间同步
  - 僵尸进程
  - Exec执行进程
- 守护进程

## 进程的概念

### 1. 进程的概念

- 通俗的讲,进程就是一个在运行的程序.
  - 进程是一个具有独立功能的程序关于某个数据集合的一次可以并发执行的运行活动，是处于活动状态的计算机程序 。
  - 有一些特殊情况,多进程程序.一个程序的主进程,可能创建若干个子进程.
  - 进程作为构成系统的基本细胞，不仅是系统内部独立运行的实体，而且是独立竞争资源的基本实体。
    - 这里资源一般指CPU运行时间,内存,外部设备访问权

### 2. 可执行文件

一般讲软件是一个包含可以执行代码的文件,或者说是一些规定格式的二进制文件. 存在计算机的文件系统里面.而进程是一个开始执行但是还没有结束的程序的实例. 存在系统的内存之中.就是可执行文件的具体实现.

不同操作系统可执行程序的格式都不一样,如 Linux 下采用称为 ELF 的格式.而 Windows 采用称为 PE 的可执行格式.但总体上大体的结构类似.每一个程序都有一些固定段.而运行时进程操作系也是一样,大体上分为比较固定的几个段.

### 3. Linux 查看进程的命令

- ps 查看进程的标准状况
- top 看进程的 CPU 占用率和内存使用情况
- /proc/<进程编号>
  - 这是一个虚拟目录,下面各个文件表示进程各种情息,可以用 cat 查看
  - 比如 1 号进程,可以用查看各种状态 cat /proc/1/status

# 程序在内存分区

- 在长期计算机科学的发展当中,程序在内存分区已经形成固定的布局.无论 Linux/Unix 下的应用程序.还是 Windows 下的程序,甚至是在没有操作系统的下运行底层程序,如 bootloader,以及 ARM 的单纯的 C 程序.它们都按大致固定的段来进行布局.
- 其它的开发语言,如 Java 之类也采用同样布局方式

**1. 保存时的分段:**

对于一个 C 的程序而言,当一个程序还是一个可执行文件时.至少包含如下几个段

- Text 段: 保存是机器代码
- Data 段 : 保存是已经初始化代码
- BSS 段(Block Storage Start).:保存是未初始化的代码

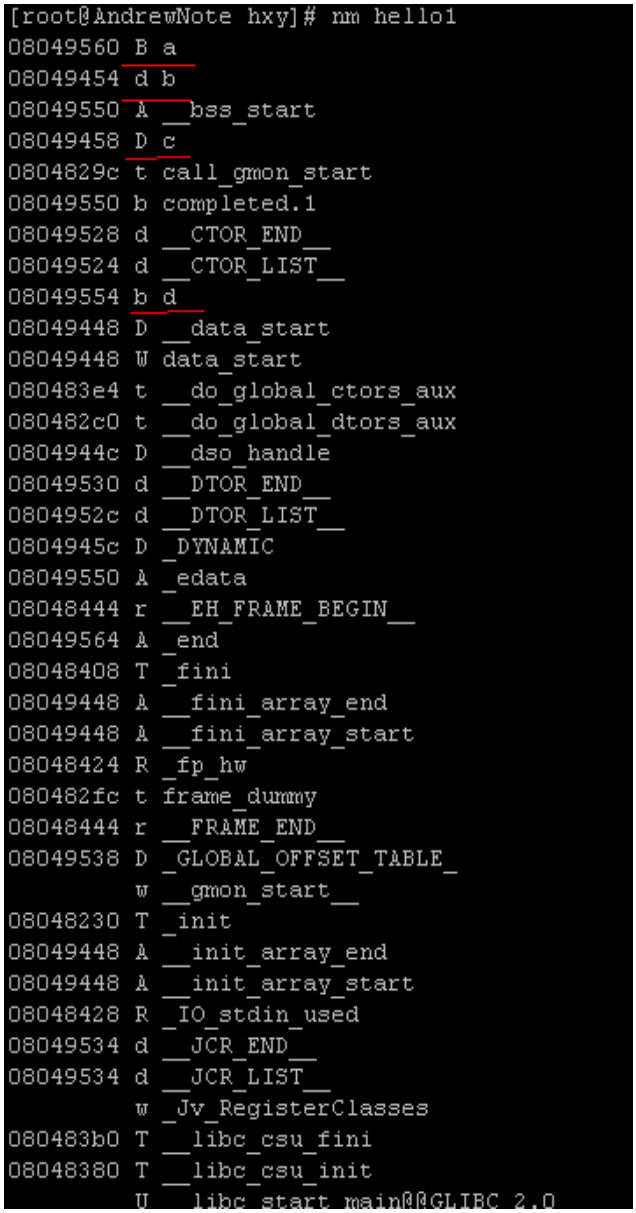
清晰的知道不同类型的变量分布在哪一个段里,是一个 C 开发者必备知识之一.

编译器会把源代码各部分按如下规则把各部分放到可执行程序文件各个段中

- **代码段(text segment)**：存放 CPU 执行的机器指令(machine instructions)。也就是存储你的程序代码编译后的机器代码,在内存中,这一段是只读的.**所有可执行代码即 C 的语句和函数都会被编译到可执行段中.**
- **初始化数据段/数据段(initialized data segment/data segmen**,包含静态初始化的数据，**所以有初值的全局变量和 static 变量在 data 区。**
- **未初始化数据段/bss 段**,,bss 是英文 Block Started by Symbol 的简称，通常是指用来存放程序中未初始化的全局变量和静态变量的一块内存区域，在程序载入时由内核清 0,**所有未初始化的全局变量,包括静态或非静态的,均保存在这一个段.**
  - 因为没有初始化数据,,为了节约可执行文件空间,所以所有在 BSS 段里的程序只占一个标识符空间,和记录所占空间大小,其变量所占空间没有展开,
  - 所有在 BSS 段里变量在装入时由操作系统统一清 0
- **注意在可执行文件是不会有局域变量的空间的,因为他们要等于执行后在进程的空间动态在 stack 区创建.**

```
#include <stdio.h>
int a;      /* 未初始,分配到bss 段 */
static int b=3; /*   已初始化变量,分配到data段中*/
int c=4;    /*已初始化变量,分配到data段中*/
static char  d[10]={3,4};/*已初始始化变量,分配到data段中,其中数据直接写在data空间*/
main()
{
    int e=3; /* 局域变量,在程序文件里没有位置*/
    printf("hello !");
  printf("%d,%d,%d,%s,%d",a,b,c,d,e);
}
```

在 Linux 下,可以用 **nm** 来导出符号表,来查看变量是属于哪一个区的.



```
[root@AndrewNote hxy]# nm hello1
08049560 B a
08049454 d b
08049550 A __bss_start
08049458 D c
0804829c t call_gmon_start
08049550 b completed.1
08049528 d __CTOR_END__
08049524 d __CTOR_LIST__
08049554 b d
08049448 D __data_start
08049448 W data_start
080483e4 t __do_global_ctors_aux
080482c0 t __do_global_dtors_aux
0804944c D __dso_handle
08049530 d __DTOR_END__
0804952c d __DTOR_LIST__
0804945c D _DYNAMIC
08049550 A _edata
08048444 r __EH_FRAME_BEGIN__
08049564 A _end
08048408 T _fini
08049448 A __fini_array_end
08049448 A __fini_array_start
08048424 R _fp_hw
080482fc t frame_dummy
08048444 r __FRAME_END__
08049538 D _GLOBAL_OFFSET_TABLE_
         w __gmon_start__
08048230 T _init
08049448 A __init_array_end
08049448 A __init_array_start
08048428 R _IO_stdin_used
08049534 d __JCR_END__
08049534 d __JCR_LIST__
         w _Jv_RegisterClasses
080483b0 T __libc_csu_fini
08048380 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
```

## 2. 运行时的分区:

当程序运行后.除了上述段会装入进程空间,还有两个新的区分配在进程空间中.

- **栈段(stack),** 保存函数的局部变量和函数参数。是一种“后进先出”（Last In First Out，LIFO）的数据结构，这意味着最后放到栈上的数据，将会是第一个从栈上移走的数据。
  - 对于哪些暂时存贮的信息，和不需要长时间保存的信息来说，LIFO 这种数据结构非常理想。在调用函数或过程后，系统通常会清除栈上保存的局部变量、函数调用信息及其它的信息。栈另外一个重要的特征是，它的地址空间“向下减少”，即当栈上保存的数据越多，栈的地址就越低。栈（stack）的顶部在可读写的 RAM 区的最后。因为栈是有限度的.因此.无限递归之类调用会将栈空间用关.

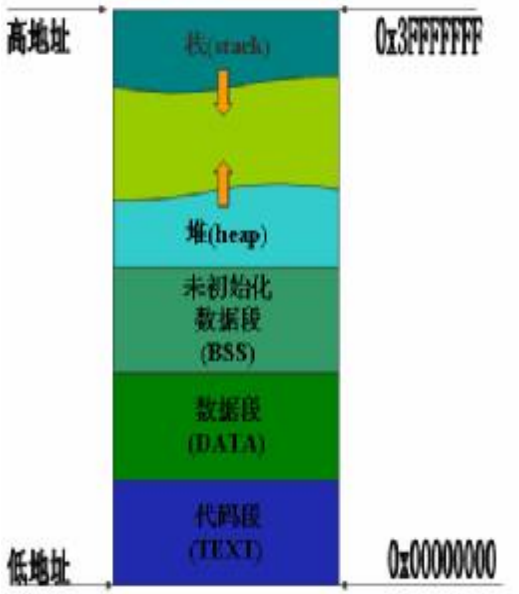
- **堆段(heap)：用于动态内存分配(dynamic memory allocation)。所有 malloc 的分配的内存空间都是从这一个区域分配出来**.保存函数内部动态分配内存，是另外一种用来保存程序信息的数据结构，更准确的说是保存程序的动态变量。
  - 堆是“先进先出”（First In first Out，FIFO）数据结构。它只允许在堆的一端插入数据，在另一端移走数据。堆的地址空间“向上增加”，即当堆上保存的数据越多，堆的地址就越高。

**堆栈分配原因**

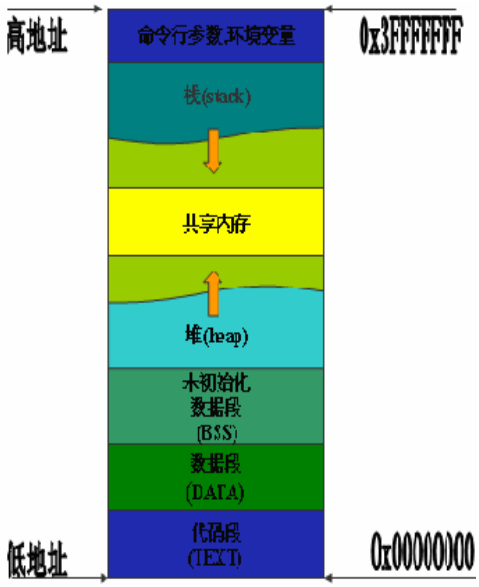
- 函数参数,函数返回值.局域变量均是保存在栈(stack)中.
- 所有 malloc 分配的空间,都是从堆(heap)中进行分配.用 free 把不用空间送回堆
  - C++的 new/delete 是 malloc/free 的封装

**进程典型分区**

- 每个 32 位 CPU 上的进程通常有 4G 虚拟空间,但一般高端地址被操作系统控制,一般是 2G 的范围,是进程自己无法控制.只有低端的 2G 才是进程真正能控制区.
- 代码段(text segment)从地址低端开始(往上)，栈底从地址高端地址往下（在这个特定的表示结构中，栈段从高地址向低地址扩展）。在堆顶和栈顶之间的虚拟地址空间是很大的（这保证了 2 个段不会互相干扰）。




程序运行的布局




Liunx 进程内存布局

**.bss 段和.data 段的区别**

```
/* program1.c*/
int ar[30000];
void main()
{
   printf(“hello”);
}
```

```
/* program2.c*/
int ar[300000] = {1, 2, 3, 4, 5, 6 };
void main()
{
   printf(“hello”);
}
```

用 gcc 编译可以发现 program2 编译之后所得的可执行文件比 program1 的要大得多。这是因为未初始化的变量会保存 BSS 段,并且只是这个段里放一个全局变量 ar 标识符和尺寸,并未占用太多空间.所以程序 1 较小,而程序 2 中的变量 ar 被初始化,将在 data 段中展开,因为整个程序也变得较大

**栈区与堆区的区别**

在一个应用程序的虚拟空间里,堆的尺寸远大于栈.而栈被频繁的使用,局域变量.函数参数等都需要栈的来处理.栈比一般开发者想象还要小.有时会小到只有 4096 字节.

因此在一个应用程序,分配一个巨大的自动变量和分配一个巨大的全局变量一样,是一件非常不明智的事情,在运行时,会立刻造成程序段错误.比较好的做法是用动态分配的方法来分配巨大的结构.

/*采用栈来分配,
运行立刻会造成段错误*/

```
/*采用栈来分配,
在某一些 OS 可能会造成段错误
*/
main()
{
    char buf[25000];
    strcpy(buf,"hxy");
    printf(buf);
}
```

```
/*采用堆来分配空间*/
main()
{
    char buf = malloc(25000);
    strcpy(buf,"hxy");
    printf(buf);
    free(buf);
}
```

## 进程控制

**1. 常见的进程控制函数**

- 常见进程控制函数
  - fork – 创建一个新进程
  - exec – 用一个新进程去执行一个命令行
  - exit – 正常退出进程
  - abort – 非正常退出一个进程
  - kill – 杀死进程或向一个进程发送信号
  - wait – 等待子进程结束
  - sleep – 将当前进程休眠一段时间
  - getpid – 取得进程编号
  - getppid – 取得父进程编号

**2. 进程标示符**

- 进程 ID 也被称作进程标识符，是一个非负的整数，在 Linux 操作系统中唯一地标志一个进程

| PID | Associated Process |
|---|---|
| 0 | swapper (scheduler) |
| 1 | init (/sbin/init) |
| 2 | pagedaemon<br>(virtual memory paging) |
| 3, 4, … | other processes |

**进程 ID 相关的系统调用**

- #include <sys/types.h>
- #include <unistd.h>
- **pid_t getpid(void);** 取得当前进程标识
- **pid_t getppid(void);** 取得当前进程父母进程标识
- **uid_t getuid(void);** 来取得执行目前进程的用户识别码
- **uid_t geteuid(void);** 用来取得执行目前进程有效的用户识别码
  - uid,euid 区别请看下页
- **gid_t getgid(void);** 取得当前进程组编号
- **gid_t getegid(void);** 取得当前进程有效组编号

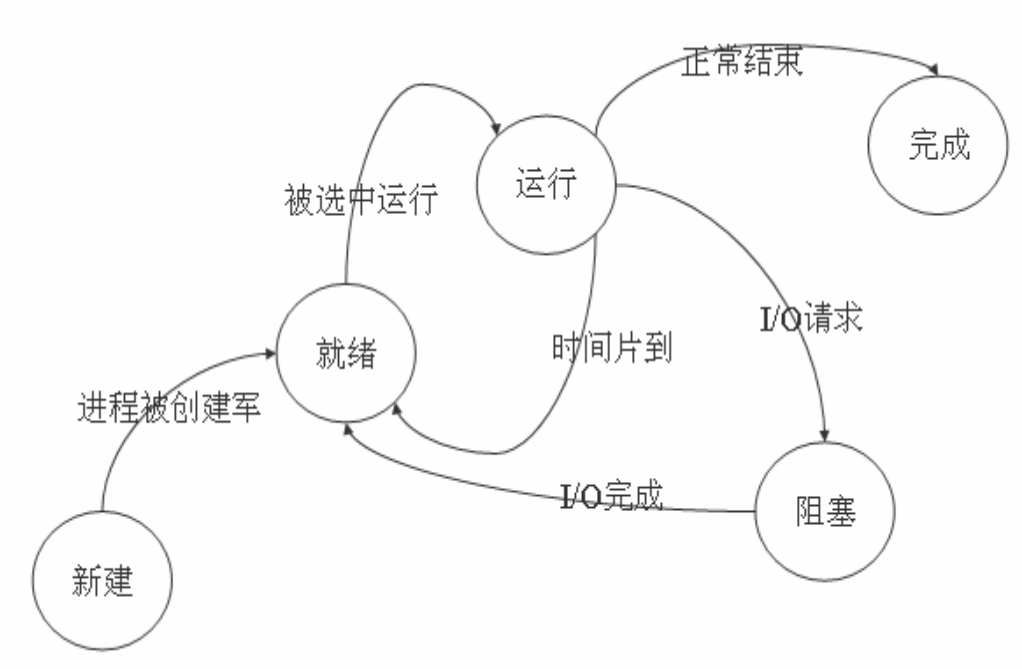
**真实 ID 与有效 ID**

- 每个进程会被分配三个 ID
  - 真实，有效和保存 ID(Real, Effective, Saved UID)
  - 内核会给每个进程关联两个和进程 ID 无关的用户 ID，一个是真实用户 ID，还有一个是有效用户 ID 或者称为 setuid（set user ID）。
    - 真实用户 ID 用于标识由谁为正在运行的进程负责。
    - 有效用户 ID 用于为新创建的文件分配所有权、检查文件访问许可，还用于通过 kill 系统调用向其它进程发送信号时的许可检查。内核允许一个进程以调用 exec 一个 setuid 程序或者显式执行 setuid 系统调用的方式改变它的有效用户 ID。

## 3. 进程状态

- 进程状态（state)说明了它在某一个特定时刻的状况。大多数操作系统都允许如下几种状态
  - 新建(new) – 正在创建
  - 运行(running) – 正在执行指令
  - 阻塞(blocked) – 等待像 I/O 这样的事件
  - 就绪(ready) – 等待分配处理器
  - 完成(done) – 结束

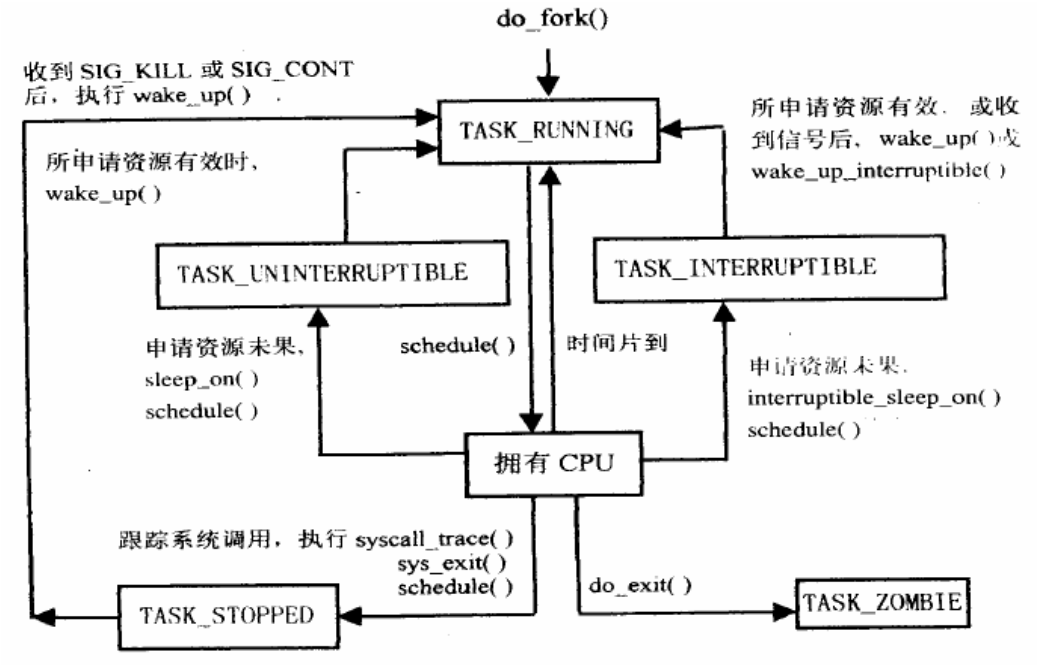
**进程状态迁移图**




**Linux 进程的状态**

- 每个进程从创建到激活的整个生存周期.Linux 进程分为几个状态,
  - 用户状态：进程在用户状态下运行的状态。
  - 内核状态：进程在内核状态下运行的状态。
  - 内存中就绪：进程没有执行，但处于就绪状态，只要内核调度它，就可以执行。
  - 内存中睡眠：进程正在睡眠并且进程存储在内存中，没有被交换到 SWAP 设备。
  - 就绪且换出：进程处于就绪状态，但是必须把它换入内存，内核才能再次调度它进行运行。
  - 睡眠且换出：进程正在睡眠，且被换出内存。
  - 被抢先：进程从内核状态返回用户状态时，内核抢先于它，做了上下文切换，调度了另一个进程。原先这个进程就处于被抢先状态。
  - 创建状态：进程刚被创建。该进程存在，但既不是就绪状态，也不是睡眠状态。这个状态是除了进程 0 以外的所有进程的最初状态。
  - 僵尸状态（zombie）：进程调用 exit 结束，进程不再存在，但在进程表项中仍有纪录，该纪录可由父进程收集。
- 各种状态可以互相转换,进程在它的生命周期里并不一定要经历所有的状态。

**Linux 各种状态翻转**




**ps查看进程状态**

- PS显示状态
  - D 不可中断睡眠 (通常是在IO操作)
  - R 正在运行或可运行（在运行队列排队中）
  - S 可中断睡眠 (在等待某事件完成)
  - T Stopped, either by a job control signal or because it is being traced.
  - W 正在换页(2.6.内核之前有效)
  - X 死进程 (should never be seen)
  - Z 僵尸

**3. 进程调度**

- CPU 一个时刻只能运行一个程序。在操作系统实现的多进程,看起来好象在同时运行多个程序。实际是是 OS 的制造的假象
- 像一个人只有一双手，但是可以同时操作 N 个桔子一样。这牵涉到一个调度的问题(schedule )。
- 在 Linux 中，每个进程在创建时都会被分配一个数据结构，称为进程控制块（Process Control Block，简称 PCB）。PCB 中包含了很多重要的信息，供系统调度和进程本身执行使用.
- 在调度时，操作系统不断进行上下文切换（context switch),即将一个进程从运行状态退出，并运行另一个进程。

**4. 进程的创建**

- 在Linux 中，创造新进程的方法只有一个:fork(), 创建子进程, 其它调用system,exec 最后也是调用fork.
  - **pid_t fork()**;

- shell 执行一个命令相当于调用了 fork

l 当一个进程调用了fork 以后，系统会创建一个子进程. 这个子进程和父进程不同的地方只有他的进程ID 和父进程ID，其他的都是一样. 就象符进程克隆(clone) 自己一样
  - 参考代码.fork_test.c

**fork 的常用结构**

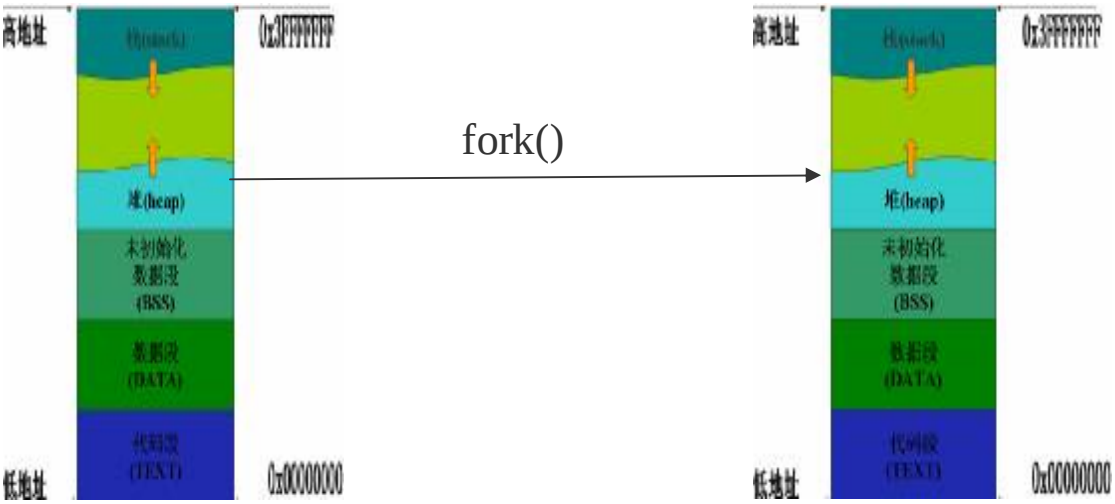
l 为了区分父进程和子进程,我们必须跟踪fork的返回值. 当fork掉用失败的时候(内存不足或者是用户的最大进程数已到)fork返回-1,否则fork的返回值有重要的作用.对于父进程fork返回子进程的ID,而对于fork子进程返回0.我们就是根据这个返回值来区分父子进程的.

```
#include<sys/types.h>
#inlcude<unistd.h>
main()
{
        pid_t pid;

        /*此时仅有一个进程*/
        pid=fork();
        /*此时已经有两个进程在同时运行*/
        if(pid<0)
                printf("error in fork!");
        else if(pid==0)
                printf("I am the child process, my process ID is %d\n",getpid());
        else
                printf("I am the parent process, my process ID is %d\n",getpid());
}
```

**fork()的特点**

l fork()的子进程是完整从父进程里克隆一个内存 w 分区出来.
  - 创建后,子进程运行就跟父进程没有什么关系了,两个进程各自独立运行



- fork()是 Linux 机制，这种多进程执行代码在 C 语言上是解释不通的。
  - 即一段代码由子进程执行，一段代码由父进程执行
- 在 fork 时.子进程从父进程拷贝完整的一套 text,data，bss 和堆和栈。然后在内存中另外的地方建立一模一样的环境。
  - 因此 fork 一瞬间，子进程的全局变量，局域变量，打开的文件操作符跟父进程一模一样。
  - 唯一的区别在于 fork 返回值,如果是用一个局域变量来保存,那么在父进程空间这个值大于 0,子进程空间等于 0
  - 但随着代码的变化，各个变量的值开始发生不同变化

**5. 进程的退出**

- exit,_exit 或者 main 函数里 return 来设置进程退出值.
- 为了得到这个值 Linux 定义了几个宏来测试这个返回值.
  - WIFEXITED:判断子进程退出值是非 0
    WEXITSTATUS:判断子进程的退出值(当子进程退出时非 0).
    WIFSIGNALED:子进程由于有没有获得的信号而退出.
    WTERMSIG:子进程没有获得的信号号(在 WIFSIGNALED 为真时才有意义).
  - 在 shell 用$?做同样事情.
- exit()函数与_exit()函数最大的区别就在于exit()函数在调用 exit 系统调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是清理 I/O 缓冲

**6. 进程的阻塞**

- 一旦子进程被创建,父子进程一起从 fork 处继续执行,相互竞争系统的资源.有时候我们希望子进程继续执行,而父进程阻塞直到子进程完成任务.这个时候我们可以调用 wait 或者 waitpid 系统调用.
  - **pid_t wait(int *stat_loc);**
    **pid_t waitpid(pid_t pid,int *stat_loc,int options);**
- wait 系统调用会使父进程阻塞直到一个子进程结束或者是父进程接受到了一个信号.如果没有父进程没有子进程或者他的子进程已经结束了 wait 回立即返回.成功时(因一个子进程结束)wait 将返回子进程的 ID,否则返回-1,并设置全局变量 errno.stat_loc 是子进程的退出状态.子进程调用
- 从本质上讲，系统调用 waitpid 和 wait 的作用是完全相同的，但 waitpid 多出了两个可由用户控制的参数 pid 和 options，从而为我们编程提供了另一种更灵活的方式。下面我们就来详细介绍一下这两个参数：
  - **static inline pid_t wait(int * wait_stat) { return waitpid(-1,wait_stat,0); }**

**wait**

| | wait（等待子进程中断或结束） |
|---|---|
| 相关函数 | waitpid，fork |
| 表头文件 | #include<sys/types.h><br>#include<sys/wait.h> |
| 定义函数 | pid_t wait (int * status); |
| 函数说明 | wait()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 wait()时子进程已经结束，则 wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一快返回。如果不在意结束状态值，则 |
| 参数 | status 可以设成 NULL。子进程的结束状态值请参考 waitpid()。 |
| 返回值 | 如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1。失败原因存于 errno 中。 |
| 附加说明 | |
| 范例 | #include<stdlib.h><br>#include<unistd.h><br>#include<sys/types.h><br>#include<sys/wait.h><br>main()<br>{<br>pid_t pid;<br>int status,i;<br>if(fork()= =0){<br>printf(“This is the child process .pid =%d\n”,getpid());<br>exit(5);<br>}else{<br>sleep(1);<br>printf(“This is the parent process ,wait for child...\n”;<br>pid=wait(&status);<br>i=WEXITSTATUS(status);<br>printf(“child’s pid =%d .exit status=^d\n”,pid,i);<br>}<br>} |
| 执行 | This is the child process.pid=1501<br>This is the parent process .wait for child...<br>child’s pid =1501,exit status =5 |

**waitpid**

| | waitpid（等待子进程中断或结束） |
|---|---|
| 相关函数 | wait，fork |
| 表头文件 | #include<sys/types.h><br>#include<sys/wait.h> |
| 定义函数 | pid_t waitpid(pid_t pid,int * status,int options); |
| 函数说明 | waitpid()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 wait()时子进程已经结束，则 wait()会立即返回子进程结 束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一快返回。如果不在意结束状态值，则参数 status 可以设成 NULL。参数 pid 为欲等待的子进程识别码，其他数值意义如下:<br>pid<-1 等待进程组识别码为 pid 绝对值的任何子进程。<br>pid=-1 等待任何子进程，相当于 wait()。<br>pid=0 等待进程组识别码与目前进程相同的任何子进程。<br>pid>0 等待任何子进程识别码为 pid 的子进程。<br>参数 option 可以为 0 或下面的 OR 组合<br>WNOHANG 如果没有任何已经结束的子进程则马上返回，不予以等待。<br>WUNTRACED 如果子进程进入暂停执行情况则马上返回，但结束状态不予以理会。<br>子进程的结束状态返回后存于 status，底下有几个宏可判别结束情况<br>WIFEXITED(status)如果子进程正常结束则为非 0 值。<br>WEXITSTATUS(status)取得子进程 exit()返回的结束代码，一般会先用 WIFEXITED 来判断是否正常结束才能使用此宏。<br>WIFSIGNALED(status)如果子进程是因为信号而结束则此宏值为真<br>WTERMSIG(status)取得子进程因信号而中止的信号代码，一般会先用 WIFSIGNALED 来判断后才使用此宏。<br>WIFSTOPPED(status)如果子进程处于暂停执行情况则此宏值为真。一般只有使用 WUNTRACED 时才会有此情况。<br>WSTOPSIG(status)取得引发子进程暂停的信号代码，一般会先用 WIFSTOPPED 来判断后才使用此宏。 |
| 返回值 | 如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1。失败原因存于 errno 中。 |
| 范例 | 参考 wait()。 |

- waitpid的返回值比wait稍微复杂一些，一共有3种情况：
  - 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

**用wait取返回值的实例**

```
if(pid) {
    int stat_val;
    pid_t child_pid;
    //父进程比子进程运行时间快，如果不 WAIT，父进程将直接退出
    printf("parent proccess %d wait...\n",getpid());
    child_pid = wait(&stat_val);

    printf("Child has finished: PID = %d\n", child_pid);
    if(child_pid==-1)
        fprintf(stderr,"Wait Error:%s\n",strerror(errno));
    else if(WIFEXITED(stat_val))
        printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
    else if(WIFSIGNALED(stat_val))
        printf("Child %ld terminated due to signal %d znot caught\n",
child_pid,WTERMSIG(stat_val));
    else
        printf("Child terminated abnormally\n");
  }
```

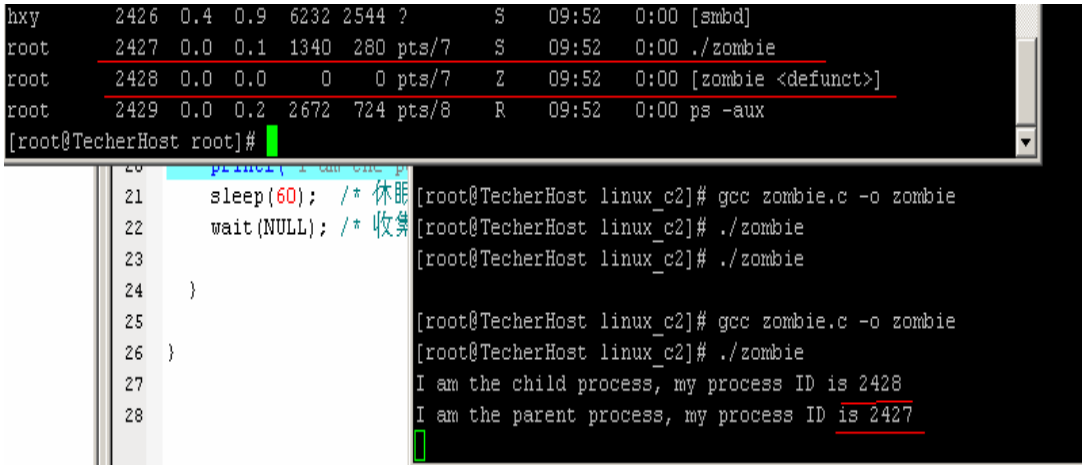
## 7. 进程间同步

- 进程同步就是要协调好 2 个以上的进程，使之以安排好地次序依次执行,比如
- 进程间同步有多种方法,其中用 wait 是一种方法之一
  - wait 只能用于有亲戚关系的进程之间进行同步

## 8. 僵尸进程(Zombie )

- 在一个进程调用了exit 之后，该进程并非马上就消失掉，而是留下一个称为僵尸进程（Zombie ）的数据结构。在Linux 进程的5 种状态中，僵尸进程是非常特殊的一种，它已经放弃了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其他进程收集，除此之外，僵尸进程不再占有任何内存空间。
- 参见Zombie.c
- 僵尸进程虽然对其他进程几乎没有什么影响，不占用CPU 时间，消耗的内存也几乎可以忽略不计

**Zombie 实例**




**9. exec 一组执行外部命令调用**

- exec指的是一组函数，一共有6个
  - #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[]);**
- 其中只有execve是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。
- exec函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何Linux下可执行的脚本文件。
- 上面6条函数看起来似乎很复杂，但实际上无论是作用还是用法都非常相似，只有很微小的差别。
- 完全整的main 主函数定义
  - int main(int argc, char *argv[], char *envp[])
  - 参数argc 指出了运行该程序时命令行参数的个数，数组argv 存放了所有的命令行参数，数组envp 存放了所有的环境变量。
- int execve(const char *path, char *const argv[], char *const envp[]);
- 前3 个函数都是以execl 开头的，后3 个都是以execv 开头的，它们的区别在于，execv 开头的函数是以¡°char *argv[]¡± 这样的形式传递命令行参数，而execl 开头的函数采用了我们更容易习惯的方式，把参数一个一个列出来，然后以一个NULL 表示结束。这里的NULL 的作用和argv 数组里的NULL 作用是一样的。
- 全部6 个函数中，只有execle 和execve 使用了char *envp[] 传递环境变量，其它的4 个函数都没有这个参数，这并不意味着它们不传递环境变量，这4 个函数将把默认的环境变量不做任何修改地传给被执行的应用程序。而execle 和execve 会用指定的环境变量去替代默认的那些。

**1)execl**

| | execl（执行文件） |
|---|---|
| 相关函数 | fork，execle，execlp，execv，execve，execvp |
| 表头文件 | #include<unistd.h> |
| 定义函数 | int execl(const char * path,const char * arg,....); |
| 函数说明 | execl()用来执行参数 path 字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的 argv(0)、argv[1]……，最后一个参数必须用空指针(NULL)作结束。 |
| 返回值 | 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。 |
| 范例 | #include<unistd.h><br>main()<br>{<br>execl(“/bin/ls”,”ls”,”-al”,”/etc/passwd”,(char * )0);<br>} |
| 执行 | /*执行/bin/ls -al /etc/passwd */<br>-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd |

**2)execlp**

| | execlp（从 PATH 环境变量中查找文件并执行） |
|---|---|
| 相关函数 | fork，execl，execle，execv，execve，execvp |
| 表头文件 | #include<unistd.h> |
| 定义函数 | int execlp(const char * file,const char * arg,……); |
| 函数说明 | execlp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的 argv[0]、argv[1]……，最后一个参数必须用空指针(NULL)作结束。 |
| 返回值 | 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。 |
| 错误代码 | 参考 execve()。 |
| 范例 | /* 执行 ls -al /etc/passwd execlp()会依 PATH 变量中的/bin 找到/bin/ls */<br>#include<unistd.h><br>main()<br>{<br>execlp(“ls”,”ls”,”-al”,”/etc/passwd”,(char *)0);<br>} |
| 执行 | -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd |

**3)execv**

| | execv（执行文件） |
|---|---|
| 相关函数 | fork，execl，execle，execlp，execve，execvp |
| 表头文件 | #include<unistd.h> |
| 定义函数 | int execv (const char * path, char * const argv[ ]); |
| 函数说明 | execv()用来执行参数 path 字符串所代表的文件路径，与 execl()不同的地方在于 execve()只需两个参数，第二个参数利用数组指针来传递给执行文件。 |
| 返回值 | 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。 |
| 错误代码 | 请参考 execve（）。 |
| 范例 | /* 执行/bin/ls -al /etc/passwd */<br>#include<unistd.h><br>main()<br>{<br>char * argv[ ]={“ls”,”-al”,”/etc/passwd”,(char*) }};<br>execv(“/bin/ls”,argv);<br>} |
| 执行 | -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd |

**4)execve**

| | execve（执行文件） |
|---|---|
| 相关函数 | fork，execl，execle，execlp，execv，execvp |
| 表头文件 | #include<unistd.h> |
| 定义函数 | int execve(const char * filename,char * const argv[ ],char * const envp[ ]); |
| 函数说明 | execve()用来执行参数 filename 字符串所代表的文件路径，第二个参数系利用数组指针来传递给执行文件，最后一个参数则为传递给执行文件的新环境变量数组。 |
| 返回值 | 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。 |

| | |
|---|---|
| 错误代码 | EACCES<br>1. 欲执行的文件不具有用户可执行的权限。<br>2. 欲执行的文件所属的文件系统是以 noexec 方式挂上。<br>3.欲执行的文件或 script 翻译器非一般文件。<br>EPERM<br>1.进程处于被追踪模式，执行者并不具有 root 权限，欲执行的文件具有 SUID 或 SGID 位。<br>2.欲执行的文件所属的文件系统是以 nosuid 方式挂上，欲执行的文件具有 SUID 或 SGID 位元，但执行者并不具有 root 权限。<br>E2BIG 参数数组过大<br>ENOEXEC 无法判断欲执行文件的执行文件格式，有可能是格式错误或无法在此平台执行。<br>EFAULT 参数 filename 所指的字符串地址超出可存取空间范围。<br>ENAMETOOLONG 参数 filename 所指的字符串太长。<br>ENOENT 参数 filename 字符串所指定的文件不存在。<br>ENOMEM 核心内存不足<br>ENOTDIR 参数 filename 字符串所包含的目录路径并非有效目录<br>EACCES 参数 filename 字符串所包含的目录路径无法存取，权限不足<br>ELOOP 过多的符号连接<br>ETXTBUSY 欲执行的文件已被其他进程打开而且正把数据写入该文件中<br>EIO I/O 存取错误<br>ENFILE 已达到系统所允许的打开文件总数。<br>EMFILE 已达到系统所允许单一进程所能打开的文件总数。<br>EINVAL 欲执行文件的 ELF 执行格式不只一个 PT_INTERP 节区<br>EISDIR ELF 翻译器为一目录<br>ELIBBAD ELF 翻译器有问题。 |
| 范例 | #include<unistd.h><br>main()<br>{<br>char * argv[ ]={“ls”,”-al”,”/etc/passwd”,(char *)0};<br>char * envp[ ]={“PATH=/bin”,0}<br>execve(“/bin/ls”,argv,envp);<br>} |
| 执行 | -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd |

**5)execvp**

| | |
|---|---|
| | execvp（执行文件） |
| 相关函数 | fork，execl，execle，execlp，execv，execve |
| 表头文件 | #include<unistd.h> |

| | |
|---|---|
| 定义函数 | int execvp(const char *file ,char * const argv []); |
| 函数说明 | execvp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行该文件，然后将第二个参数 argv 传给该欲执行的文件。 |
| 返回值 | 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。 |
| 错误代码 | 请参考 execve（）。 |
| 范例 | /*请与 execlp（）范例对照*/<br>#include<unistd.h><br>main()<br>{<br>char * argv[ ] ={ “ls”,”-al”,”/etc/passwd”,0};<br>execvp(“ls”,argv);<br>} |
| 执行 | -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd |

**10. System 函数**

- system 是直接用一个字符串执行一个命令
  - **system(“ls –l”)**
  - **system(“ifconfig eth0 192.168.0.104”);**
  - 定义:int system(const char * *c m d s t r i n g*) ;
- 在ANSI C 定义了，但是对操作系统依赖很高，通常在Linux 使用.
- s y s t e m 可以看成是一个阻塞执行的函数，在其实现中调用了f o r k 、e x e c 和w a i t p i d ，因此有三种返回值
  - 如果f o r k失败或者w a i t p i d返回除E I N T R之外的出错，则s y s t e m返回－1，而且e r r n o中设置了错误类型。
  - 如果e x e c 失败( 表示不能执行s h e l l) ，则其返回值如同s h e l l 执行了e x i t ( 1 2 7 ) 一样。
  - 否则所有三个函数( f o r k , e x e c 和w a i t p i d ) 都成功，并且s y s t e m 的返回值是s h e l l 的终止状态，
  - 其格式已在w a i t p i d 中说明。

**System**

| | system（执行 shell 命令） |
|---|---|
| 相关函数 | fork，execve，waitpid，popen |
| 表头文件 | #include<stdlib.h> |
| 定义函数 | int system(const char * string); |
| 函数说明 | system()会调用 fork()产生子进程，由子进程来调用/bin/sh-c string 来执行参数 string 字符串所代表的命令，此命令执行完后随即返回原调用的进程。在调用 system()期间 SIGCHLD 信号会被暂时搁置，SIGINT 和 SIGQUIT 信号则会被忽略。 |

| | |
|---|---|
| 返回值 | 如果 system()在调用/bin/sh 时失败则返回 127，其他失败原因返回-1。若参数 string 为空指针(NULL)，则返回非零值。如果 system()调用成功则最后会返回执行 shell 命令后的返回值，但是此返回值也有可能为 system()调用/bin/sh 失败所返回的 127，因 此最好能再检查 errno 来确认执行成功。 |
| 附加说明 | 在编写具有 SUID/SGID 权限的程序时请勿使用 system()，system()会继承环境变量，通过环境变量可能会造成系统安全的问题。 |
| 范例 | #include<stdlib.h><br>main()<br>{<br>system("ls -al /etc/passwd /etc/shadow");<br>} |
| 执行 | -rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd<br>-r-------- 1 root root 572 Sep 2 15 :34 /etc/shadow |

## 守护进程

**1. daemon 进程**

- Daemon 运行在后台，有时人们也把它们称作¡°后台服务进程¡±。 类似于 Windows 的服务
- 几乎所有的服务器程序，包括我们熟知的Apache 和wu-FTP ，都用daemon 进程的形式实现。很多Linux 下常见的命令如inetd 和ftpd ，末尾的字母d 就是指daemon 。

**2. Daemon 的结构**

- **daemon进程的编程规则**
  - 调用fork产生一个子进程，同时父进程退出。我们所有后续工作都在子进程中完成。
    - 这样做我们可以交出控制台的控制权,并为子进程作为进程组长作准备
    - 由于父进程已经先于子进程退出，会造成子进程没有父进程，变成一个孤儿进程（orphan）。每当系统发现一个孤儿进程，就会自动由1号进程收养它，这样，原先的子进程就会变成1号进程的子进程。
  - 调用setsid系统调用。这是整个过程中最重要的一步。的作用是创建一个新的会话（session），并自任该会话的组长（session leader）。
    - 让进程摆脱原会话的控制；
    - 让进程摆脱原进程组的控制；
    - 让进程摆脱原控制终端的控制；
  - 把当前工作目录切换到根目录。以防止在整个进程运行期间该文件所在当前目系统系统都无法被卸下（umount），
  - 将文件权限掩码设为0。这需要调用系统调用umask ,这样Deamon创建文件不会有太大麻烦
  - 关闭所有不需要的文件。同文件权限掩码一样，我们的新进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不被我们的daemon进程读或写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法卸下。需要指出的是，

文件描述符为0、1和2的三个文件（文件描述符的概念将在下一章介绍），也就是我们常说的输入、输出和报错这三个文件也需要被关闭。

**守护进程基本结构**

```
pid_t pid;
        int i;
  pid=fork();

        if(pid<0){
                printf("error in fork\n");
                exit(1);
        }
        else if(pid == 0)
                {
                  printf("child process gid ,%d ppid %d\n",getpid(),getppid());
                  /*.... */
                }
        else if(pid>0)
                {
                /*  父进程退出  */
                printf("parent process gid %d\n",getpid());
                exit(0);

        }

        printf("current process gid %d\n",getpid());

        /*  调用 setsid */
        setsid();
        /*  切换当前目录  */
        chdir("/");
        /*  设置文件权限掩码  */
        umask(0);
        /*  关闭所有可能打开的不需要的文件  */
        for(i=0;i<MAXFILE;i++)
                close(i);
```

## 课堂练习

**练习题:**

1) 指出以下变量数据存储位置

全局变量 int(*g_pFun)(int);g_pFun=myFunction;g_pFun 存储的位置( )

指向空间的位置( )

函数内部变量 static int nCount; ( )

函数内部变量 char p[]=”AAA”; p 指向空间的位置( )

函数内部变量 char *p=”AAA”; p 指向空间的位置( )

函数内部变量 char *p=new char; p 的位置( ) 指向空间的位置( )

A. 数据段

B. 代码段

C. 堆栈

D. 堆

E. 不一定, 视情况而定

2)请指出下列哪一些变量不存储在栈区

A.函数返回值, B.函数参数 C.给 volatie 修饰的局域变量. D.函数内定义的静态变量

3)请问屏幕一共输出几句 printf?

假设子进程运行父进程略快,请问程序会在屏幕上输出什么内容?

```
int main()
{
   int var=100;
   pid_t pt;
   pt = fork();
   printf(“%d\n”,var);
   if(pt==0)
   {
       var+=10;
       printf(“%d\n”,var);
   }
   else (pt> 0)
   {
       var += 33;
   }

   printf(“%d\n”,var++);
}
```

**编程题:**

- 用system加date命令写一个类似mydate程序
  - date 设置命令行格式的设置日期 date -s YYMMDD
    - date -s 090102 # 设置2009 年1 月2 号
  - date 设置命令行格式的设置时间 date -s HH:MM:SS
    - date -s 12:01:02 设置为12 点01 分02 秒
  - 程序要有ROOT 权限
- 用exec 的函数写一个用ifconfig 设置和显示IP 程序
  - 程序要有ROOT 权限
- 将Deamon 创建过程写在一个子函数RunAsDeamon() 里
  - 不运行这一个函数，程序变成普通程序
  - 运行这一函数，程序变成守护进程

**扩展练习**

- 用waitpid,exec*,fork编写一个类似于system函数的调用.
- 完善自写的的gcc ，把参数解析以后，通过system或exec*函数调用真正的gcc来编译.
- 设计一个守护进程
  - 加入上一次处理文件代码
  - 扩展的练习:加入配置文件读写