

第三章：进程

3.1 进程概述

3.1.1 进程的定义

程序：

程序是存放在存储介质上的一个可执行文件。

进程：

进程是程序的执行实例，包括程序计数器、寄存器和变量的当前值。

程序是静态的，进程是动态的：

程序是一些指令的有序集合，而进程是程序执行的过程。进程的状态是变化的，其包括进程的创建、调度和消亡。

在 linux 系统中，进程是管理事务的基本单元。进程拥有自己独立的处理环境和系统资源（处理器、存储器、I/O 设备、数据、程序）。

可使用 `exec` 函数由内核将程序读入内存，使其执行起来成为一个进程。

3.1.2 进程的状态及转换

进程整个生命周期可以简单划分为三种状态：

就绪态：

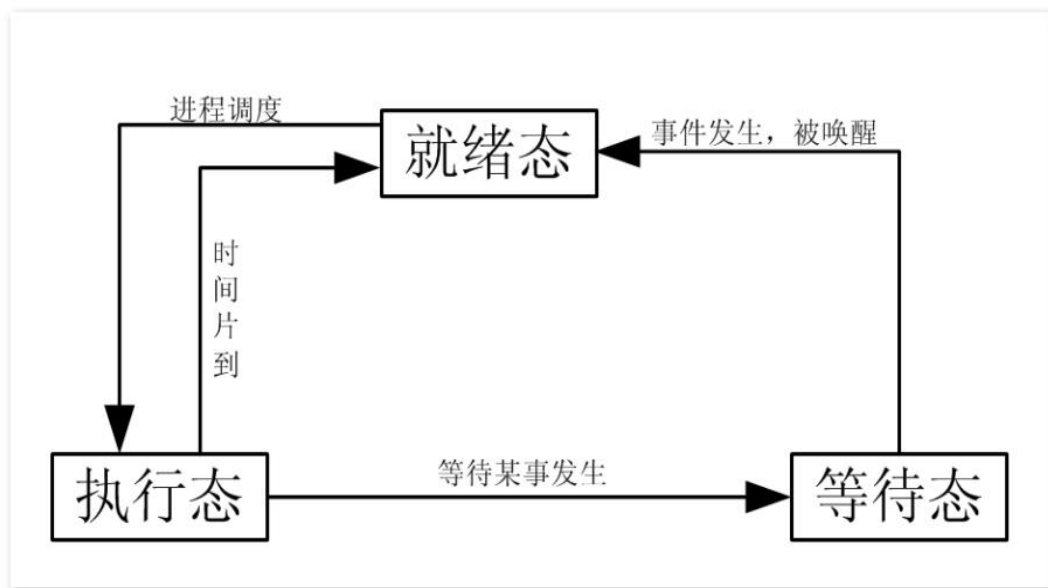
进程已经具备执行的一切条件，正在等待分配 CPU 的处理时间。

执行态：

该进程正在占用 CPU 运行。

等待态：

进程因不具备某些执行条件而暂时无法继续执行的状态。



进程三种状态的转换关系

3.1.3 进程控制块

OS 是根据 PCB 来对并发执行的进程进行控制和管理。系统在创建一个进程的时候会开辟一段内存空间存放与此进程相关的 PCB 数据结构。

PCB 是操作系统中最重要的记录型数据结构。PCB 中记录了用于描述进程进展情况及控制进程运行所需的全部信息。

PCB 是进程存在的唯一标志，在 Linux 中 PCB 存放在 `task_struct` 结构体中。

调度数据

进程的状态、标志、优先级、调度策略等。

时间数据

创建该进程的时间、在用户态的运行时间、在内核态的运行时间等。

文件系统数据

`umask` 掩码、文件描述符表等。

内存数据、进程上下文、进程标识（进程号）

...

3.2 进程控制

3.2.1 进程号

每个进程都由一个进程号来标识，其类型为 `pid_t`，进程号的范围：0~32767。

做真实的自己，用良心做教育

进程号总是唯一的，但进程号可以重用。当一个进程终止后，其进程号就可以再次使用了。

在 linux 系统中进程号由 0 开始。

进程号为 0 及 1 的进程由内核创建。

进程号为 0 的进程通常是调度进程，常被称为交换进程(swapper)。进程号为 1 的进程通常是 init 进程。

除调度进程外，在 linux 下面所有的进程都由进程 init 进程直接或者间接创建。

进程号(PID)

标识进程的一个非负整型数。

父进程号(PPID)

任何进程(除 init 进程)都是由另一个进程创建，该进程称为被创建进程的父进程，对应的进程号称为父进程号(PPID)。

进程组号(PGID)

进程组是一个或多个进程的集合。他们之间相互关联，进程组可以接收同一终端的各种信号，关联的进程有一个进程组号(PGID)。

Linux 操作系统提供了三个获得进程号的函数 getpid()、getppid()、getpgid()。

需要包含头文件：

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

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

pid_t getpid(void)

功能：获取本进程号(PID)

pid_t getppid(void)

功能：获取调用此函数的进程的父进程号(PPID)

pid_t getpgid(pid_t pid)

功能：获取进程组号(PGID)，参数为 0 时返回当前 PGID，否则返回参数指定的进程的 PGID

例：01_pid.c

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

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

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

```
int main(int argc, char *argv[])
```

```
{
```

```
    pid_t pid, ppid, pgid;
```

```
    pid = getpid();
```

```
    printf("pid = %d\n", pid);
```

```
    ppid = getppid();
```

```
    printf("ppid = %d\n", ppid);
```

```
    pgid = getpgid(pid);
```

```
    printf("pgid = %d\n", pgid);
```

```
    return 0;
```

}

3.2.2 进程的创建 fork 函数

在 linux 环境下，创建进程的主要方法是调用以下两个函数：

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

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

```
pid_t fork(void);
```

```
pid_t vfork(void);
```

创建一个新进程

```
pid_t fork(void)
```

功能：

fork()函数用于从一个已存在的进程中创建一个新进程，新进程称为子进程，原进程称为父进程。

返回值：

成功：子进程中返回 0，父进程中返回子进程 ID。

失败：返回-1。

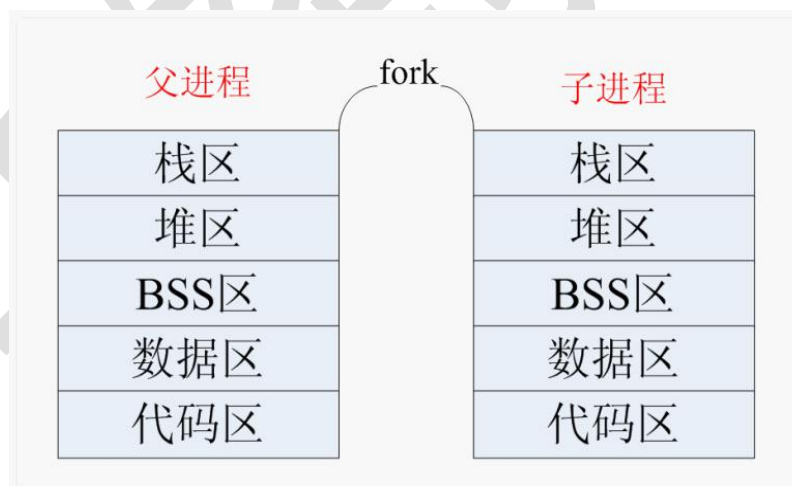
使用 fork 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间。

地址空间：

包括进程上下文、进程堆栈、打开的文件描述符、信号控制设定、进程优先级、进程组号等。

子进程所独有的只有它的进程号，计时器等。因此，使用 fork 函数的代价是很大的。

fork 函数执行结果：



例：02_fork_1.c 创建一个子进程实现多任务

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

```
#include <stdlib.h>
```

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

```
int main(int argc, char *argv[])
```

```
{
```

```
    pid_t pid;
```

做真实的自己，用良心做教育

```
pid=fork();
if(pid<0)
    perror("fork");
if(pid==0)
{
    while(1)
    {
        printf("this is son process\n");
        sleep(1);
    }
}
else
{
    while(1)
    {
        printf("this is father process\n");
        sleep(1);
    }
}
return 0;
}
```

例：02_fork_2.c 验证父子进程分别有各自独立的地址空间

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int var=10;
int main(int argc, char *argv[])
{
    pid_t pid;
    int num=9;
    pid=fork();
    if(pid<0)
    {
        perror("fork");
    }
    if(pid==0)
    {
        var++;
        num++;
        printf("in son process var=%d,num=%d\n", var, num);
    }
    else
```

```
{
    sleep(1);
    printf("in father process var=%d,num=%d\n", var, num);
}
printf("common code area\n");
return 0;
}
```

从 02_fork_2.c 程序可以看出，子进程对变量所做的改变并不影响父进程中该变量的值，说明父子进程各自拥有自己的地址空间。

一般来说，在 fork 之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用的调度算法。

如要求父子进程之间相互同步，则要求某种形式的进程间通信。

例：02_fork_3.c 验证子进程继承父进程的缓冲区

提示：

标准 I/O 提供三种类型的缓冲：

全缓冲：(大小不定)

在填满标准 I/O 缓冲区后，才进行实际的 I/O 操作。术语冲洗缓冲区的意思是进行标准 I/O 写操作。

行缓冲：(大小不定)

在遇到换行符时，标准 I/O 库执行 I/O 操作。这种情况允许我们一次输入一个字符，但只有写了一行后才进行实际的 I/O 操作。

不带缓冲

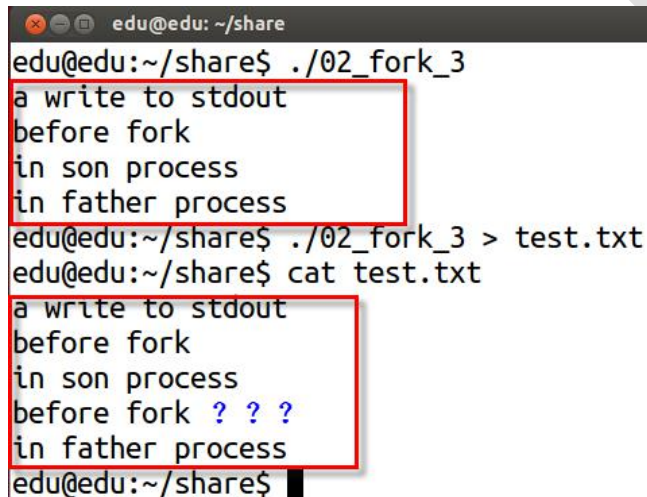
代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char *argv[])
{
    pid_t pid;
    int length = 0;
    char buf[] = "a write to stdout\n";

    length = write(1, buf, strlen(buf));
    if(length != strlen(buf))
    {
        printf("write error\n");
    }
    printf("before fork\n");
    pid=fork();
```

```
if(pid<0)
{
    perror("fork");
}
else if(pid==0)
{
    printf("in son process\n");
}
else
{
    sleep(1);
    printf("in father process\n");
}
return 0;
}
```

运行方式:



```
edu@edu: ~/share
edu@edu:~/share$ ./02_fork_3
a write to stdout
before fork
in son process
in father process
edu@edu:~/share$ ./02_fork_3 > test.txt
edu@edu:~/share$ cat test.txt
a write to stdout
before fork
in son process
before fork ???
in father process
edu@edu:~/share$
```

提示:

调用 fork 函数后，父进程打开的文件描述符都被复制到子进程中。在重定向父进程的标准输出时，子进程的标准输出也被重定向。

write 函数是系统调用，不带缓冲。

标准 I/O 库是带缓冲的，当以交互方式运行程序时，标准 I/O 库是行缓冲的，否则它是全缓冲的。

3.2.3 进程的挂起

进程在一定的时间内没有任何动作，称为进程的挂起

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

```
unsigned int sleep(unsigned int sec);
```

做真实的自己，用良心做教育

功能:

进程挂起指定的秒数，直到指定的时间用完或收到信号才解除挂起。

返回值:

若进程挂起到 `sec` 指定的时间则返回 0，若有信号中断则返回剩余秒数。

注意:

进程挂起指定的秒数后程序并不会立即执行，系统只是将此进程切换到就绪态。

3.2.4 进程的等待

父子进程有时需要简单的进程间同步，如父进程等待子进程的结束。

linux 下提供了以下两个等待函数 `wait()`、`waitpid()`。

需要包含头文件:

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

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

3.2.4.1 wait 函数

```
pid_t wait(int *status);
```

功能:

等待子进程终止，如果子进程终止了，此函数会回收子进程的资源。

调用 `wait` 函数的进程会挂起，直到它的一个子进程退出或收到一个不能被忽视的信号时才被唤醒。

若调用进程没有子进程或它的子进程已经结束，该函数立即返回。

参数:

函数返回时，参数 `status` 中包含子进程退出时的状态信息。子进程的退出信息在一个 `int` 中包含了多个字段，用宏定义可以取出其中的每个字段。

返回值:

如果执行成功则返回子进程的进程号。

出错返回-1，失败原因存于 `errno` 中。

取出子进程的退出信息

```
WIFEXITED(status)
```

如果子进程是正常终止的，取出的字段值非零。

```
WEXITSTATUS(status)
```

返回子进程的退出状态，退出状态保存在 `status` 变量的 8~16 位。在用此宏前应先用宏 `WIFEXITED` 判断子进程是否正常退出，正常退出才可以使用此宏。

注意:

此 `status` 是个 `wait` 的参数指向的整型变量。

例: 03_wait.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    pid_t pid;

    pid=fork();
    if(pid<0)
        perror("fork");
    if(pid==0)
    {
        int i = 0;
        for(i=0;i<5;i++)
        {
            printf("this is son process\n");
            sleep(1);
        }
        _exit(2);
    }
    else
    {
        int status = 0;

        wait(&status);
        if(WIFEXITED(status)!=0)
        {
            printf("son process return %d\n", WEXITSTATUS(status));
        }
        printf("this is father process\n");
    }
    return 0;
}
```

3.2.4.2 waitpid 函数

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

功能:

等待子进程终止，如果子进程终止了，此函数会回收子进程的资源。

返回值:

如果执行成功则返回子进程 ID。

出错返回-1，失败原因存于 `errno` 中。

参数 pid 的值有以下几种类型:

pid>0:

等待进程 ID 等于 pid 的子进程。

pid=0

等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，`waitpid` 不会等待它。

pid=-1:

等待任一子进程，此时 `waitpid` 和 `wait` 作用一样。

pid<-1:

等待指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

`status` 参数中包含子进程退出时的状态信息。

`options` 参数能进一步控制 `waitpid` 的操作:

0:

同 `wait`，阻塞父进程，等待子进程退出。

WNOHANG:

没有任何已经结束的子进程，则立即返回。

WUNTRACED

如果子进程暂停了则此函数马上返回，并且不予以理会子进程的结束状态。（跟踪调试，很少用到）

返回值:

成功:

返回状态改变了的子进程的进程号；如果设置了选项 `WNOHANG` 并且 pid 指定的进程存在则返回 0。

出错:

返回-1。当 pid 所指示的子进程不存在，或此进程存在，但不是调用进程的子进程，`waitpid` 就会出错返回，这时 `errno` 被设置为 `ECHILD`。

例: 03_waitpid.c

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

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

int main(int argc, char *argv[])
{
    pid_t pid;

    pid=fork();
    if(pid < 0)
        perror("fork");
    if(pid == 0)
    {
        int i = 0;
        for(i=0;i<5;i++)
        {
            printf("this is son process\n");
            sleep(1);
        }
        _exit(2);
    }
    else
    {
        waitpid(pid, NULL, 0);
        printf("this is father process\n");
    }
    return 0;
}
```

特殊进程

僵尸进程 (Zombie Process)

进程已运行结束，但进程的占用的资源未被回收，这样的进程称为僵尸进程。

子进程已运行结束，父进程未调用 wait 或 waitpid 函数回收子进程的资源是子进程变为僵尸进程的原因。

孤儿进程 (Orphan Process)

父进程运行结束，但子进程未运行结束的子进程。

守护进程 (精灵进程) (Daemon process)

守护进程是个特殊的孤儿进程，这种进程脱离终端，在后台运行。

做真实的自己，用良心做教育

3.2.5 进程的终止

在 linux 下可以通过以下方式结束正在运行的进程:

```
void exit(int value);  
void _exit(int value);
```

3.2.5.1 exit 函数

结束进程执行

```
#include <stdlib.h>
```

```
void exit(int value)
```

参数:

status: 返回给父进程的参数(低 8 位有效)。

3.2.5.2 _exit 函数

结束进程执行

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

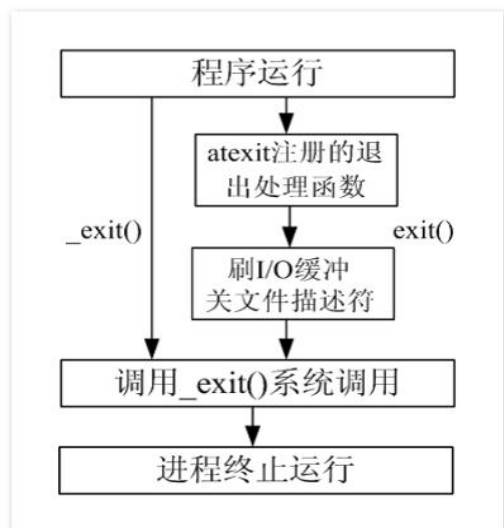
```
void _exit(int value)
```

参数:

status:返回给父进程的参数(低 8 位有效)。

exit 和 _exit 函数的区别:

exit 为库函数, 而 _exit 为系统调用



3.2.6 进程退出清理

进程在退出前可以用 `atexit` 函数注册退出处理函数。

```
#include <stdlib.h>
```

```
int atexit(void (*function)(void));
```

功能:

注册进程正常结束前调用的函数，进程退出执行注册函数。

参数:

function: 进程结束前，调用函数的入口地址。

一个进程中可以多次调用 `atexit` 函数注册清理函数，正常结束前调用函数的顺序和注册时的顺序相反。

例: 04_atexit.c

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

void clear_fun1(void)
{
    printf("perform clear fun1 \n");
}

void clear_fun2(void)
{
    printf("perform clear fun2 \n");
}
```

做真实的自己，用良心做教育

```
}  
  
void clear_fun3(void)  
{  
    printf("perform clear fun3 \n");  
}  
  
int main(int argc, char *argv[])  
{  
    atexit(clear_fun1);  
    atexit(clear_fun2);  
    atexit(clear_fun3);  
    printf("process exit 3 sec later!!!\n");  
    sleep(3);  
    return 0;  
}
```

3.2.7 进程的创建 -- vfork 函数

pid_t vfork(void)

功能:

vfork 函数和 fork 函数一样都是在已有的进程中创建一个新的进程，但它们创建的子进程是有区别的。

返回值:

创建子进程成功，则在子进程中返回 0，父进程中返回子进程 ID。出错则返回-1。

fork 和 vfork 函数的区别:

vfork 保证子进程先运行，在它调用 exec 或 exit 之后，父进程才可能被调度运行。

vfork 和 fork 一样都创建一个子进程，但它并不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用 exec(或 exit)，于是也就不访问该地址空间。

相反，在子进程中调用 exec 或 exit 之前，它在父进程的地址空间中运行，在 exec 之后子进程会有自己的进程空间。

例：05_vfork_1.c 验证 vfork 创建子进程先执行，父进程挂起

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
int main(int argc, char *argv[])  
{
```

```
pid_t pid;

pid = vfork();
if(pid<0)
    perror("vfork");
if(pid==0)
{
    int i = 0;
    for(i=0;i<5;i++)
    {
        printf("this is son process\n");
        sleep(1);
    }
    _exit(0);
}
else
{
    while(1)
    {
        printf("this is father process\n");
        sleep(1);
    }
}
return 0;
}
```

例：05_vfork_2.c 验证 vfork 创建子进程与父进程共用一个地址空间

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

int var = 10;
int main(int argc, char *argv[])
{
    pid_t pid;
    int num = 9;

    pid = vfork();
```

```
if(pid<0)
{
    perror("vfork");
}
if(pid == 0)
{
    var++;
    num++;
    printf("in son process var=%d,num=%d\n", var, num);
    _exit(0);
}
else
{
    printf("in father process var=%d,num=%d\n", var, num);
}
return 0;
}
```



3.2.8 进程的替换

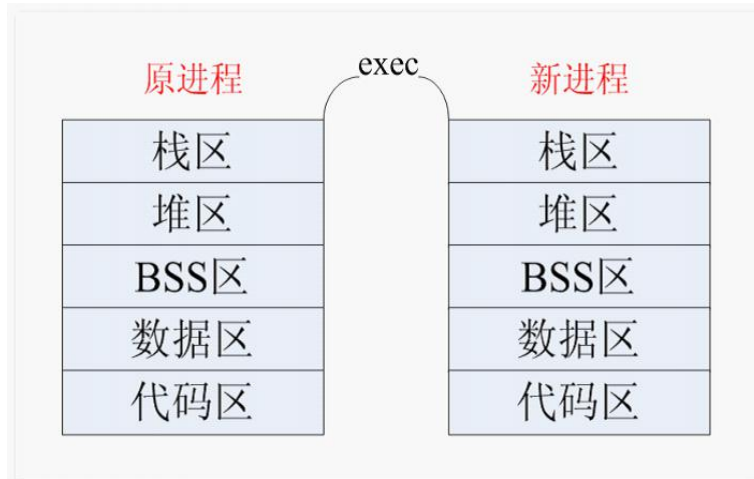
进程的替换

exec 函数族，是由六个 exec 函数组成的。

- 1、exec 函数族提供了六种在进程中启动另一个程序的方法。
- 2、exec 函数族可以根据指定的文件名或目录名找到可执行文件。
- 3、调用 exec 函数的进程并不创建新的进程，故调用 exec 前后，进程的进程号并不会改变，其执行的程序完全由新的程序替换，而新程序则从其 main 函数开始执行。

exec 函数族取代调用进程的数据段、代码段和堆栈段。

做真实的自己，用良心做教育



exec 函数族

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

```
int execl(const char *pathname,  
          const char *arg0, ...,  
          NULL);
```

```
int execlp(const char *filename,  
           const char *arg0, ...,  
           NULL);
```

```
int execlxe(const char *pathname,  
            const char *arg0, ..., NULL,  
            char *const envp[]);
```

```
int execlv(const char *pathname,  
           char *const argv[]);
```

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

```
int execlve(const char *pathname,  
            char *const argv[],  
            char *const envp[]);
```

六个 exec 函数中只有 execlve 是真正意义的系统调用(内核提供的接口)，其它函数都是在此基础上经过封装的库函数。

l(list):

参数地址列表，以空指针结尾。

做真实的自己，用良心做教育

参数地址列表

`char *arg0, char *arg1, ..., char *argn, NULL`

`v(vector):`

存有各参数地址的指针数组的地址。

使用时先构造一个指针数组，指针数组存各参数的地址，然后将该指针数组地址作为函数的参数。

`p(path)`

按 PATH 环境变量指定的目录搜索可执行文件。

以 p 结尾的 `exec` 函数取文件名做为参数。当指定 `filename` 作为参数时，若 `filename` 中包含 /，则将其视为路径名，并直接到指定的路径中执行程序。

`e(environment):`

存有环境变量字符串地址的指针数组的地址。`execle` 和 `execve` 改变的是 `exec` 启动的程序的环境变量（新的环境变量完全由 `environment` 指定），其他四个函数启动的程序则使用默认系统环境变量。

注意：

`exec` 函数族与一般的函数不同，`exec` 函数族中的函数执行成功后不会返回。只有调用失败了，它们才会返回 -1。失败后从原程序的调用点接着往下执行。

在平时的编程中，如果用到了 `exec` 函数族，一定要记得加错误判断语句。

例：06_test.c

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

int main(int argc, char *argv[])
{
    printf("USER=%s\n",getenv("USER"));
    printf("GONGSI=%s\n",getenv("GONGSI"));
    return 0;
}
```

例：06_exec1.c

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

int main(int argc, char *argv[])
{
    execl("/bin/ls", "ls", "-a", "-l", "-h", NULL);
    perror("execl");
    return 0;
}
```

例：06_execlp.c

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

int main(int argc, char *argv[])
{
    execlp("ls", "ls", "-a", "-l", "-h", NULL);
    perror("execlp");
    return 0;
}
```

例：06_execle.c

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

int main(int argc, char *argv[])
{
    char *env[]={"USER=ME", "GONGSI=QF", NULL};

    execl("./test", "test", NULL, env);
    perror("execle");
    return 0;
}
```

例：06_execv.c

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

int main(int argc, char *argv[])
{
    char *arg[]={"ls", "-a", "-l", "-h", NULL};

    execv("/bin/ls", arg);
    perror("execv");
    return 0;
}
```

例：06_execvp.c

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

int main(int argc, char *argv[])
{
    char *arg[]={"ls", "-a", "-l", "-h", NULL};

    execvp("ls", arg);
    perror("execvp");
    return 0;
}
```

例：06_execve.c

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

int main(int argc, char *argv[])
{
    char *arg[]={"test", NULL};
    char *env[]={"USER=ME", "GONGSI=QF", NULL};

    execve("./test", arg, env);
    perror("execve");
    return 0;
}
```

一个进程调用 exec 后，除了进程 ID，进程还保留了下列特征不变：

- 父进程号
- 进程组号
- 控制终端
- 根目录
- 当前工作目录
- 进程信号屏蔽集
- 未处理信号

...

做真实的自己，用良心做教育

3.2.9 system 函数

```
#include <stdlib.h>
```

```
int system(const char *command);
```

功能:

system 会调用 fork 函数产生子进程，子进程调用 exec 启动/bin/sh -c string 来执行参数 string 字符串所代表的命令，此命令执行完后返回原调用进程。

参数:

要执行的命令的字符串。

返回值:

如果 command 为 NULL，则 system() 函数返回非 0，一般为 1。

如果 system() 在调用/bin/sh 时失败则返回 127，其它失败原因返回-1。

注意:

system 调用成功后会返回执行 shell 命令后的返回值。其返回值可能为 1、127 也可能为-1，故最好应再检查 errno 来确认执行成功。

例: 07_system.c

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

int main(int argc, char *argv[])
{
    int status;

    status = system("ls -alh");
    if(WIFEXITED(status))
    {
        printf("the exit status is %d \n", status);
    }
    else
    {
        printf("abnormamal exit\n");
    }
    return 0;
}
```