

嵌入式系统工程师



进程

- 进程概述
- 进程控制

➤ 进程概述

➤ 进程的定义

➤ 进程的状态及转换

➤ 进程控制块

➤ 进程控制

➤ 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。
- 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。

➤ 进程概述

- 进程的定义

- 进程的状态及转换

- 进程控制块

➤ 进程控制

➤ 进程整个生命周期可以简单划分为三种状态:

➤ 就绪态:

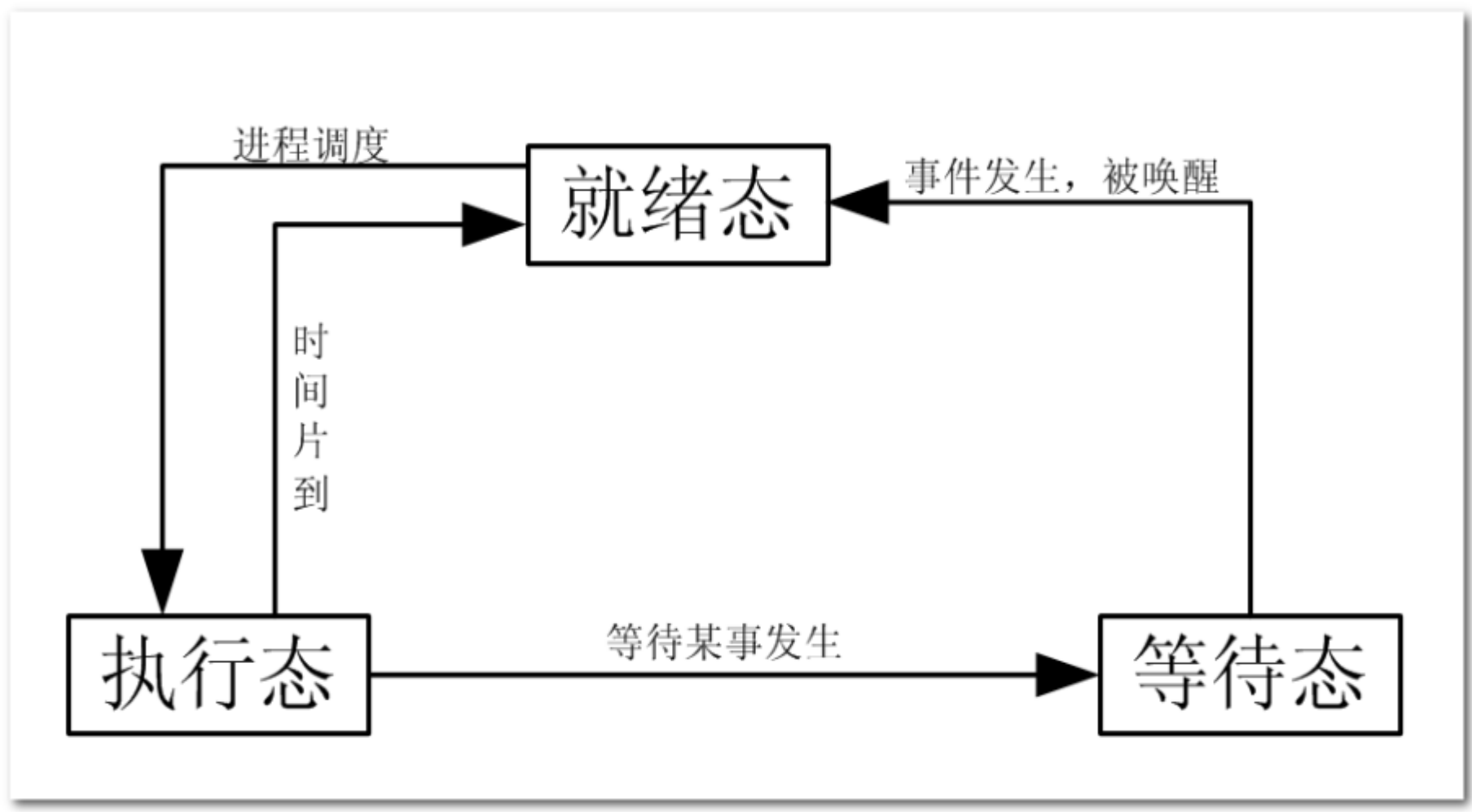
进程已经具备执行的一切条件, 正在等待分配CPU的处理时间。

➤ 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

➤ 进程概述

- 进程的定义

- 进程的状态及转换

- 进程控制块

➤ 进程控制

➤ 进程控制块 (PCB)

- OS是根据PCB来对并发执行的进程进行控制和管理
的。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。

➤ 进程控制块 (PCB)

➤ 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)

➤ ...

- 进程概述
- 进程控制
 - 进程号
 - 进程的创建
 - 进程的挂起
 - 进程的等待
 - 进程的终止
 - 进程的替换

- 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。
- 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)`
 - 返回值: 本进程号 (PID)
- `pid_t getppid(void)`
 - 返回值: 调用此函数的进程的父进程号 (PPID)
- `pid_t getpgid(pid_t pid)`
 - 参数: 0 当前 PGID, 否则为指定进程的 PGID
 - 返回值: 进程组号 (PGID)

例: 01_pid.c

➤ 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);

➤ fork函数：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。

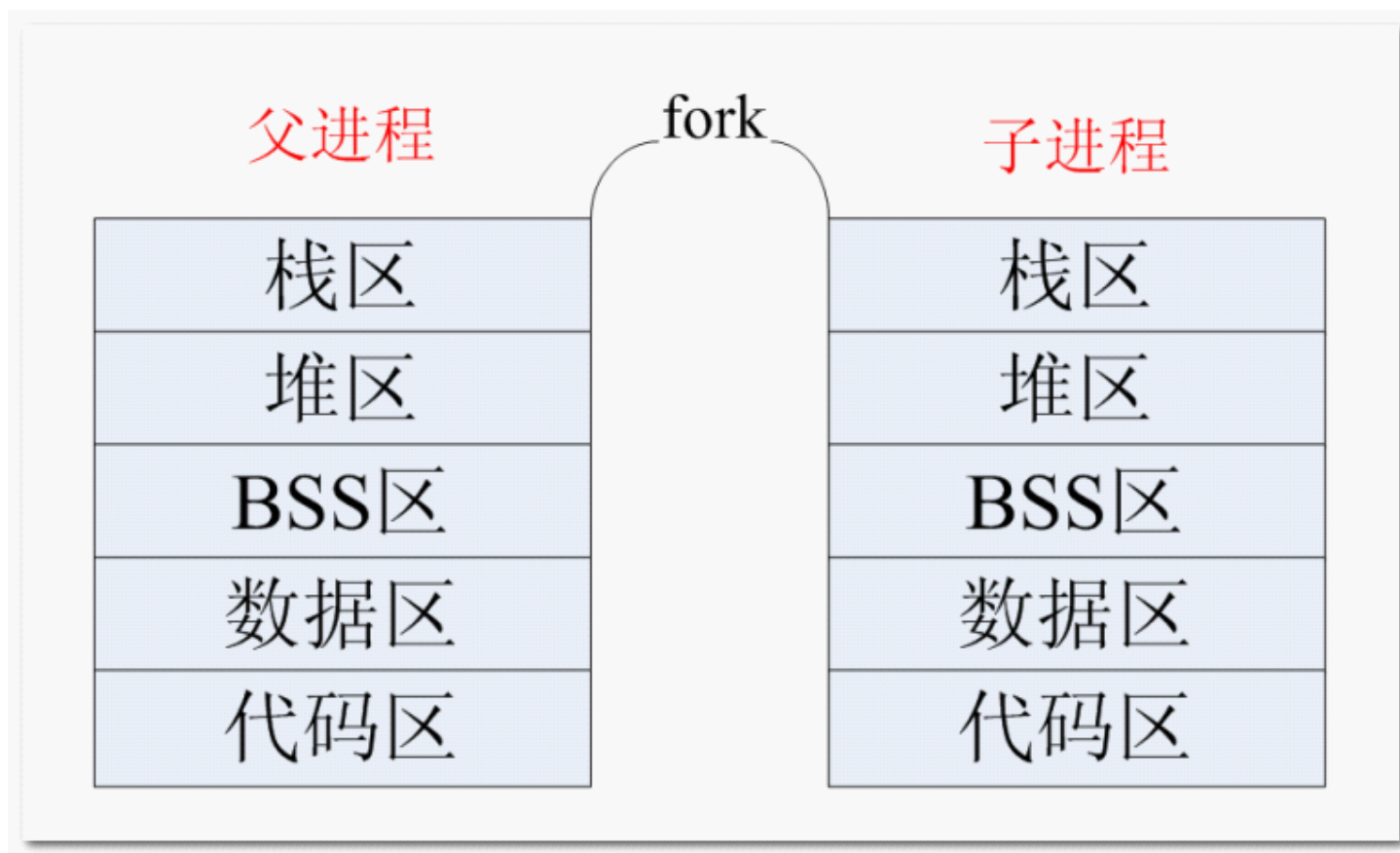
返回值：

- 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](#)

例: 02_fork_2.c

- 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

➤ 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。

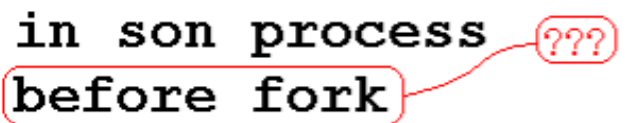
➤ 行缓冲: (大小不定)

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。进程的等待

➤ 不带缓冲

➤ 运行方法:

```
[root@localhost fork]# gcc 02_fork_3.c -o 02_fork_3
[root@localhost fork]# ./02_fork_3
a write to stdout
before fork
in son process
in father process
[root@localhost fork]# ./02_fork_3 > test
[root@localhost fork]# cat test
a write to stdout
before fork
in son process
before fork
in father process
[root@localhost fork]#
```



- 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。

➤ 僵尸进程 (Zombie Process)

父进程未运行结束，已运行结束的子进程。

➤ 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)

守护进程是个孤儿进程，它提供系统服务，常常在系统启动时启动，仅在系统关闭时才终止。这种进程脱离终端，在后台运行。

➤ vfork函数：创建一个新进程

```
pid_t vfork(void)
```

功能：

vmfork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。

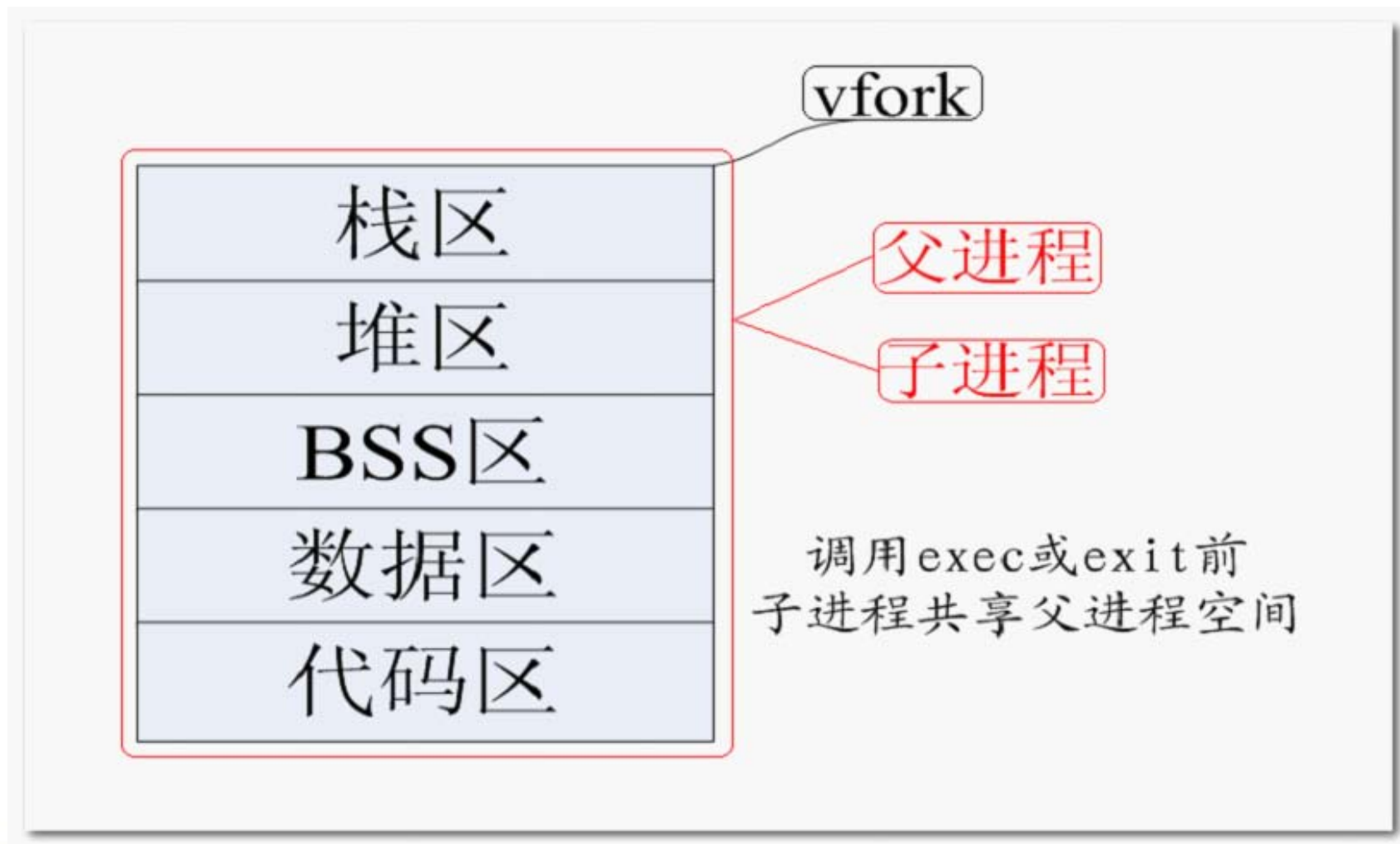
返回值：

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: 03_vfork_1.c 03_vfork_2.c



- 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);
```

- 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。

- 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>

➤ `pid_t wait(int *status);`

功能:

等待子进程改变状态, 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):

取出的字段值为子进程调用exit函数返回的值（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

例: 04_wait.c

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

功能:

等待子进程改变状态，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
如果子进程已暂停则马上返回, 且子进程的结束状态不予以理会。

➤ 返回值:

如果设置了选项WNOHANG，调用waitpid时若没有任何已经结束的子进程可收集，返回0；若有已经结束的子进程可收集，则返回子进程进程号。

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: [04_waitpid.c](#)

- 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);`

➤ 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>
```

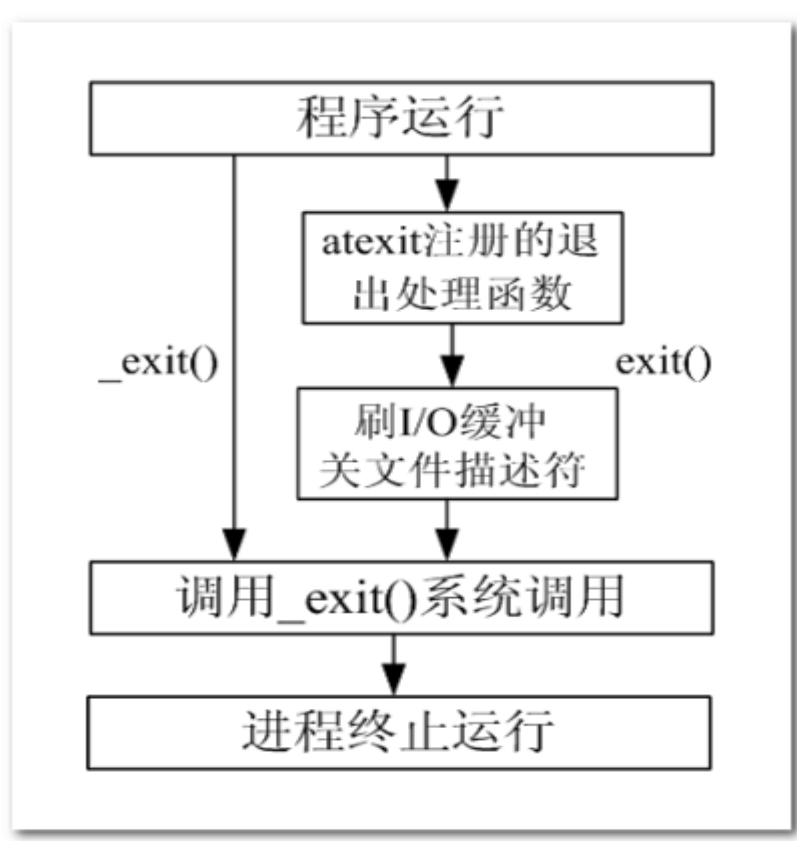
```
void _exit(int value)
```

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



- 进程在退出时可以用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。

参数:

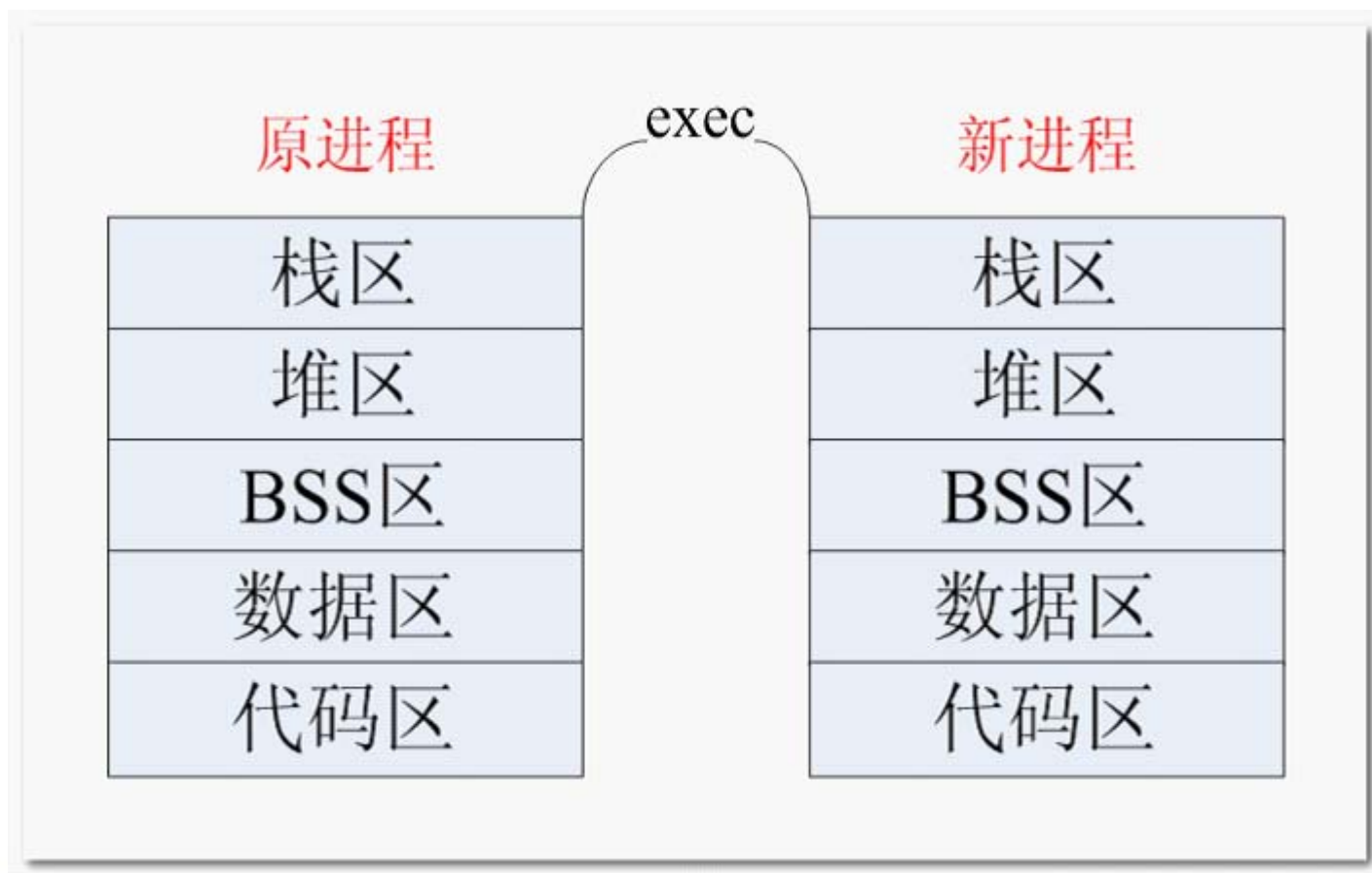
function: 进程结束前, 调用函数的入口地址。

- 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: 05_atexit.c

- 进程的替换
- exec函数族，是由六个exec函数组成的。
 - exec函数族提供了六种在进程中启动另一个程序的方法。
 - 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。
 - 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。



➤ 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 execlp(const char *pathname,
 const char *arg0, ..., NULL,
 char *const envp[]);`

➤ exec函数族

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

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

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

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

- 六个exec函数中只有execve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。
- 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。

- 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
 - 例: 06_exec1.c
 - 例: 06_exec1p.c
 - 例: 06_exec1e.c
 - 例: 06_execv.c
 - 例: 06_execvp.c
 - 例: 06_execve.c

- 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：
 - 进程号和父进程号
 - 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
 - ...

➤ #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](#)

➤ 练习

➤ 题目：编写shell命令解释器（外部命令）

➤ 外部命令：

在/bin/ 目录下能找到命令的可执行程序的可被
称为外部命令。

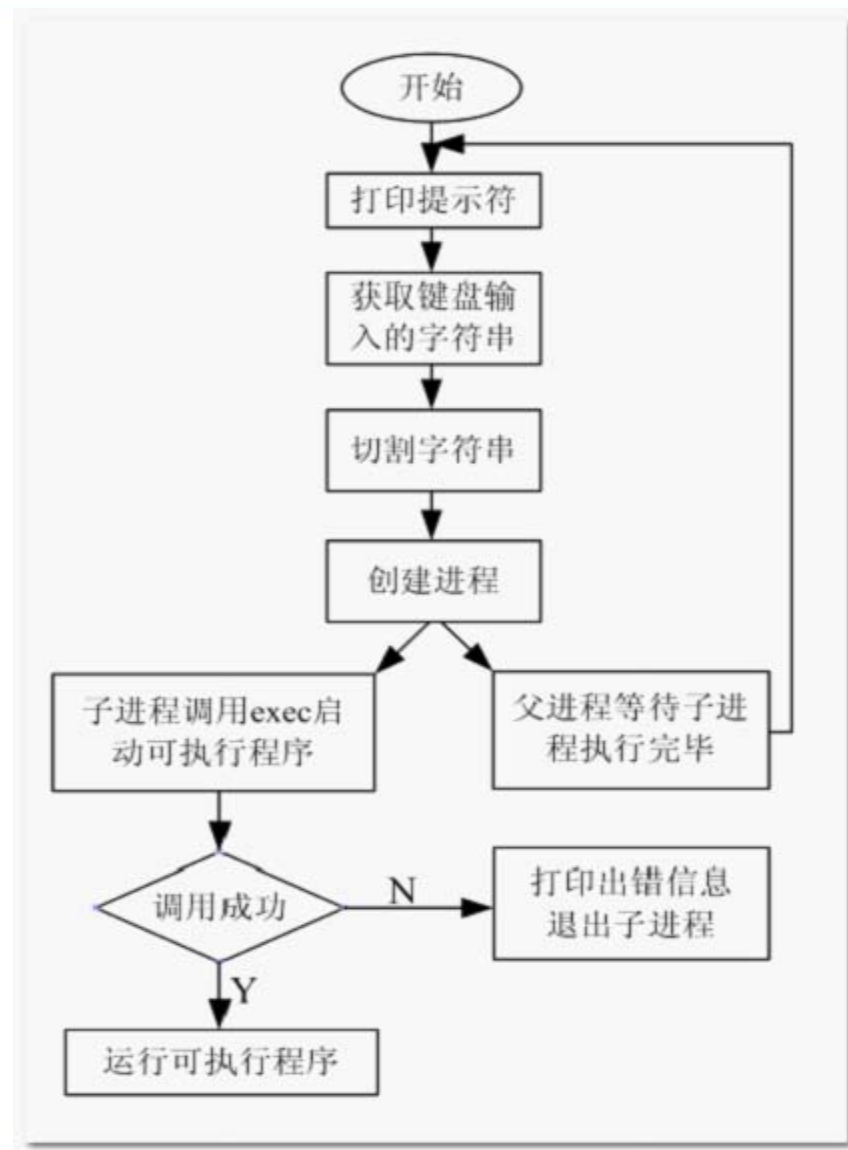
如：ls、pwd等，如：ls、pwd等，通过exec来执
行可执行程序实现命令功能。

➤ 内部命令：

在/bin/ 目录下找不到可执行程序的命令被称为
内部命令。

如：cd、exit、export等，shell程序内部通过调
用函数来实现命令功能（如shell通过调用chdir函数
来实现cd命令）。

➤ 程序流程图



- 提示：通过getenv函数可以得到用户名、主机名和当前路径。
 - `char *p1 = getenv("USER");`
 - `char *p3 = getenv("PWD");`
- 获取当前工作目录的绝对路径：`getcwd()`；详情参考Linux c函数.chm或系统自带的manual。



值得信赖的教育品牌

Tel: 400-705-9680, Email: edu@sunplusapp.com, BBS: bbs.sunplusedu.com

