APUE - 7

阅读目的：了解 UNIX 创建新进程、执行程序和进程终止相关的进程控制。
阅读时间：4 小时
阅读概况：第 8 章

第 8 章　进程控制

1. 进程标识

每个进程都有一个非负整数表示的唯一进程 ID。

`ID 0` 的进程通常是调度进程，又被称为交换进程（ seapper ）或系统进程。该进程是内核的一部分，它并不执行任何磁盘上的程序。

`ID 1` 的进程通常是 init 进程，在自举过程结束时由内核调用。init 通常读取与系统有关的初始化文件，并将系统引导到一个状态。init 进程绝不会终止。

常用的其他标识符函数：

```
#include <unistd.h>
//调用进程的进程 ID
pid_t getpid(void);
//调用进程的父进程 ID
pid_t getppid(void);
```

2. 函数 fork

一个现有的进程可以调用 fork 函数创建一个新进程。

```
#include <unistd.h>
pid_t fork(void);
```

由 fork 创建的新进程被称为子进程（ child process ）。

fork 函数被调用一次，但返回两次。子进程的返回值是 0，父进程的返回值是新建子进程的进程 ID。

子进程和父进程继续执行 fork 调用之后的指令。子进程是父进程的副本。

子进程获得父进程**数据空间、堆和栈的副本**。注意，这是子进程所拥有的副本。父进程和子进程并不共享这些存储空间部分。父进程和子进程**共享正文段**。

fork 有以下两种用法：

- 一个父进程希望复制自己，使父进程和子进程同时执行不同的代码段。

> 这在网络服务中是常见的，父进程等待客户端的服务请求。当这种请求到达时，父进程调用 fork，使子进程处理此请求。父进程则继续等待下一个服务请求。

- 一个进程要执行一个不同的程序。子进程 fork 返回后立即调用 exec。

3. 函数 exit

不管进程如何终止，最后都会执行内核的同一段代码。这段代码为相应进程关闭所有打开描述符，释放它所使用的存储器等。

对于正常终止和异常终止任意一种终止情形，我们都希望终止进程能通知其父进程它是如何终止的。

实现这一点的方法是：在正常终止情况，将其**退出状态（ exit status ）**作为参数传递给函数。在异常终止情况，内核（不是进程本身）产生一个指示其异常终止原因的**终止状态（ termination status ）**。

在任意一种情况下，该终止进程的父进程都能用 wait 或 waitpid 函数取得其终止状态。

如果父进程在 fork 出的子进程之前终止，该如何处理呢?

对于父进程已经终止的所有进程，它们的父进程都改变为 init 进程。我们程这些进程由 init 进程收养。

> 其过程大致是：在一个进程终止时，内核逐个检查所有活动进程，以判断它是否是正要终止进程的子进程，如果是，则该进程的父进程 ID 就更改为 1（ init 进程的 ID ）。这种处理方法保证了每个进程有一个父进程。

如果子进程在父进程之前终止，那么父进程又如何能在做相应检查时得到子进程的终止状态呢?

内核为每个终止子进程保存了一定量的信息，当终止进程的父进程调用 wait 或 waitpid 时，可以得到这些信息。这些信息至少包括进程ID、该进程的终止状态以及该进程使用的 CPU 时间总量。

在 UNIX 中，一个已经终止、但是其父进程尚未对其进行善后处理（获取终止子进程的有关信息、释放它仍占用的资源）的进程被称为**僵死进程（ zombie ）**

4. 函数 wait 和 waitpid

```
#include <sys/wait.h>
pid_t wait(int *staloc);

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

这两个函数区别如下：

- 在一个子进程终止前，wait 使其调用者阻塞，而 waitpid 有一选项，可使调用者不阻塞。
- waitpid 并不等待在其调用之后的第一个终止子进程，它有若干个选项，可以控制它所等待的进程。

如果子进程已经终止，并且是一个僵死进程，则 wait 立即返回并取得该子进程的状态；否则 wait 使其调用者阻塞，直到一个子进程终止。如调用者阻塞而且它有多个子进程，则在某一子进程终止时，wait 就立即返回。因为 wait 返回终止子进程的进程 ID，所以它总能了解是哪一个子进程终止了。

5. 函数 exec

当进程调用一种 exec 函数时，该进程执行的程序完全替换为新程序，而新程序则从其 main 函数开始执行。因为调用 exec 并不创建新进程，所以前后的进程 ID 并未改变。exec 只是用磁盘上的一个新程序替换了当前进程的正文段、数据段、堆段和栈段。

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

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

int execle(const char *pathname, const char *arg0, ... /* (char *)0, char *const envp[]
 */ );

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

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

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

int fexecve(int fd, char *const argv[], char *const envp[]);
```

这些函数之间的第一个区别是前四个函数取路径名作为参数，后
两个函数则取文件名作为参数，最后一个取文件描述符作为参数。

第二个区别于参数表的参数表的传递有关（ l 表示列表 list，v 表示矢量 vector ）。函数 execl、execlp 和 execle 要求将新程序的每个命令行参数都说明为一个单独的参数。这种参数表以空指针结尾。对于另外四个函数，则应先构造一个指向各参数的指针数组，然后将该数组地址作为这四个函数的参数。

最后一个区别与向新程序传递环境表相关。以 e 结尾的三个函数可以传递一个指向环境字符串指针数组的指

针。其他四个函数则使用调用进程中的 environ 变量为新程序复制现有的环境。

这七个函数中只有 execve 是内核的系统调用。另外六个只是库函数，它们最终都要调用该系统调用。