第五课　进程管理

一、基本概念

1. 进程与程序

1) 进程就是运行中的程序。一个运行着的程序，
可能有多个进程。进程在操作系统中执行特定的任务。

2) 程序是存储在磁盘上，
包含可执行机器指令和数据的静态实体。
进程或者任务是处于活动状态的计算机程序。

2. 进程的分类

1) 进程一般分为交互进程、批处理进程和守护进程三类。

2) 守护进程总是活跃的，一般是后台运行。
守护进程一般是由系统在开机时通过脚本自动激活启动，
或者由超级用户root来启动。

3. 查看进程

1) 简单形式

```
# ps
```

以简略方式显示当前用户有控制终端的进程信息。

2) BSD风格常用选项

```
# ps axu
```

a - 所有用户有控制终端的进程
x - 包括无控制终端的进程
u - 以详尽方式显示
w - 以更大列宽显示

3) SVR4风格常用选项

```
# ps -efl
```

-e或-A　　- 所有用户的进程
-a　　- 当前终端的进程
-u 用户名或用户ID - 特定用户的进程
-g 组名或组ID　　- 特定组的进程
-f　　- 按完整格式显示
-F　　- 按更完整格式显示
-l　　- 按长格式显示

4) 进程信息列表

USER/UID: 进程属主。

PID: 进程ID。

%CPU/C: CPU使用率。

%MEM: 内存使用率。

VSZ: 占用虚拟内存大小(KB)。

RSS: 占用物理内存大小(KB)。

TTY: 终端次设备号，“?”表示无控制终端，如后台进程。

STAT/S: 进程状态。可取如下值：

O - 就绪。等待被调度。
R - 运行。Linux下没有O状态，就绪状态也用R表示。
S - 可唤醒睡眠。系统中断，获得资源，收到信号，
    都可被唤醒，转入运行状态。
D - 不可唤醒睡眠。只能被wake_up系统调用唤醒。
T - 暂停。收到SIGSTOP信号转入暂停状态，
    收到SIGCONT信号转入运行状态。
W - 等待内存分页(2.6内核以后被废弃)。
X - 死亡。不可见。
Z - 僵尸。已停止运行，但其父进程尚未获取其状态。
< - 高优先级。
N - 低优先级。
L - 有被锁到内存中的分页。实时进程和定制IO。
s - 会话首进程。
l - 多线程化的进程。
+ - 在前台进程组中。

START/STIME: 进程开始时间。

TIME: 进程运行时间。

COMMAND/CMD: 进程指令。

F: 进程标志。可由下列值取和：

1 - 通过fork产生但是没有exec。
4 - 拥有超级用户特权。

PPID: 父进程ID。

NI: 进程nice值，-20到19，可通过系统调用或命令修改。

PRI: 进程优先级。

静态优先级 = 80 + nice，60到99，值越小优先级越高。
内核在静态优先级的基础上，
根据进程的交互性计算得到实际(动态)优先级，
以体现对IO消耗型进程的奖励，

和对处理器消耗型进程的惩罚。

ADDR: 内核进程的内存地址。普通进程显示“-”。

SZ: 占用虚拟内存页数。

WCHAN: 进程正在等待的内核函数或事件。

PSR: 进程被绑定到哪个处理器。

4. 父进程、子进程、孤儿进程和僵尸进程

```
内核进程(0)
  init(1)
    xinetd
      in.telnetd <- 用户登录
        login
          bash
            vi
```

1) 父进程启动子进程后，
   子进程在操作系统的调度下与其父进程同时运行。

2) 子进程先于父进程结束，
   子进程向父进程发送SIGCHLD(17)信号，
   父进程回收子进程的相关资源。

3) 父进程先于子进程结束，子进程成为孤儿进程，
   同时被init进程收养，即成为init进程的子进程。

4) 子进程先于父进程结束，
   但父进程没有回收子进程的相关资源，
   该子进程即成为僵尸进程。

5. 进程标识符(进程ID)

1) 每个进程都有一个以非负整数表示的唯一标识，
   即进程ID/PID。

2) 进程ID在任何时刻都是唯一的，但可以重用，
   当一个进程退出时，其进程ID就可以被其它进程使用。

3) 延迟重用。

```
a.out - 1000
a.out - 1010
a.out - 1020
...
```

范例：delay.c

二、getxxxid

```
#include <unistd.h>

getpid  - 获取进程ID
getppid - 获取父进程ID
getuid  - 获取实际用户ID
geteuid - 获取有效用户ID
getgid  - 获取实际组ID
getegid - 获取有效组ID

范例：id.c

假设a.out文件的属主和属组都是root。以其它用户身份登录并执行
$ a.out
输出
    进程ID：...
  父进程ID：...
实际用户ID：1000 - 实际用户ID取父进程(shell)的实际用户ID
有效用户ID：1000 - 有效用户ID取实际用户ID
  实际组ID：1000 - 实际组ID取父进程(shell)的实际组ID
  有效组ID：1000 - 有效组ID取实际组ID

执行
# ls -l a.out
输出
-rwxr-xr-x. 1 root root ...


为a.out的文件权限添加设置用户ID位和设置组ID位
# chmod u+s a.out
# chmod g+s a.out

执行
# ls -l a.out
输出
-rwsr-sr-x. 1 root root ...


以其它用户身份登录并执行
$ a.out
输出
    进程ID：...
  父进程ID：...
实际用户ID：1000 - 实际用户ID取父进程(shell)的实际用户ID
有效用户ID：0    - 有效用户ID取程序文件的属主ID
  实际组ID：1000 - 实际组ID取父进程(shell)的实际组ID
  有效组ID：0    - 有效组ID取程序文件的属组ID

进程的访问权限由其有效用户ID和有效组ID决定。
通过此方法可以使进程获得比登录用户更高的权限。
比如通过passwd命令修改登录口令。

执行
ls -l /etc/passwd
输出
-rw-r--r--. 1 root root 1648 Nov  9 14:05 /etc/passwd
```

该文件中存放所有用户的口令信息，仅root用户可写，
但事实上任何用户都可以修改自己的登录口令，
即任何用户都可以通过/usr/bin/passwd程序写该文件。

执行
ls -l /usr/bin/passwd
输出
-rwsr-xr-x. 1 root root 28816 Feb 8 2011 /usr/bin/passwd

该程序具有设置用户ID位，且其属主为root。
因此以任何用户登录系统，执行passwd命令所启动的进程，
其有效用户ID均为root，对/etc/passwd文件有写权限。

三、fork

```
#include <unistd.h>

pid_t fork (void);
```

1. 创建一个子进程，失败返回-1。

2. 调用一次，返回两次。
   分别在父子进程中返回子进程的PID和0。
   利用返回值的不同，
   可以分别为父子进程编写不同的处理分支。

范例：fork.c

3. 子进程是父进程的副本，
   子进程获得父进程数据段和堆栈段(包括I/O流缓冲区)的拷贝，
   但子进程共享父进程的代码段。

范例：mem.c、os.c、is.c

4. 函数调用后父子进程各自继续运行，
   其先后顺序不确定。
   某些实现可以保证子进程先被调度。

5. 函数调用后，
   父进程的文件描述符表(进程级)也会被复制到子进程中，
   二者共享同一个文件表(内核级)。

范例：ftab.c

6. 总进程数或实际用户ID所拥有的进程数，
   超过系统限制，该函数将失败。

7. 一个进程如果希望创建自己的副本并执行同一份代码，
   或希望与另一个程序并发地运行，都可以使用该函数。

8. 孤儿进程与僵尸进程。

范例：orphan.c、zombie.c

注意：fork之前的代码只有父进程执行，

fork之后的代码父子进程都有机会执行，
受代码逻辑的控制而进入不同分支。

四、vfork

```
#include <unistd.h>

pid_t vfork (void);
```

该函数的功能与fork基本相同，二者的区别：

1. 调用vfork创建子进程时并不复制父进程的地址空间，
   子进程可以通过exec函数族，
   直接启动另一个进程替换自身，
   进而提高进程创建的效率。

2. vfork调用之后，子进程先被调度。

五、进程的正常退出
---

1. 从main函数中return。

```
int main (...) {
    ...
    return x;
}
```

等价于：

```
int main (...) {
    ...
    exit (x);
}
```

2. 调用标准C语言的exit函数。

```
#include <stdlib.h>

void exit (int status);
```

1) 调用进程退出，
   其父进程调用wait/waitpid函数返回status的低8位。

2) 进程退出之前，
   先调用所有事先通过atexit/on_exit函数注册的函数，
   冲刷并关闭所有仍处于打开状态的标准I/O流，
   删除所有通过tmpfile函数创建的文件。

```
#include <stdlib.h>

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

function - 函数指针，
           指向进程退出前需要被调用的函数。

该函数既没有返回值也没有参数。

成功返回0，失败返回非零。

int on_exit (void (*function) (int, void*), void* arg);

function - 函数指针，
指向进程退出前需要被调用的函数。
该函数没有返回值但有两个参数：
第一参数来自exit函数的status参数，
第二个参数来自on_exit函数的arg参数。

arg - 任意指针，
将作为第二个参数被传递给function所指向的函数。

成功返回0，失败返回非零。

3) 用EXIT_SUCCESS/EXIT_FAILURE常量宏
(可能是0/1)作参数，调用exit()函数表示成功/失败，
提高平台兼容性。

4) 该函数不会返回。

5) 该函数的实现调用了_exit/_Exit函数。

3. 调用_exit/_Exit函数。

#include <unistd.h>

void _exit (int status);

1) 调用进程退出，
其父进程调用wait/waitpid函数返回status的低8位。

2) 进程退出之前，
先关闭所有仍处于打开状态的文件描述符，
将其所有子进程托付给init进程(PID为1的进程)收养，
向父进程递送SIGCHILD信号。

3) 该函数不会返回。

4) 该函数有一个完全等价的标准C版本：

#include <stdlib.h>

void _Exit (int status);

4. 进程的最后一个线程执行了返回语句。

5. 进程的最后一个线程调用pthread_exit函数。

范例：exit.c

六、进程的异常终止
---

1. 调用abort函数，产生SIGABRT信号。

2. 进程接收到某些信号。

3. 最后一个线程对“取消”请求做出响应。

七、wait/waitpid

等待子进程终止并获取其终止状态。

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

pid_t wait (int* status);

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

成功返回终止子进程的PID，失败返回-1。

1. 当一个进程正常或异常终止时，
   内核向其父进程发送SIGCHLD信号。
   父进程可以忽略该信号，
   或者提供一个针对该信号的信号处理函数，默认为忽略。

2. 父进程调用wait函数：

1) 若所有子进程都在运行，则阻塞。

2) 若有一个子进程已终止，
   则返回该子进程的PID和终止状态(通过status参数)。

3) 若没有需要等待子进程，则返回失败，errno为ECHILD。

3. 在任何一个子进程终止前，wait函数只能阻塞调用进程，
   而waitpid函数可以有更多选择。

4. 如果有一个子进程在wait函数被调用之前，
   已经终止并处于僵尸状态，wait函数会立即返回，
   并取得该子进程的终止状态。

5. 子进程的终止状态通过输出参数status返回给调用者，
   若不关心终止状态，可将此参数置空。

6. 子进程的终止状态可借助
   sys/wait.h中定义的参数宏查看：

WIFEXITED(): 子进程是否正常终止，
是则通过WEXITSTATUS()宏，
获取子进程调用exit/_exit/_Exit函数，
所传递参数的低8位。
因此传给exit/_exit/_Exit函数的参数最好不要超过255。

WIFSIGNALED(): 子进程是否异常终止，
是则通过WTERMSIG()宏获取终止子进程的信号。

WIFSTOPPED(): 子进程是否处于暂停，
是则通过WSTOPSIG()宏获取暂停子进程的信号。

WIFCONTINUED(): 子进程是否在暂停之后继续运行

范例：wait.c、loop.c

7. 如果同时存在多个子进程，又需要等待特定的子进程，
   可使用waitpid函数，其pid参数：

 -1 - 等待任一子进程，此时与wait函数等价。

> 0 - 等待由该参数所标识的特定子进程。

  0 - 等待其组ID等于调用进程组ID的任一子进程，
      即等待与调用进程同进程组的任一子进程。

<-1 - 等待其组ID等于该参数绝对值的任一子进程，
      即等待隶属于特定进程组内的任一子进程。

范例：waitpid.c

8. waitpid函数的options参数可取0(忽略)或以下值的位或：

WNOHANG   - 非阻塞模式，
             若没有可用的子进程状态，则返回0。

WUNTRACED - 若支持作业控制，且子进程处于暂停态，
             则返回其状态。

WCONTINUED - 若支持作业控制，且子进程暂停后继续，
             则返回其状态。

范例：nohang.c

八、exec
----

1. exec函数会用新进程完全替代调用进程，
   并开始从main函数执行。

2. exec函数并非创建子进程，新进程取调用进程的PID。

3. exec函数所创建的新进程，
   完全取代调用进程的代码段、数据段和堆栈段。

4. exec函数若执行成功，则不会返回，否则返回-1。

5. exec函数包括六种形式：

```
#include <unistd.h>

int execl  (
    const char* path,
    const char* arg, ...
);
```

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

int execle (
    const char* path,
    const char* arg,
    ...,
    char* const envp[]
);

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

int execlp (
    const char* file,
    const char* arg,
    ...
);

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

l: 新程序的命令参数以单独字符串指针的形式传入
   (const char* arg, ...)，参数表以空指针结束。

v: 新程序的命令参数以字符串指针数组的形式传入
   (char* const argv[])，数组以空指针结束。

e: 新程序的环境变量以字符串指针数组的形式传入
   (char* const envp[])，数组以空指针结束，
   无e则从调用进程的environ变量中复制。

p: 若第一个参数中不包含“/”，则将其视为文件名，
   根据PATH环境变量搜索该文件。

范例：argenv.c、exec.c

九、system

```
#include <stdlib.h>

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

1. 标准C函数。执行command，
   成功返回command对应进程的终止状态，失败返回-1。

2. 若command取NULL，返回非零表示shell可用，
   返回0表示shell不可用。

3. 该函数的实现，
   调用了fork、exec和waitpid等函数，
   其返回值：

1) 如果调用fork或waitpid函数出错，则返回-1。

2) 如果调用exec函数出错，则在子进程中执行exit(127)。

3) 如果都成功，则返回command对应进程的终止状态
   (由waitpid的status输出参数获得)。

4. 使用system函数而不用fork+exec的好处是，
   system函数针对各种错误和信号都做了必要的处理。

范例：system.c、fexe.c