相关概念

终端

■ 在 UNIX 系统中，用户通过终端登录系统后得到一个 shell 进程，这个终端成为 shell 进程的控制终端（Controlling Terminal），进程中，控制终端是保存在 PCB 中的信息，而 fork() 会复制 PCB 中的信息，因此`由 shell 进程启动的其它进程的控制终端也是这个终端`。
■ 默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上。
■ 在控制终端输入一些特殊的控制键可以给前台进程发信号，例如 Ctrl + C 会产生 SIGINT 信号，Ctrl + \ 会产生 SIGQUIT 信号。

进程组

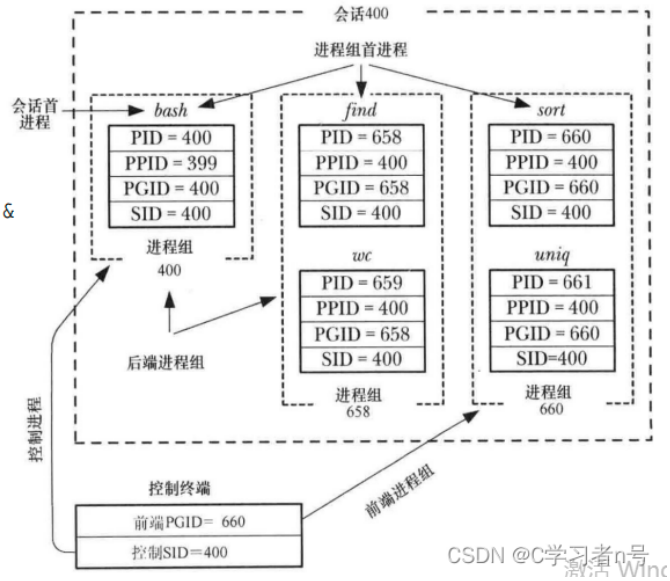
■ 进程组和会话在进程之间形成了一种两级层次关系：`进程组是一组相关进程的集合，会话是一组相关进程组的集合`。进程组和会话是为支持 shell 作业控制而定义的抽象概念，用户通过 shell 能够交互式地在前台或后台运行命令。
■ 进行组由一个或多个共享同一进程组标识符（PGID）的进程组成。一个进程组拥有一个进程组首进程，该进程是创建该组的进程，其进程 ID 为该进程组的 ID，新进程会继承其父进程所属的进程组 ID。
■ 进程组拥有一个生命周期，其开始时间为首进程创建组的时刻，结束时间为最后一个成员进程退出组的时刻。一个进程可能会因为终止而退出进程组，也可能会因为加入了另外一个进程组而退出进程组。`进程组首进程无需是最后一个离开进程组的成员`。

会话

■ 会话是一组进程组的集合。会话首进程是创建该新会话的进程，其进程 ID 会成为会话 ID。新进程会继承其父进程的会话 ID。
■ 一个会话中的所有进程共享单个控制终端。控制终端会在会话首进程首次打开一个终端设备时被建立。一个终端最多可能会成为一个会话的控制终端。
■ 在任一时刻，会话中的其中一个进程组会成为终端的前台进程组，其他进程组会成为后台进程组。只有前台进程组中的进程才能从控制终端中读取输入。当用户在控制终端中输入终端字符生成信号后，该信号会被发送到前台进程组中的所有成员。
■ 当控制终端的连接建立起来之后，会话首进程会成为该终端的控制进程。

三者关系

- `find / 2 > /dev/null | wc -l &`
- `sort < longlist | uniq -c`



进程组和会话的相关函数

- `pid_t getpgrp(void);`
- `pid_t getpgid(pid_t pid);`
- `int setpgid(pid_t pid, pid_t pgid);`
- `pid_t getsid(pid_t pid);`
- `pid_t setsid(void);`

守护进程

■ 守护进程（Daemon Process），也就是通常说的 Daemon 进程（精灵进程），是Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以 d 结尾的名字。

■ 守护进程具备下列特征：

1、 生命周期很长，守护进程会在系统启动的时候被创建并一直运行直至系统被关闭。

2、 它在后台运行并且不拥有控制终端。没有控制终端确保了内核永远不会为守护进程自动生成任何控制信号以及终端相关的信号（如 SIGINT、SIGQUIT）。

■ Linux 的大多数服务器就是用守护进程实现的。比如，Internet 服务器 inetd，Web 服务器 httpd 等。

创建守护进程的步骤

■ 执行一个 fork()，之后父进程退出，子进程继续执行。

■ 子进程调用 setsid() 开启一个新会话。

■ 清除进程的 umask 以确保当守护进程创建文件和目录时拥有所需的权限。

■ 修改进程的当前工作目录，通常会改为根目录（/）。

■ 关闭守护进程从其父进程继承而来的所有打开着的文件描述符。

■ 在关闭了文件描述符0、1、2之后，守护进程通常会打开/dev/null 并使用dup2() 使所有这些描述符指向这个设备。指向/dev/null之后会自动丢弃传过来的数据。

■ 核心业务逻辑

示例

实现每隔两秒向文件输出当前时间。

```c
/*
    写一个守护进程，每隔2s获取一下系统时间，将这个时间写入到磁盘文件中。
*/

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/time.h>
#include <signal.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

void work(int num)
{
    // 捕捉到信号之后，获取系统时间，写入磁盘文件
    time_t tm = time(NULL);
    struct tm *loc = localtime(&tm);

    char *str = asctime(loc); //以某种格式返回当前时间信息
    //该文件会在新的工作目录中出现
    int fd = open("time.txt", O_RDWR | O_CREAT | O_APPEND, 0664);
    write(fd, str, strlen(str));
    close(fd);
}

int main()
{

    // 1.创建子进程，退出父进程
    pid_t pid = fork();

    if (pid > 0)
    {
        exit(0); //父进程结束，一下的都是子进程代码
    }

    // 2.将子进程重新创建一个会话
    setsid();

    // 3.设置掩码
    umask(022);

    // 4.更改工作目录，通常是根目录/，但是可能会涉及到一些权限问题
    chdir("/home/kiko/lesson/myprocess");

    // 5. 关闭、重定向文件描述符
    int fd = open("/dev/null", O_RDWR);
    dup2(fd, STDIN_FILENO);
    dup2(fd, STDOUT_FILENO);
    dup2(fd, STDERR_FILENO);
```

```
    // 6.业务逻辑

    // 捕捉定时信号
    struct sigaction act;
    act.sa_flags = 0;
    act.sa_handler = work;
    sigemptyset(&act.sa_mask);
    sigaction(SIGALRM, &act, NULL); //捕捉定时信号

    struct itimerval val;
    val.it_value.tv_sec = 2;
    val.it_value.tv_usec = 0;
    val.it_interval.tv_sec = 2;
    val.it_interval.tv_usec = 0;

    // 创建定时器
    setitimer(ITIMER_REAL, &val, NULL); //发送定时信号

    // 不让进程结束
    while (1)
    {
        sleep(10);
    }

    return 0;
}
```