第6章 进程控制程序设计

6.1 进程控制概述

6.2 进程控制编程

6.1 进程控制概述

- 什么是进程?
- 进程的特点
- 进程控制块和标示符
- 进程状态
- 进程同步、互斥
- 进程调度·
- 进程地址空间
- Linux进程的管理

（1）什么是进程？

- 进程的概念首先是在60年代初期由MIT的Multics系统和IBM的TSS/360系统引入的。在40多年的发展中，人们对进程有过各种各样的定义。现列举较为著名的几种。
  - 进程是一个独立的可调度的活动（E. Cohen，D. Jofferson）
  - 进程是一个抽象实体，当它执行某个任务时，要分配和释放各种资源（P. Denning）
  - 进程是可以并行执行的计算单位。（S. E. Madnick，J. T. Donovan）
- 以上进程的概念都不相同，但其本质是一样的。它指出了进程是一个程序的一次执行的过程，同时也是资源分配的最小单元。它和程序是有本质区别的，程序是静态的，它是一些保存在磁盘上的指令的有序集合，没有任何执行的概念；而进程是一个动态的概念，它是程序执行的过程，包括了动态创建、调度和消亡的整个过程。它是程序执行和资源管理的最小单位。
- **进程：进程是一个具有一定独立功能的程序的一次运行活动。**

（2）进程的特点

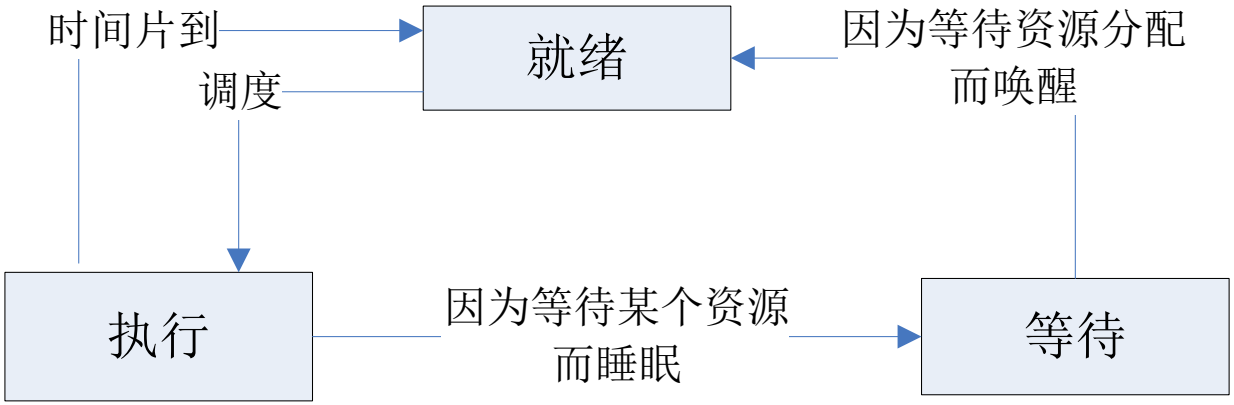
- 动态性
  - 动态性是进程最基本的特征
- 并发性
  - 指多个进程实体，同时存在于内存中，能在同一时间内同时运行，并发是进程的重要特征，也是OS的重要特征。
- 独立性
  - 进程实体是一个能独立运行的基本单位，同时也是获得资源，独立调度的基本单位
- 异步性
  - 进程按各自独立的，不可预知的速度向前执行

（3）进程控制块和标识符

- 进程是Linux系统的基本调度和管理资源的单位，它是通过进程控制块来描述的。进程控制块包含了进程的描述信息、控制信息以及资源信息，它是进程的一个静态描述。在Linux中，进程控制块是一个task_struct结构(include/linux/sched.h)。
- 在Linux中最主要的进程标识有进程号（PID，Process Idenity Number）和它的父进程号（PPID，parent process ID）。其中PID惟一地标识一个进程。PID和PPID都是非零的正整数。
- 在Linux中获得当前进程的PID和PPID的系统调用函数为getpid()和getppid()，通常程序获得当前进程的PID和PPID之后，可以将其写入日志文件以做备份。
- 另外，进程标识还有用户和用户组标识、进程时间、资源利用情况等

（4）进程的状态

- 进程是程序的执行过程，根据它的生命周期可以划分成3种状态。
  - 执行态：该进程正在运行，即进程正在占用CPU
  - 就绪态：进程已经具备执行的一切条件，正在等待分配CPU的处理时间片。
  - 等待（阻塞）态：进程不能使用CPU，若等待事件发生（等待的资源分配到）则可将其唤醒。



(5) 进程同步、互斥

- 一组并发进程按一定的顺序执行的过程称为**进程间的同步**。具有同步关系的一组并发进程称为合作进程，合作进程间互相发送的信号称为消息或事件。
- **进程互斥**是指当有若干进程都要使用某一共享资源时，任何时刻最多允许一个进程使用，其他要使用该资源的进程必须等待，直到占用该资源者释放了该资源为止。
- 临界资源：操作系统中将一次只允许一个进程访问的资源称为临界资源。进程中访问临界资源的那段程序代码称为临界区。为实现对临界资源的互斥访问，应保证诸进程互斥的进入各自的临界区。
- 死锁：多个进程因竞争资源而形成一种僵局，若无外力作用，这些进程都将永远不能再向前推进。

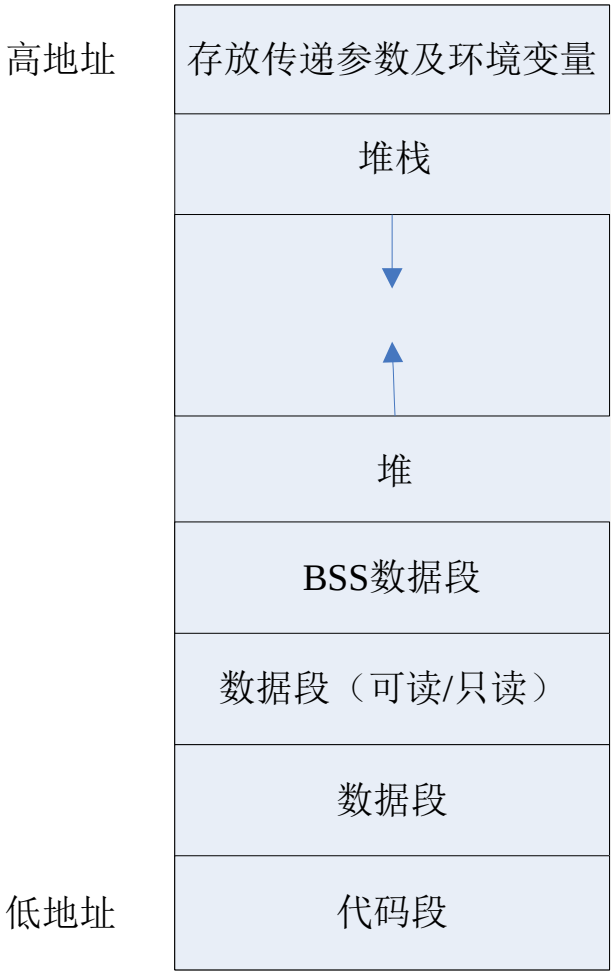
（6）进程调度

- 概念：按一定的算法，从一组待运行的进程中选出一个来占用CPU运行。
- 调度方式：
  - 抢占式
  - 非抢占式
- 调度算法
  - 先来先服务调度算法
  - 短进程优先调度算法
  - 高优先级优先调度算法
  - 时间片轮转法

（7）Linux下进程地址空间

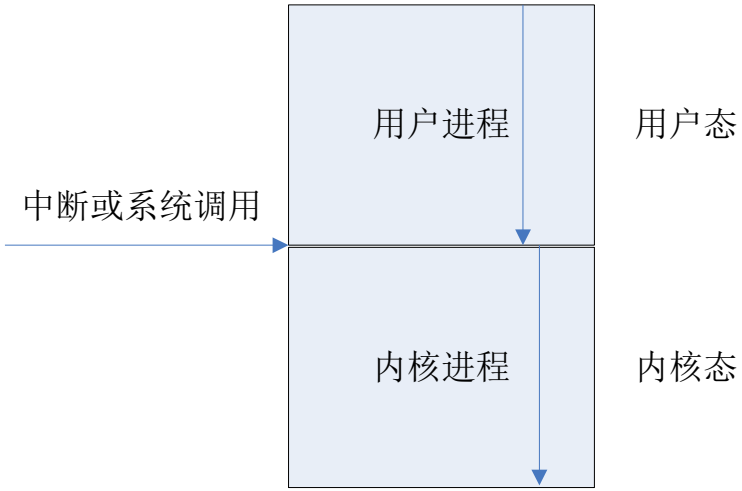
- Linux系统是一个多进程的系统，它的进程之间具有并行性、互不干扰等特点。也就是说，每个进程都是一个独立的运行单位，拥有各自的权利和责任。其中，各个进程都运行在独立的虚拟地址空间，因此，即使一个进程发生异常，它也不会影响到系统中的其他进程。
- Linux中的进程包含3个段，分别为“数据段”、“代码段”和“堆栈段”。
  - “数据段”存放的是全局变量、常数以及动态数据分配的数据空间，根据存放的数据，数据段又可以分成普通数据段（包括可读可写/只读数据段，存放静态初始化的全局变量或常量）、BSS数据段（存放未初始化的全局变量）以及堆（存放动态分配的数据）。
  - “代码段”存放的是程序代码的数据。
  - “堆栈段”存放的是子程序的返回地址、子程序的参数以及程序的局部变量等。

（7）Linux下进程地址空间



（8）Linux下进程的模式和类型

- 进程的执行模式分为：用户模式和内核模式
- 用户程序、应用程序或者内核之外的系统程序都运行在用户模式下；
- 如果用户模式下的程序执行系统调用或者发生中断事件，那么就要运行操作系统(即核心)程序，进程模式变为内核模式。



（9）Linux下的进程管理

- 启动进程
  - 手工启动
  - 调度启动
- 进程调度常用命令

ps	**查看系统中的进程**
top	**动态的现实系统中的进程**
nice	**按用户的指定的优先级运行**
renice	**改变正在运行进程的优先级**
kill	**终止进程**
bg	**将挂起的进程放到后台执行**

6.2 Linux进程控制编程

- 获取进程标识符
- 进程的创建：fork、vfork
- 进程的终止：exit、_exit
- 进程的等待：wait、waitpid
- exec函数族
- 守护进程

（1）获取进程标识

- 获取进程PID、PPID
  - #include <sys/types.h>
  - #include <unistd.h>
- pid_t getpid(void)
  获取本进程ID。
- pid_t getppid()
  获取父进程ID。

获取ID举例：

- getpid.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
  printf(“PID = %d\n”,getpid());
  printf((“PPID = %d\n”,getppid());
  return 0;
}
```

（2）进程创建-fork

- 在Linux中创建一个新进程的方法是使用fork()函数。
- fork()函数用于从已存在的进程中创建一个新进程。新进程称为子进程，而原进程称为父进程。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，包括进程上下文、代码段、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组号、当前工作目录、根目录、资源限制和控制终端等，而子进程所独有的只有它的进程号、资源使用和计时器等。
- 因为子进程几乎是父进程的完全复制，所以父子两个进程会运行同一个程序。因此需要用一种方式来区分它们，并使它们照此运行，否则，这两个进程不可能做不同的事。

（2）进程创建-fork

- 实际上是在父进程中执行fork()函数时，父进程会复制出一个子进程，而且父子进程的代码从fork()函数的返回开始分别在两个地址空间中同时运行。从而两个进程分别获得其所属fork()的返回值，其中在父进程中的返回值是子进程的进程号，而在子进程中返回0。因此，可以通过返回值来判定该进程是父进程还是子进程。
- 同时可以看出，使用fork()函数的代价是很大的，它复制了父进程中的代码段、数据段和堆栈段里的大部分内容，使得fork()函数的系统开销比较大，而且执行速度也不是很快。

（2）进程创建-fork

#include<sys/types.h>

#include<unistd.h>

pid_t fork(void)

创建子进程：

- fork的奇妙之处在于它被调用一次，却返回两次，它可能有三种不同的返回值
- 在父进程中，fork返回新创建的子进程的PID
- 在子进程中，fork返回0
- 如果出现错误，fork返回一个负值（-1）

（2）进程创建-fork

实例：fork1.c

```
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t pid;
   //此时仅有一个进程
   pid =fork();
   //此时有两个进程同时运行
   if(pid < 0)
         printf(“error in fork”);
   else if(pid == 0)
         printf(“I am the child process, ID is %d\n”,getpid());
   else
         printf(“I am the parent process, ID is %d\n”,getpid());
}
```

（2）进程创建-fork

- 在pid=fork()之前，只有一个进程在执行，但在这条语句执行之后，就变成两个进程在执行了，这两个进程的代码段，将要执行的下一条语句都是if(pid<0)。两个进程中，原来就存在的那个进程被称作“父进程”，新出现的那个进程被称作“子进程”，父子进程的区别在于进程标识符（PID）不同。

（2）进程创建-fork

- 思考下面程序运行结果？

```
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t pid;
   int count=0;
   pid =fork();

   count++;
   printf("count = %d\n",count);

   return 0;
}
```

输出：
count = 1
count = 1

（2）进程创建-fork

- count++被父进程、子进程一共执行了两次，为什么count的第二次输出不等于2?
- 子进程的数据空间、堆栈空间都会从父进程得到一个拷贝，而不是共享。在子进程中对count进行加1的操作，并没有影响到父进程中的count值，父进程中的count值任然为0。

（2）进程创建-fork

- fork有下面两种用法：
  - 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。这在网络服务进程中比较常见，父进程等待客户端的服务请求，当请求到达时，父进程调用fork，使子进程去处理该请求，父进程则继续等待下一个服务请求到达。
  - 一个进程要执行一个不同的程序。这对shell是常见的情况，这种情况下，子进程从fork返回后立即调用exec执行。

（2）进程创建-vfork

#include <sys/types.h>

#include <unistd.h>

pid_t vfork()

功能：创建子进程

目的：该子进程的目的是exec一个新的程序

（2）进程创建-vfork

- fork 与 vfork的区别：
  - fork：子进程拷贝父进程的数据段、堆栈段
  - vfork：子进程与父进程共享数据段

  - fork：父、子进程的执行顺序不确定
  - vfork：保证子进程先运行，在它调用了exec或exit之后，父进程运行(如果在调用这两个函数之前子进程依赖父进程的进一步动作，则会导致死锁)

vfork举例：

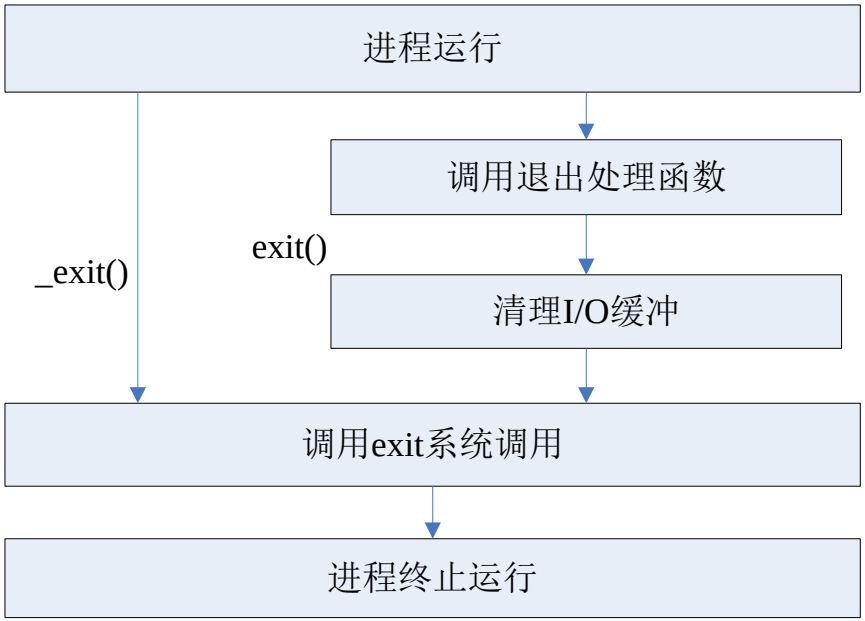
```
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t pid;
   int count=0;
   pid =vfork();

   count++;
   printf("count = %d\n",count);

   return 0;
}
```

（3）进程终止exit()和_exit()

- exit()和_exit()函数都是用来终止进程的。
- _exit()函数的作用是：直接使进程停止运行，清除其使用的内存空间，并清除其在内核中的各种数据结构；exit()函数则在这些基础上做了一些包装，在执行退出之前加了若干道工序。exit()函数与_exit()函数最大的区别就在于exit()函数在调用exit系统之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是图中的“清理I/O缓冲”一项。
- 由于在Linux的标准函数库中，有一种被称作“缓冲I/O（buffered I/O）”操作，其特征就是对应每一个打开的文件，在内存中都有一片缓冲区。每次读文件时，会连续读出若干条记录，这样在下次读文件时就可以直接从内存的缓冲区中读取；同样，每次写文件的时候，也仅仅是写入内存中的缓冲区，等满足了一定的条件（如达到一定数量或遇到特定字符等），再将缓冲区中的内容一次性写入文件。

（3） exit()和_exit()

- 这种技术大大增加了文件读写的速度，但也为编程带来了一些麻烦。比如有些数据，认为已经被写入到文件中，实际上因为没有满足特定的条件，它们还只是被保存在缓冲区内，这时用_exit()函数直接将进程关闭，缓冲区中的数据就会丢失。因此，若想保证数据的完整性，就一定要使用exit()函数。



（3）exit()和_exit()

- 原型：void _exit(int status);
  - status：该参数指定进程退出时的返回值，该返回值可以在shell中通过“echo $?”命令查看，也可以通过system函数的返回值取得，还可以在父进程中通过调用wait函数获得。
  - 通常进程返回0表示正常退出（如exit(0)），返回非零表示异常退出（如exit(1)/exit(-1)）。

（3）exit()和_exit()

例程:exit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
   printf(“Using exit\n”);
   printf(“This is the content in buffer”);
   exit(0);
   //_exit(0);
}
```

flush() 清理缓存

//对比程序运行结果，printf(“This is the content in buffer”);后加\n结果

（4）进程等待wait()和waitpid()

- wait()函数是用于使父进程（也就是调用wait()的进程）阻塞，直到一个子进程结束或者该进程接到了一个指定的信号为止。如果该父进程没有子进程或者他的子进程已经结束，则wait()就会立即返回。
- waitpid()的作用和wait()一样，但它并不一定要等待第一个终止的子进程，它还有若干选项，如可提供一个非阻塞版本的wait()功能，也能支持作业控制。实际上wait()函数只是waitpid()函数的一个特例，在Linux内部实现wait()函数时直接调用的就是waitpid()函数。

（4）进程等待wait()和waitpid()

- 调用wait或waitpid的进程可能会：
  - 如果其所有子进程都还在运行，则父进程阻塞
  - 如果一个子进程已终止，正等待父进程获取其终止状态，则取得该子进程的终止状态立即返回
  - 如果它没有任何子进程，则立即出错返回

（4）Wait()、waitpid()

- pid_t wait(int *status);
- pid_t waitpid(pid_t pid,int *status,int options);
  - status：用于保存子进程的结束状态。
  - pid：为欲等待的子进程ID，其数值意义如下：
    - pid<-1：等待进程组ID为 pid 绝对值的任何子进程
    - pid=-1：等待任何子进程,相当于 wait()
    - pid=0：等待进程组ID与目前进程相同的任何子进程
    - pid>0：等待任何子进程ID为 pid 的子进程

（4）Wait()、waitpid()

- options：该参数提供了一些额外的选项来控制waitpid，可有以下几个取值或它们的按位或组合：
  - 0：同wait，阻塞父进程，等待子进程退出；
  - WNOHANG：若pid指定的子进程没有结束，则waitpid()函数返回0，不予以等待。若结束，则返回该子进程的ID。
  - WUNTRACED：若子进程进入暂停状态，则马上返回，但子进程的结束状态不予以理会。
- **返回值：**
  - -1：调用失败。
  - 其他：调用成功，返回值为退出的子进程ID。

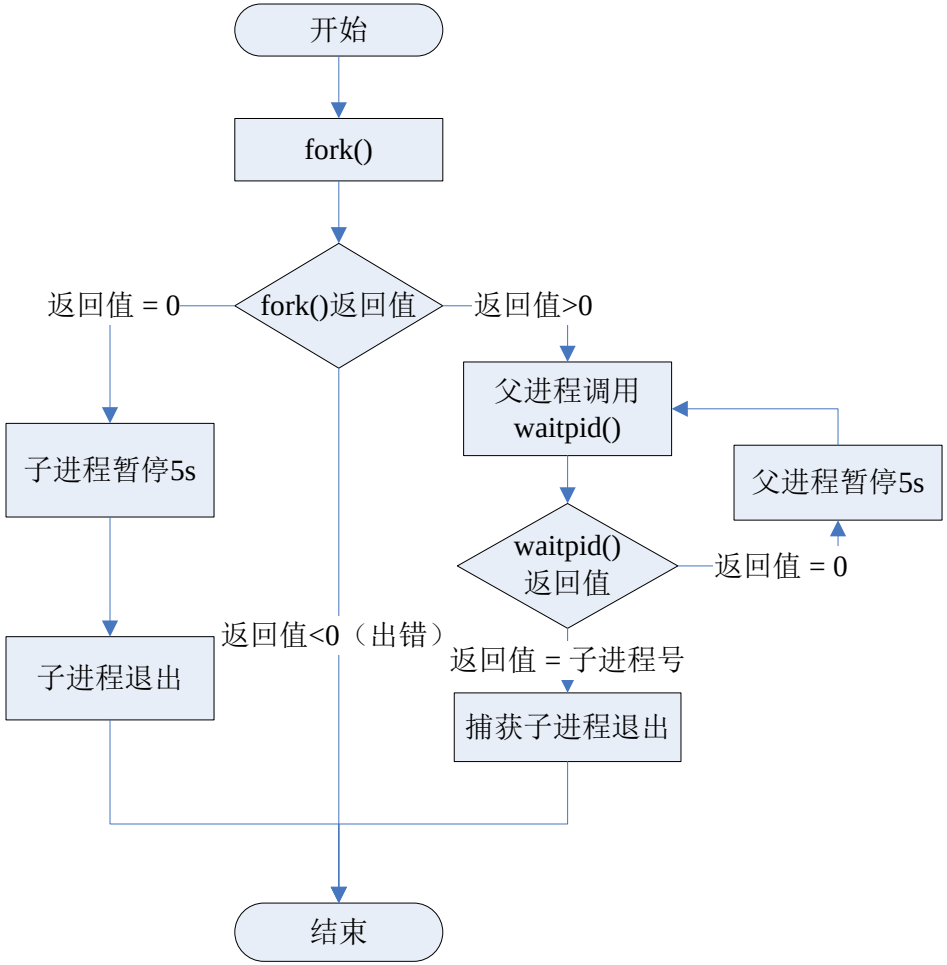
（4）进程等待wait()和waitpid()

- 例程wait.c

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
   pid_t pc,pr;
   pc=fork();
   if(pc==0){
          printf(“this is child process with pid of %d\n”,getpid());
          sleep(10);}
   else if(pc>0){
          pr=wait(NULL);
          printf(“I catched a child process with pid of %d\n”,pr);
          }
   exit(0);
}
```

示例

- 例程：waitpid.c



（5）exec函数族

- exec函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新的进程替换了。另外，这里的可执行文件既可以是二进制文件，也可以是Linux下任何可执行的脚本文件。
- 使用exec函数族主要有两种情况
  - 当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用exec函数族中的任意一个函数让自己重生；
  - 如果一个进程想执行另一个程序，那么它就可以调用fork()函数新建一个进程，然后调用exec函数族中的任意一个函数，这样看起来就像通过执行应用程序而产生了一个新进程（这种情况非常普遍）。

（5）exec函数族

- exec系列函数共有6种不同的形式，统称为exec函数族。我们把这6个函数划分为两组：
  - execl、execle 和 execlp
  - execv、execve 和 execvp
  - 这两组函数的不同在于exec后的第一个字符，第一组是l，在此称为execl系列；第二组是v，在此称为execv系列。这里的l是list（列表）的意思，表示execl系列函数**需要将每个命令行参数作为函数的参数进行传递**。而v是vector（矢量）的意思，表示execv系列函数**将所有函数包装到一个矢量数组中传递**即可。

（5）exec函数族

- exec函数的原型为：
- int execv(const char *path,char* const argv[]);
- int execve(const char *path, char* const argv[],char* const envp[]);
- int execvp(const char *file,char* const argv[]);
- int execl(const char *path,const char *arg,...);
- int execle(const char *path, const char *arg, char* const envp[]);
- int execlp(const char *file,const char *arg,...);
- 参数说明：
  - path：要执行的程序路径。可以是绝对路径或者是相对路径。在execv、execve、execl和execle这四个函数中，使用带路径名的文件名作为参数。

（5）exec函数族

- file：要执行的程序名称。如果该参数中包含“/”字符，则视为路径名直接执行；否则视为单独的文件名，系统将根据PATH环境变量指定的路径顺序搜索指定的文件。
- argv：命令行参数的矢量数组。
- envp：带有该参数的exec函数，可以在调用时指定一个环境变量数组。其他不带该参数的exec函数，则使用调用进程的环境变量。
- arg：程序的第0个参数，即程序名自身，相当于argv[0]。
- ...：命令行参数列表。调用相应程序时有多少命令行参数，就需要有多少个输入参数项。注意：在使用此类函数时，在所有命令行参数的最后，应该增加一个空的参数项（NULL），表明命令行参数结束。

- 返回值：
  - -1：表明调用exec失败。无返回：表明调用成功

（5）exec函数族

#include <unistd.h>

int execl(const char* path,const char* arg1,……)

参数:

- path：被执行程序名(含完整路径)
- arg1-argn：被执行程序所需的命令行参数，含程序名。以空指针NULL结束

例:execl.c

```
int main()
{
  execl(“/bin/ls”,”ls”,”-al”,”/etc/passwd”,(char*)0);
}
```

（5）exec函数族

#include <unistd.h>

int execlp(const char* file,const char* arg1,……)

参数:

- file：被执行程序名(不含路径，将从path环境变量中查找该程序)
- arg1-argn：被执行程序所需的命令行参数，含程序名。以空指针NULL结束

例:execlp.c

```
int main()
{
  execlp(“ls”,”ls”,”-al”,”/etc/passwd”,(char*)0);
}
```

（5）exec函数族

#include <unistd.h>

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

参数:

- path：被执行程序名(含完整路径)
- argv[]：被执行程序所需的命令行参数数组

例:execv.c

```
int main()
{
  char* argv[]={“ls”,”-al”,” /etc/passwd”,(char*)0};
  execv(“/bin/ls”,argv);
}
```

（5）exec函数族

```
#include <unistd.h>
int execve(const char* path, char* const argv[],char* const envp[])
```

参数:

- **path：被执行程序名(含完整路径)**
- **argv[]：被执行程序所需的命令行参数数组**
- **envp[]：指向环境字符串指针数组**

例:execve.c

```
int main()
{
   char* argv[]={“ls”,”-al”,” /etc/passwd”,(char*)0};
   char* envp[]={“PATH=/tmp”,”USER=wxq”,NULL};
   if(fork()==0)
   {
          if(execve(“/usr/bin/env”,argv,envp)<0)
                 printf(“Execve error\n”);
   }
}
```

（5）exec函数族

int system(const char* string)

- 功能：调用fork产生子进程，由子程序来调用/bin/sh –c string来执行参数string所代表的命令。
- 与exec不同的是system将外部可执行程序加载执行完毕后继续返回调用进程。

例:system.c

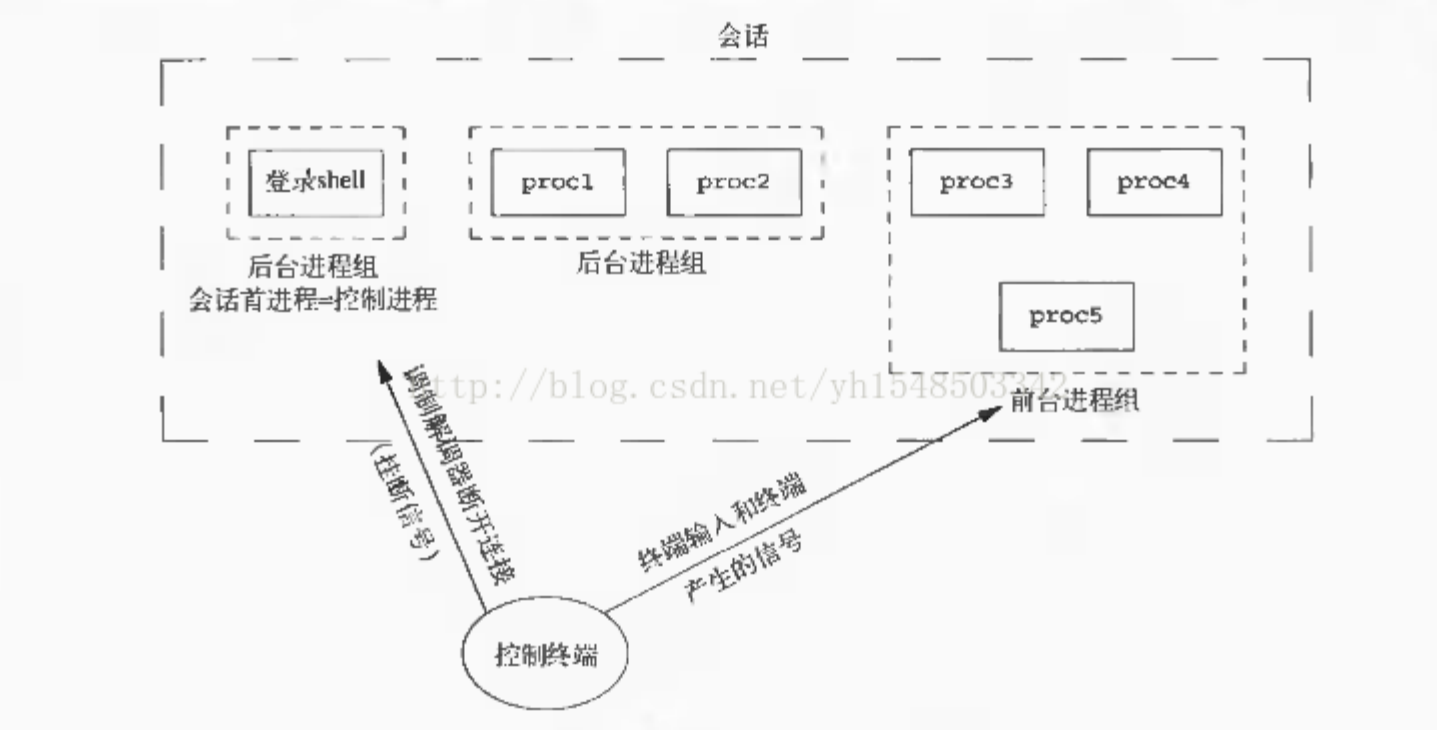
```
#include <stdlib.h>
int main()
{
  system(“ls –al /etc/passwd”);
}
```

（6）Linux守护进程

- 守护进程就是后台服务进程，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。守护进程常常在系统引导载入时启动，在系统关闭时终止。Linux有很多系统服务，大多数服务都是通过守护进程实现的，守护进程还能完成许多系统任务，例如，侦听网络接口服务xinetd、打印进程lqd等（这里的尾字母d是Daemon的意思）。
- 在Linux中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能够突破这种限制，它从被执行开始运转，直到整个系统关闭时才会退出。如果想让某个进程不因为用户、终端或者其他的变化而受到影响，那么就必须把这个进程变成一个守护进程。可见，守护进程是非常重要的。

（6）Linux守护进程

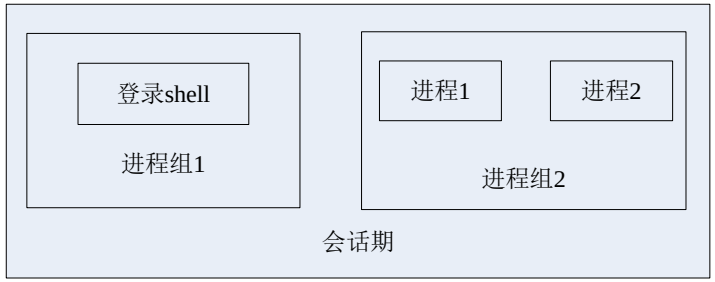
终端、会话、进程组、进程之间的关系：



（6） Linux守护进程

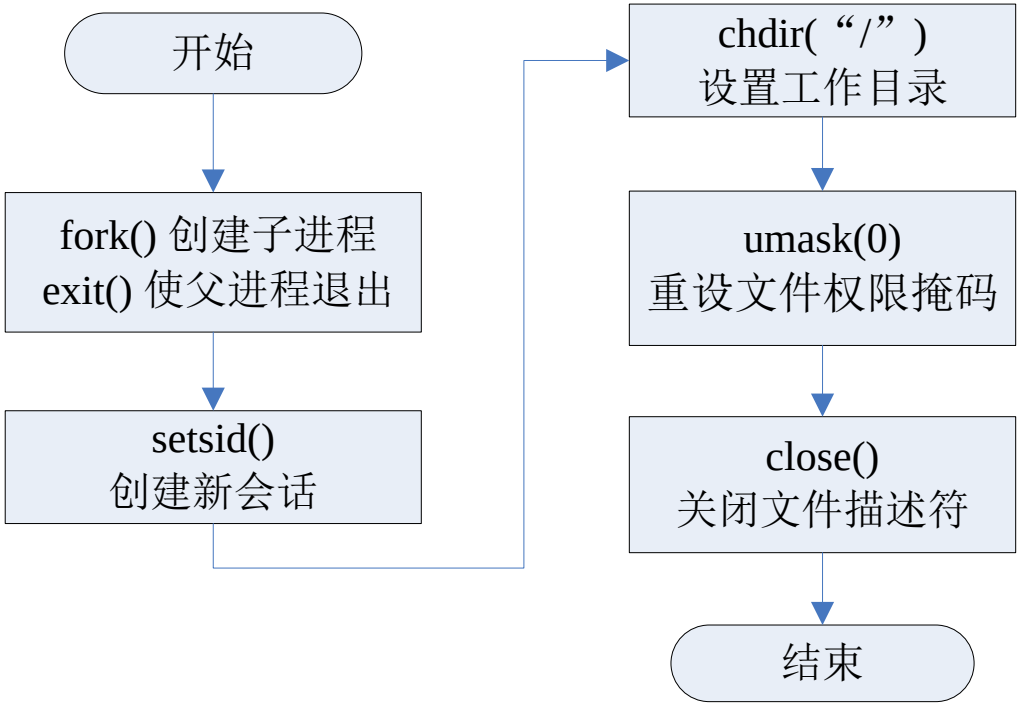
守护进程的编写流程：

- 创建子进程，父进程退出(由于守护进程是脱离控制终端的，杀死父进程使其脱离控制终端)
- 在子进程中创建新会话(重要) :setsid()
  - 创建新会话，并担任会话组的组长
  - 让进程摆脱原会话的控制
  - 让进程摆脱原进程组的控制
  - 让进程摆脱原控制终端的控制
- 改变当前目录为根目录（chdir()）
- 重设文件权限掩码（umask(0)）
- 关闭文件描述符 for(i=0;i<MAXFILE;i++)
  close(i);



（6） Linux守护进程

- **守护进程的编写流程**



例程 Daemon.c

（6） Linux守护进程

- 守护进程的出错处理
  - 由于守护进程完全脱离了控制终端，因此，不能像其他普通进程一样将错误信息输出到控制终端来通知程序员。守护进程的一种通用的办法是使用syslog服务，将程序中的出错信息输入到系统日志文件中，从而可以直观地看到程序的问题所在。
  - syslog是Linux中的系统日志管理服务，通过守护进程syslogd来维护。该守护进程在启动时会读一个配置文件“/etc/syslog.conf”。该文件决定了不同种类的消息会发送向何处。如，紧急消息被送向系统管理员并在控制台上显示，而警告消息则可被记录到一个文件中。

（6） Linux守护进程

- 守护进程的出错处理
  - openlog()函数用于打开系统日志服务的一个连接；syslog()函数是用于向日志文件中写入消息，在这里可以规定消息的优先级、消息输出格式等；closelog()函数是用于关闭系统日志服务的连接。

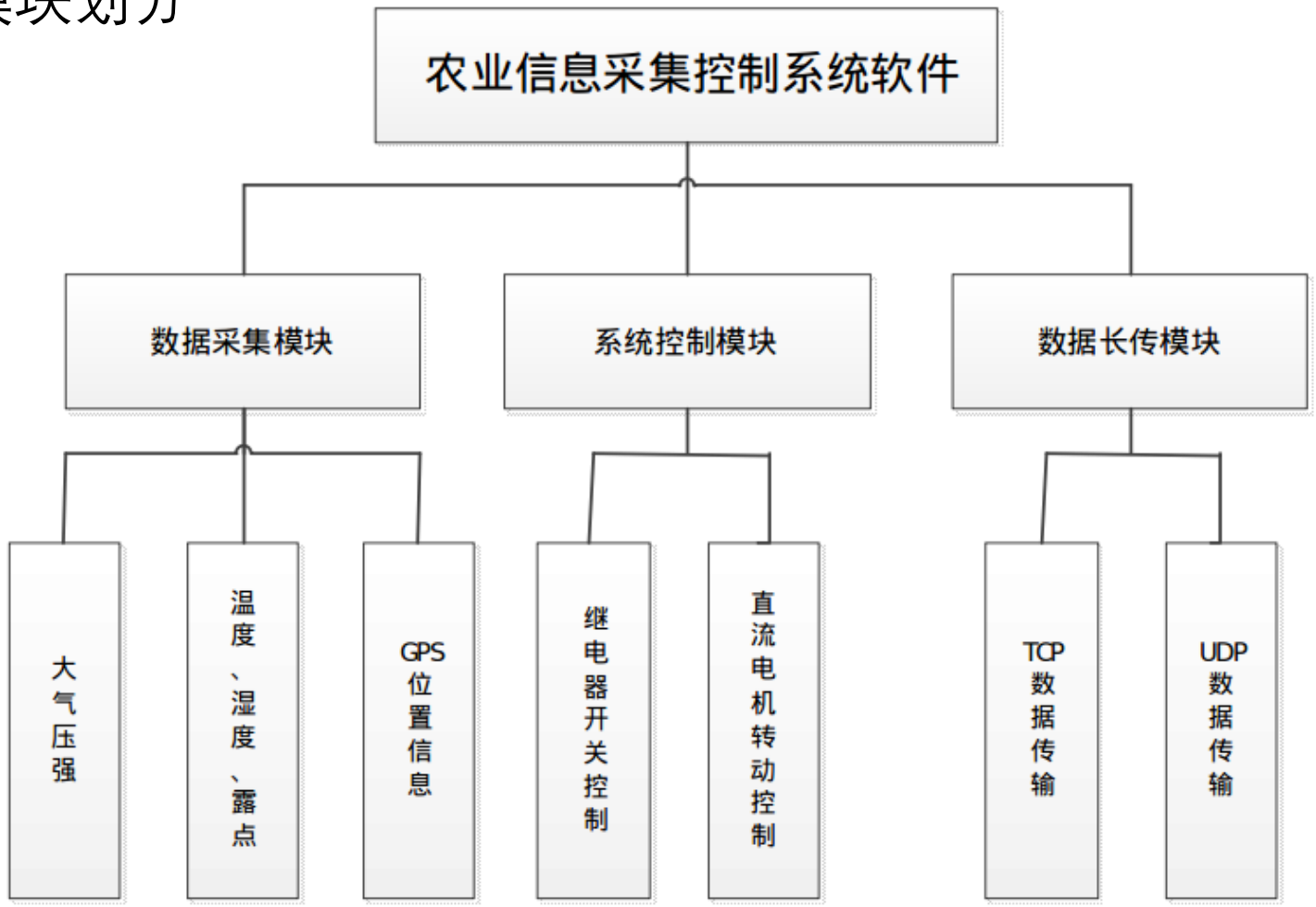
例程 syslog_daemon.c

农业信息采集控制系统主程序设计

- 农业信息采集控制系统总体由三大模块构成，一是数据采集，我们主要完成农田温湿度信息、大气压强、农田位置等数据的实时采集。二是输出控制模块，用直流电机、继电器等来控制农田大棚卷帘的升降、水管的灌溉等，实现上通过键盘按键完成对应的控制功能；三是数据上传模块，通过TCP/IP协议将采集的数据传输到远程服务器。

农业信息采集控制系统主程序设计

- 模块划分



农业信息采集控制系统主程序设计

- 主进程流程:
  - 系统主要功能分为三部分，为了实现数据采集与控制功能相互独立互不影响，我们采用多进程实现数据采集和控制。父进程完成控制功能，即扫描键盘，根据按键发出控制动作。子进程负责数据采集，假设每隔5s钟完成一次数据采集。采集后的数据通过命名管道（也可以采用其他通信方式）的方式发送给数据上传的进程。

农业信息采集控制系统主程序设计