Linux进程相关系统调用

Version 1.0

西安電子科技大學

需要掌握的要点

- 掌握Linux进程相关的基本概念系
- 理解进程的生命周期，重点理解僵尸态
- 掌握Linux下进程创建及进程管理
- 掌握Linux下进程创建相关的系统调用
  - fork, vfork, clone
  - execl, execv
  - exit, _exit
  - wait, waitpid
- 掌握守护进程的概念
- 学会编写守护进程

西安電子科技大學

Linux中关于进程的基本概念

- 程序是静态的一段代码，文件系统上的可执行文件
- 进程是程序的一次执行过程，是资源分配的最小单位
- 在Linux下，进程也CPU调度的最小单位；
- 在Linux下，线程是利用轻量级进程机制实现的。
- 在Linux下，任务（task）等价于进程
- Linux利用数据结构task_struck(进程控制块PCB)记录进程的描述信息、控制信息和资源信息。

操作系统课程中线程的概念

- 线程的概念
  - 它是进程内独立的一条运行路线，处理器调度的最小单元，也可以称为轻量级进程。线程可以对进程的内存空间和资源进行访问，并与同一进程中的其他线程共享。因此，线程的上下文切换的开销比创建进程小得多。
- 线程与进程间的关系

Linux 系统中的线程

- 线程的种类
  - 用户级线程
  - 轻量级进程
  - 内核线程
  - https://blog.csdn.net/gatieme/article/details/51481863
  - https://blog.csdn.net/gatieme/article/details/51892437
- Linux中的线程基于进程实现
  - 相对于UNIX，进程创建开销小
  - 用户线程对象与内核线程对象的关系是“一对一”
  - pthread

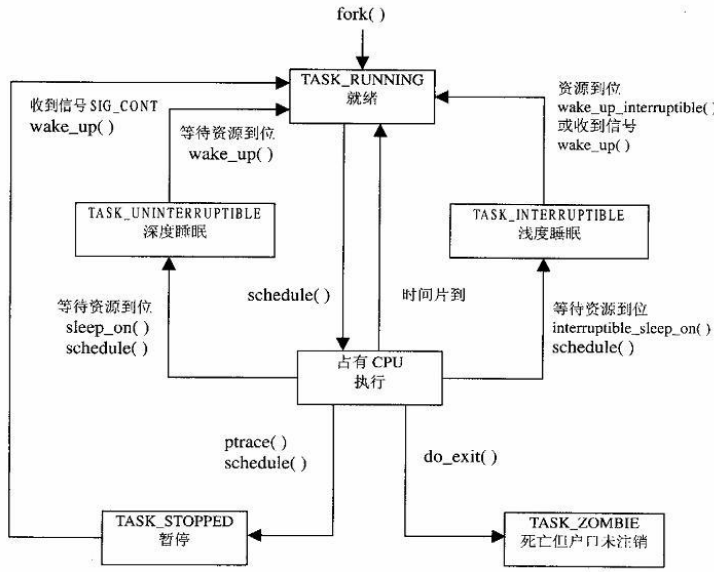
进程标识PID

- Linux内核通过惟一的进程标识符PID来标识每个进程。PID存放在进程描述符task_struct的pid字段中 。
- 在Linux中获得当前进程的进程号（PID）和父进程号（PPID）的系统调用函数分别为：
  - pid_t getpid();
  - pid_t getppid();

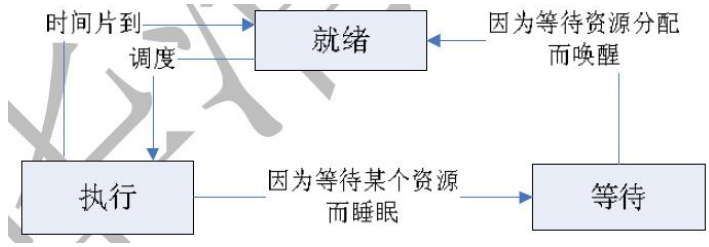
Linux进程状态

- Linux中任务和进程是相同的术语，每个进程由task_struct 结构来描述，即PCB （进程控制块）
- Linux 将进程状态主要分为五种：
  - 运行状态（TASK_RUNNING)R
  - 可中断的阻塞状态(TASK_INTERRUPTIBLE)S
  - 不可中断的阻塞状态(TASK_UNINTERRUPTIBLE)D
  - 可终止的阻塞状态(TASK_KILLABLE)
  - 暂停状态(TASK_STOPPED)T
  - 跟踪状态(TASK_TRACED)T
  - 僵尸状态(TASK_ZOMBIE)Z
  - 僵尸撤销状态(EXIT_DEAD)X
- 进程的状态随着进程的调度发生改变

Linux进程状态转换



Linux进程状态转换



进程启动

- 手工启动
  - 前台启动：shell中输入命令：program
  - 后台启动：shell中输入命令：program&
- 调度启动
  - 利用at命令在指定时刻启动
  - 利用cron命令定期启动

进程管理相关命令

ps	查看系统中的进程
top	动态显示系统中的进程
nice	按用户指定的优先级运行
renice	改变正在运行进程的优先级
kill	向进程发送信号（包括后台进程）
crontab	用于安装、删除或者列出用于驱动 cron 后台进程的任务。
bg	将挂起的进程放到后台执行

在程序中创建与终止进程

- **进程的创建和执行：**
  - Linux中进程的创建进程被分解到两个单独的函数中取执行：fork()和exec函数族。首先，fork()通过拷贝当前进程创建一个子进程，子进程与父进程的区别仅仅在于不同的PID、PPID和某些资源及统计量。exec函数族负责读取可执行文件并将其载入地址空间开始运行。
- **进程的终止：**
  - 进程终结也需要做很多繁琐的收尾工作，系统必须保证进程所占用的资源回收，并通知父进程。Linux首先把终止的进程设置为僵尸状态，这个时候，进程无法投入运行了，它的存在只为父进程提供信息，申请死亡。父进程得到信息后，开始调用wait函数族，最终赐死子进程，子进程占用的所有资源被全部释放。

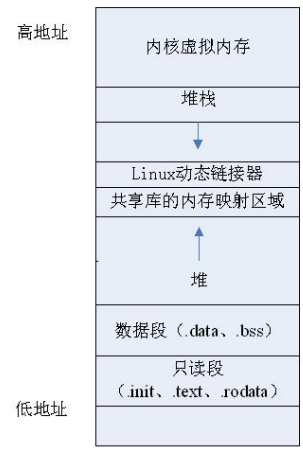
fork函数

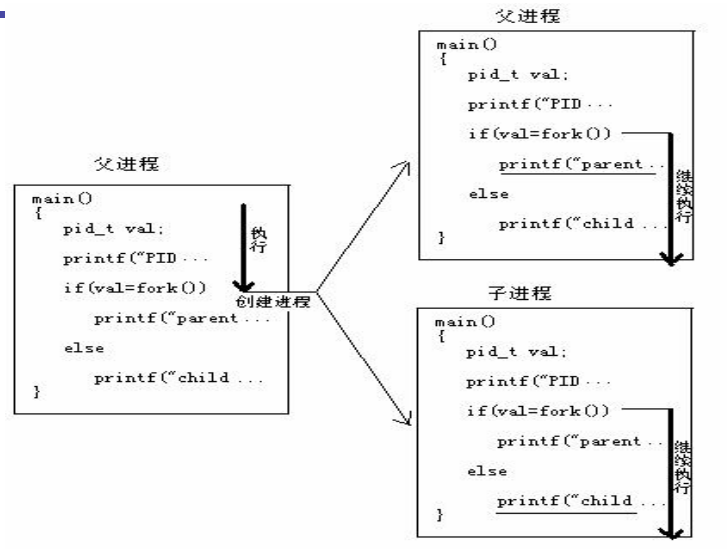
- **fork()**
  - fork()函数用于从已存在的进程中创建一个新进程。新进程称为子进程，而原进程称为父进程。
  - 使用fork()函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间，包括进程上下文、代码段、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组号、当前工作目录、根目录、资源限制和控制终端等，而子进程所独有的只有它的进程号、资源使用和计时器等。
- **fork()函数语法：**

所需头文件	#include <sys/types.h> /* 提供类型 pid_t 的定义 */ #include <unistd.h>
函数原型	pid_t fork(void)
函数返回值	0：子进程
	子进程 ID（大于 0 的整数）：父进程
	-1：出错

进程地址空间

- **Linux虚拟内存管理，4G的地址空间，0-3G为用户地址空间，3G-4G为内核地址空间，内核地址空间有自己的页表，用户进程有自己的页表**
- **进程的内存结构**
- **查看**
  - cat /proc/pid/maps
- **确定进程上限**
  - cat /proc/sys/kernel/pid_max





fork的另一个例子

```
int main(){
        int var;
        pid_t pid;
        var = 88;
        if((pid = fork()) < 0) {
                printf("fork error\n");
        }else if (pid == 0) {
                var++;
        }
        else {
                sleep(2);
        }
        printf("pid = %d, var = %d\n", getpid(), var);
        return 0;
}
```

关于fork的两个问题

- 连续调用2次fork，共可产生多少个进程?
- 下面的程序一共输出多少个“-”？

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    int i;
    for(i=0; i<2; i++){
        fork();
        printf("-");
    }
    return 0;
}
```


子进程复制了父进程的哪些资源?

- 用户ID、用户组ID、进程组ID、会话ID
- 当前工作目录、根目录、环境变量
- 文件访问权限、资源访问权限
- 信号屏蔽位
- 打开的文件描述符
- 进程地址空间（数据段、代码段、堆栈段）
- …

Fork函数的应用逻辑—孙悟空逻辑

```
int main()
{
    …
    ….  // 遇到两个需要并行执行的任务：任务1和任务2
    pid = fork(); // 分身术
    if (pid == 0) {
    …… // 子进程处理任务1
    }
    else {
     …… // 父进程处理任务2
    }
    return 0;
}
```


Fork函数应用例1

```
int main()  {
      int fd1 = open("data_file1", …);
      int fd2 = open("data_file2", …);
      pid_t pid;
      if((pid = fork()) ==0) {
            close(fd2);
            read(fd1, buffer, len);
            … // 处理文件1的数据
     }
      else {
            close(fd1);
            read(fd2, buffer, len);
            … // 处理文件2的数据
      }
}
```

Windows创建进程API—CreateProcess

```
BOOL CreateProcess
(
   LPCTSTR lpApplicationName,     //新进程将要使用的可执行文件的名字(路径)
   LPTSTR lpCommandLine,     //递给新进程的命令行字符串
   LPSECURITY_ATTRIBUTES lpProcessAttributes,
   LPSECURITY_ATTRIBUTES lpThreadAttributes,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
   LPVOID lpEnvironment,
   LPCTSTR lpCurrentDirectory,
   LPSTARTUPINFO lpStartupInfo,
   LPPROCESS_INFORMATION lpProcessInformation
);
```


CreateProcess的应用逻辑—黑社会逻辑

Main.exe： 主控程序， 黑社会Boss

A.exe： 专门处理任务A的程序， 黑社会小弟

B.exe： 专门处理任务A的程序， 黑社会小弟

```
int main()
{
    …
    ….  // 遇到两个需要并行执行的任务：任务A和任务B
    CreateProcess(“A.exe”, …); // 叫个小弟来处理任务A
    CreateProcess(“B.exe”, …); // 叫个小弟来处理任务B
    ….  // 自己继续享受生活
}
```

exec函数

- exec函数用于创建一个新的进程，新进程以另一个可执行程序为执行脚本。
- exec创建的新进程“占用了”原进程的绝大部分资源，进程地址空间中装入了新的可执行程序。exec执行成功之后，原进程就“消失了”。
- #include <unistd.h>
- int execl(const char *path, const char *arg, ...)
- 参数path：可执行文件的路径和名字构成的字符串
- arg：新程序的命令行参数1
- execl是一个不定参数函数，还可以传入多个命令行参数，最后一个参数必须是NULL。
- 返回值：-1 表示出错

exec函数的例子1

```
#include <unistd.h>
int main(int argc, char *argv[]) {
    if(execl("/bin/echo", "echo", "executed byexecl", NULL)<0)
        perror("Err on execl");
}
```

exec函数的例子2

如果还想保留父进程怎么办？？

```
#include <unistd.h>
int main(int argc, char *argv[]) {
   if(fork()==0) {
      if(execl("/bin/echo", "echo", "executed byexecl",
NULL)<0)
      perror("Err on execl");
      }
    // 父进程做其他事情
    return 0;
}
```


exec函数族

```
#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[])
```

exec函数族使用区别

- exec函数族使用区别
  - 查找方式
    - 表中的前四个函数的查找方式都是完整的文件目录路径，而最后两个函数(以p结尾的函数)可以只给出文件名，系统就会自动从环境变量“$PATH”所指出的路径中进行查找。
  - 参数传递方式
    - 两种方式：逐个列举、将所有参数整体构造指针数组传递
    - 以函数名的第五位字母来区分的，字母为“l”(list)的表示逐个列举的方式，其语法为char *arg；字母为“v”(vertor)的表示将所有参数整体构造指针数组传递，其语法为const argv[]
  - 环境变量
    - exec函数族可以默认系统的环境变量，也可以传入指定的环境变量。这里，以“e”(Enviromen)结尾的两个函数execle、execve就可以在envp[]中指定当前进程所使用的环境变量
  - 真正的系统调用是execve()

exec与fork配合使用的效率问题

- 在Unix时代exec经常与fork配合使用，但这样做了大量的无用功，效率低下（为什么？）。
- 为了解决该问题，在Unix时代创建了vfork函数与exec配合。vfork函数不复制父进程的资源，而是共享父进程资源，直到碰到exec函数才开始复制父进程的部分资源（不包括进程地址空间）。
- 在Linux时代，fork函数实现中引入了“写时拷贝Copy On Write”技术，fork+exec的配合效率也很高。
- Linux时代，vfork几乎没有存在的必要，只被用在极少数场合。

vfork函数

- **vfork()**

  其功能类似于fork()，但是有以下两点显著的不同：

  – vfork()不同于fork()，它没有复制自己的进程地址空间，而是共享父进程的，所以，子进程的改变也会引起父进程的改变

  – vfork()创建后子进程总是立即优先于父进程执行的，在子进程exec或者exit后，才会执行父进程。

- **vfork()函数语法：**

  – pid_t vfork (void);

exit()和_exit()函数

- **exit()和_exit()**

  – exit()和_exit()函数都是用来终止进程的。当程序执行到exit()或_exit()时，进程会无条件地停止剩下的所有操作，清除包括各种数据结构，并终止本进程的运行。

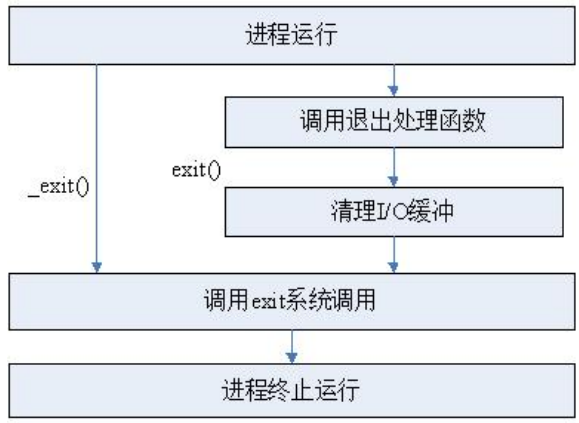
- **int atexit(void (*function)(void));**

  – function是一个函数指针，指向程序退出时候调用的一个回调函数。利用该函数程序能够在最终关闭之前提供一个或者多个运行的清理函数

  – atexit用于注册一个或多个退出函数

exit()和_exit()的执行过程

- exit()和 exit()的执行过程



exit()和_exit()的区别

- exit()和_exit()的区别
  - _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，并销毁其在内核中的各种数据结构；
  - exit()函数则在这些基础上作了一些包装，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()函数在终止当前进程之前要检查该进程打开过哪些文件，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。
    - “标准文件I/O”

exit()和_exit()函数

– exit()和_exit函数语法：

所需头文件	Exit：#include <stdlib.h> _exit：#include <unistd.h>
函数原型	Exit：void exit(int status) _exit：void _exit(int status)
函数传入值	status 是一个整型的参数，可以利用这个参数传递进程结束时的状态。一般来说，0 表示正常结束；其他的数值表示出现了错误，进程非正常结束。 在实际编程时，可以用 wait()系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理

僵尸进程

- 僵尸进程产生条件
  - 子进程执行完毕；
  - 父进程没有回收其状态；
- 子进程退出前会向父进程发送SIGCHLD信号
- 父进程用wait和waitpid回收

wait()和waitpid()

- **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。
  - wait函数语法：

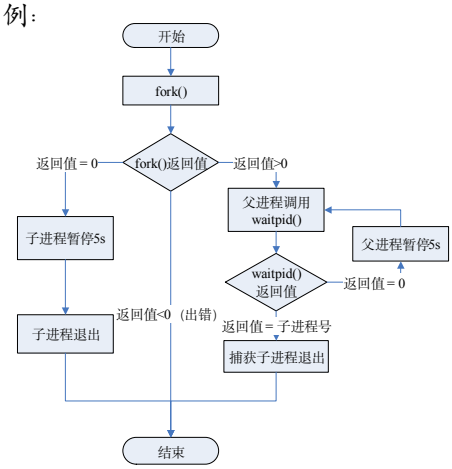
所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t wait(int *status)
函数传入值	这里的 status 是一个整型指针，是该子进程退出时的状态。status 若不为空，则通过它可以获得子进程的结束状态。另外，子进程的结束状态可由 Linux 中一些特定的宏来测定
函数返回值	成功：已结束运行的子进程的进程号 失败：-1

wait()和waitpid()

- **wait()和waitpid()**

所需头文件	#include <sys/types.h> #include <sys/wait.h>	
函数原型	pid_t waitpid(pid_t pid, int *status, int options)	
函数传入值	Pid	pid > 0：只等待进程 ID 等于 pid 的子进程，不管已经有其他子进程运行结束退出了，只要指定的子进程还没有结束，waitpid()就会一直等下去
		pid = -1：等待任何一个子进程退出，此时和 wait()作用一样
		pid = 0：等待其组 ID 等于调用进程的组 ID 的任一子进程
		pid < -1：等待其组 ID 等于 pid 的绝对值的任一子进程
函数传入值	Status	同 wait()
	options	WNOHANG：若由 pid 指定的子进程没有结束，则 waitpid()不阻塞而立即返回，此时返回值为 0
		WUNTRACED：为了实现某种操作，由 pid 指定的任一子进程已被暂停，且其状态自暂停以来还未报告过，则返回其状态
		0：同 wait()，阻塞父进程，等待子进程退出
函数返回值	正常：已经结束运行的子进程的进程号	
	使用选项 WNOHANG 且没有子进程退出：0	
	调用出错：-1	

- **wait()和waitpid()**
  - waitpid实例：



Linux守护进程

- 守护进程，也就是通常所说的Daemon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的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装入时启动，在系统关闭时终止
- Linux系统有很多守护进程，大多数服务都是用守护进程实现的
- 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，相应的进程都会被自动关闭。
- 守护进程能够突破这种限制，它从被执行开始运转，直到整个系统关闭才会退出。如果想让某个进程不因为用户或终端或其他的变化而受到影响，就必须把这个进程变成一个守护进程。

Linux守护进程编写规范

- **编写守护进程**
  - 创建子进程，父进程退出
  - 在子进程中创建新会话
  - 改变当前目录为根目录
  - 重设文件权限掩码
  - 关闭文件描述符

Linux守护进程编写规范

- **创建子进程，父进程退出**
  - 这是编写守护进程的第一步。由于守护进程是脱离控制终端的，因此，完成第一步后就会在shell终端里造成一种程序已经运行完毕的假象。之后的所有工作都在子进程中完成，而用户在shell终端里则可以执行其他的命令，从而在形式上做到了与控制终端的脱离。
  - 由于父进程已经先于子进程退出，会造成子进程没有父进程，从而变成一个孤儿进程。在Linux中，每当系统发现一个孤儿进程，就会自动由1号进程（也就是init进程）收养它，这样，原先的子进程就会变成init进程的子进程了。
  - 实例：

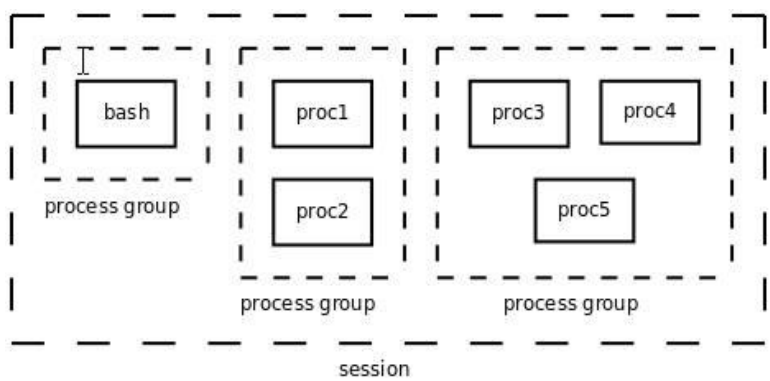
```
pid = fork();
if (pid > 0)
{
   exit(0); /*父进程退出*/
}
```

Linux守护进程编写规范

- **在子进程中创建新会话**
  - 进程组
    - Shell上的一条命令形成一个进程组
    - 进程组是一个或多个进程的集合。进程组由进程组ID来惟一标识。除了进程号（PID）之外，进程组ID也是一个进程的必备属性。
    - 每个进程组都有一个组长进程，其组长进程的进程号等于进程组ID。且该进程组ID不会因组长进程的退出而受到影响。
  - 会话期
    - 会话是一个或多个进程组的集合。一个会话开始于用户登录，终止于用户退出，在此期间该用户运行的所有进程都属于这个会话期
    - 会话会有一个控制终端，控制终端上的输入和信号会发送给进程组中的每个进程
    - 一个会话可包含多个进程组，但只能有一个前台进程组
    - 只有不是进程组长的进程才能创建新会话

-

```
$ proc1 | proc2 &
$ proc3 | proc4 | proc5
```



Linux守护进程编写规范

- **在子进程中创建新会话**
  - setsid()函数作用：
    - setsid()函数用于创建一个新的会话，并担任该会话组的组长。调用setsid()有下面的3个作用。
    - 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。
  - setsid()函数格式：

所需头文件	#include <sys/types.h> #include <unistd.h>
函数原型	pid_t setsid(void)
函数返回值	成功：该进程组 ID 出错：-1

Linux守护进程编写规范

- **改变当前目录为根目录**
  - 通常的做法是让“/”作为守护进程的当前工作目录。
  - 使用fork创建的子进程继承了父进程的当前工作目录。

Linux守护进程编写规范

- **重设文件权限掩码**
  - 文件权限掩码是指屏蔽掉文件权限中的对应位。由于使用fork新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多的麻烦。因此，把文件权限掩码设置为0，可以大大增加该守护进程的灵活性。设置文件权限掩码的函数是umask
  - 通常的使用方法为umask(0)

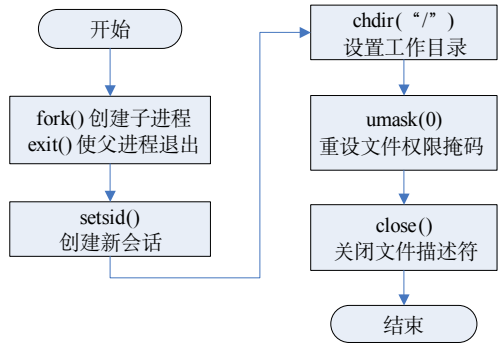
Linux守护进程编写规范

- **关闭文件描述符**
  - 同文件权限掩码一样，用fork新建的子进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读或写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法卸下。
  - 在上面的第二步后，守护进程已经与所属的控制终端失去了联系。因此从终端输入的字符不可能达到守护进程，守护进程中用常规的方法(如printf)输出的字符也不可能在终端上显示出来。所以，文件描述符为0、1和2的三个文件(常说的输入、输出和报错这三个文件)已经失去了存在的价值，也应被关闭。
  - 实例：

```
for(i=0;i<MAXFILE;i++)
{
   close(i);
}
```

Linux守护进程编写规范

- **Linux守护进程创建流程**



Linux守护进程

- **守护进程出错处理**
  - 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。例如，紧急消息可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。
  - 该机制提供了3个syslog相关函数，分别为openlog()、syslog()和closelog()。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。

Linux守护进程

- **syslog相关函数格式**

所需头文件	#include <syslog.h>	
函数原型	void openlog (char *ident, int option , int facility)	
函数传入值	ident	要向每个消息加入的字符串，通常为程序的名称
	option	LOG_CONS：如果消息无法送到系统日志服务，则直接输出到系统控制终端
		LOG_NDELAY：立即打开系统日志服务的连接。在正常情况下，直接发送到第一条消息时才打开连接
		LOG_PERROR：将消息也同时送到stderr上
		LOG_PID：在每条消息中包含进程的PID
函数传入值	facility：指定程序发送的消息类型	LOG_AUTHPRIV：安全/授权讯息
		LOG_CRON：时间守护进程（cron及at）
		LOG_DAEMON：其他系统守护进程
		LOG_KERN：内核信息
		LOG_LOCAL[0~7]：保留
		LOG_LPR：行打印机子系统
		LOG_MAIL：邮件子系统
		LOG_NEWS：新闻子系统
		LOG_SYSLOG：syslogd内部所产生的信息
		LOG_USER：一般使用者等级讯息
		LOG_UUCP：UUCP子系统

Linux守护进程

- **syslog相关函数格式**

所需头文件	#include <syslog.h>	
函数原型	void syslog(int priority, char *format, ...)	
函数传入值	priority：指定消息的重要性	LOG_EMERG：系统无法使用
		LOG_ALERT：需要立即采取措施
		LOG_CRIT：有重要情况发生
		LOG_ERR：有错误发生
		LOG_WARNING：有警告发生
		LOG_NOTICE：正常情况，但也是重要情况
		LOG_INFO：信息消息
		LOG_DEBUG：调试信息
	format	以字符串指针的形式表示输出的格式，类似printf中的格式

所需头文件	#include <syslog.h>
函数原型	void closelog(void)