Linux C++通讯架构实战\_卷1—学习笔记

视频课程地址：

<https://study.163.com/course/introduction/1006470001.htm?share=1&shareId=1396930938>

目录

[1. 环境搭建 4](#_Toc35630494)

[1.1. 安装Ubuntu虚拟机 4](#_Toc35630495)

[1.2. 配置Ubuntu的固定IP地址 4](#_Toc35630496)

[1.3. 配置远程连接 5](#_Toc35630497)

[1.4. 安装gcc,g++等 6](#_Toc35630498)

[2. 进入nginX之门 7](#_Toc35630499)

[2.1. 为什么选择nginX 7](#_Toc35630500)

[2.2. 安装nginX，搭建web服务器 7](#_Toc35630501)

[2.2.1. 安装前提 7](#_Toc35630502)

[2.2.2. nginX文件目录 7](#_Toc35630503)

[2.2.3. nginx的编译和安装 8](#_Toc35630504)

[2.3. nginx的启动和简单实用 9](#_Toc35630505)

[3. nginx整体结构、进程模型 10](#_Toc35630506)

[3.1. nginx的整体结构 10](#_Toc35630507)

[3.1.1. master进程和worker进程概览（父子关系） 10](#_Toc35630508)

[3.1.2. nginx进程模型 10](#_Toc35630509)

[3.1.3. 调整worker进程数量 11](#_Toc35630510)

[3.2. nginx进程模型细说 11](#_Toc35630511)

[4. 终端和进程的关系 12](#_Toc35630512)

[4.1. 终端与bash进程 12](#_Toc35630513)

[4.2. 终端上的开启进程 12](#_Toc35630514)

[4.3. 进程关系进一步分析 12](#_Toc35630515)

[4.4. strace工具的使用 13](#_Toc35630516)

[4.5. 终端关闭时如何让进程不退出 14](#_Toc35630517)

[4.6. 后台运行 & 17](#_Toc35630518)

[5. 信号的概念、认识、处理动作 18](#_Toc35630519)

[5.1. 信号的基本概念 18](#_Toc35630520)

[5.2. 通过kill命令认识一些信号 18](#_Toc35630521)

[5.3. 进程的状态 19](#_Toc35630522)

[5.4. 常用信号列举： 20](#_Toc35630523)

[5.5. 信号处理的相关动作 21](#_Toc35630524)

[6. Unix/Linux体系结构 、信号编程初步 22](#_Toc35630525)

[6.1. Unix/Linux操作系统体系结构 22](#_Toc35630526)

[6.2. signal函数范例 25](#_Toc35630527)

[6.2.1. 可重入函数 25](#_Toc35630528)

[6.2.2. 不可重入函数的错误演示 27](#_Toc35630529)

[7. 信号编程进阶、sigprocmask范例 30](#_Toc35630530)

[7.1. 信号集 30](#_Toc35630531)

[7.2. 信号相关函数 30](#_Toc35630532)

[7.3. sigprocmask等信号函数范例演示 31](#_Toc35630533)

[8. fork函数详解、范例演示 34](#_Toc35630534)

[8.1. fork()函数简单认识 34](#_Toc35630535)

[8.1.1. fork()函数简单范例 34](#_Toc35630536)

[8.1.2. 僵尸进程的产生、解决，SIGCHLD 36](#_Toc35630537)

[8.2. fork()函数进一步认识 38](#_Toc35630538)

[8.3. 完善一下fork()代码 39](#_Toc35630539)

[8.4. fork()失败的可能性 41](#_Toc35630540)

[9. 守护进程详解、范例演示 41](#_Toc35630541)

[9.1. 普通进程运行观察 41](#_Toc35630542)

[9.2. 守护进程基本概念 42](#_Toc35630543)

[9.3. 守护进程编写规则 43](#_Toc35630544)

[9.3.1. 文件描述符 43](#_Toc35630545)

[9.3.2. 输入输出重定向 45](#_Toc35630546)

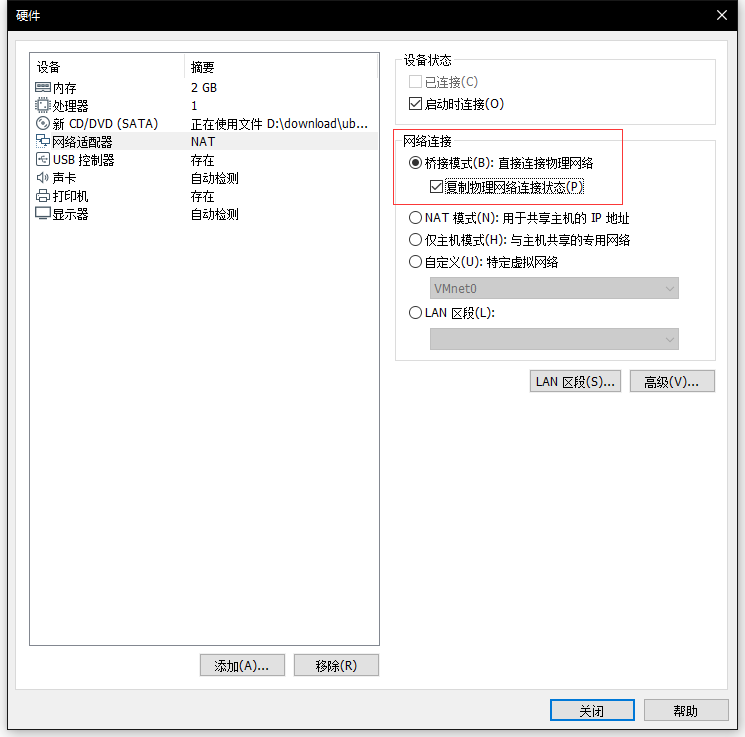
[9.3.3. 空设备 45](#_Toc35630547)

[9.3.4. 实现范例 46](#_Toc35630548)

# 环境搭建

## 安装Ubuntu虚拟机

注意事项：网络适配器选择“桥接”，因为服务端需要固定的IP地址。



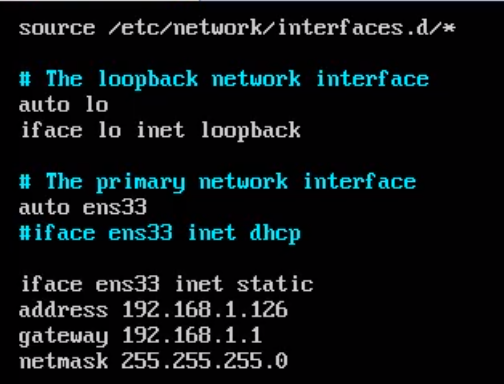
## 配置Ubuntu的固定IP地址

服务器端需要固定的IP地址方便客户端访问。

1. 要修改配置文件，需要vim编辑器，Ubuntu下安装vim的命令：

sudo apt-get install vim-gtk

1. Ubuntu下查看网络信息命令：ifconfig
2. 修改 /etc/network/interfaces 文件：sudo vim interfaces



1. 修改DNS

命令：sudo vim /etc/resolvconf/resolv.conf.d/base



1. 然后重启Ubuntu：sudo reboot

## 配置远程连接

1. 需要在linux上安装ssh服务:

检查有没有ssh服务：ps -e|grep ssh (没有输出代表没有安装)

安装ssh：sudo apt-get install openssh-server

1. 安装Xshell

## 安装gcc,g++等

sudo apt-get install build-essential

sudo apt-get install gcc

sudo apt-get install g++

# 进入nginX之门

## 为什么选择nginX

linux epoll技术； windows IOCP

单机10万并发；

epoll这种高并发技术好处就是：高并发只是占用更多内存就能做到。

## 安装nginX，搭建web服务器

### 安装前提

1. epoll, linux 内核版本为2.6或者以上；

命令：cat /proc/version

1. gcc，g++；
2. pcre库：函数库；支持解析正则表达式；

命令：sudo apt-get install libpcre3-dev

1. zlib库：压缩解压缩功能

命令：sudo apt-get install libz-dev

1. openssl库：ssl功能相关库。用于网站加密通讯

命令：sudo atp-get install libssl-dev

### nginX文件目录

解压nginx压缩包：tar -zxvf nginx-1.16.1.tar.gz

auto/:编译相关的脚本，可执行文件configure一会会用到这些脚本

cc/:检查编译器的脚本

lib/:检查依赖库的脚本

os/:检查操作系统类型的脚本

type/:检查平台类型的脚本

CHANGES:修复的bug，新增加的功能说明

CHANGES.ru:俄语版的CHANGES

conf/:默认的配置文件

configure:编译nginx之前必须先执行本脚本以生成一些必要的中间文件

contrib/:脚本和工具，典型的是vim高亮工具

vim/:vim高亮工具

html/:欢迎界面和错误界面相关的html文件

man/:nginx帮助文件目录

src/:nginx源码目录

core:核心代码

event:event(事件)模块相关代码

http:http(web服务)模块相关代码

mail:邮件模块相关代码

os:操作系统相关代码

stream:流处理相关代码

### nginx的编译和安装

1. 编译的第一步：用configure来进行编译之前的配置工作 ./configure

--prefix:指定最终安装到的目录：默认值 /usr/local/nginx

--sbin-path:用来指定可执行文件目录：默认值 sbin/ nginx

--conf-path:用来指定配置文件目录：默认值 conf/nginx.conf

执行configura文件后，会生成一些文件：

objs/:执行了configure生成的中间文件目录

ngx\_modules.c:内容决定了一会编译nginx的时候有哪些模块会被编译到nginx里边来

Makefile:执行了configure脚本产生的编译规则文件，执行make命令 时用到

1. 用make来编译，生成了可执行文件 make
2. 用make命令开始安装 sudo make install

## nginx的启动和简单实用

启动： sudo ./nginx

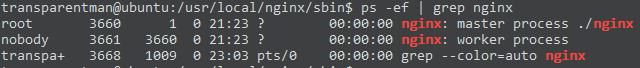
# nginx整体结构、进程模型

## nginx的整体结构

### master进程和worker进程概览（父子关系）

启动nginx，看到了一个master进程，一个worker进程

ps -ef命令：



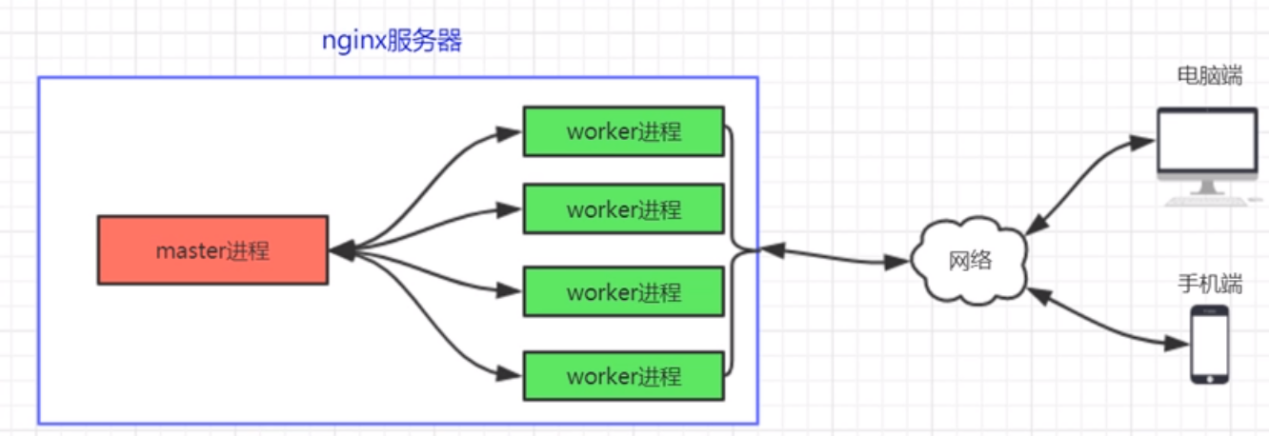
第一列：UID, 进程所属的用户id

第二列：进程ID（PID），用来唯一的标识一个进程

第三列：父进程ID（PPID）。fork(), worker进程是被master进程通过fork()创建出来的——worker进程是master进程的子进程

### nginx进程模型

1个master进程，1到多个worker进程，这种工作机制来对外服务的。这种工作机制保证了nginx能够稳定、灵活的运行。



1. master进程的责任：监控进程，不处理具体业务，专门用来管理和监控worker进程；master的角色是监工，比较清闲。
2. worker进程：用来干主要的活的，（和用户交互）。
3. master进程和worker进程之间要通讯，可以用 信号 ，也可以用 共享内存 。
4. 稳定性、灵活性的体现之一：worker进程一旦挂掉，那么master进程会立即fork()一个新的worker进程投入工作中去。

### 调整worker进程数量

worker进程几个合适呢？公认的做法：多核计算机，就让每个worker运行在一个单独的内核上，最大限度减少CPU进程切换成本，提高系统运行效率。

## nginx进程模型细说

nginx重载配置文件：sudo ./nginx -s reload

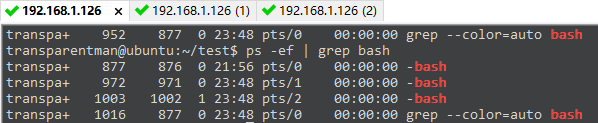
nginx能够热升级、能够热回滚

nginx的关闭：sudo ./nginx -s quit

# 终端和进程的关系

## 终端与bash进程

输入命令：ps -ef | grep bash



其中，pts是虚拟终端，每连接一个虚拟终端到linux操作系统，就会出现一个bash进程（shell[壳]），就是这个黑窗口，用来解释用户输入的命令。

bash = shell = 命令行解释器

## 终端上的开启进程

用命令查看：ps -la

随着终端的退出，这个终端上开启的正在运行的进程也退出了。

开启的可执行程序 是 bash的子进程。

## 进程关系进一步分析

查看命令：ps -eo pid,ppid,sid,tty,pgrp,comm,cmd | grep -E ‘bash|PID|test’

每个进程还属于一个进程组：一个或者多个进程的集合，每个进程组有一个唯一的进程组ID，可以调用系统函数来创建进程组、加入进程组。

“会话”（session）：是一个或者多个进程组的集合。

一般，只要不进行特殊的函数调用，一个bash（shell）上边运行的所有程序都属于一个会话，而这个会话有一个session leader；这个bash（shell）通常就是session leader；你也可以调用系统函数创建新的session。

1. 如果虚拟终端要断开的话，操作系统就会发送SIGHUP信号（终端断开信号），给session leader，一般来说就是这个bash进程。
2. bash进程受到SIGHUP信号后，bash会把这个信号发送给session里边的所有进程，收到这个SIGHUP信号的进程的缺省动作就是退出。

## strace工具的使用

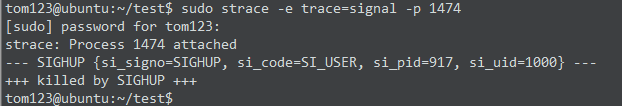
linux下调试分析诊断工具：可以跟踪程序执行时进程的系统调用以及所收到的信号；

跟踪test进程：sudo strace -e trace=signal -p 1474

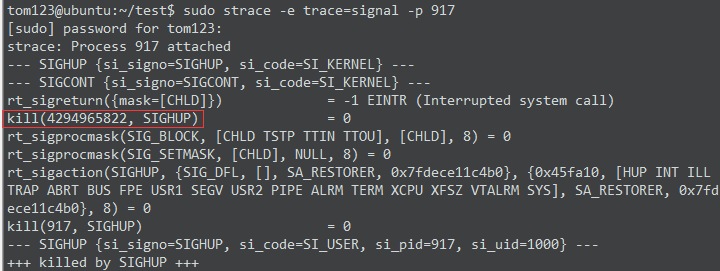
跟踪开启test的bash进程：sudo strace -e trace=signal -p 917

关闭终端后：

跟踪test输出：



跟踪bash输出：



kill(4294965822, SIGHUP) ：发送信号SIGHUP给这个-1474的绝对值所在的进程组；所以test进程就收到了SIGHUP信号。

综合来讲，这个bash先发送SIGHUP给同一个session里边的所有进程，然后再发送SIGHUP给自己。

## 终端关闭时如何让进程不退出

设想：

1. test进程拦截（test进程收到这个信号并告诉操作系统，我不想死，不要把我杀掉）SIGHUP信号，是不是可以？
2. test进程和bash进程不在同一个session里，是不是可以？

验证：

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

int *main*()

{

*printf*("test进程开始执行了\n");

// 系统函数， 设置某个信号来的时候处理程序（用哪个函数处理）

*signal*(SIGHUP ,*SIG\_IGN*); // SIG\_IGN标志：我要求忽略这个信号，请操作系统不要用缺省的处理方式来对待我

while (1) {

*printf*("休息3秒\n");

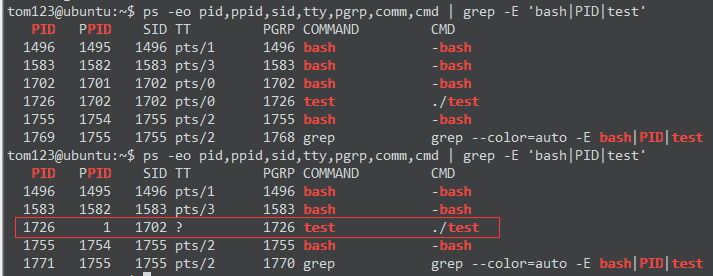
sleep(3);

}

return 0;

}

关闭终端后：



观察到开启test的bash进程1702被杀死了，但是test进程还在，而且父id变成了1，1就是Init这个老祖宗；因为本身的父进程被杀死了，test进程没被杀死，所以test进程就变成了孤儿进程，然后被Init收养了。

经过验证，设想A成立！

#include <stdio.h>

#include <unistd.h>

int main()

{

printf("test进程开始执行了\n");

pid\_t pid;

pid = fork(); // 系统函数，用来创建新进程。子进程会从fork()调用之后开始执行

if (pid < 0) {

printf("fork()进程出错！\n");

}

else if (pid == 0) {

// 子进程pid为0

printf("子进程开始执行！\n");

setsid(); // 设置新的sessionID

while (1) {

sleep(3);

printf("子进程休息3秒\n");

}

return 0;

}

else {

// 父进程会走到这里

while (1) {

sleep(3);

printf("父进程休息3秒\n");

}

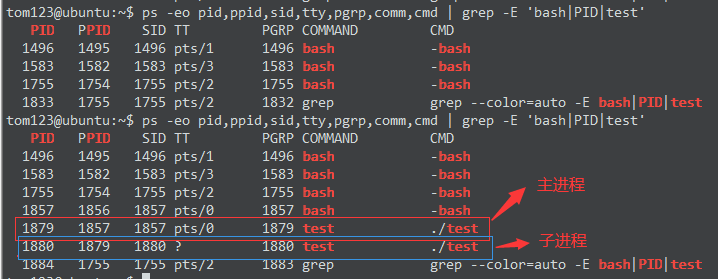
return 0;

}

return 0;

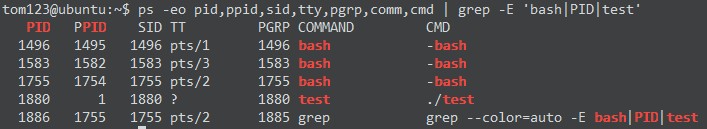
}

关闭终端前：



主进程和子进程的sessionID不一样！

关闭终端后：



主进程被杀死，子进程还在运行，因为sessionID不一样。

经过验证，设想B成立！

还有其他方法可以使终端退出时进程不退出：

1. setsid命令：启动一个进程，而且能够使启动的进程在一个新的session中，这样的话，终端关闭时该进程就不会退出。

命令：setsid ./test

1. nohup(no hang up不要挂断)，用该命令启动的进程跟上边忽略掉SIGHUP信号的道理相同，该命令会把输出内容存到当前目录下的nohup.out文件中。

命令：nohup ./test

## 后台运行 &

后台执行就是执行这个程序的同时，你的终端可以干其他事情；如果不用后台执行，那么执行这个程序后，你的终端就只能等这个程序完成后才能继续执行其他的操作。

可以用命令fg切换到前台。

# 信号的概念、认识、处理动作

## 信号的基本概念

进程间常用的通信手段：发送信号。

信号：通知（事件通知），用来通知某个进程发生了某一个事情；事情、信号都是突发事件，也就是说信号是异步发生的；信号也被称呼为“软件中断”。

信号是如何产生的呢？

1. 某个进程发送给另外一个进程或者发送给自己（比如说热更新时，新的进程给老的进程发信号）；
2. 由内核（操作系统）发送个某个进程：
   1. 通过在键盘输入命令ctrl+c[中断信号]，kill命令。
   2. 内存访问异常，除数为0等等...硬件都会检测到并且通知内核；

信号的名字，都是以SIG开头；

UNIX以及类UNIX操作系统支持的信号数量各不相同（·10~60多个之间）；

信号是宏定义（数字，从1开始的正整数常量）；

在/目录下查找文件内容中关键词的命令：

sudo find / -name “signal.h” | xargs grep -in “SIGHUP”

## 通过kill命令认识一些信号

kill：kill 进程id，它的工作是给进程发个信号；

kill能给进程发送多种信号；

1. 如果单纯的用kill进程id，那么就是往这个进程发送SIGTERM信号（终止信号）
2. 如果用kill -数字 进程id，那么就能发出跟这个数字对应的信号。

kill 命令不同数字所能发出的不同信号

|  |  |  |
| --- | --- | --- |
| kill的参数 | 该参数发出的信号 | 操作系统缺省动作 |
| -1 | SIGHUP（连接断开） | 终止掉进程（进程没了） |
| -2 | SIGINT（终端中断符，比如ctrl+c） | 终止掉进程（进程没了） |
| -3 | SIGQUIT（终端退出符，比如ctrl+\） | 终止掉进程（进程没了） |
| -9 | **SIGKILL（终止）** | 终止掉进程（进程没了） |
| -18 | SIGCONT（使暂停的进程继续） | 忽略（进程依旧在运行不受影响） |
| -19 | **SIGSTOP（停止），可用SIGCONT继续，但任务被放到了后台** | 停止进程（**不是终止，进程还在**） |
| -20 | SIGTSTP（终端停止符，比如ctrl+z），但任务被放到了后台，可用SIGCONT继续 | 停止进程（**不是终止，进程还在**） |

## 进程的状态

ps -eo pid,ppid,sid,tty,pgrp,comm,stat | grep -E ‘bash|PID|test’

ps aux | grep -E ‘bash|PID|test’ // aux是BSD风格的显示格式

进程状态：

|  |  |
| --- | --- |
| 状态 | 含义 |
| D | 不可中断的休眠状态(通常是I/O的进程)，可以处理信号，有 延迟 |
| R | 可执行状态&运行状态(在运行队列里的状态) |
| S | 可中断的休眠状态之中（等待某事件完成），可以处理信号 |
| T | 停止或被追踪（被作业控制信号所停止） |
| Z | 僵尸进程 |
| X | 死掉的进程 |
| < | 高优先级的进程 |
| N | 低优先级的进程 |
| L | 有些页被锁进内存 |
| s | Session leader（进程的领导者），在它下面有子进程 |
| t | 追踪期间被调试器所停止 |
| + | 位于前台的进程组 |

kill只是发个信号，而不是单纯的杀死的意思。

## 常用信号列举：

|  |  |
| --- | --- |
| 信号名 | 信号含义 |
| SIGHUP（连接断开） | 是终端断开信号，如果终端接口检测到一个连接断开，发送此信号到该终端所在的会话首进程（前面讲过），缺省动作会导致所有相关的进程退出(上节课也重点讲了这个信号，xshell断开就有这个信号送过来)；  Kill -1 进程号也能发送此信号给进程； |
| SIGALRM（定时器超时） | 一般调用系统函数alarm创建定时器，定时器超时了就会这个信号； |
| SIGINT（中断） | 从键盘上输入ctrl+C（中断键）【比如你进程正跑着循环干一个事】，这一ctrl+C就能打断你干的事，终止进程；  **但shell会将后台进程对该信号的处理设置为忽略（也就是说该进程若在后台运行则不会收到该信号）**； |
| SIGSEGV（无效内存） | 内存访问异常，除数为0等，硬件会检测到并通知内核；其实这个SEGV代表段违例（segmentation violation），你有的时候运行一个你编译出来的可执行的c程序，如果内存有问题，执行的时候就会出现这个提示； |
| SIGIO（异步I/O） | 通用异步I/O信号，咱们以后学通讯的时候，如果通讯套接口上有数据到达，或发生一些异步错误，内核就会通知我们这个信号； |
| SIGCHLD（子进程改变） | 一个进程终止或者停止时，这个信号会被发送给父进程；（我们想象下nginx，worker进程终止时 master进程应该会收到内核发出的针对该信号的通知）； |
| SIGUSR1,SIGUSR2（都是用户定义信号） | 用户定义的信号，可用于应用程序，用到再说； |
| SIGTERM（终止） | 一般你通过在命令行上输入kill命令来杀一个进程的时候就会触发这个信号，收到这个信号后，你有机会退出前的处理，实现这种所谓优雅退出的效果； |
| SIGKILL（终止） | 不能被忽略，这是杀死任意进程的可靠方法，不能被进程本身捕捉 |
| SIGSTOP（停止） | 不能被忽略，使进程停止运行，可以用SIGCONT继续运行，但进程被放入到了后台 |
| SIGQUIT（终端退出符） | 从键盘上按ctrl+\  **但shell会将后台进程对该信号的处理设置为忽略（也就是说该进程若在后台运行则不会收到该信号）**； |
| SIGCONT（使暂停进程继续） | 使暂停的进程继续运行 |
| SIGTSTP（终端停止符） | 从键盘上按ctrl+z，进程被停止，并被放入后台，可以用SIGCONT继续运行 |

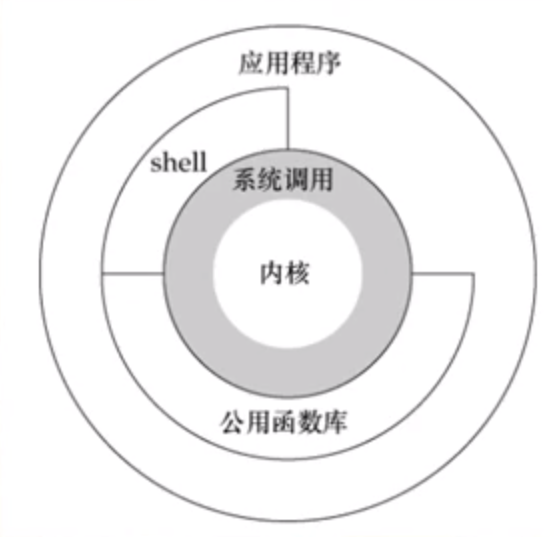
## 信号处理的相关动作

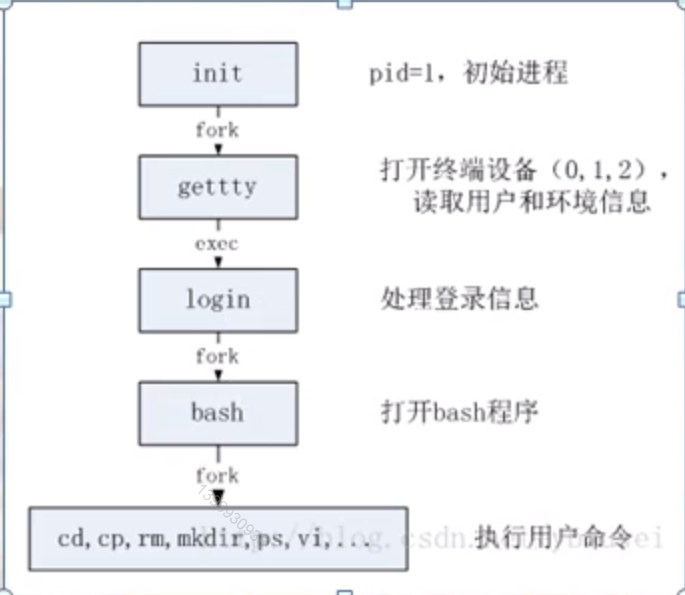
当某个信号出现时，我们可以按三种方式之一进行处理，我们称之为信号的处理或者与信号相关的动作。

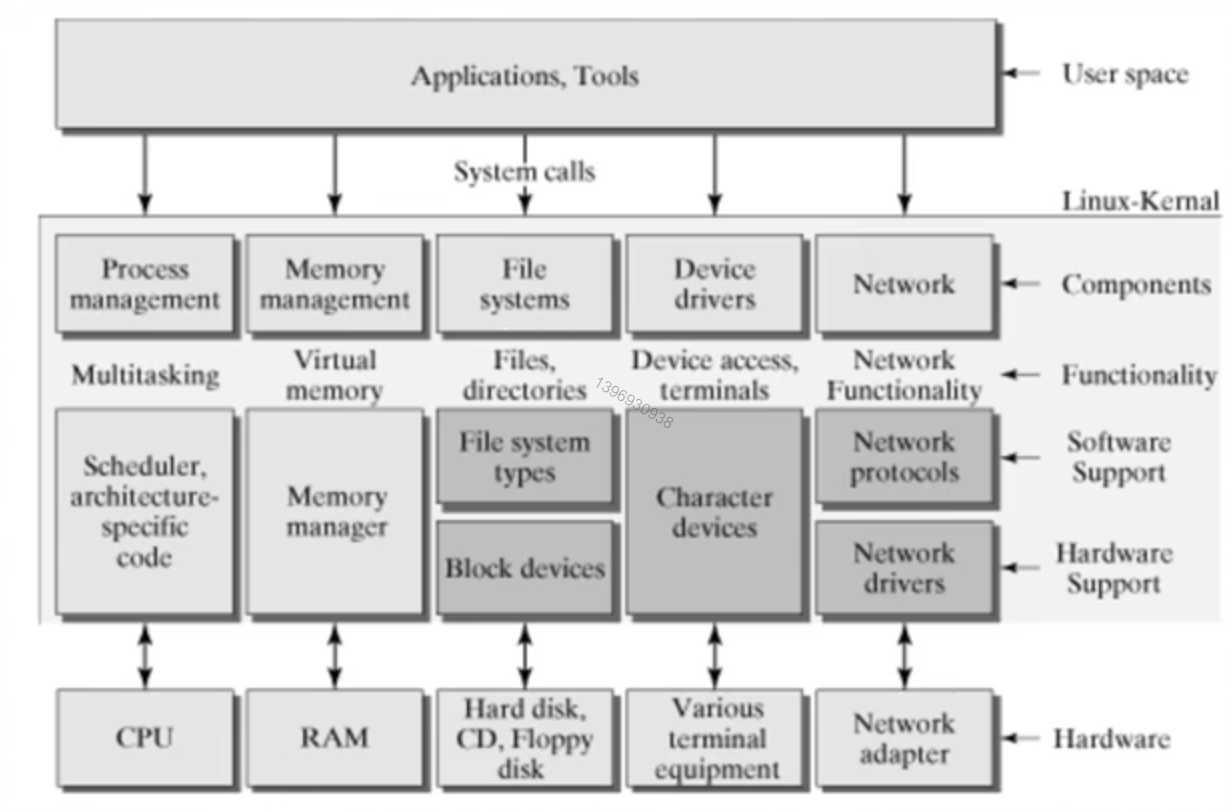
1. 执行系统默认动作，绝大多数信号的默认动作是杀死你这个进程。
2. 忽略此信号（但是不包括SIGKILL和SIGSTOP）
3. 捕捉该信号：我写个处理函数，信号来的时候，我就用处理函数去处理；（但是不包括SIGKILL和SIGSTOP）

# Unix/Linux体系结构 、信号编程初步

## Unix/Linux操作系统体系结构







类Unix操作系统体系结构分为两个状态：用户态 和 内核态

1. 操作系统/内核：用来控制计算机硬件资源，提供应用程序运行的环境；

我们写的程序，要么运行在用户态，要么运行在内核态。一般运行在用户态；

换个角度理解：用户态就是最外圈应用程序的活动空间；

1. 系统调用：就是一些函数（系统函数），只需要调用这些函数就可以了；
2. shell： bash(borne again shell[重新装配的shell])，bash是shell的一种，linux上默认采用的是bash这种shell；

通俗一点理解，bash是一个可执行程序，主要作用是 把用户输入的命令翻译给操作系统（命令解释器）;

shell分隔系统调用和引用程序，有胶水的感觉。

1. 用户态、内核态之间的切换

运行于用户态的进程可以执行的操作和访问的资源会受到极大限制（用户态权限小）；

而运行在内核态的进程可以执行任何操作并且在资源的使用上没有限制（内核态权限大）；

一个进程执行的时候，大部分时间是处于用户态下的，只有需要内核所提供的服务时才会切换到内核态，内核态做的事情完成后，又转回用户态；

比如说malloc() printf()：这种状态的转换时操作系统干的，不需要我们介入；

疑问：为什么要区分用户态、内核态？

大概有两个目的：

1. 一般情况下，程序都运行在用户态状态，权限小，不至于危害到系统的其他部分；当你要干一些危险的事情的时候，系统给你提供接口，让你去干。
2. 既然这些接口是系统提供给你的，那么这些接口也是操作系统统一管理的。

因为资源是有限的，如果大家都来访问这些资源，又不加以管理，问题一个是访问冲突，一个是被访问的资源如果耗尽，那么系统还可能崩溃；

因此系统提供这些接口，就是为了减少有限的资源的访问以及使用上的冲突。

那么什么时候从用户态切换到内核态去呢？

1. 系统调用，比如调用malloc();
2. 异常事件，比如来了个信号；
3. 外围设备中断；

总结起来，只需要知道用户态、内核态 两者根据需要自动切换即可。

## signal函数范例

信号来了之后，我们可以用signal函数来忽略或者捕捉。

### 可重入函数

可重入函数就是我们在信号处理函数中调用它是安全的。

可重入函数：在信号处理程序中保证调用安全的函数，这些函数是可重入的并被称为异步信号安全的。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <errno.h>

int g\_mysign = 0;

void muNEfunc(int value) //我这个函数能够修改这个全局变量g\_mysign的值

{

//.....其他处理代码

g\_mysign = value;

//.....其他处理代码

}

//信号处理函数

void sig\_usr(int signo)

{

//int tmpsign = g\_mysign;

//muNEfunc(22); //因为一些实际需求必须要在sig\_user这个信号处理函数里调用muNEfunc

int myerrno = errno; // 将errno的值保存下来，防止被更改了

if (signo == SIGUSR1)

{

printf("收到了SIGUSR1信号!\n"); // 商业环境中不可以这么写，因为printf是不可重入函数

}

else if (signo == SIGUSR2)

{

printf("收到了SIGUSR2信号!\n");

}

else

{

printf("收到了未捕捉的信号%d!\n", signo);

}

//g\_mysign = tmpsign;

errno = myerrno; // 还原errno的值

}

int main(int argc, char\* const\* argv)

{

if (signal(SIGUSR1, sig\_usr) == SIG\_ERR) //系统函数，参数1：是个信号，参数2：是个函数指针，代表一个针对该信号的捕捉处理函数

{

printf("无法捕捉SIGUSR1信号!\n");

}

if (signal(SIGUSR2, sig\_usr) == SIG\_ERR)

{

printf("无法捕捉SIGUSR2信号!\n");

}

for (;;)

{

sleep(1); //休息1秒

printf("休息1秒\n");

muNEfunc(15);

printf("g\_mysign=%d\n", g\_mysign);

//拿g\_mysign做一些其他用途；

}

printf("再见!\n");

return 0;

}

在写信号处理函数的时候，要注意的事项：

1. 在信号处理函数中，尽量使用简单的语句做简单的事情，尽量不要调用系统函数以免引起麻烦；
2. 如果必须要在信号处理函数中调用一些系统函数，那么要保证在信号处理函数中调用的系统函数一定要是可重入的；
3. 如果必须要在信号处理函数中调用那些可能修改errno值的可重入的系统函数，那么就得事先备份errno的值，在信号处理函数返回之前，将errno的值恢复；

### 不可重入函数的错误演示

一旦在信号处理函数中用了不可重入函数，可能导致程序错乱，不正常...

#include <stdio.h>

#include <stdlib.h> //malloc

#include <unistd.h>

#include <signal.h>

//信号处理函数

void sig\_usr(int signo)

{

//这里也malloc，这是错用，不可重入函数不能用在信号处理函数中；

int\* p;

p = (int \*) malloc (sizeof(int)); //用了不可重入函数；

free(p);

if (signo == SIGUSR1)

{

printf("收到了SIGUSR1信号!\n");

}

else if (signo == SIGUSR2)

{

printf("收到了SIGUSR2信号!\n");

}

else

{

printf("收到了未捕捉的信号%d!\n", signo);

}

}

int main(int argc, char\* const\* argv)

{

if (signal(SIGUSR1, sig\_usr) == SIG\_ERR)

{

printf("无法捕捉SIGUSR1信号!\n");

}

if (signal(SIGUSR2, sig\_usr) == SIG\_ERR)

{

printf("无法捕捉SIGUSR2信号!\n");

}

for (;;)

{

int\* p;

p = (int\*)malloc(sizeof(int));

free(p);

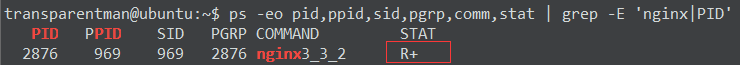
}

printf("再见!\n");

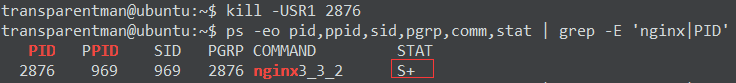
return 0;

}

刚启动时的状态：



发送信号后：



nginx3\_3\_2进程窗口没有任何反应，它没有处理信号，而且已经休眠了。

signal函数因为兼容性，可靠性等等一些历史问题，不建议使用（我们的策略就是坚决不用），建议使用sigaction()函数代替。

# 信号编程进阶、sigprocmask范例

## 信号集

如果信号处理函数还没执行完，又收到了这个信号，进程就会将这个信号阻塞或者忽略。

一个进程，必须能够记住我这个进程当前阻塞了哪些信号。

我们需要“信号集”这么一种数据结构，把这60多个信号是否阻塞的状态都表示出来。

0000000000，0000000000，0000000000，00，0000000000，0000000000，0000000000，00

linux中是用sigset\_t结构类型来表示信号集的；

typedef struct {

unsigned long sig[2]; // 一个long 4个字节，32位，两个long就是64位

} sigset\_t;

信号集的定义：信号集表示一组信号的来（1）或者没来（0）。

信号集相关的数据类型：sigset\_t

## 信号相关函数

1. sigemptyset()：把信号集中的所有信号都清0，清0后就表示这60多个信号都没有来。
2. sigfillset()：把信号集中的所有信号都设置为1，跟sigemptyset()正好相反。
3. 用sigaddset()，sigdelset()就可以往信号集中增加屏蔽信号，或者从信号集中删除特定的信号。
4. sigprocmask，sigismember

一个进程里边会有一个信号集，用来记录当前屏蔽（阻塞）了哪些信号；

如果我们把这个信号集 中的某个信号位设置为1，就表示屏蔽了同类信号，此时再来个同类信号，那么同类信号就会被屏蔽，不能传递给进程；

如果这个信号集 中有很多个信号位都被设置为1，那么所有这些被设置为1的信号都是属于当前被阻塞的而不能传递到该进程的信号；

sigprocmask()函数，就能够设置该进程所对应的信号集中的内容。

## sigprocmask等信号函数范例演示

#include <stdio.h>

#include <stdlib.h> // malloc

#include <unistd.h>

#include <signal.h>

// 信号处理函数

void sig\_quit(int signo)

{

printf("收到了SIGQUIT信号!\n");

if (signal(SIGQUIT, SIG\_DFL) == SIG\_ERR)

{

printf("无法为SIGQUIT信号设置缺省处理(终止进程)!\n");

exit(1);

}

}

int main(int argc, char\* const\* argv)

{

sigset\_t newmask, oldmask; // 信号集，新的信号集，原有的信号集

if (signal(SIGQUIT, sig\_quit) == SIG\_ERR) // 注册信号对应的信号处理函数,"ctrl+\"

{

printf("无法捕捉SIGQUIT信号!\n");

exit(1); // 退出程序，参数是错误代码，0表示正常退出，非0表示错误，但具体什么错误，没有特别规定，这个错误代码一般也用不到，先不管他；

}

sigemptyset(&newmask); // newmask信号集中所有信号都清0（表示这些信号都没有来）；

sigaddset(&newmask, SIGQUIT); // 设置newmask信号集中的SIGQUIT信号位为1，说白了，再来SIGQUIT信号时，进程就收不到，设置为1就是该信号被阻塞掉呗

// sigprocmask()：设置该进程所对应的信号集

if (sigprocmask(SIG\_BLOCK, &newmask, &oldmask) < 0) // 第一个参数用了SIG\_BLOCK表明设置 进程 新的信号屏蔽字 为 “当前信号屏蔽字 和 第二个参数指向的信号集的并集

{ // 一个 ”进程“ 的当前信号屏蔽字，刚开始全部都是0的；所以相当于把当前 "进程"的信号屏蔽字设置成 newmask（屏蔽了SIGQUIT)；

// 第三个参数不为空，则进程老的(调用本sigprocmask()之前的)信号集会保存到第三个参数里，用于后续，这样后续可以恢复老的信号集给线程

printf("sigprocmask(SIG\_BLOCK)失败!\n");

exit(1);

}

printf("我要开始休息10秒了--------begin--，此时我无法接收SIGQUIT信号!\n");

sleep(10); // 这个期间无法收到SIGQUIT信号的；

printf("我已经休息了10秒了--------end----!\n");

if (sigismember(&newmask, SIGQUIT)) // 测试一个指定的信号位是否被置位(为1)，测试的是newmask

{

printf("SIGQUIT信号被屏蔽了!\n");

}

else

{

printf("SIGQUIT信号没有被屏蔽!!!!!!\n");

}

if (sigismember(&newmask, SIGHUP)) // 测试另外一个指定的信号位是否被置位,测试的是newmask

{

printf("SIGHUP信号被屏蔽了!\n");

}

else

{

printf("SIGHUP信号没有被屏蔽!!!!!!\n");

}

// 现在我要取消对SIGQUIT信号的屏蔽(阻塞)--把信号集还原回去

if (sigprocmask(SIG\_SETMASK, &oldmask, NULL) < 0) // 第一个参数用了SIGSETMASK表明设置 进程 新的信号屏蔽字为 第二个参数 指向的信号集，第三个参数没用

{

printf("sigprocmask(SIG\_SETMASK)失败!\n");

exit(1);

}

printf("sigprocmask(SIG\_SETMASK)成功!\n");

if (sigismember(&oldmask, SIGQUIT)) // 测试一个指定的信号位是否被置位,这里测试的当然是oldmask

{

printf("SIGQUIT信号被屏蔽了!\n");

}

else

{

printf("SIGQUIT信号没有被屏蔽，您可以发送SIGQUIT信号了，我要sleep(10)秒钟!!!!!!\n");

int mysl = sleep(10); // sleep函数返回未睡够的时间，如果被打断时(来了某个信号)睡了3秒，则返回7

if (mysl > 0)

{

printf("sleep还没睡够，剩余%d秒\n", mysl);

}

}

printf("再见了!\n");

return 0;

}

# fork函数详解、范例演示

## fork()函数简单认识

fork()函数是用来创建进程的。

进程的概念：一个可执行程序，执行起来就是一个进程，再执行起来一次，它就又是一个进程（多个进程可以共享同一个可执行文件）；文雅的说法：进程定义为程序执行的一个实例。

在一个进程（程序）中，可以用fork()创建一个子进程，当该子进程创建时，它从fork()指令的下一条（或者说从fork()的返回处）开始执行与父进程相同的代码。

说白了：fork()函数产生了一个和当前进程完全一样的新进程，并和当前进程一样从fork()函数里返回。

fork()：一分二

### fork()函数简单范例

#include <stdio.h>

#include <stdlib.h> //malloc,exit

#include <unistd.h> //fork

#include <signal.h>

//信号处理函数

void sig\_usr(int signo)

{

printf("收到了SIGUSR1信号，进程id=%d!\n", getpid());

}

int main(int argc, char\* const\* argv)

{

pid\_t pid;

printf("进程开始执行!\n");

//先简单处理一个信号

if (signal(SIGUSR1, sig\_usr) == SIG\_ERR)

{

printf("无法捕捉SIGUSR1信号!\n");

exit(1);

}

//---------------------------------

pid = fork(); //创建一个子进程

//要判断子进程是否创建成功

if (pid < 0)

{

printf("子进程创建失败，很遗憾!\n");

exit(1);

}

//现在，父进程和子进程同时开始 运行了

for (;;)

{

sleep(1); //休息1秒

printf("休息1秒，进程id=%d!\n", getpid());

}

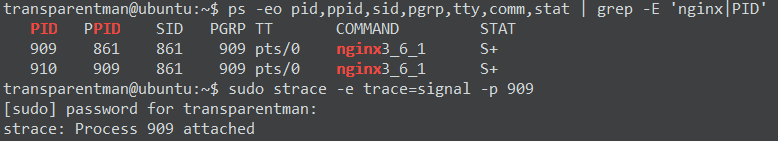
printf("再见了!\n");

return 0;

}

fork()之后，是父进程fork()之后的代码先执行还是子进程fork()之后的代码先执行是不一定的，这个跟内核调度算法有关。

我们kill掉子进程，观察父进程收到什么信号？

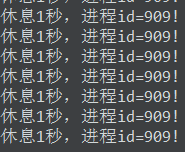




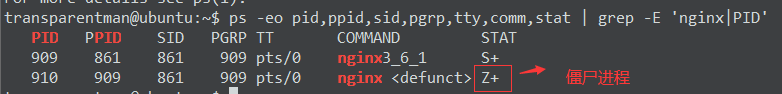
观察到 收到了SIGCHLD信号：



此时子进程已经不运行了：



再次查看进程状态：



### 僵尸进程的产生、解决，SIGCHLD

僵尸进程的产生：在Unix系统中，一个子进程结束了，但是他的父进程还活着，但该父进程没有调用（wait/waitpid）之类的函数来进行额外的处置，那么这个子进程就会变成一个僵尸进程。

僵尸进程：已经被终止，它不干活了，但是依旧没有被内核丢弃掉，因为内核认为父进程可能还需要该子进程的一些信息。

作为开发者，坚决不允许僵尸进程的存在。

那么如何干掉僵尸进程？

1. 重启电脑
2. 手动把僵尸进程的父进程kill掉，僵尸进程就会自动消失
3. 利用SIGCHLD信号：一个进程被终止或者停止时，这个信号会被发送给父进程，所以，对于源码中有fork()行为的进程，我们应该拦截并处理SIGCHLD信号。

#include <stdio.h>

#include <stdlib.h> //malloc,exit

#include <unistd.h> //fork

#include <signal.h>

#include <sys/wait.h> //waitpid

//信号处理函数

void sig\_usr(int signo)

{

int status;

switch (signo)

{

case SIGUSR1:

printf("收到了SIGUSR1信号，进程id=%d!\n", getpid());

break;

case SIGCHLD:

printf("收到了SIGCHLD信号，进程id=%d!\n", getpid());

//这里大家学了一个新函数waitpid，有人也用wait,但老师要求大家掌握和使用waitpid即可；

//这个waitpid说白了获取子进程的终止状态，这样，子进程就不会成为僵尸进程了；

pid\_t pid = waitpid(-1, &status, WNOHANG); //第一个参数为-1，表示等待任何子进程，

//第二个参数：保存子进程的状态信息(大家如果想详细了解，可以百度一下)。

//第三个参数：提供额外选项，WNOHANG表示不要阻塞，让这个waitpid()立即返回

if (pid == 0) //子进程没结束，会立即返回这个数字，但这里应该不是这个数字

return;

if (pid == -1) //这表示这个waitpid调用有错误，有错误也理解返回出去，我们管不了这么多

return;

//走到这里，表示 成功，那也return吧

return;

break;

} //end switch

}

int main(int argc, char\* const\* argv)

{

pid\_t pid;

printf("进程开始执行!\n");

if (signal(SIGUSR1, sig\_usr) == SIG\_ERR)

{

printf("无法捕捉SIGUSR1信号!\n");

exit(1);

}

if (signal(SIGCHLD, sig\_usr) == SIG\_ERR)

{

printf("无法捕捉SIGCHLD信号!\n");

exit(1);

}

//---------------------------------

pid = fork();

if (pid < 0)

{

printf("子进程创建失败，很遗憾!\n");

exit(1);

}

for (;;)

{

sleep(1);

printf("休息1秒，进程id=%d!\n", getpid());

}

printf("再见了!\n");

return 0;

}

## fork()函数进一步认识

fork()产生新进程的速度非常快，fork()产生的新进程并不复制原进程的内存空间，而是和原来的进程（父进程）一起共享一个内存空间，但这个内存空间的特性是“写时复制”，也就是说：原来的进程和fork()出来的子进程可以同时、自由的读取内存，但如果子进程或者父进程对内存进行修改的话，那么这个内存就会复制一份给该进程单独使用，以免影响到共享这个内存空间的其他进程使用。

#include <stdio.h>

#include <stdlib.h> //malloc,exit

#include <unistd.h> //fork

#include <signal.h>

int main(int argc, char\* const\* argv)

{

//一般fork都会成功所以不判断返回值了,我们假定成功

fork(); // 一分二

fork(); // 二分四

for (;;)

{

sleep(1); //休息1秒

printf("休息1秒，进程id=%d!\n", getpid());

}

printf("再见了!\n");

return 0;

}

最后会生成4个进程。

## 完善一下fork()代码

fork()会返回两次：父进程中返回一次，子进程中返回一次，而且，fork()在父进程中返回的值和在子进程中返回的值是不同的：

#include <stdio.h>

#include <stdlib.h> //malloc,exit

#include <unistd.h> //fork

#include <signal.h>

int g\_mygbltest = 0;

int main(int argc, char\* const\* argv)

{

pid\_t pid;

printf("进程开始执行!\n");

//---------------------------------

pid = fork(); //创建一个子进程

//要判断子进程是否创建成功

if (pid < 0)

{

printf("子进程创建失败，很遗憾!\n");

exit(1);

}

//走到这里，fork()成功，执行后续代码的可能是父进程，也可能是子进程

if (pid == 0)

{

//子进程，因为子进程的fork()返回值会是0；

//这里专门针对子进程的处理代码

while (1)

{

g\_mygbltest++;

sleep(1); //休息1秒

printf("真是太高兴了，我是子进程的，我的进程id=%d,g\_mygbltest=%d!\n", getpid(), g\_mygbltest);

}

}

else

{

//这里就是父进程，因为父进程的fork()返回值会 > 0（实际返回的是子进id程）

//这是专门针对父进程的处理代码

while (1)

{

g\_mygbltest++;

sleep(5); //休息5秒

printf("......。。，我是父进程的，我的进程id=%d,g\_mygbltest=%d!\n", getpid(), g\_mygbltest);

}

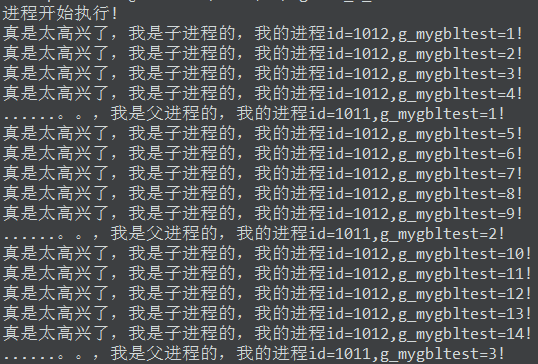
}

return 0;

}

子进程的fork()返回值是0；

父进程的fork()返回值是新建立的子进程的ID，因为全局量g\_mygbltest的值发生改变，导致主、子进程内存被单独的分开，所以每个的g\_mygbltest值也不同。



## fork()失败的可能性

1. 系统中进程太多

缺省情况，最大的pid：32767

1. 每个用户有个允许开启的进程总数：

printf("每个实际用户ID的最大进程数=%ld\n", sysconf(\_SC\_CHILD\_MAX));

# 守护进程详解、范例演示

## 普通进程运行观察

1. 进程有对应的终端，如果终端退出，那么对应的进程也就消失了；它的父进程是一个bash。
2. 终端被占住了，输入各种命令都没有反应。

## 守护进程基本概念

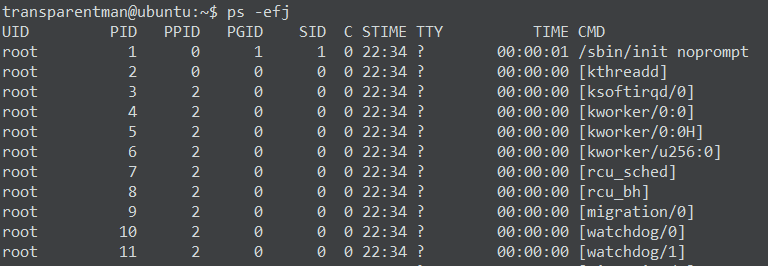
守护进程：一种长期运行的进程，这种进程在后台运行，并且不跟任何的控制终端关联。

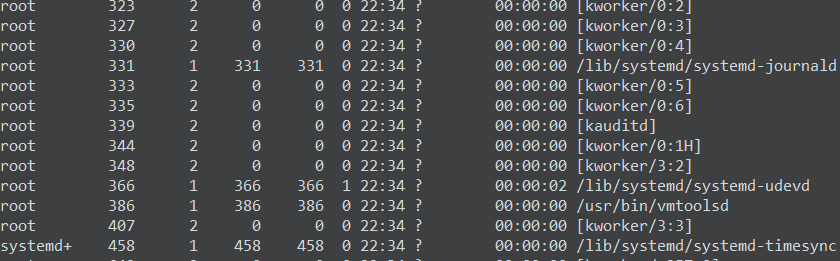
基本特点：

1. 生存期长[不是必须，但一般应该这样做]，一般是操作系统启动的时候它就启动，操作系统关闭的时候它才关闭；
2. 守护进程跟终端无关联，也就是说它们没有控制终端，所以你控制终端退出，也不会导致守护进程退出；
3. 守护进程是在后台运行，不会占用终端，终端可以执行其它命令；

linux操作系统本身有很多的守护进程在默默的运行，维持着系统的日常活动。大概30-50个。

-j 显示与任务或者作业有关的信息





1. ppid = 0：内核进程，跟随系统启动而启动，生命周期贯穿整个系统；
2. cmd列名字带[]这种，叫内核守护进程；
3. 老祖宗init：也是系统守护进程，它负责启动各运行层次特定的系统服务，所以很多进程的PPID是init。而且这个init也负责收养孤儿进程。
4. cmd列中名字不带[]的叫普通守护进程（用户级守护进程）

共同点总结：

1. 大多数守护进程都是以超级用户特权运行的；
2. 守护进程没有控制终端，TT这列显示“?”

内核守护进程以无控制终端方式启动，普通守护进程可能是守护进程调用了setsid的结果（无控制端）；

## 守护进程编写规则

1. 调用umask(0)：umask是个函数，用来限制（屏蔽）一些文件权限的。
2. fork()一个子进程出来（目的是脱离终端），然后父进程退出（目的是把终端空出来，不让终端卡住）；固定套路。

fork()的目的是想成功调用setsid()来建立新会话，目的是：让子进程有单独的sid；而且子进程也成为了一个新进程组的组长进程，同时，子进程不关联任何终端了。

-----------讲解一些概念-----------

### 文件描述符

是个正数，用来标识一个文件。

当你打开一个存在的文件或者创建一个新文件，操作系统都会返回这个文件描述符（其实就是代表这个文件的），后续对这个文件操作的一些函数，都会用到这个文件描述符作为参数。

linux中三个特殊的文件描述符，数字分别是0，1，2

0：标准输入【键盘】，对应的符号常量叫STDIN\_FILENO

1：标准输出【屏幕】，对应的符号常量叫STDOUT\_FILENO

2：标准错误【屏幕】，对应的符号常量叫STDERR\_FILENO

类Unix操作系统，默认从STDIN\_FILENO读数据，向STDOUT\_FILENO来写数据，向STDERR\_FILENO来写错误。

类Unix操作系统有个说法：一切皆文件，所以它把标准输入、标准输出、标准错误都看成文件。

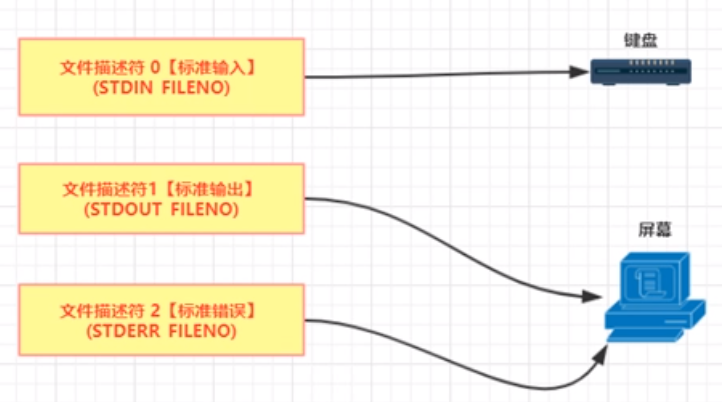
与其说把 标准输入、标准输出、标准错误 都看成文件，倒不如说：

像看待文件一样看待 标准输入、标准输出、标准错误；

像操作文件一样操作 标准输入、标准输出、标准错误；

同时，你的程序一旦运行起来，这三个文件描述符0，1，2会被自动打开（自动只想对应的设备）。

文件描述符虽然是数字，但是，如果我们把文件描述符直接理解成指针（指针里边保存的是地址，说白了也是个数字），会更好理解一些。



### 输入输出重定向

输出重定向：标准输出文件描述符，不指向屏幕了，假如我指向（重定向）一个文件。在命令行中用>即可。

输入重定向：在命令行中用<即可。

范例：cat < myinfile > myoutfile 从myinfile中读，写到myoutfile中去

### 空设备

/dev/null : 是一个特殊的设备文件，它丢弃一切写入其中的数据（像黑洞一样）

---------------------------------

守护进程虽然可以通过终端启动，但是和终端不挂钩。

守护进程是在后台运行，它不应该从键盘上接收任何东西，也不应该把输出结果打印到屏幕或者终端上来；所以，一般按照江湖规矩，我们要把守护进程的 标准输入，标准输出，重定向到 空设备（黑洞），从而确保守护进程不从键盘接收任何东西，也不把输出结果打印到屏幕。

int fd;

fd = open("/dev/null", O\_RDWR); // 打开空设备

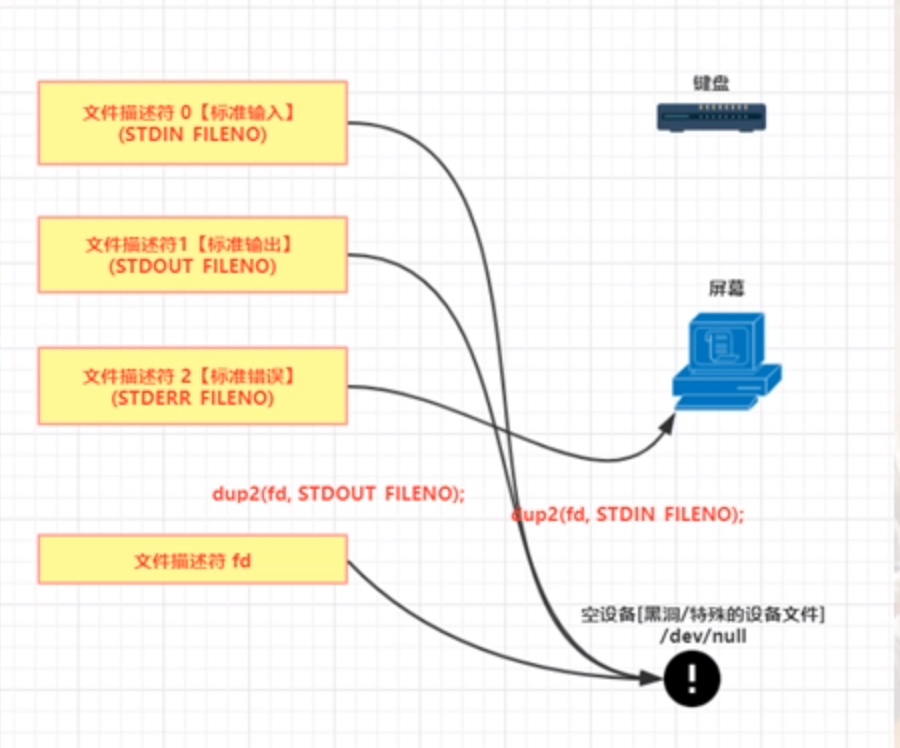
dup2(fd, STDIN\_FILENO); // 复制文件描述符，像个指针赋值，把第一个参数指向的内容赋给了第二个参数

dup2(fd, STDOUT\_FILENO);

if (fd > STDERR\_FILENO) {

close(fd); // 等价于fd = null;

}



### 实现范例

#include <stdio.h>

#include <stdlib.h> //malloc

#include <unistd.h>

#include <signal.h>

#include <sys/stat.h>

#include <fcntl.h>

//创建守护进程

//创建成功则返回1，否则返回-1

int ngx\_daemon()

{

int fd;

switch (fork()) //fork()子进程

{

case -1:

//创建子进程失败，这里可以写日志......

return -1;

case 0:

//子进程，走到这里，直接break;

break;

default:

//父进程，直接退出

exit(0);

}

//只有子进程流程才能走到这里

if (setsid() == -1) //脱离终端，终端关闭，将跟此子进程无关

{

//记录错误日志......

return -1;

}

umask(0); //设置为0，不要让它来限制文件权限，以免引起混乱

fd = open("/dev/null", O\_RDWR); //打开黑洞设备，以读写方式打开

if (fd == -1)

{

//记录错误日志......

return -1;

}

if (dup2(fd, STDIN\_FILENO) == -1) //先关闭STDIN\_FILENO[这是规矩，已经打开的描述符，动他之前，先close]，类似于指针指向null，让/dev/null成为标准输入；

{

//记录错误日志......

return -1;

}

if (dup2(fd, STDOUT\_FILENO) == -1) //先关闭STDOUT\_FILENO，类似于指针指向null，让/dev/null成为标准输出；

{

//记录错误日志......

return -1;

}

if (fd > STDERR\_FILENO) //fd应该是3，这个应该成立

{

if (close(fd) == -1) //释放资源这样这个文件描述符就可以被复用；不然这个数字【文件描述符】会被一直占着；

{

//记录错误日志......

return -1;

}

}

return 1;

}

int main(int argc, char\* const\* argv)

{

if (ngx\_daemon() != 1)

{

//创建守护进程失败，可以做失败后的处理比如写日志等等

return 1;

}

else

{

//创建守护进程成功,执行守护进程中要干的活

for (;;)

{

sleep(1); //休息1秒

printf("休息1秒，进程id=%d!\n", getpid()); //你就算打印也没用，现在标准输出指向黑洞（/dev/null），打印不出任何结果【不显示任何结果】

}

}

return 0;

}

守护进程可以用命令启动，如果想开机启动，则需要借助系统初始化脚本来启动

## 守护进程不会收到的信号

内核发给你的，另外的进程发给你的

### SIGHUP信号

守护进程不会收到来自内核的SIGHUP信号；

潜台词就是如果守护进程收到了SIGHUP信号，那么肯定是另外的进程发给你的；

很多守护进程把这个信号作为通知信号，表示配置文件已经发生改动，守护进程应该重新读入其配置文件，比如nginx。

### SIGINT、SIGWINCH信号

守护进程不会收到来自内核的SIGINT(ctrl + C)，SIGWINCH（终端窗口大小改变）信号。

## 守护进程和后台进程的区别

1. 守护进程和终端不挂钩，后台进程和终端挂钩（能往终端上输出东西）。
2. 关闭终端时守护进程不受影响，后台进程会随着终端的退出而退出。
3. ......其他的细微区别。

# 服务器程序目录规划、makefile编写

## 信号高级认识范例

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <errno.h>

//信号处理函数

void sig\_usr(int signo)

{

if (signo == SIGUSR1)

{

printf("收到了SIGUSR1信号，我休息10秒......!\n");

sleep(10);

printf("收到了SIGUSR1信号，我休息10秒完毕，苏醒了......!\n");

}

else if (signo == SIGUSR2)

{

printf("收到了SIGUSR2信号，我休息10秒......!\n");

sleep(10);

printf("收到了SIGUSR2信号，我休息10秒完毕，苏醒了......!\n");

}

else

{

printf("收到了未捕捉的信号%d!\n", signo);

}

}

int main(int argc, char\* const\* argv)

{

if (signal(SIGUSR1, sig\_usr) == SIG\_ERR) //系统函数，参数1：是个信号，参数2：是个函数指针，代表一个针对该信号的捕捉处理函数

{

printf("无法捕捉SIGUSR1信号!\n");

}

if (signal(SIGUSR2, sig\_usr) == SIG\_ERR)

{

printf("无法捕捉SIGUSR2信号!\n");

}

for (;;)

{

sleep(1); //休息1秒

printf("休息1秒~~~~!\n");

}

printf("再见!\n");

return 0;

}

用kill发送USR1信号给进程

1. 执行信号处理函数被卡住了10秒，这个时候因为流程回不到main()，所以main中的语句无法得到执行；
2. 在触发了SIGUSR1信号并因此sleep了10秒钟期间，就算你多次触发SIGUSR1信号，也不会重新执行SIGUSR1信号对应的信号处理函数，而是会等待上一个SIGUSR1信号处理函数执行完毕才第二次执行SIGUSR1信号处理函数；

换句话说：在信号处理函数被调用时，操作系统建立的新信号屏蔽字（sigprocmask()），自动包括了正在被递送的信号，因此，保证了在处理一个给定信号的时候，如果这个信号再次发生，那么它会阻塞到对前一个信号处理结束为止。