Unix/Linux 核心编程

课程内容

- Unix/Linux操作系统简介
- GNU编译工具GCC
- GNU C
- 内存管理
- 文件I/O
- 进程管理
- 信号
- 进程间通信
- 多线程
- 网络通信

UNIX/LINUX操作系统

Unix操作系统

- UNIX操作系统，是美国AT&T公司于1971年在PDP-11上运行的操作系统。具有多用户、多任务的特点，支持多种处理器架构，最早由肯·汤普逊（Kenneth Lane Thompson）、丹尼斯·里奇（Dennis MacAlistair Ritchie）和Douglas McIlroy于1969年在AT&T的贝尔实验室开发。
- Unix的三大派生版本
  - System V
  - Berkley
  - Hybrid

System V

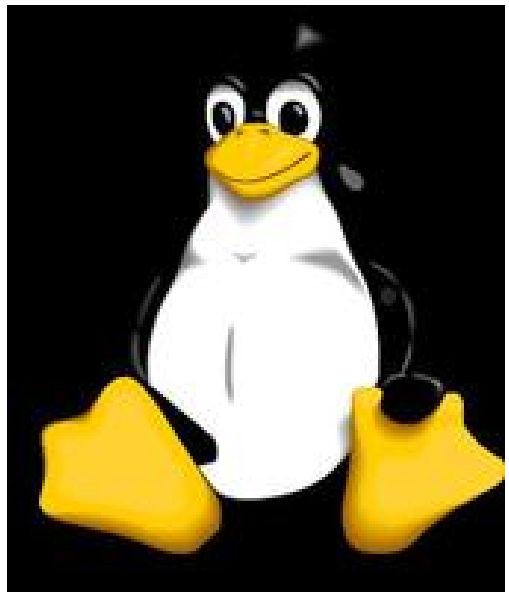
- AIX
- Solaris
- HP-UX
- IRIX

Berkley

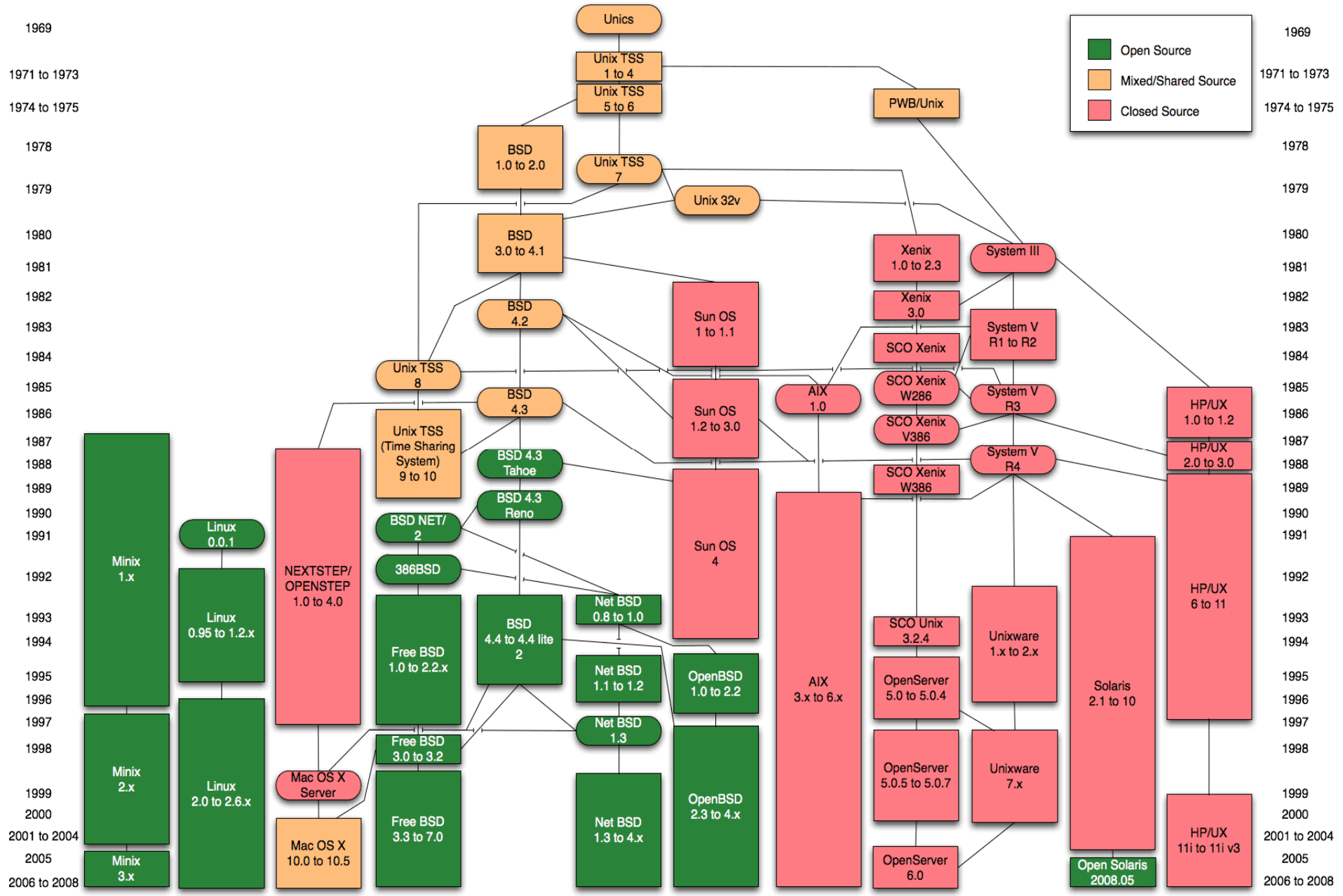
- FreeBSD
  - 一种类UNIX操作系统，但不是真正意义上的UNIX操作系统，它是由经过BSD、386BSD和4.4BSD发展而来的Unix的一个重要分支
- NetBSD
  - 是一份免费，安全的具有高度可定制性的类Unix操作系统，适于多种平台，从64位AMDAthlon服务器和桌面系统到手持设备和嵌入式设备
- OpenBSD
  - 一个从NetBSD衍生出来的类Unix操作系统
- Mac OS X
  - 是苹果公司开发的专属操作系统Mac OS的最新版本。它是一套Unix基础的操作系统，包含两个主要的部份：核心名为Darwin，是以FreeBSD源代码和Mach微核心为基础，由苹果公司和独立开发者社区协力开发；及一个由苹果电脑开发，名为Aqua之专有版权的图形用户界面。

Hybrid

- Minix
  - 名称取自英语Mini UNIX，是一个迷你版本的类Unix操作系统(约300MB)
- Linux
  - 是一类Unix计算机操作系统的统称



Unix族谱


1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2008
Open Source
Mixed/Shared Source
Closed Source
Unics
Unix TSS 1 to 4
Unix TSS 5 to 6
PWB/Unix
BSD 1.0 to 2.0
Unix TSS 7
Unix 32v
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
BSD 4.2
Sun OS 1 to 1.1
System V R1 to R2
SCO Xenix
Unix TSS 8
SCO Xenix W286
AIX 1.0
System V R3
BSD 4.3
Sun OS 1.2 to 3.0
HP/UX 1.0 to 1.2
Unix TSS (Time Sharing System) 9 to 10
SCO Xenix V386
HP/UX 2.0 to 3.0
BSD 4.3 Tahoe
System V R4
SCO Xenix W386
BSD 4.3 Reno
Minix 1.x
Linux 0.0.1
BSD NET/ 2
Sun OS 4
NEXTSTEP/ OPENSTEP 1.0 to 4.0
386BSD
AIX 3.x to 6.x
HP/UX 6 to 11
Net BSD 0.8 to 1.0
Linux 0.95 to 1.2.x
SCO Unix 3.2.4
Unixware 1.x to 2.x
Free BSD 1.0 to 2.2.x
BSD 4.4 to 4.4 lite 2
Net BSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.0.4
Solaris 2.1 to 10
Net BSD 1.3
Minix 2.x
Linux 2.0 to 2.6.x
Free BSD 3.0 to 3.2
OpenServer 5.0.5 to 5.0.7
Unixware 7.x
Mac OS X Server
OpenBSD 2.3 to 4.x
Free BSD 3.3 to 7.0
Net BSD 1.3 to 4.x
HP/UX 11i to 11i v3
Mac OS X 10.0 to 10.5
OpenServer 6.0
Minix 3.x
Open Solaris 2008.05

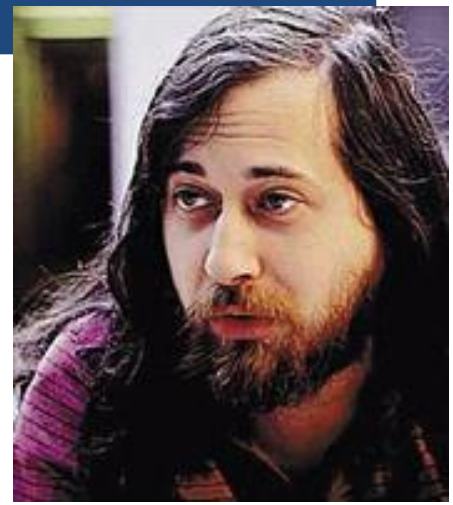
Linux简介

- Linux是一种自由和开放源码的类Unix操作系统。目前存在着许多不同的Linux,但它们都使用了Linux内核。Linux可安装在各种计算机硬件设备中，从手机、平板电脑、路由器和视频游戏控制台，到台式计算机、大型机和超级计算机。Linux是一个领先的操作系统，世界上运算最快的10台超级计算机运行的都是Linux操作系统。严格来讲，Linux这个词本身只表示Linux内核，但实际上人们已经习惯了用Linux来形容整个基于Linux内核，并且使用GNU 工程各种工具和数据库的操作系统。Linux得名于计算机业余爱好者Linus Torvalds。
- Tux（一只企鹅，全称为tuxedo）是Linux的标志

和Linux相关

- MINIX 操作系统
  - MINIX 系统是由Andrew S. Tanenbaum（AST）开发的。AST 是在荷兰Amsterdam 的Vrije 大学数学与计算机科学系统工作，是ACM 和IEEE 的资深会员
- GNU 计划
- POSIX 标准
- GPL通用公共许可证

GNU计划



- GNU Project由Richard Stallman发起开始于1984年，由自由软件基金(FSF :Free Software Foundation)支持。
- GNU的基本原则是共享。
- GNU的主旨在于发展一个类似 Unix ，并且为自由软件的完整操作系统： GNU 系统。
- 目前已经有各种使用 Linux 作为内核的 GNU 操作系统正被广泛地使用著；这些系统通常被称作为 “Linux” ，但准确的说应该被称GNU/Linux” 系统 。

POSIX 标准

- POSIX(Portable Operating System Interface for Computing Systems)是由IEEE 和ISO/IEC 开发的一簇标准。该标准是基于现有的UNIX 实践和经验，描述了操作系统的调用服务接口，用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植运行。
- POSIX.1 仅规定了系统服务应用程序编程接口（API），仅概括了基本的系统服务标准
- 在90 年代初，POSIX 标准的制定正处在最后投票敲定的时候，Linux刚刚起步，这个UNIX 标准为Linux 提供了极为重要的信息，使得Linux 的能够在标准的指导下进行开发，能够与绝大多数UNIX 系统兼容。

GPL通用公共许可证

- GNU通用公共许可证（GPL）
  - 一个法定的版权声明，但附带(或，在技术上去除了某些限制)，在条款中，允许对某项成果以及由它派生的其余成果的重用，修改和复制对所有人都是自由的。
- 非版权(copyleft)
  - copyleft带有标准的Copyright声明，确认作者的所有权和标志。但它放弃了标准copyright中的某些限制。它声明：任何人不但可以自由分发该成果，还可以自由地修改它。但你不能声明你做了原始的工作，或声明是由他人做的。最终，所有派生的成果必须遵循这一条款(相当于继承关系）

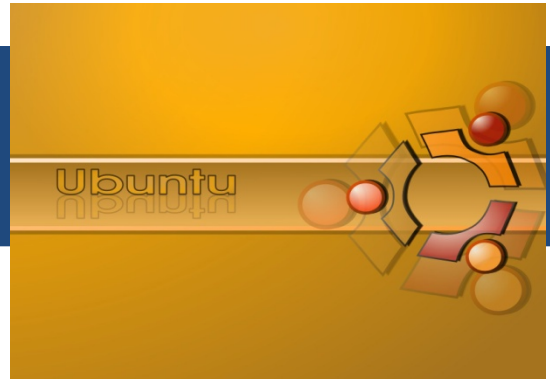
版本命名

- 早期版本
  - 第一个版本的内核是0.01。其次是0.02,0.03,0.10,0.11,0.12（第一GPL版本），0.95,0.96,0.97,0.98,0.99及1.0。
- 旧计划（1.0和2.6版之间），版本的格式为A.B.C，其中A,B,C代表：
  - A大幅度转变的内核。这是很少发生变化，只有当发生重大变化的代码和核心发生才会发生。在历史上曾改变两次的内核：1994年的1.0及1996年的2.0。
  - B是指一些重大修改的内核。
    - 内核使用了传统的奇数次要版本号码的软件号码系统（用偶数的次要版本号码来表示稳定版本）。
  - C是指轻微修订的内核。这个数字当有安全补丁，bug修复，新的功能或驱动程序，内核便会有变化。
- 第三次，自2.6.0(2003年12月）发布后，人们认识到，更短的发布周期将是有益的。自那时起，版本的格式为A.B.C.D

Linux特点

- 遵循GNU/GPL
- 开放性
- 多用户
- 多任务
- 设备独立性
- 供了丰富的网络功能
- 可靠的系统安全
- 良好的可移植性

Linux发行版



- 大众的Ubuntu
- 优雅的Linux Mint
- 锐意的Fedora
- 华丽的openSUSE
- 自由的Debian
- 简洁的Slackware
- 老牌的RedHat

slackware linux

debian

redhat.

GNU 编译工具GCC

编译工具----GCC

- 编译,如C、C++、Object C、Java、Fortran、Pascal、Ada等语言。GCC是可以在多种硬件平台上编译出可执行
- 在使用GCC编译程序时,编译过程可以细分为4个阶段：
  - a.预处理。
  - b.编译。
  - c.汇编。
  - d.链接。
- 程序员可以对编译过程进行控制,同时GCC提供了强大的代码优化功能。
- 查看gcc的版本：gcc –v

C程序中的文件后缀名

扩展名	说明
.a	静态对象库
.c	需要预处理的C语言源代码
.h	C语言源代码头文件
.i	不需要预处理的C语言源代码
.o	目标文件
.s	汇编语言代码
.so	共享对象库

编译单源程序

- 语法：gcc [选项参数] c文件
- 通用选项参数说明如下：
  - 1、指定输出文件名

    -o 指定输出文件

    例子：gcc -o main ch01.c
  - 2、警告与提示.

    -pedantic 检测不符合ANSI/ISO C语言标准的源代码,使用扩展语法的地方将产生警告信息。

    -Wall 生成尽可能多的警告信息。

    -Werror 要求编译器将警告当做错误进行处理。

– 3、指定编译文件类型

-x 指定编译代码类型，c、c++、assembler，none。None根据扩展名自动确认。

例子：gcc –x c -Wall –o main ch01.c

– 4、生成调试信息与优化

-g　　生成调试信息

-O　　优化

– 5、建议：在编译任何程序的时候都带上-Wall选项。

编译多源程序

- 语法：gcc [选项] C源代码1 C源代码2 C源代码3
- 思考：头文件的作用是什么？

预处理

- 语法：gcc -E C源代码文件
  - 示例：
  - gcc -E -o ch01.i ch01.c
  - gcc -E -o ch01_1.i ch01_1.c
  -
- 注意：
  - 预处理每次只能处理一个文件。不能处理多个文件，就是每个.c文件对应一个.i文件。
  - 不指定 –o 选项，预处理的结果输出到标准输出设备。

预处理指令介绍

预编译指示符号	说明
#define	定义宏
#elif	else if 多选分支
#else	与#if、#ifndef、#ifdef结合使用
#error	产生错误,挂起预处理程序
#if	判定
#endif	结束判定
#ifdef	判定宏是否定义
#ifndef	判定宏是否定义
#include	将指定的文件插入#include的位置
#include_next	与#include一样,但从当前目录之后的目录查找
#line	指定行号
#pragma	提供额外信息的标准方法，可用来指定平台
#undef	删除宏
#warning	创建一个警告
##	连接操作符号，用于宏内连接两个字符串

#error、#warning

```
#include <stdio.h>
#define VERSION  3
/* 演示编译器gcc */
#if (VERSION < 2)
        #error "版本低"
#else
        #warning "版本高"
#endif

int main()
{
        printf("Hello gcc使用!\n");
        return 0;
}
```

#include、#include_next

1.系统头文件使用#include <…>
2.用户头文件使用#include “…”
规则：
1.系统头文件会在I参数指定得目录中查找。
2.用户头文件会在当前目录查找。
3.Unix标准系统目录
/usr/local/include
/usr/lib/gcc-lib/…/版本/include
/usr/include
4.编译C++优先查找/usr/include/g++
5.#include <sys/time.h>会在所有标准目录的子目录sys中查找time.h
6.#include的文件名含扩展，*、?无意义。除非文件名中包含*。

#line

```
int re=0;
printf("Hello gcc使用!\n");
for(int i=0;i<200)
{
        re+=i;
}
printf("out:%d\n",re);
//代码行数被修改
#line 200
printf("out:%d\n",re,a);//人为错误
printf("out:%d\n",re);
```

#pragma

- 所有GCC的pragma都定义两个词GCC +其他

  #pragma GCC dependency 文件名 提示符号

  测试文件的时间戳，当指定文件比当前文件新的时候产生警告。

  #pragma GCC poison

  每次使用指定名字就会产生警告

  pragma pack(1)

  ……

#pragma有一个等价的宏_Pragam

```
#include <stdio.h>
#pragma GCC dependency "ch02.c"
//#pragma GCC poison printf add
int main()
{
    int re=0;
    printf("Hello gcc使用!\n");
    int i;
    _Pragma("GCC poison printf add")
    for(i=0;i<200;i++){
      re+=i;
    }
    printf("out:%d\n",re);
    return 0;
}
```

预定义宏介绍

宏	说明
__BASE_FILE__	源代码的完整路径
__cplusplus	C++有效，程序不符合标准为1,否则是标准的年月
__DATE__	日期
__FILE__	源代码文件名
__func__	当前函数名
__FUNCTION__	同上
__INCLUDE_LEVEL__	包含层数,基本的为0
__LINE__	行数
__TIME__	时间

编译环境变量

- C_INCLUDE_PATH　　：查找头文件的目录。C。
- CPATH　　：查找头文件，相当于-I选项。
- CPLUS_INCLUDE_PATH　：查找头文件的目录。C++。
- LD_LIBRARY_PATH　　：编译没有影响，主要影响运行。指定目录便于定位共享库。
- LIBRARY_PATH　　：查找连接文件，相当于-l选项

生成汇编

- 编译成汇编
  - gcc -S ch01.c ch01_1.c
- 编译汇编
  - gcc ch01.s ch01_1.s -o main

创建静态库

- 编译静态库
  - gcc -c -static ch01_1.c
  - 其中-static可选，可阻止gcc使用共享库
  - 不使用共享库会使可执行文件变大，但会减少运行时间开销
- ar指令
  - ar -r libmy.a ch01_1.o
  - 语法：ar [选项] 归档文件名 目标文件列表
  - 指令ar的常用选项

选项	说明
-d	从归档文件删除指定目标文件列表。
-q	将指定目标文件快速附加到归档文件末尾。
-r	将指定目标文件插入文档，如果存在则更新。
-t	显示目标文件列表
-x	把归档文件展开为目标文件

使用静态库

- gcc -o main ch01.c libmy.a
- 如果libmy.a在LIBRARY_PATH的指定目录中，还可以采用如下方式编译。
  - gcc ch01.c -o main –lmy

创建共享库

- 编译共享库
  - 编译共享库分成两个部分：
  - 编译成位置独立代码的目标文件，选项-fpic
  - 编译成共享库，选项-shared
    - gcc -c -fpic ch01_1.c
    - gcc -shared ch01_1.o -o libmy.so
  - 使用一条指令的效果一样
    - gcc -fpic -shared ch01_1.c -o libmy.so

定位共享库

- 共享库编译的时候与静态库一样依赖LIBRARAY_PATH，运行的时候依赖LD_LIBRARY_PATH。
- 规则：
  - 查找LD_LIBRARY_PATH，目录使用冒号分隔。
  - /ect/ld.so.cache中找到的列表。工具ldconfig维护。
  - 目录/lib
  - 目录/usr/lib

使用共享库

- gcc ch01.c libmy.so -o main
- 在代码中动态加载共享库：
  - 共享库代码

```
int add(int a,int b)
{
  int c=a+b;
  c=c/2;
  return c;
}
```

共享库的四个函数

- #include <dlfcn.h>

  void *dlopen(const char *filename, int flag);

  char *dlerror(void);

  void *dlsym(void *handle, const char *symbol);

  int dlclose(void *handle);

- 其中dlopen的参数flag的含义如下：

  RTLD_LAZY：符号查找时候才加载。

  RTLD_NOW：马上加载。

其他工具简介

库工具程序介绍

- ldconfig
  - ldconfig是一个动态链接库管理命令，为了让动态链接库为系统所共享,还需运行动态链接库的管理命令--ldconfig.
  - ldconfig 命令的用途,主要是在默认搜寻目录(/lib和/usr/lib)以及动态库配置文件/etc/ld.so.conf内所列的目录下,搜索出可共享的动态链接库(格式如前介绍,lib*.so*),进而创建出动态装入程序(ld.so)所需的连接和缓存文件.缓存文件默认为/etc/ld.so.cache,此文件保存已排好序的动态链接库名字列表
  - ldconfig通常在系统启动时运行,而当用户安装了一个新的动态链接库时,就需要手工运行这个命令.

ldconfig选项

(1) -v或--verbose：用此选项时,ldconfig将显示正在扫描的目录及搜索到的动态链接库,还有它所创建的连接的名字.

(2) -n：用此选项时,ldconfig仅扫描命令行指定的目录,不扫描默认目录(/lib,/usr/lib),也不扫描配置文件/etc/ld.so.conf所列的目录.

(3) -N：此选项指示ldconfig不重建缓存文件(/etc/ld.so.cache).若未用-X选项,ldconfig照常更新文件的连接.

(4) -X：此选项指示ldconfig不更新文件的连接.若未用-N选项,则缓存文件正常更新.

(5) -f CONF：此选项指定动态链接库的配置文件为CONF,系统默认为/etc/ld.so.conf.

(6) -C CACHE：此选项指定生成的缓存文件为CACHE,系统默认的是/etc/ld.so.cache,此文件存放已排好序的可共享的动态链接库的列表.

(7) -r ROOT：此选项改变应用程序的根目录为ROOT(是调用chroot函数实现的).选择此项时,系统默认的配置文件/etc/ld.so.conf,实际对应的为 ROOT/etc/ld.so.conf.如用-r /usr/zzz时,打开配置文件/etc/ld.so.conf时,实际打开的是/usr/zzz/etc/ld.so.conf文件.用此选项,可以大大增加动态链接库管理的灵活性.

(8) -l：通常情况下,ldconfig搜索动态链接库时将自动建立动态链接库的连接.选择此项时,将进入专家模式,需要手工设置连接.一般用户不用此项.

(9) -p或--print-cache：此选项指示ldconfig打印出当前缓存文件所保存的所有共享库的名字.

(10) -c FORMAT 或 --format=FORMAT：此选项用于指定缓存文件所使用的格式,共有三种:old(老格式),new(新格式)和compat(兼容格式,此为默认格式).

(11) -V：此选项打印出ldconfig的版本信息,而后退出.(12) -? 或 --help 或 --usage：这三个选项作用相同,都是让ldconfig打印出其帮助信息,而后退出.

nm

- nm 用来列出目标文件的符号清单
- 常用参数：

选项	说明
-p	不排序显示
-r	逆序排列显示
--size-sort	大小排列显示
-o	用源文件标识成员符号
-s	包含模块的索引信息
-D	对共享库显示动态符号,而不是静态符号
-g	显示定义为外部的符号

strip

- 去除指定目标文件与静态库中的调试信息 ：strip ch01_1.o libmy.a

ldd

- 列出共享库的依赖关系:ldd libmy.so

objdump

- 显示二进制文件信息
  以一种可阅读的格式让你更多地了解二进制文件可能带有的附加信息
  --source
  -S 尽可能反汇编出源代码，尤其当编译的时候指定了-g这种调试参数时，
  效果比较明显。隐含了-d参数。
  --show-raw-insn
  反汇编的时候，显示每条汇编指令对应的机器码，除非指定了--prefix-addresses，这将是缺省选项。

  --no-show-raw-insn
  反汇编时，不显示汇编指令的机器码，这是指定 --prefix-addresses
  选项时的缺省设置。

错误处理

异常处理方式

- 根据函数返回值判断异常
  - 返回一般用户数据
    - -1:表示异常,其他就是用户数据
  - 返回指针用户数据
    - NULL指针，OXFFFFFFFF　指针表示错误
    - 其他就是指针用户数据
  - 返回值不是用户数据，只是用来指明函数调用状态。
    - 0:成功
    - -1:失败
  - 返回void。一般不会发生错误

异常处理方式

- 使用外部全局变量errno获取异常原因
  - 根据errno得到异常编号
    - errno在函数调用正确不会被修改
    - 绝对不要通过errno判定错误
  - 把errno转换为字符串
    - strerror函数
    - perror函数
    - printf函数
      - printf()的格式输出%m格式

GNU C 简介

GNU C

- gcc(GNU CC)是一个功能非常强大的跨平台C编译器
- 它对标准C（ANSI/ISO C）进行了很多扩展，这些扩展对优化目标代码布局安全性的检测方面提供很强的支持，我们把支持GNU扩展的C语言称为GNU C。
- Linux内核代码使用了大量的GNU C扩展，以至于唯一能够编译Linux内核的编译器就是gcc

GNU C扩展的一些特殊之处

- 允许零长度数组
- case范围
- 语句表达式
- typeof关键字
- 可变参数的宏
- 标号元素
- 特殊属性声明
- 内建函数

允许零长度数组

```
struct var_data s{
        int len;
        char data[0];
};
```

- char data[0]仅仅意味着程序中通过var_data的结构体实例的data[index]成员可以访问len之后的第index个地址，并没有为data[0]分配内存
- 假设struct var_data的数据域保存在struct var_data紧接着的内存区域，通过如下代码可以遍历这些数据：

```
struct var_data s;
for (i=0;i<s.len;i++){
    printf("%02x",s.data[i]);
}
```

case范围

- GNU C 支持case x...y 这样的语法，区间[x,y]的数都会满足这个case的条件

```
switch (  c  ){
        case '0'...'9': c-='0';
        break;
        case 'a'...'f': c-='a'-10;
        break;
        case 'A'...'F': c-='A'-10;
        break;
}
```

语句表达式

- GNU C把包含在括号里的复合语句看做是一个表达式，称为语句表达式，它可以出现在任何允许表达式的地方。我们可以在语句表达式中使用原本只能在复合语句中使用的循环变量、局部变量等，例如

```
#define min_t(type,x,y) \
({type __x=(x); type __y=(y);__x<__y?__x:__y})
int ia,ib,mini;
mini=min_t(int,ia,ib);
```

- 这样，因为重新定义了__x和__y这两个局部变量，所以上述方法定义的宏将不会有副作用。在标准C中，对应的宏通常会有副作用：

```
#define min(x,y) ((x)<(y)?(x):(y))
```

而代码min(++ia,++ib)将会被展开为

((++ia)<(++ib)?(++ia):(++ib)) 传入宏的参数会被增加两次

typeof关键字

- typeof(x)语句可以获得x的类型，因此，我们可以借助typeof重新定义第3条提到的min_t这个宏

```
#define min(x,y)({ \
const typeof(x) _x=(x);\
const typeof(y) _y=(y);\
(void) (&_x==&_y);\
_x<_y ? _x: _y ; })
```

- 我们不需要像第三条时那样传一个type进去，因为通过typeof(x)可以得到type。
- 代码 (void) (&_x==&_y);的作用是检查_x和_y的类型是否一致。

可变参数的宏

- 而在GNU C中，宏也可以接受可变数目的参数，例如
  - #define pr_debug(fmt,arg...) printk(fmt,##arg)
- 这里arg表示其余的参数可以是零个或多个，这些参数以及参数之间的逗号构成arg的值，在宏扩展时替换arg ，例如
  - pr_debug("%s:%d",filename,line);被扩展为
  - printk("%s:%d",filename,line);
  - 使用##的原因是为了处理arg不代表任何参数的情况，这时候，前面的逗号就变得多余了。
  - 使用##之后，GNU C预处理器会丢弃前面的逗号，这样代码pr_debug("success!\n") 会被正确扩展为 printk("success!\n");而不是 printk("success!\n",);

标号元素

- 标准c要求数组或结构体的初始化值必须以固定的顺序出现，在GNU C中，通过指定索引或结构体成员名，允许初始化值得以任意顺序出现。
- 指定数组索引的方法是在初始化值前添加[INDEX]= ，当然也可以用 [FIRST...LAST]= 的形式指定一个范围。例如下面的代码定义一个数组，并把其中的所有元素赋值为0：
- unsigned char data[MAX] ={[0...MAX-1]=0 };

标号元素

- 下面的代码借助结构体成员名初始化结构体：

```
struct file_operations DEMO_fops = {
    owner :    THIS_MODULE,
    llseek:    DEMO_llseek,
    read:      DEMO_read,
    write:     DEMO_write,
    ioctl:     DEMO_ioctl,
    open:       DEMO_open,
    release:   DEMO_release,
};
```

标号元素

- 但是Linux 2.6还是推荐采用标准C的方式，如下

```
struct file_operations DEMO_fops = {
    .owner =    THIS_MODULE,
    .llseek =   DEMO_llseek,
    .read =     DEMO_read,
    .write =    DEMO_write,
    .ioctl =    DEMO_ioctl,
    .open =     DEMO_open,
    .release =  DEMO_release,
};
```

attribute

- GNU C的一大特色就是__attribute__机制
- __attribute__可以设置:

  函数属性（Function Attribute）

  变量属性（Variable Attribute）

  类型属性（Type Attribute）
- __attribute__语法格式为：

  __attribute__ ((attribute-list))

  其位置约束为：放于声明的尾部“；”之前。

函数属性（Function Attribute）

- 函数属性可以帮助开发者把一些特性添加到函数声明中，从而可以使编译器在错误检查方面的功能更强大。__attribute__机制也很容易同非GNU应用程序做到兼容之功效。
- GNU CC需要使用 –Wall编译器来激活该功能，这是控制警告信息的一个很好的方式。
- 后面是常见的一些属性参数

__attribute__ format

- 该__attribute__属性可以给被声明的函数加上类似printf或者scanf的特征，它可以使编译器检查函数声明和函数实际调用参数之间的格式化字符串是否匹配。
- format的语法格式为：
  format (archetype, string-index, first-to-check)
  format属性告诉编译器，按照printf, scanf, strftime或strfmon的参数表格式规则对该函数的参数进行检查。“archetype”指定是哪种风格；“string-index”指定传入函数的第几个参数是格式化字符串；“first-to-check”指定从函数的第几个参数开始按上述规则进行检查。
- 具体使用格式如下：
  __attribute__((format(printf,m,n)))
  __attribute__((format(scanf,m,n)))
  其中参数m与n的含义为：
  m：第几个参数为格式化字符串（format string）；
  n：参数集合中的第一个，即参数“...”里的第一个参数在函数参数总数排在第几

测试用例

```
1：
2：extern void myprint(const char *format,...)
__attribute__((format(printf,1,2)));
3：
4：void test()
5：{
6：  myprint("i=%d\n",6);
7：  myprint("i=%s\n",6);
8：  myprint("i=%s\n","abc");
9：  myprint("%s,%d,%d\n",1,2);
10：}
```

运行$gcc –Wall –c attribute.c attribute后，输出结果为：

```
attribute.c: In function `test':
attribute.c:7: warning: format argument is not a pointer (arg 2)
attribute.c:9: warning: format argument is not a pointer (arg 2)
attribute.c:9: warning: too few arguments for format
```

__attribute__ noreturn

- 该属性通知编译器函数从不返回值，当遇到类似函数需要返回值而却不可能运行到返回值处就已经退出来的情况，该属性可以避免出现错误信息。
- C库函数中的abort（ ）和exit（ ）的声明格式就采用了这种格式，如下所示：
  - extern void exit(int) __attribute__((noreturn));
  - extern void abort(void) __attribute__((noreturn));

示例

```
extern void myexit();
int test(int n)
{
    if ( n > 0 ){
            myexit();
            /* 程序不可能到达这里*/
    } else
            return 0;
}
```

编译显示的输出信息为：

```
    $gcc –Wall –c noreturn.c
    noreturn.c: In function `test':
```

加上__attribute__((noreturn))则可以很好的处理类似这种问题。

把extern void myexit();修改为：

```
    extern void myexit() __attribute__((noreturn));
```

之后，编译不会再出现警告信息。

constructor/destructor

- 若函数被设定为constructor属性，则该函数会在main（ ）函数执行之前被自动的执行。类似的，若函数被设定为destructor属性，则该函数会在main（ ）函数执行之后或者exit（ ）被调用后被自动的执行。拥有此类属性的函数经常隐式的用在程序的初始化数据方面。

函数调用约定

- 函数调用时，调用者依次把参数压栈，然后调用函数，函数被调用以后，在堆栈中取得数据，并进行计算。函数计算结束以后，或者调用者、或者函数本身修改堆栈，使堆栈恢复原装。
- 在参数传递中，有两个很重要的问题必须得到明确说明：
  - 当参数个数多于一个时，按照什么顺序把参数压入堆栈
  - 函数调用后，由谁来把堆栈恢复原装
  - 在高级语言中，通过函数调用约定来说明这两个问题。常见的调用约定有：
    - stdcall
    - cdecl
    - fastcall
    - thiscall
    - naked call

变量属性（Variable Attributes）

- 关键字__attribute__也可以对变量（variable）或结构体成员（structure field）进行属性设置。
- 在使用__attribute__参数时，你也可以在参数的前后都加上“__”（两个下划线），例如，使用__aligned__而不是aligned
  - aligned (alignment)
    - 该属性规定变量或结构体成员的最小的对齐格式，以字节为单位。
  - packed
    - 使用该属性可以使得变量或者结构体成员使用最小的对齐方式，即对变量是一字节对齐，对域（field）是位对齐

类型属性（Type Attribute）

- 关键字__attribute__也可以对结构体（struct）或共用体（union）进行属性设置。大致有六个参数值可以被设定，即：aligned, packed, transparent_union, unused, deprecated 和 may_alias。

变量属性与类型属性举例

```
struct p{
    int a;
    char b;
    char c;
}__attribute__((aligned(4))) pp;
struct q{
    int a;
    char b;
    struct n qn;
    char c;
}__attribute__((aligned(8))) qq;
int main(){
    printf("sizeof(int)=%d,sizeof(short)=%d.sizeof(char)=%d\n",sizeof(int),
    sizeof(short),sizeof(char));
    printf("pp=%d,qq=%d \n", sizeof(pp),sizeof(qq));
}
```

内存管理

环境变量

- 环境表
  - 每个程序都会接收到一张环境表，是一个字符指针数组。数组以null做为结束。
  - 全局变量environ保存了该数组的首地址。
- 环境变量操作函数

函数	描述
getenv	返回指向name关联的value的指针
putenv	将形式为name=value的环境变量放入环境表
setenv	将name设置为value,每三个参数决定是否替代已有变量
unsetenv	删除定义
clearenv	删除环境表中所有项

内存管理在语言结构上的变化

- 从malloc/free到new/delete
  - C++是强类型语言，new/delete的主要成果也就是加强了类型观念，减少了强制类型转换的需求。但是从内存管理角度看，这个变革并没有多少的突破性
- 从new/delete到内存配置器（allocator）
  - allocator的引入也是C++内存管理一个突破。留意一下你就可以发现，整个STL所有组件的内存均从allocator分配。也就是说，STL并不推荐使用new/delete进行内存管理，而是推荐使用allocator.

Unix/Linux内存管理

- Unix/Linux低层采用三层结构，实际使用中可以方便映射到两层或者三层结构，以适用不同的硬件结构。最下层的申请内存函数get_free_page。之上有三种类型的内存分配函数
  - 1.kmalloc类型。内核进程使用，基于切片(slab)技术，用于管理小于内存页的内存申请。思想出发点和应用层面的内存缓冲池同出一辙。但它针对内核结构，特别处理，应用场景固定，不考虑释放。
  - 2.vmalloc类型。内核进程使用。用于申请不连续内存。
  - 3.brk/mmap类型。用户进程使用。malloc/free实现的基础。

进程与内存

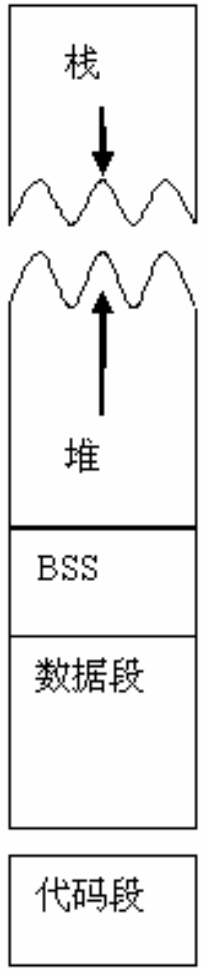
- 所有进程（执行的程序）都必须占用一定数量的内存
- 对任何一个普通进程来讲，它都会涉及到5种不同的数据段
  - 代码段：代码段是用来存放可执行文件的操作指令，也就是说是它是可执行程序在内存种的镜像。代码段需要防止在运行时被非法修改，所以只准许读取操作，而不允许写入（修改）操作——它是不可写的。
  - 数据段：数据段用来存放可执行文件中已初始化全局变量，换句话说就是存放程序静态分配的变量和全局变量。
  - BSS段：BSS段包含了程序中未初始化全局变量，在内存中bss段全部置零。
    - Block started by symbol

堆和栈

- 堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）
- 栈：栈是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括static声明的变量，static意味这在数据段中存放变量）。除此以外在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也回被存放回栈中。由于栈的后进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上将我们可以把堆栈看成一个临时数据寄存、交换的内存区。

进程如何组织这些区域

- 上述几种内存区域中数据段、BSS和堆通常是被连续存储的——内存位置上是连续的，而代码段和栈往往会被独立存放。有趣的是堆和栈两个区域关系很“暧昧”，他们一个向下“长”（i386体系结构中栈向下、堆向上），一个向上“长”，相对而生。但你不必担心他们会碰头，因为他们之间间隔很大



查看内存结构

- /proc/进程ID目录
- cat maps
- size 文件名
  - 报告正文段、数据段和bss段的长度

虚拟内存管理技术

- Linux操作系统采用虚拟内存管理技术，使得每个进程都有各自互不干涉的进程地址空间。该空间是块大小为4G的线性虚拟空间，用户所看到和接触的都是该虚拟地址，无法看到实际的物理内存地址。利用这种虚拟地址不但能起到保护操作系统的效果（用户不能直接访问物理内存），而且更重要的是用户程序可使用比实际物理内存更大的地址空间
- 4G的进程地址空间被人为的分为两个部分——用户空间与内核空间。用户空间从0到3G（0xC0000000），内核空间占据3G到4G。用户进程通常情况下只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址。例外情况只有用户进程进行系统调用（代表用户进程在内核态执行）等时刻可以访问到内核空间。

虚拟内存管理技术

- 用户空间对应进程，所以每当进程切换，用户空间就会跟着变化；而内核空间是由内核负责映射，它并不会跟着进程改变，是固定的。内核空间地址有自己对应的页表（init_mm.pgd），用户进程各自有不同的页表
- 每个进程的用户空间都是完全独立、互不相干的。你可以把程序同时运行10次（当然为了同时运行，让它们在返回前一同睡眠100秒吧），你会看到10个进程占用的线性地址一模一样。

进程内存管理

- 进程内存管理的对象是进程线性地址空间上的内存镜像，这些内存镜像其实就是进程使用的虚拟内存区域（memory region）。进程虚拟空间是个32或64位的“平坦”（独立的连续区间）地址空间（空间的具体大小取决于体系结构）。要统一管理这么大的平坦空间可绝非易事，为了方便管理，虚拟空间被化分为许多大小可变的(但必须是4096的倍数)内存区域，这些区域在进程线性地址中像停车位一样有序排列。这些区域的划分原则是“将访问属性一致的地址空间存放在一起”，所谓访问属性在这里无非指的是“可读、可写、可执行等”。

物理内存管理（页管理）

- Linux内核管理物理内存是通过分页机制实现的，它将整个内存划分成无数4k(在i386体系结构中)大小页，从而分配和回收内存的基本单位便是内存页了。利用分页管理有助于灵活分配内存地址，因为分配时不必要求必须有大块的连续内存，系统可以东一页、西一页的凑出所需要的内存供进程使用。虽然如此，但是实际上系统使用内存还是倾向于分配连续的内存块，因为分配连续内存时，页表不需要更改，因此能降低刷新率（频繁刷新会很大增加访问速度）。
- getpagesize();

brk/sbrk的虚拟内存管理

- void *sbrk(int size)
  - Size=0 返回sbrk/brk上次的末尾地址
  - size>0　分配内存空间，返回上一次末尾地址
  - size<0　　释放空间
- int brk(void* ptr)
  - 直接修改访问的有效范围的末尾地址
  - 释放空间形成一个完整的page，则该页影射被解除
  - 返回：
    - 0：分配成功
    - -1：失败

系统底层的内存映射(mmap/munmap)

- #include <sys/mman.h>
- void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
- int munmap(void *start, size_t length);
  - 参数start：指向欲映射的内存起始地址，通常设为 NULL，代表让系统自动选定地址，映射成功后返回该地址。
  - 参数length：代表将文件中多大的部分映射到内存。
    - 映射空间大小。建议4k倍数，不是4K倍数，自动对齐
  - 参数prot：

    映射区域的保护方式。可以为以下几种方式的组合：

    1.PROT_EXEC 映射区域可被执行

    2.PROT_READ 映射区域可被读取

    3.PROT_WRITE 映射区域可被写入

    4.PROT_NONE 映射区域不能存取

mmap

- 参数flags：影响映射区域的各种特性。在调用mmap()时必须要指定MAP_SHARED 或MAP_PRIVATE。

  1.MAP_FIXED 如果参数start所指的地址无法成功建立映射时，则放弃映射，不对地址做修正。通常不鼓励用此标志。

  2.MAP_SHARED对映射区域的写入数据会复制回文件内，而且允许其他映射该文件的进程共享。

  3.MAP_PRIVATE 对映射区域的写入操作会产生一个映射文件的复制，即私人的“写入时复制”（copy on write）对此区域作的任何修改都不会写回原来的文件内容。

  4.MAP_ANONYMOUS建立匿名映射。此时会忽略参数fd，不涉及文件，而且映射区域无法和其他进程共享。

  5.MAP_DENYWRITE只允许对映射区域的写入操作，其他对文件直接写入的操作将会被拒绝。

  6.MAP_LOCKED 将映射区域锁定住，这表示该区域不会被置换（swap）。

mmap

- 参数fd 要映射到内存中的文件描述符。如果使用匿名内存映射时，即flags中设置了MAP_ANONYMOUS，fd设为-1。有些系统不支持匿名内存映射，则可以使用fopen打开/dev/zero文件，然后对该文件进行映射，可以同样达到匿名内存映射的效果。
- 参数offset： 文件映射的偏移量，通常设置为0，代表从文件最前方开始对应，offset必须是分页大小的整数倍。
- 返回值： 若映射成功则返回映射区的内存起始地址，否则返回MAP_FAILED(- 1)，错误原因存于errno 中。

errno

- 错误代码：
  - 1.EBADF 参数fd 不是有效的文件描述词
  - 2.EACCES 存取权限有误。如果是MAP_PRIVATE 情况下文件必须可读，使用MAP_SHARED则要有PROT_WRITE以及该文件要能写入。
  - 3.EINVAL 参数start、length 或offset有一个不合法。
  - 4.EAGAIN 文件被锁住，或是有太多内存被锁住。
  - 5.ENOMEM 内存不足。

系统调用

系统调用

- Linux 大部分的系统功能是通过系统调用(System Call)来实现的.如open,send之类.
- 这些函数在C程序调用起来跟标准C库函数(printf...)非常类似.但是实现机制完全不同.
- 库函数仍然是运行在Linux 用户空间程序.很多时候内部会调用系统调用.
- 但系统调用是内核实现的.在C库封装成函数.但通过系统软中断进行调用.
  - 用time命令测试时间,系统时间实际就是系统调用时间累积
    - time ./demo1
  - 用strace 可以跟踪一种程序系统调用使用情况
    - strace ./demo1 #不需要调试信息

库函数与系统调用的关系

- 以是C库函数malloc与系统调用sbrk的关系

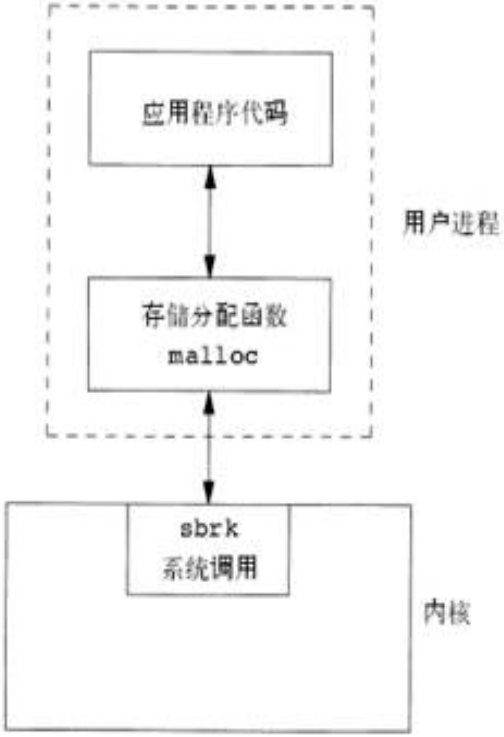


图1-2 malloc函数和sbrk系统调用

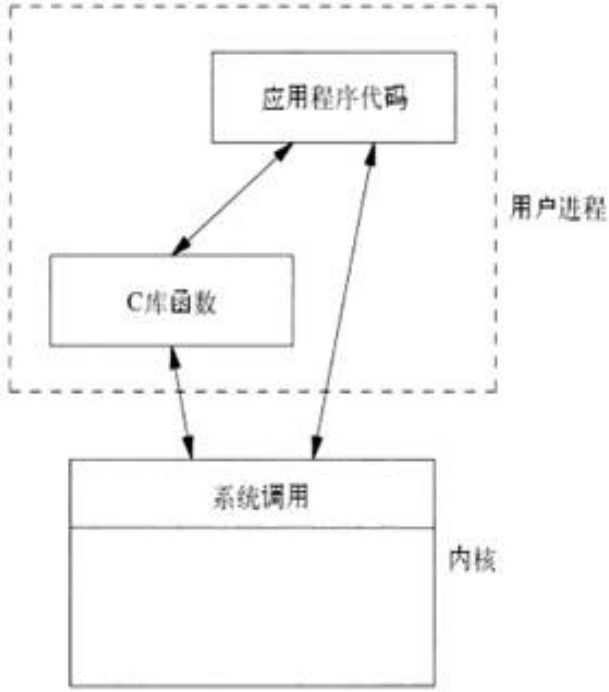


图1-3 C库函数和系统调用之间的差别

文件控制

Linux文件结构

- Linux环境中的文件具有特别重要的意义，因为它们为操作系统服务和设备提供了一个简单而统一的接口.在Linux中,一切（或几乎一切）都是文件。
- 通常程序完全可以像使用文件那样使用磁盘文件、串行口、打印机和其他设备。
- 大多数情况下，你只需要使用五个基本的函数——open、close、read、write和ioctl
- Linux中的任何事物都可以用一个文件代表，或者可以通过特殊的文件进行操作。

Linux文件结构(2)

- 一些特殊文件
  - 目录
  - 设备文件
  - /dev/console
  - /dev/tty
  - /dev/null

底层库函数

- Linux 在底层实现一整套处理文件函数.
  - 这一些函数能处理普通文件,网络socket文件,设备文件等
  - 全部是系统调用实现的函数
- 文件处理函数
  - open – 打开或创建一个文件
  - creat – 建立一个空文件
  - close – 关闭一个文件
  - read – 从文件读入数据
  - write – 向文件写入一个数据
  - lseek – 在文件中移动读写位置
  - unlink – 删除一个文件
  - remove – 删除一个文件本身
  - fcntl – 控制一个文件属性

文件描述符

- 值为一个非负整数
- 用于表示一个打开文件
- 在内核空间被引用,并且由系统调用(open)所创建
- read,write使用文件描述符
- 内核缺省打开三个文件描述符
  - 1-标准输出
  - 2-错误输出
  - 0-标准输入
- unistd.h中，0,1,2应当替换成STDIN_FILENO,STDOUT_FILENO,STDERR_FILENO.
- 文件描述符的变化范围是0~OPEN_MAX。OPEN_MAX有可能是63,linux允许更大的值。

Open函数

- #include <fcntl.h>
- int open(const char *pathname, int flags);
- int open(const char *pathname, int flags, mode_t mode);
- int creat(const char *pathname, mode_t mode);
  - 第三个参数（...）仅当创建新文件时才使用，用于指定文件的访问权限位（access permission bits）
  - flags 用于指定文件的打开/创建模式，这个参数可由以下常量（定义于 fcntl.h）通过逻辑或构成。
    - 1.O_RDONLY　只读模式
    - 2.O_WRONLY　只写模式
    - 3.O_RDWR　读写模式
  - 打开/创建文件时，至少得使用上述三个常量中的一个

Open函数

- 以下常量是选用的：
  1.O_APPEND　　每次写操作都写入文件的末尾
  2.O_CREAT　　如果指定文件不存在，则创建这个文件
  3.O_EXCL　　如果要创建的文件已存在，则返回 -1，并且修改 errno 的值
  4.O_TRUNC　　如果文件存在，并且以只写/读写方式打开，则清空文件全部内容
  5.O_NOCTTY　　如果路径名指向终端设备，不要把这个设备用作控制终端。
  6.O_NONBLOCK　　如果路径名指向 FIFO/块文件/字符文件，则把文件的打开和后继 I/O设置为非阻塞模式（nonblocking mode）

open函数

- 以下三个常量同样是选用的，它们用于同步输入输出
  1.O_DSYNC　　等待物理 I/O 结束后再 write。在不影响读取新写入的数据的前提下，不等待文件属性更新。
  2.O_RSYNC　　read 等待所有写入同一区域的写操作完成后再进行
  3.O_SYNC　　等待物理 I/O 结束后再 write，包括更新文件属性的 I/O
- open 返回的文件描述符一定是最小的未被使用的描述符
- 一个进程同时打开文件的个数是有限的，这个限制通常由limits.h头文件中的常量OPEN_MAX决定

  POSIX要求最少16

  通常被设置成256

函数说明：write

- #include <unistd.h>
- ssize_t write(int filedes, const void *buf, size_t nbytes);
- 返回值：写入文件的字节数（成功）；-1（出错）
- write 函数向 filedes 中写入 nbytes 字节数据，数据来源为 buf 。返回值一般总是等于 nbytes，否则就是出错了。常见的出错原因是磁盘空间满了或者超过了文件大小限制。

函数说明：read

- #include <unistd.h>
- ssize_t read(int filedes, void *buf, size_t nbytes);
- 返回值：读取到的字节数；0（读到 EOF）；-1（出错）
- read 函数从 filedes 指定的已打开文件中读取 nbytes 字节到 buf 中。以下几种情况会导致读取到的字节数小于 nbytes ：
  A. 读取普通文件时，读到文件末尾还不够 nbytes 字节。例如：如果文件只有 30 字节，而我们想读取 100 字节，那么实际读到的只有 30 字节，read 函数返回 30 。此时再使用 read 函数作用于这个文件会导致 read 返回 0 。
  B. 从终端设备（terminal device）读取时，一般情况下每次只能读取一行。

函数说明：read

- C. 从网络读取时，网络缓存可能导致读取的字节数小于 nbytes 字节。
- D. 读取 pipe 或者 FIFO 时，pipe 或 FIFO 里的字节数可能小于 nbytes 。
- E. 从面向记录（record-oriented）的设备读取时，某些面向记录的设备（如磁带）每次最多只能返回一个记录。
- F. 在读取了部分数据时被信号中断。

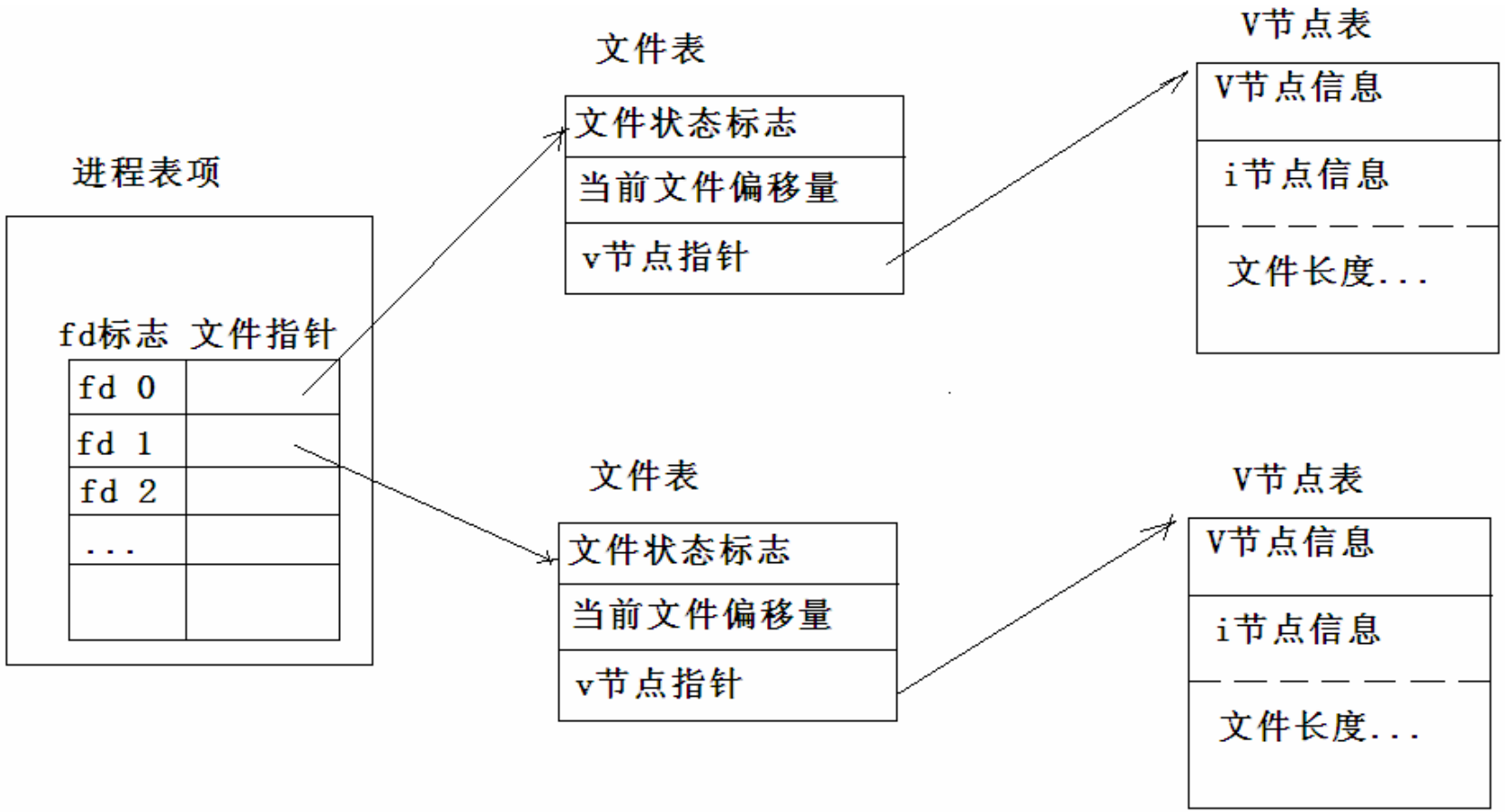
函数说明： close

- close调用终止一个文件描述符fildes与其对应文件之间的关联。文件描述符被释放并能够重新使用。close调用成功就返回0，出错就返回-1。
- 有时检查close调用的返回结果十分重要。有的文件系统，特别是网络文件系统，可能不会在关闭文件之前报告文件写操作中出现的错误，因为执行写操作时，数据可能未被确认写入。
- 关闭一个文件时会释放该进程加在文件上的所有记录锁
- 当一个进程终止，内核自动关闭它所有打开的文件。

lseek函数

- 每个打开的文件都有一个与其相关的“当前文件偏移量”
- 偏移量通常是一个非负整数，用以度量从文件开始处计算的字节数。
- 读、写操作都从当前文件偏移量处开始，并使偏移量增加所读写的字节数。
- 当打开一个文件时，除非指定O_CREAT，否则偏移量被设置为0
- 该函数仅将当前的文件偏移量记录在内核中，它并不会引起任何I/O操作。
- 文件偏移量可以大于文件的当前长度。对该文件的下一次读写会加长该文件，并形成文件空洞，这个是允许的
- 文件空洞并不要求在磁盘上占用存储区，但对于新写的数据需要分配磁盘块

打开文件的内核数据结构



dup,dup2函数

- 复制一个现在的文件描述符
- Dup返回的返回的一定是当前可用描述符的最小值
- dup2可由第二个参数指定描述符值，如果指定的文件已经打开，那么先关闭文件
- 一定程度上和fcntl功能相同

sync,fsync,fdatasync

- 大多数磁盘IO都通过缓冲进行，写入文件时先写入缓冲区，如果缓冲未满，则不将其排入输出队列，这种方式叫做延迟写。
- 延迟写减少了磁盘写次数，但降低了文件内容更新速度
- 这三个函数可以保证缓冲区和实际文件系统的数据一致。
- sync将所有修改过的缓冲区排入写队列，然后就返回，并不等实际写磁盘
- fsync只对一个文件，并且等实际写磁盘完成才返回
- fdatasync只更新数据，不更新文件属性

fcntl函数

- #include<fcntl.h>
  - int fcntl(int fd , int cmd);
  - int fcntl(int fd,int cmd,long arg);
  - int fcntl(int fd,int cmd,struct flock * lock);
- 函数说明:
  - fcntl()用来操作文件描述词的一些特性。
  - 参数fd代表欲设置的文件描述词.
  - 参数cmd代表欲操作的指令。

fcntl函数

- cmd有以下几种情况:
- F_DUPFD用来查找大于或等于参数arg的最小且仍未使用的文件描述符，并且复制参数fd的文件描述符。执行成功则返回新复制的文件描述符。请参考 dup2()。
- F_SETFD 设置新的文件标志，新标志按第三个参数设置
  - 通常仅用来设置FD_CLOEXEC
  - FD_CLOEXEC的作用是决定是否在成功调用了某个exec系列的系统调用之后关闭该文件描述符
- F_GETFL 取得文件描述符状态标志，此标志为open（ ）的参数flags。
- F_SETFL 设置文件描述词状态标志 ，参数arg为新标志 ，但只允许O_APPEND、O_NONBLOCK和O_ASYNC位的改变，其他位的改变将不受影响。

fcntl函数

- F_GETLK 取得文件锁定的状态。
- F_SETLK 设置文件锁定的状态。此时flcok 结构的l_type 值必须是F_RDLCK、F_WRLCK或F_UNLCK。如果无法建立锁定，则返回-1，错误代码为EACCES 或EAGAIN。
- F_SETLKW F_SETLK 作用相同，但是无法建立锁定时，此调用会一直等到锁定动作成功为止。若在等待锁定的过程中被信号中断时，会立即返回-1，错误代码为EINTR。参数lock指针为flock 结构指针，定义如下

```
struct flcok        {
        short int l_type; /* 锁定的状态*/
        short int l_whence;/*决定l_start位置*/
        off_t l_start; /*锁定区域的开头位置*/
        off_t l_len; /*锁定区域的大小*/
        pid_t l_pid; /*锁定动作的进程*/
};
```

fcntl函数

- l_type 有三种状态:
  - F_RDLCK 建立一个供读取用的锁定
  - F_WRLCK 建立一个供写入用的锁定
  - F_UNLCK 删除之前建立的锁定
- l_whence 也有三种方式:
  - SEEK_SET 以文件开头为锁定的起始位置。
  - SEEK_CUR 以目前文件读写位置为锁定的起始位置
  - SEEK_END 以文件结尾为锁定的起始位置。

stat，fstat, lstat

- int stat(const char *path, struct stat *buf);
  - 提供文件名字，获取文件对应属性
- int fstat(int filedes, struct stat *buf);
  - 通过文件描述符获取文件对应的属性。
- int lstat(const char *path, struct stat *buf);
  - 连接文件描述符，获取文件属性

struct stat

```
struct stat {
        mode_t     st_mode;       //文件对应的模式，文件，目录等
        ino_t      st_ino;       //inode节点号
        dev_t      st_dev;       //设备号码
        dev_t      st_rdev;      //特殊设备号码
        nlink_t    st_nlink;     //文件的连接数
        uid_t      st_uid;       //文件所有者
        gid_t      st_gid;       //文件所有者对应的组
        off_t      st_size;      //普通文件，对应的文件字节数
        time_t     st_atime;     //文件最后被访问的时间
        time_t     st_mtime;     //文件内容最后被修改的时间
        time_t     st_ctime;     //文件状态改变时间
        blksize_t st_blksize;    //文件内容对应的块大小
        blkcnt_t   st_blocks;    //文件内容对应的块数量
      };
```

st_mode

S_IFSOCK 0140000 scoket
S_IFLNK 0120000 符号连接
S_IFREG 0100000 一般文件
S_IFBLK 0060000 区块装置
S_IFDIR 0040000 目录
S_IFCHR 0020000 字符装置
S_IFIFO 0010000 先进先出
S_ISUID 04000 文件的（set user-id on execution）位
S_ISGID 02000 文件的（set group-id on execution）位
S_ISVTX 01000 文件的sticky位
S_IRUSR（S_IREAD） 00400 文件所有者具可读取权限
S_IWUSR（S_IWRITE）00200 文件所有者具可写入权限
S_IXUSR（S_IEXEC） 00100 文件所有者具可执行权限
S_IRGRP 00040 用户组具可读取权限
S_IWGRP 00020 用户组具可写入权限
S_IXGRP 00010 用户组具可执行权限
S_IROTH 00004 其他用户具可读取权限
S_IWOTH 00002 其他用户具可写入权限
S_IXOTH 00001 其他用户具可执行权限

sys/stat.h中的文件类型宏函数

宏	文件类型
S_ISREG()	普通文件
S_ISDIR()	目录文件
S_ISCHR()	字符特殊文件
S_ISBLK()	块特殊文件
S_ISFIFO	管道或FIFO
S_ISLNK()	符号链接
S_ISSOCK()	套接字

- 返回值:

  执行成功则返回0，失败返回-1，错误代码存于errno
- 错误代码:
  - ENOENT 参数file_name指定的文件不存在
  - ENOTDIR 路径中的目录存在但却非真正的目录
  - ELOOP 欲打开的文件有过多符号连接问题，上限为16符号连接
  - EFAULT 参数buf为无效指针，指向无法存在的内存空间
  - EACCESS 存取文件时被拒绝
  - ENOMEM 核心内存不足
  - ENAMETOOLONG 参数file_name的路径名称太长

文件访问权限

- st_mode值也包含了文件访问权限，屏蔽位如下表

St_mode屏蔽	意义
S_IRUSR	用户读
S_IWUSR	用户写
S_IXUSR	用户执行
S_IRGRP	组读
S_IWGRP	组写
S_IXGRP	组执行
S_IROTH	其他读
S_IWOTH	其他写
S_IXOTH	其他执行

access函数

- 按实际用户ID和实际组ID进行访问权限测试。
- 测试成功返回0，出错返回-1
- 测试中的mode 常量

Mode	说明
R_OK	测试读权限
W_OK	写权限
X_OK	执行权限
F_OK	文件是否存在

umask

- 为进程设置文件模式创建屏蔽字，并返回以前的值
- 参数是由9个常量(S_IRUSR,S_IWUSR等)中的若干位按位或构成的

chmod、fchmod

- 更改文件访问权限，Chmod是在指定文件上操作，fchmod是在打开的文件上操作
- Chmod函数的mode常量，来自sys/stat.h

Mode	说明
S_IRWX	用户读写执行
S_IRUS	用户读
S_IWUS	用户写
S_IXUS	用户执行
S_IRWX	组读写执行
S_IRGR	组读
S_IWG	组写
S_IXGR	组执行
S_IRWX	其他读写执行
S_IROT	其他读
S_IWO	其他写
S_IXOTH	其他执行

chown、fchown、lchown

- 更改文件的用户id和组id
- 如果任意id参数为-1，则保持不变
- 只有超级用户进程改变文件的用户id
- 如果进程拥有些文件，非超级用户也可以修改

truncate、ftruncate

- 文件截短成指定长度

link、unlink、remove和rename

- link,创建一个指向现在文件的链接
- unlink,删除一个文件的链接
  - 只有当链接数达到0时，文件删除。另外，当一个进程已经打开了该文件，其内容也不能删除，当进程关闭时，文件才会删除。
- remove,对于文件，功能和unlink相同，对于目录，和rmdir相同。
- rename,文件或目录改名。

symlink和readlink函数

- 创建软链接文件
  - 并不要求目标文件已经存在
  - 并不需要位于同一个文件系统中
- 因为open函数无法直接读取软件链接文件，可以使用readlink

创建目录

- mkdir
  - 创建一个空目录
  - mode参数指定权限
- rmdir
  - 删除一个空目录

读目录

- DIR* opendir()
  - 打开目录，成功返回指针，失败NULL
- struct dirent* readdir(DIR*)
  - 读目录,返回第一个目录项
  - Dirent结构：
    - struct dirent{
    - ino_t d_ino;//i-node号
    - char d_name[256];//文件名
    - }
- rewinddir
- telldir
- seekdir

chdir、fchdir、getcwd

- chdir、fchdir
  - 更改当前工作目录
  - 工作目录只是进程的一个属性，所以它只影响进程本身
- getcwd
  - 获取当前工作目录的绝对路径

进程控制

进程和程序

- 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。
- 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。

进程分类

- 进程一般分为交互进程、批处理进程和守护进程三类。
- 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超级管理用户root来启动。

ps aux 查看进程

- 显示进程的信息
  - USER　进程的属主；
  - PID　　进程的ID；
  - PPID　父进程；
  - %CPU　进程占用的CPU百分比；
  - %MEM 占用内存的百分比；
  - NI　进程的NICE值，数值大，表示较少占用CPU时间；
  - VSZ 进程虚拟大小；
  - RSS　驻留中页的数量；
  - WCHAN
  - TTY　终端ID

- STAT 进程状态
- D Uninterruptible sleep (usually IO)
- R 正在运行可中在队列中可过行的；
- S 处于休眠状态；
- T 停止或被追踪；
- W 进入内存交换（从内核2.6开始无效）；
- X 死掉的进程（从来没见过）；
- Z 僵尸进程；
- < 优先级高的进程
- N 优先级较低的进程
- L 有些页被锁进内存；
- s 进程的领导者（在它之下有子进程）；

进程描述符

- 每个进程都有一个非负整形表示的唯一进程ID
- 进程ID是唯一的，但可以重用。当一个进程终止时，其进程ID就可以再次使用。
  - 延迟重用法

    getpid 获得进程ID
    getppid 获得父进程ID
    getuid 获得实际用户ID
    geteuid 获得有效用户ID
    getgid 获得实际组ID
    getegid 获得有效组ID

创建进程fork

- fork 创建一个新进程
  - 出错返回-1
  - 由fork函数创建的进程叫子进程(child proccess)
  - 此函数调用一次，返回两次。
  - 分别在子进和和父进程中返回，子进程中返回0，父进程返回子进程的PID
  - 子进程是父进程的副本，子进程获得父进程的数据空间，堆和栈的副本，但子进程共享父进程的正文段
  - fork之后父子进程会继续执行。
  - fork之后 父进程先执行还是子进程先执行不确定
  - fork时，文件描述符也会被复制，那么两个进程可能会共享同一个文件表。

fork

- fork失败的原因
  - 系统中有太多的进程
  - 实际用户ID的进程总数已经超过系统限制。
- fork的用法
  - 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
  - 一个进程要执行一个不同的程序

vfork

- 基本功能和fork相同
- 区别：
  - vfork创建新进程的主要目的是exec一个新程序。
  - vfork并不复制父进程的地址空间，因为子进程会立即调用exec
  - vfork保证子进程先运行

进程终止的5种正常情况

- 在main函数中执行return
- 调用exit函数，并不处理文件描述符，多进程
- 调用_exit或_Exit.
- 进程的最后一个线程执行了返回语句
- 进程的最后一个线程调用pthread_exit函数

进程的3种异常终止方式

- 调用abort,产生SIGABRT信号
- 进程接收到某些信号
- 最后一个线程对“取消”请求做出响应

wait和waitpid函数

- 当一个进程正常或异常终止时，内核就向其父进程发送SIGCHLD信号。父进程可以忽略该信号，或者提供一个该该信号的处理函数。默认情况下，系统会忽略该信号
- 如果父进程调用了wait或waitpid时
  - 如果其子进程都还在运行，则阻塞
  - 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，则立即返回
- 区别
  - 在一个子进程终止前，wait使其调用者阻塞，而waitpid 很多选择
- 如果一个子进程已经终止，并且是僵死进程，wait会立即返回并取得该子进程的状态，否则阻塞。

参数statloc

- 终止进程的状态将保存在此指针指向的位置，如果不关心状态，可置空。
- 得到的状态由<sys/wait.h>中的各个宏来查看。有四个互斥的宏用来查看进程终止的原因

宏	说明
WIFEXITED(status)	若为正常终止子进程返回的状态，则为真，对于这种情况可执行WEXITSTATUS(status)取子程序传给exit、_exit或_Exit参数的低8位
WIFSIGNALEND(status)	若为异常终止进程返回的状态，则为真。可执行WTERMSIG(status)取信号编号
WIFSTOPPED(status)	基为当前暂停子进程的返回状态，则为真。可执行WSTOPSIG(status)，取信号编号
WIFCONTINUED(status)	若在作业控制暂停后已经继续的子进程返回了状态，则为真。

waitpid

- 如果进程有几个子进程，那么要等待指定的进程终止，可使用waitpid
- pid参数
  ==-1,等待任意子进程，与wait等效
  >0 等待指定子进程
  ==0 等待其组ID等于调用进程组ID的任一子进程
  <-1 等待其组ID等于pid绝对值的任一子进程
- Options参数可以为0,或是以下参数，参数可“或”叠加

常量	说明
WCONTINUED	若实现文件作业控制，子进程暂停后继续，但状态未报告，则返回其状态
WNOHANG	若pid指定的子进程并不立即可用，则waitpid不阻塞，此时返回值为0
WUNTRACED	若支持作业控制，子进程处于暂停状态，返回其状态

exec函数

- exec函数会用新程序完全替代掉现有程序，并开始从main函数执行。
- exec并不创建新的进程，所以pid并未改变
- exec只是用一个全新的程序替换了当前进程的正文、数据、堆和栈。
- 有6种不同的exec函数可供使用

```
int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execve(const char *path, const char *arg[], char * const envp[]);
int execvp(const char *file, char *const argv[]);
```

区别

- 第一个参数是路径名或文件名，当是文件名时，如果其中包含/则为路径名，否则查找PATH环境变量获取文件
- 参数表的传递，l表示list,v表示vector.
- 以e结尾的两个函数可以传递一个指向环境字符串指针数据的指针。

system函数

- ISO C定义了此函数
- 如果参数是空，返回非零值。此特性经常用来测试平台是否支持此函数
- 函数本质上调用了fork、exec和waitpid，返回值如
  - 如果fork失败或waitpid出错，则返回-1
  - 如果exec失败，如果执行了exit(127)
  - 如果都成功，返回shell的终止状态，见waitpid
- 使用system而不是fork和exec的优点是，system 进行了所需的各种出错处理，以及各种信号处理。

信号

信号概念

- 信号是软件中断。它即可以作为进程间通信的一种机制，更重要的是，信号总是中断一个进程的正常运行，它更多地被用于处理一些非正常情况。ctrl+c就是一个信号
- 信号是异步的，进程并不知道信号什么时候到达。
- 进程既可以处理信号，也可以发送信号给特定进程。
- 每个信号都有一个名字，这些名字都以SIG开头。例如：SIGABRT是进程异常终止信号。

信号的来源

- 硬件异常产生信号：除数为0、无效的存储访问等等。这些条件通常由硬件检测到，并将其通知内核。然后内核为该条件发生时正在运行的进程产生适当的信号。
- 软件产生异常信号，可以用kill、raise、alarm、setitimer和 sigqueue产生信号。

信号的种类

- 不可靠的信号：Linux信号机制基本上是从Unix系统中继承过来的。早期Unix系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题，因此，把那些建立在早期机制上的信号叫做“不可靠信号”，信号值小于SIGRTMIN的叫不可靠信号(1~31)。
- 每次信号处理后，该信号对应的处理函数会恢复到默认值。但现代的Linux已经对其进行了改进，信号处理函数一直是用户指定的或者是系统默认的。
- 信号可能丢失。
- 不可靠信号不支持信号排队，同一个信号产生多次，只要程序还未处理该信号，那么实际只处理此信号一次。

信号的种类

- 可靠信号：信号值位于SIGRTMIN和SIGRTMAX之间的信号都是可靠信号，可靠信号克服了信号可能丢失的问题 。
- 实时信号与非实时信号：Linux目前定义了64种信号（将来可能会扩展），前面32种为非实时信号，后32种为实时信号。非实时信号都不支持排队，都是不可靠信号，实时信号都支持排队，都是可靠信号。
- 信号排队意味着无论产生多少次信号，信号处理函数就会被调用同样的次数。

Linux系统信号

信号名称	信号说明	默认处理
SIGABRT	由程序调用 abort时产生该信号。程序异常结束。	进程终止并且产生core文件
SIGALRM	timer到期，有alarm或者setitimer	进程终止
SIGBUS	总线错误，地址没对齐等。取决于具体硬件。	结束终止并产生core文件
SIGCHLD	进程停止或者终止时，父进程会收到该信号。	忽略该信号
SIGCONT	让停止的进程继续执行	继续执行或者忽略
SIGFPE	算术运算异常，除0等。	进程终止并且产生core文件。
SIGHUP	终端关闭时产生这个信号	进程终止
SIGILL	代码中有非法指令	进程终止并产生core文件
SIGINT	终端输入了中断字符ctrl+c	进程终止

Linux系统信号

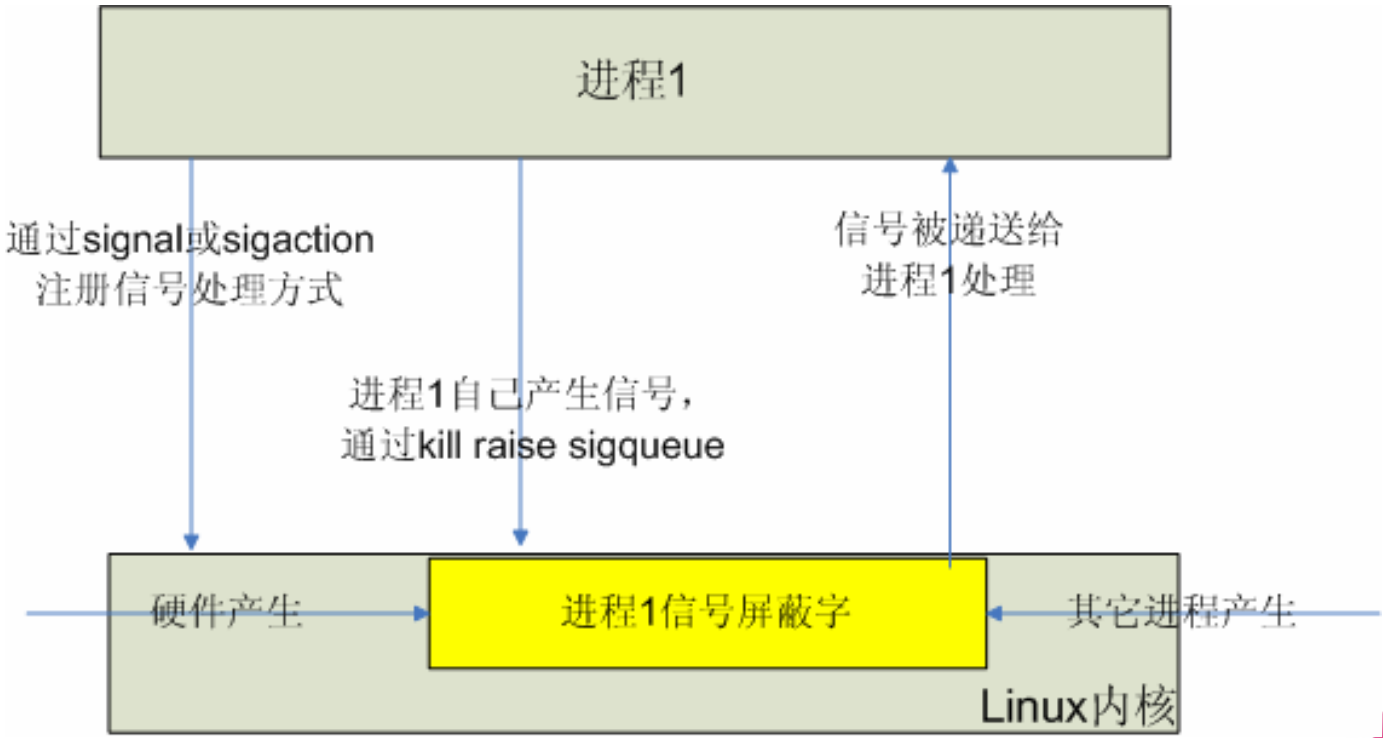
SIGIO	异步I/O,跟SIGPOLL一样。	进程终止
SIGIOT	执行I/O时产生硬件错误	进程终止并且产生core文件
SIGKILL	这个信号用户不能去捕捉它。	进程终止
SIGPIPE	往管道写时，读者已经不在了，或者往一个已断开数据流socket写数据。	进程终止
SIGPOLL	异步I/O，跟SIGIO一样。	进程终止
SIGPROF	有setitimer设置的timer到期引发 。	进程终止
SIGPWR	Ups电源切换时	进程终止
SIGQUIT	Ctrl+\，不同于SIGINT，这个是会产生core dump文件的。	进程终止并且产生core文件
SIGSEGV	内存非法访问，默认打印出segment fault	进程终止并且产生core文件
SIGSTOP	某个进程停止执行，该信号不能被用户捕捉。	进程暂停执行

Linux系统信号

SIGSYS	调用操作系统不认识的系统调用。	进程终止并且产生core文件
SIGTERM	有kill函数调用产生。	进程终止
SIGTRAP	有调试器使用，gdb	进程终止并且产生core文件
SIGTSTP	Ctrl+z，挂起进程。	进程暂停
SIGTTIN	后台程序要从终端读取成数据时。	进程暂停
SIGTTOU	后台终端要把数据写到终端时。	进程暂停
SIGURG	一些紧急的事件，比如从网络收到带外数据。	忽略
SIGUSR1	用户自定义信号	进程终止
SIGUSR2	用户自定义信号	进程终止
SIGVTALRM	有setitimer产生。	进程终止

信号处理流程

- 信号处理取决于Linux内核屏蔽字。当信号是不可靠信号时进程只处理一个；当信号是可靠信号的时候，进程则依据排队一个一个的处理。



信号分析汇总

- SIGHUP:将进程关闭或者网络连接断开时产生该信号(可以使用捕捉函数体system(“echo abcd > a.txt”)去验证)。当使用“nohup ./* &”在后台运行程序时，用kill -SIGHUP id杀不掉它；当取消nohup运行时就可以杀掉（但要注意，要去捕捉的函数信息需要延时，否则在没有去执行杀死的信号前，它自己已经结束了）。
- SIGALRM:它用alarm(seconds)+pause()来设置SIGALRM信号在经过seconds指定的秒数后传送给目前的进程 .
- SIGABRT:由abort()产生，并终止程序并生成core文件(但需设置：ulimit –a 后+unlimited)
- SIGCHLD：子进程终止或停止时，父进程会收到该信号。用fork产生父子进程，然后让父进程sleep(5)，则子进程先结束就会发此信号给父进程；或者让子进程exit()则也会发信号给父进程。

信号术语

- 信号的产生：当引发信号的事件发生时，为进程产生一个信号（或向一个进程发送一个信号）。信号产生时，内核会在进程表中设置一位标识。
- 信号的递送(delivery)：当进程对信号采取动作（执行信号处理函数或忽略）时称为递送。
- 信号产生和递送之间的时间间隔内称信号是未决的（pending）。
- 信号递送阻塞(block)：进程可指定对某个信号采用递送阻塞。如果此时信号的处理时默认或者捕捉的，该信号就会处于未决的状态，直到进程解除对该信号的递送阻塞或者处理方式改为忽略。
- 如果信号的处理方式是忽略该信号，那么该信号永远不会处于递送或者递送未决状态

信号的使用

- 进程可以从三个方面使用信号：
  - 指定进程的信号处理函数（信号处理）。
  - 阻塞一个信号（也就是推迟它的发生），比如处于一段临界代码，执行完临界代码后在启用这个信号。
  - 向另外一个进程发送信号。

信号的处理

- 忽略此信号：大多数信号都可使用这种方式进行处理，但有两种信号却不能被忽略。它们是：SIGKILL和SIGSTOP。这两种信号不能被忽略的原因时：它们向超级用户提供一种进程终止或或停止的可靠方法；
- 捕捉信号：为了做到这一点要通知内核在某种信号发生时，调用一个用户函数。在用户函数中，可执行用户希望对这种事件进行的处理。
- 执行系统默认动作：对大多数信号的系统默认动作是终止该进程。

信号注册函数：signal

- signal函数用来通知内核如何处理某个特定信号（忽略、捕捉、默认处理）。

```
#include <signal.h>
typedef void (*sighandler_t)(int);
void function( int num )
sighandler_t    signal(int signum, sighandler_t handler);
void (*signal(int signo, void *func)(int)))(int);
```

  - 参数signum是信号名称，可以用kill –l列出所有的信号（可以用整数表示）。
  - 参数handler是信号处理方式，a）SIG_IGN进程忽略此信号，b）SIG_DFL以系统默认方式处理此信号，c）用户自定义的函数地址，当信号递送时linux内核会调用此函数。

信号注册函数：signal

- 失败：返回SIG_ERR，成功：返回前一次注册的信号处理函数。

```
#define  SIG_ERR   (void (*)())-1
#define  SIG_DFL   (void (*)()) 0
#define  SIG_IGN   (void (*)()) 1
```

- 进程对SIGCHLD的默认处理为忽略，请问此时的信号处理函数是什么？ 终止产生core文件

信号等待函数：pause

- pause会使调用进程进入睡眠状态直到有信号递送。如果你对某个信号采取了SIG_IGN，那么pause是不会被唤醒的。

```
#include <unistd.h>
int pause(void);
```

pause返回值为-1，errno设置成EINTR。

特殊信号

- SIGKILL 和 SIGSTOP 既不能忽略也不能被捕捉，它们只能按照默认方式处理。
- 进程正在运行，当使用kill + -SIGKILL + id会立即终止进程。
- 进程正在运行，当使用kill + - SIGSTOP + id会立即停止进程
- 让内核和超级用户有一种确定方式可以杀死或者停止某个进程。

子进程信号处理

- 子进程会继承父进程的信号处理方式，直到子进程调用exec函数。
- 子进程调用exec函数后，exec将父进程中设置为捕捉的信号变为默认处理方式，其余不变。
- 例如在父进程中把SIGTERM设置为捕捉，SIGINT设置为忽略。子进程执行exec和不执行exec的区别。
  - 当子进程调用exec函数时，如果父进程里设置的信号为捕捉方式，则变成默认方式处理(关闭终端)，其它方式则不变。当在子父进程里各自放一个pause()时，它就会等待信号进来，再响应。
  - 当进程没有调用exec函数时，一切不变。

信号集

- 目前Linux系统定义了64个信号（平行64位，即用8个字节表示），以后也可能进一步扩展。所以用类型sigset_t来表示所有的信号。
- 一般情况下，sigset_t中的一个比特位表示一种信号。
- 对信号集的操作需要专用的操作函数。
- 信号0是系统保留，不被使用的。事实上信号0是被用来测试某一个进程是否有权限向另外一个进程发送信号。kill 0 id

信号集操作函数

- 信号集操作函数有以下5个
  - 把所有信号集的比特位置为0： int sigemptyset(sigset_t *set);
  - 把所有信号集的比特位设为1： int sigfillset(sigset_t *set);
  - 把信号signum加入信号集set中
    int sigaddset(sigset_t *set, int signum);
  - 把信号signum从信号集set中清除
    int sigdelset(sigset_t *set, int signum);
  - 判断某个信号是否在信号集set中（特殊信号集已有该信号则返回1，没有则返回0）
    - int sigismember(const sigset_t *set, int signum);
  - 执行成功则返回0，如果有错误则返回-1

信号注册函数：sigaction

- Linux中signal注册函数最终也是调用sigaction来实现，保留signal主要是为了兼容。

  #include <signal.h>

  int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

  – 第一个signum指定需要处理的信号（除SIGKILL和SIGSTOP）。第二个参数act设定信号的处理方式，act可以为NULL。之前设定的信号处理方式会保存到第三个参数oldact，oldact为NULL时不保存。

  – 返回0成功，返回-1失败。

sigacton.c

信号注册函数：sigaction

```
struct sigaction {
        void        (*sa_handler)(int);
        void        (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t    sa_mask;
        int   sa_flags;
        void        (*sa_restorer)(void);     /*  obsolete */
}
```

- sa_handler和signal函数的第二个参数类型一样，当信号递送给进程时会调用这个sa_handler.
- sa_sigaction也是信号处理的函数指针，它只会在sa_flags包含SA_SIGINFO时才会被调用,而它的值为0 或其它任何值都将按默认方式处理，即调用handler捕捉函数。否则linux内核默认调用sa_handler。siginfo_t 包含了信号产生的原因。
- 不用同时赋值给sa_handler和sa_sigaction，因为它们可能是一个union。

信号注册函数：sigaction

- sa_mask信号屏蔽字，当执行sa_handler信号处理函数时，sa_mask指定的信号会被阻塞，直到该信号处理函数返回。当前信号处理函数对应的信号会被自动加入到sa_mask中。
- sa_flags用来改变信号处理时的行为。当sa_flags包含SA_RESTART时，被中断的系统调用在信号处理完后会被自动启动。

sa_flags选项说明

sa_flags	说明
SA_NOCLDSTOP	若signum是SIGCHLD,当一子进程停止时，不产生此信号
SA_NOMASK/SA_NODEFER	在处理此信号未结束前不理会此信号的再次到来
SA_RESTART	由此信号中断的系统调用自动重启
SA_ONSTACK	若用sigaltstack已说明了一替换栈，则此信号递送给替换栈上进程
SA_NOCLDWAIT	若signum是SIGCHLD,则当调用进程的子进程终止时，不创建僵尸进程。若调用进程在后面调用wait，则阻塞到它所有子进程都终止，此时返回-1
SA_NODEFER	当捕捉到此信号时，在执行其信号捕捉函数时，系统不自动阻塞此信号。
SA_ONESHOT/ SA_RESETHAND	当调用新的信号处理函数前，将此信号处理方式改为系统预设（SIG_DFL）的方式
SA_SIGINFO	此选项对信号处理程序提供了附加信息。

signal函数的默认值

- 当用signal函数注册信号捕捉函数时，signal函数设置哪些默认值。可用sigaction来观察。

信号和系统调用

- 由于信号是异步的，它会在程序的任何地方发生。由此程序正常的执行路径被打破，去执行信号处理函数。
- 一般情况下，当进程正在执行某个系统调用，那么在该系统调用返回前信号是不会被递送的。但慢速系统调用除外，如读写终端、网络、磁盘，以及wait和pause。这些系统调用都会返回-1，errno置为EINTR
- 当系统调用被中断时，我们可以选择使用循环再次调用，或者设置重新启动该系统调用(SA_RESTART)。

信号处理函数的中断问题

- 由于信号处理函数可被中断，所以在处理函数中调用的函数都必须是可重入的！如果某个函数可以被多个任务并发使用，而不会造成数据错误，我们就说这个函数具有可重入性（reentrant）
- 可重入函数可以在任意一点被中断，且可以被并发执行和重复调用！
- 可重入函数不能使用静态变量，不能使用malloc/free，不能使用标准I/O库！可重入函数使用全局变量时也应该小心！
- 不可重入函数有getlogin()，gmtime()，getgrgid()，getgrnam()，getpwuid()以及getpwnam() ，malloc(),free()
- 例子演示了在程序和信号处理函数中调用不可重入函数的情况，后果是进程的行为会出现未定义的情况。

可重入函数

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf
aio_return	fdatasync	mkfifo	setsid	tcdrain
aio_suspend	fork	open	setsockopt	tcflow
alarm	fpathconf	pathconf	setuid	tcflush
bind	fstat	pause	shutdown	tcgetattr
cfgetispeed	fsync	pipe	sigaction	tcgetpgrp
cfgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cfsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cfsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time

可重入函数

chmod	getgroups	read	sigismember	timer_getoverrun
chown	getpeername	readlink	signal	timer_gettime
clock_gettime	getpgrp	recv	sigpause	timer_settime
close	getpid	recvfrom	sigpending	times
connect	getppid	recvmsg	sigprocmask	umask
creat	getsockname	rename	sigqueue	uname
dup	getsockopt	rmdir	sigset	unlink
dup2	getuid	select	sigsuspend	utime
execle	kill	sem_post	sleep	wait
execve	link	send	socket	waitpid
_Exit & _exit	listen	sendmsg	socketpair	write

设置信号暂停函数

(1)alarm（设置信号传送闹钟）:unsigned int alarm(unsigned int seconds);

函数说明 :alarm()用来设置信号SIGALRM在经过参数seconds指定的秒数后传送给目前的进程。如果参数seconds 为0，则之前设置的闹钟会被取消，并将剩下的时间返回。返回值：返回之前闹钟的剩余秒数，如果之前未设闹钟则返回0。

(2)pause（让进程暂停直到信号出现） 定义函数:int pause(void);pause()会令目前的进程暂停（进入睡眠状态），直到被信号(signal)所中断。返回值 :只返回-1。

(3) sleep（让进程暂停执行一段时间)定义函数 :

unsigned int sleep(unsigned int seconds);函数说明 :sleep()会令目前的进程暂停，直到达到参数seconds 所指定的时间，或是被信号所中断。

返回值 :若进程暂停到参数seconds 所指定的时间则返回0，若有信号中断则返回剩余秒数。

(4) wait（等待子进程中断或结束） 定义函数 :pid_t wait (int * status);

wait()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用wait()时子进程已经结束，则wait()会立即返回子进程结束状态值,它由参数status 返回，而子进程的进程识别码也会一块返回。如果不在意结束状态值，则参数status可以设成NULL。子进程的结束状态值请参考waitpid()。返回值 :如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1

信号发送函数：kill

- kill函数可以向某个进程或者进程组发送特定的信号。
- #include <sys/types.h>　#include <signal.h>
- int kill(pid_t pid, int sig);
- 参数
  - pid > 0 将信号发送给进程ID为pid的进程；
  - pid == 0将信号发送给进程组ID等于发送进程的进程组ID，而且发送进程有许可权向其发送信号的所有进程；
  - pid < -1　　将信号发送给其进程组ID等于pid绝对值，而且发送进程有许可权向其发送信号的所有进程。
  - pid == -1　　将信号发送给除进程1以外的所有进程，但发送进程必须有许可权。
  - killall　命令使用进程名称来终止进程的运行。如果系统中有多个相同名称的进程，这些进程将全部被结束

信号发送函数：kill

- 成功返回0，失败返回-1，errno会被设置。
- 有发送信号许可权的基本规则是：1）超级用户可以将信号发送给任意进程；2）发送者的实际或有效用户ID必须等于接收者实际或有效用户ID；
- 当signum为0时，则kill仍执行正常的错误检测，但不发送信号。这常被用来确定一个特定进程是否存在。如果向一个并不存在的进程发送空信号，则kill返回-1,errno则被设置为ESRCH。
- 判断语句：wait(&status);
  - If(WIFSIGNALED(status)) 如果子进程是因为信号而结束则此宏值为真
    WTERMSIG(status)取得子进程因信号而中止的信号代码，一般会先用WIFSIGNALED 来判断后才使用此宏。

信号发送函数：raise

- raise函数向调用进程发送一个信号，相当于kill(getpid(),signum)

```
#include <signal.h>
int raise(int sig);
```

- 成功返回0，失败返回非0。

进程组

- 每个进程除了有一个进程ID之外，还属于一个进程组。进程组是一个或者多个进程的组合。
- 进程组有一个唯一的进程组ID，这个ID一般为进程组组长的进程ID。
- 给进程组发送信号时，该组的所有进程都会收到信号。

进程组函数

- setpgid 把pid进程的进程组ID设置为pgid
- getpgid 获取pid进程的进程组ID
- setpgrp 把当前进程ID设置为进程组ID
- getpgrp获得当前进程的进程组ID

前后台进程转换

- fg %job 把前台进程组置为后台运行
- bg %job 把后台进程置为前台进程
- &命令把指定程序放到后台执行
  ctrl+z(SIGSTOP SIGTSTP), SIGCONT
- 可以向整个进程组发送信号

信号发送函数：sigqueue

#include <signal.h>

int sigqueue(pid_t pid, int sig, const union sigval value);

sigqueue 向一个特定进程pid发送信号sig，发送前也需要检查权限。

第三个参数只有在注册信号处理函数时，在sa_flags中包含SA_SIGINFO，系统才会把value传递给信号处理函数，在siginfo_t 中的si_value中。

```
union sigval {
        int   sival_int;
        void *sival_ptr;
};
```

成功返回0，失败返回-1，errno会被设置。

信号屏蔽字

- 每个进程都会有一个信号屏蔽字，它规定了当前哪些信号可以递送，哪些信号需要阻塞。
- 当程序执行敏感任务时（比如更新数据库），不希望外界信号中断程序的运行。在这个阶段并不是简单地忽略信号，而是阻塞这些信号，当进程处理完关键任务后，还会处理这些信号。
- 在信号处理函数中，当前的信号总是加入信号屏蔽字中。

信号屏蔽：sigprocmask

#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

sigprocmask可以检测和修改进程的信号屏蔽字。oldset非空时，当前信号屏蔽字会保存到oldset中。如果set非空，那么参数how指示如何修改当前屏蔽字。

how有三个值：

SIG_BLOCK　被阻塞的信号是当前屏蔽字加上第二个参数包含的信号

SIG_UNBLOCK 第二个参数包含的信号从当前屏蔽字中移除

SIG_SETMASK 当前屏蔽字被完全设置为第二个参数的值。

当恢复之前的信号屏蔽字时，一定要用SIG_SETMASK,而不是SIG_UNBLOCK。因为你无法确定之前这个信号是否已经在屏蔽字中 。

可靠信号和不可靠信号

- 可靠信号在屏蔽期间会以链表的形式保存。解屏蔽后信号会被递送多次。
- 不可靠信号是不会被queue的。也即在屏蔽期间，无论同一个不可靠信号产生多少次，解屏蔽后只会递送一次 。
- 实例 signal/sig_rt.c

信号屏蔽：sigpending

- #include <signal.h>
- int sigpending(sigset_t *set);
- sigpending返回当前处于阻塞递送的信号，既信号已经产生，但还没有被送给进程处理。
- 成功返回0，失败返回-1

信号屏蔽问题

- 在sigpending.c中，如果我们在解除某个信号调用sleep函数，那么未被递送的信号会首先被处理。导致pause无法被唤醒。
- 需要把解除信号和等待信号递送做成一个原子操作，这样就不会产生上述问题。

信号屏蔽：sigsuspend

- #include <signal.h>
- int sigsuspend(const sigset_t *mask);
- sigsuspend会把当前信号屏蔽字设置为mask，然后挂起调用进程直到有信号被递送。
- 总是返回-1，因为它是被信号中断后返回的！

计时器与信号

- Linux下有两个睡眠函数，原型为：
  - #include <unistd.h>
  - unsigned int sleep(unsigned int seconds);
  - void usleep(unsigned long usec);
  - 函数sleep让进程睡眠seconds秒，函数usleep让进程睡眠usec毫秒。
  - sleep睡眠函数内部是用信号机制进行处理的，用到的函数有：

时钟处理

- Linux为每个进程维护3个计时器，分别是真实计时器、虚拟计时器和实用计时器。
  - 真实计时器计算的是程序运行的实际时间；
  - 虚拟计时器计算的是程序运行在用户态时所消耗的时间(可认为是实际时间减掉(系统调用和程序睡眠所消耗)的时间)；
  - 实用计时器计算的是程序处于用户态和处于内核态所消耗的时间之和。
- 例如：有一程序运行，在用户态运行了5秒，在内核态运行了6秒，还睡眠了7秒，则真实计算器计算的结果是18秒，虚拟计时器计算的是5秒，实用计时器计算的是11秒。

为进程设定计时器

- 用指定的初始间隔和重复间隔时间为进程设定好一个计时器后，该计时器就会定时地向进程发送时钟信号。3个计时器发送的时钟信号分别为：SIGALRM,SIGVTALRM和SIGPROF。
- 获取计时器的设置
  - int getitimer(int which, struct itimerval *value);
- 设置计时器
  - int setitimer(int which, const struct itimerval *value, struct itimer val *ovalue);

计时器

- which指定哪个计时器，可选项为
  - ITIMER_REAL(真实计时器)
  - ITIMER_VITUAL(虚拟计时器
  - ITIMER_PROF(实用计时器))
- value为一结构体的传入参数，指定该计时器的初始间隔时间和重复间隔时间
- ovalue为一结构体传出参数，用于传出以前的计时器时间设置。
- 如果成功，返回0，否则-1

计时器

```
truct itimerval
{
    struct timeval it_interval; /* next value *///重复间隔
    struct timeval it_value;    /* current value *///初始间隔
};

struct timeval
{
    long tv_sec;          /* seconds */          //时间的秒数部分
    long tv_usec;        /* microseconds *///时间的微秒部分
};
```

示例

```
void TimeInt2Obj(int imSecond,timeval *ptVal){
    ptVal->tv_sec=imSecond/1000;
    ptVal->tv_usec=(imSecond%1000)*1000;
}
void SignHandler(int SignNo){
    printf("Clock\n");
}
int main(){
    signal(SIGALRM,SignHandler);
    itimerval tval;
    TimeInt2Obj(1,&tval.it_value);        //设初始间隔为1毫秒，注意不要为0
    TimeInt2Obj(1500,&tval.it_interval);//设置以后的重复间隔为1500毫秒
     setitimer(ITIMER_REAL,&tval,NULL);
    while(getchar()!=EOF);
}
```

进程间通信(IPC)

进程间通信(IPC)

- 之前进程间交换信息的方法只能是由fork或exec传送文件
- IPC—进程 间通信
  - 管道
  - 消息队列
  - 信号量
  - 共享存储
  - 套接字
  - 。。。

管道

- 管道是Unix系统IPC最古老的形式
- 历史上它是半双工的(数据只能在一个方向流动)
  - 现在很多系统都提供全双工管道
- 有名管道
  - mkfifo函数创建有名管道文件

序号	进程A	进程B	动作
1	建立管道文件		mkfifo
2	打开管道	打开管道	Open
3	读写数据	读写数据	write/read
4	关闭管道	关闭管道	close
5	删除管道		unlink

无名管道

- 适用范围:父子进程
- int pipe(int fd[2]);
  - 0：成功 -1：失败
  - 参数：返回两个文件描述符号
    - fd[0] 读
    - fd[1] 写
  - 建立起与当前进程的管道几乎是没有任何用处
  - 通常，需要用fork建立一个子进程
  - 父子进程 之间，一个关闭读端，一个关闭写端
  - 进行通信
  - 关闭

XSI IPC

- 消息队列　信号量　共享内存统称为XSI IPC
  - Inter Process Communication
- 每个内核中的IPC结构，都用一个非负整数的标识符加以引用，与文件描述符不同，IPC标识符使用时会连续加1，当达到整数的最大值时，转回0
- 标识符是IPC对象的内部名。多个进程在同一IPC对象上会合，需要提供外部名方案，为此使用Key.
- 无论何时创建IPC结构，都应指定一个键。
- 键的数据类型是基本数据类型key_t,其实是在头文件<sys/types.h>中定义的长整形。

客户进程和服务进程在IPC上会合

- 服务进程指定键IPC_PRIVATE创建新的IPC结构，将返回的标识存放在某处(如文件)以便客户进程取用。
- 在一个公用头文件中定义一个客户进程和服务器进程都认可的键，服务进程按此键创建IPC结构
- 客户进程和服务进程认同一个路径名和项目ID(0~255)，接着调用函数ftok将两个值变换为一个键。
  - key_t ftok(const char* path, int id);
  - Path必需是一个现存的文件或目录的路径。
  - 只使用id的低8位

IPC结构

- 三种IPC结构的创建函数都有两个类似的参数，一个key和一个整形flag,如满足以下两个条件 ，则创建结构
  - Key是IPC_PRIVATE
  - key当前未与特定的IPC结构相结合，并且flag指定了IPC_CREAT位
  - 创建新IPC结构时，要确保不是引用具有同一标识符的一个现行IPC结构，就需要flag指定IPC_CREAT和IPC_EXCL。这样如果IPC结构已经存在，就会出错。

共享内存

- 两个或更多的进程共享一个给定的存储区
- 最快的一种IPC，因为不需要复制信息
- 问题是同步访问
- 需要使用的函数
  - 获取共享存储标识符
    - int shmget(key_t key,size_t sz, int flag);
    - 函数会根据key创建或引用现有的共享存储，当创建时，初始化shmid_ds结构中的某些成员
    - sz为共享存储的长度。通常为系统页长的整数倍。如果创建新段，必需指定。如果引用一个现存段，则为0

共享内存

- 连接到地址空间
  - void* shmat(int shmid, const void*addr, int flag);
  - 共享存储段连接到进程的哪个地址上与addr,和flag是否指定SHM_RND有关
  - 一般情况下，addr为0，连接到第一个可用的地址上。除非只计划在一种指定的硬件上运行程序，否则不应指定连接地址。
  - 如果flag中指定了SHM_RDONLY,则以只读方式连接
  - 返回值是该段所连接的实际地址，如出错返回-1，如果成功，共享存储段shmid_ds结构中的shm_nattch计数器加1
  - flag建议使用0：默认方式挂载：读写
- 脱接内存段
  - int shmdt(void *addr)
  - addr是shmat返回的地址。如果成功，shm_mattch减1
  - 对共享存储使用完膈调用此函数。
  - 关不从系统中删除其标识符和数据结构。

对共享内存段进行操作

- int shmctl(int shmid, int cmd, struct shmid_ds* buf);
- cmd参数定义了如下5种命令，使其在shmid指定的段上
  - IPC_STAT 将此段的shmid_ds结构，存放在由 buf指定的结构中
  - IPC_SET 按buf指向结构中的值设置与此段相关结构中的下列三个字段：shm_perm.uid，shm_perm.gid以及shm_perm.mode. 只能两种进程执行
    - 有效用户ID等于shm_perm.uid或shm_perm.cuid的进程
    - 超级用户进程
  - IPC_RMID 从系统中删除该共享段。
    - 除非使用该段的最后一个进程终止或与该段脱接，否则不会实际删除该段
    - 不管此段是否仍在使用，该段标识符立即被删除，所以不能再用shmat与该段连接只能两种进程执行
      - 有效用户ID等于shm_perm.uid或shm_perm.cuid的进程
      - 超级用户进程

编程模型

序号	服务进程	客户进程	动作
1	使用约定文件创建KEY	使用约定文件创建KEY	ftok
2	使用KEY创建共享内存	使用KEY获取共享内存ID	shmget
3	挂载到共享内存	挂载到共享内存	shmat
4	使用内存	使用内存	
5	卸载共享内存	卸载共享内存	shmdt
6	释放共享内存		shmctl

查看IPC的命令

- 显示
  - ipcs -a 显示所有共享内核对象
  - ipcs -m 显示共享内存　m=memory
  - ipcs -q 显示共享队列　q=queue
  - ipcs -s 显示信号量　s=semphore
- 删除
  - ipcrm -m ID　删除共享内存
  - ipcrm -q ID　删除共享队列
  - ipcrm -s ID　删除信号量

消息队列

- 消息的链接表，存放在内核中并由消息队列标准符标识
- 用msgget有来创建或打开一个队列,msgsend将新消息加入到队列尾端，用msgrcv从队列中取消息。
- 每个消息包含一个正长整形字段，一个非负长度及实际数据字节
- 每个队列都有一个msqid_ds结构与之关联

消息队列

- 打开一个现有队列或创建一个新队列

  int msgget(key_t key, int flag);

  当创建新队列时，会初始化msqid_ds结构中的相关成员

  如果执行成功，返回非负队列ID

- 将数据放入消息队列

  int msgsnd(int msgid, const void* ptr, size_t sz, int flag);

  ptr指向一个长整数，它包含正的消息类型，其后紧跟着消息数据。比如，发送的消息是512个字节，可定义下列结构

```
struct mymesg{
    long mtype;
    char mtext[512];
};
```

  sz并不包含long类型所占的空间

  flag可以是IPC_NOWAIT，指示消息发送是否阻塞

消息队列

- 从队列中取用消息
  - ssize_t msgrvc(int msqid, void* ptr, size_t sz, long type, int flag);
  - prt和sz参数和发送相同
  - 如果sz比返回的消息小，而且flag指定了MSG_NOERROR,则消息被截断，如果没有设置该标志，消息又太长，则出错返回E2BIG(消息仍留在队列中)
  - 参数type让我们可以指定哪种消息
    - ==0 返回队列中的第一个消息
    - >0 返回队列中类型为type的第一个消息
    - <0 返回队列中消息类型小于或等于type绝对值的消息，如果有多个，则取类型值最小的消息。
  - 可以指定flag值为IPC_NOWAIT，使用其操作不阻塞。这使得如果没有指定类型的消息，则msgrcv返回-1.errno设置ENOMSG

msgctl函数

- 和共享存储段一样，消息队列的函数msgctl和shmctl操作几乎一样。
- int msgctl(int msqid, int cmd, struct msqid_ds* buf)

信号量

- 信号量与前几种IPC不同，它是一个计数器，用于多进程对共享数据对象的访问。
- 进程为获去共享资源，需要执行以下操作
  - 测试控制该资源的信号量
  - 若此信号量的值为正，则进程可以使用资源。进程将信号量减1，表示它使用了资源
  - 若此信号量值为0，则进程进入休眠状态，直到信号量大于0，进程被唤醒，返回第1步执行
  - 当进程不再使用由一个信号量控制的共享资源时，该信号量值增1。如果有进程正在休眠等待此信号量，则唤醒它

信号量

- 获得一个信号量集ID
  int semget(key_t key, int nsems, int flag)
  nsems是该集合中的信号量数，如果创建，必需指定
- 操作信号量
  int semctl(int semid, int semnum, int cmd,
  /*union senum arg*/)
  第四个参数是可选的，它是多个特定命令的参数联合
  union semun{
  int val; /*for SETVAL*/
  struct semid_ds *buf; /*for IPC_STAT IPC_SET*/
  unsigned short *array; /*for GETALL SETALL*/
  }
  注意：此联合可能需要自己定义

信号量

- 执行信号量集合上的操作数组

  int semop(int semid, struct sembuf semoparray[],size_t nops);

  参数semoparray是一个指针，它指向一个信号量操作数组，信号量操作由sembuf结构表示：

  ```
  struct sembuf{
     unsigned short sem_num;//操作信号量的下标
     short sem_op; //对信号量操作方式。 负数，0，正数
     short sem_flg; //信号量的操作标记，默认为0
  };
  ```

  如果sem_op为正，则对应于进程释放占用的资源数，sem_op值加到信号量上，如果指定了undo标志(sem_flg成员设置了SEM_UNDO位），则也从该进程的此信号量中减去sem_op

  若sem_op为负，则表示要获取该信号量控制的资源。

多线程编程

主要内容

- 线程/进程基本概念
- 线程管理
- 线程属性控制
- 线程通信

进程与线程

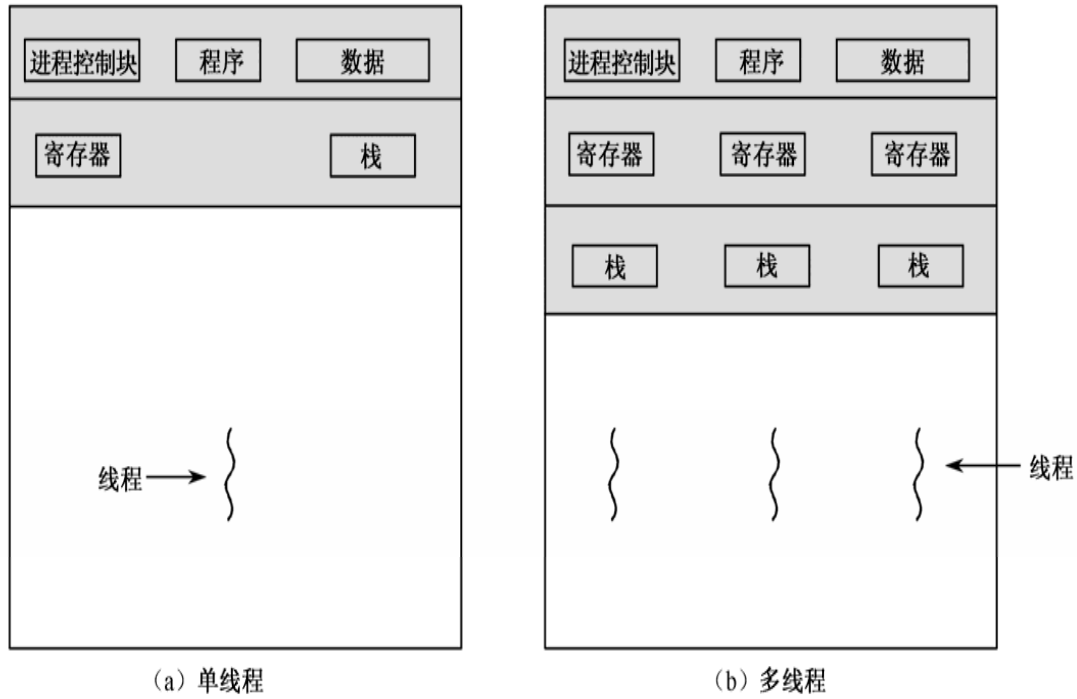
- 进程
  - 资源分配单位
  - 进程的上下文组成
    - 进程控制块PCB：包括进程的编号、状态、优先级以及正文段和数据段中数据分布的大概情况
    - 正文段(text segment)：存放该进程的可执行代码
    - 数据段(data segment)：存放进程静态产生的数据结构
    - 用户堆栈(stack)
- 线程
  - CPU调度基本单位

进程执行状态相关信息

- 进程 ID、进程组 ID、用户 ID和组 ID
- 工作环境（Environment）
- 工作目录（Working directory）
- 程序指令（Program instructions）
- 寄存器（Registers）
- 栈（Stack）
- 堆（Heap）
- 文件描述符（File descriptors）
- 信号（Signal actions ）
- 共享库（Shared libraries）
- 进程间通信手段（Inter-process communication tools）

线程与进程的关系

- 包含在进程中的一种实体
  - 有自己的运行线索，可完成特定任务
  - 可与其他线程共享进程中的共享变量及部分环境
  - 可通过相互之间
  - 线程维护的信息
    - 堆栈指针
    - 寄存器
    - 调度属性
    - 信号
    - 线程私有数据



(a) 单线程　　(b) 多线程

线程的基本特点

- 是进程的一个实体，可作为系统独立调度和分派的基本单位
- 有不同的状态，有控制线程的各种原语，包括创建和撤销线程等
- 不拥有系统资源（只拥有从属进程的全部资源，资源是分配给进程）
- 一个进程中的多个线程可并发执行
  - 进程中创建的线程可执行同一程序的不同部分
  - 也可以执行相同代码
- 系统开销小、切换快
  - 进程的多个线程都在进程的地址空间活动，共享全局变量

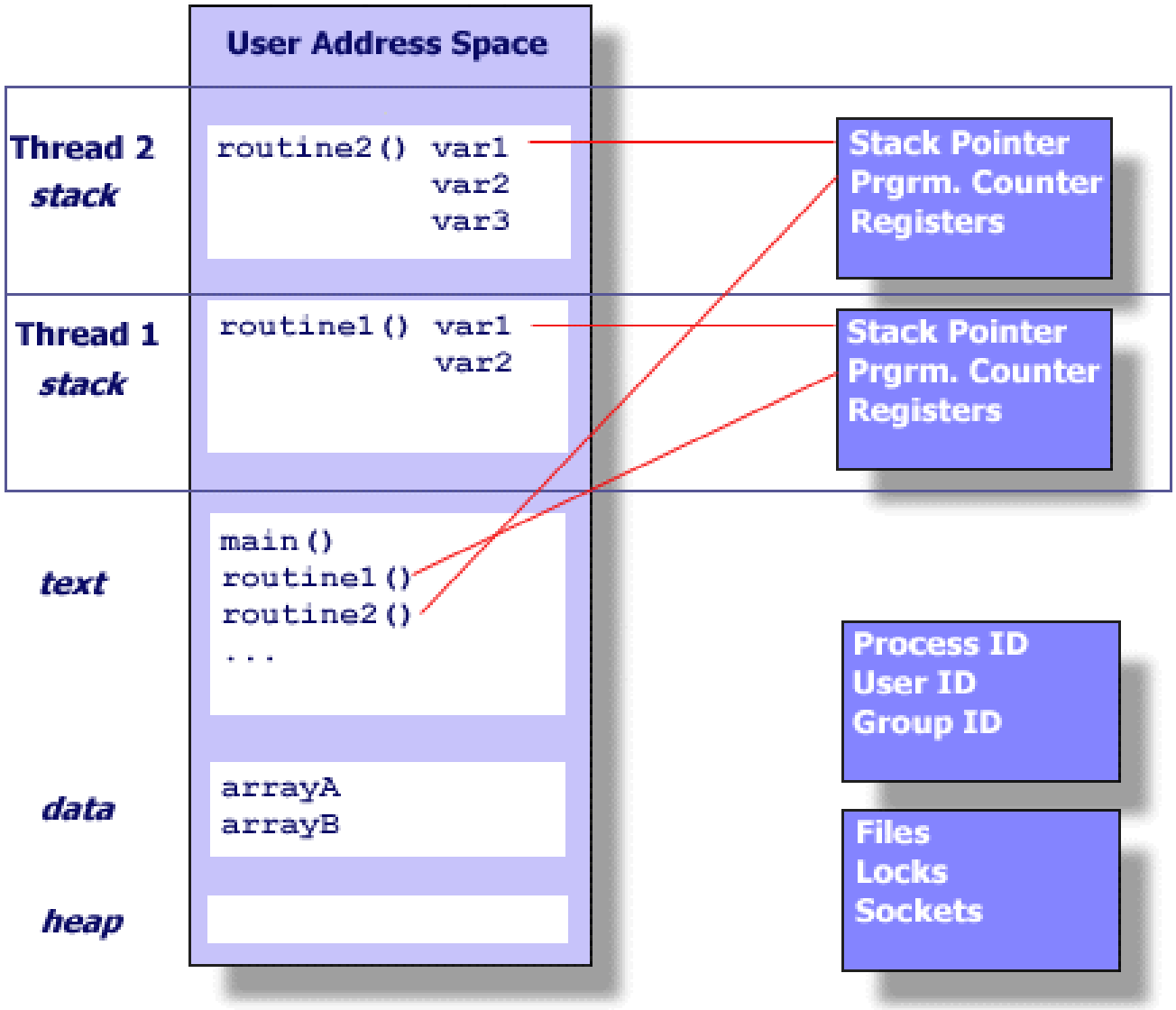
线程之间的共性特征

- 同一进程中的多个线程共享该进程的虚拟空间
  - 进程代码段
  - 进程的公有数据
    - 利用这些共享的数据，线程很容易的实现相互通讯
  - 进程打开的文件描述符
  - 信号的处理器
  - 进程的当前目录和进程用户ID与进程组ID
- 说明
  - 对不同进程来说，具有独立的数据空间，数据传递只能通过通信的方式进行，这种方式费时而不方便

线程的个性特征

- 线程是实现并发的必要条件
  - 线程ID
    - 每个线程都有自己唯一的线程ID
  - 寄存器组的值
    - 创建线程时，须将原有线程的寄存器集合的状态保存
  - 线程的堆栈
    - 线程必须拥有自己的函数堆栈，使得函数调用可以正常执行，不受其他线程的影响
  - 错误返回码
    - 不同线程应该拥有自己的错误返回码变量
  - 线程的信号屏蔽码
    - 线程的信号屏蔽码应由线程自己管理，但所有线程都共享同样的信号处理器
  - 线程的优先级

线程组成员之间的关系


User Address Space
Thread 2
stack
routine2() var1
var2
var3
Thread 1
stack
routine1() var1
var2
text
main()
routine1()
routine2()
...
data
arrayA
arrayB
heap
Stack Pointer
Prgrm. Counter
Registers
Stack Pointer
Prgrm. Counter
Registers
Process ID
User ID
Group ID
Files
Locks
Sockets

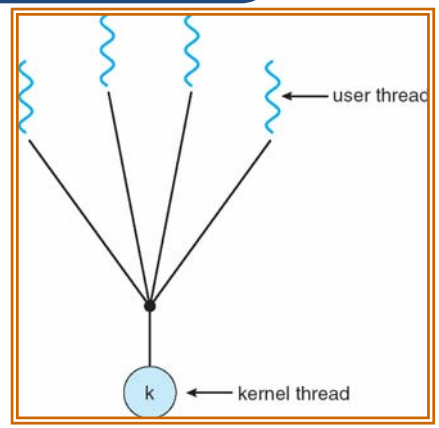
用户线程

- 用户线程存在于用户空间，通过线程库来实现
- 线程库提供对线程的创建、调度和管理的支持，而无须内核支持
- 内核并不知道用户级线程，所有线程的创建和调度都在用户空间内进行，无须内核干预
- 用户级线程的调度以进程为单位
- 优点
  - 同一进程内的线程切换不需要转换到内核，调度算法是进程专用的
- 缺点
  - 系统调度阻塞问题，不能充分利用多处理器

内核线程

- 由操作系统直接支持，内核在其空间内执行线程的创建、调度和管理
- 由于线程管理由操作系统完成，因此内核线程的创建和管理要慢于用户线程的创建和管理
- 优点
  - 支持多处理器，支持用户进程中的多线程、内核线程切换的速度快
- 缺点
  - 对用户的线程切换来说，系统开销大

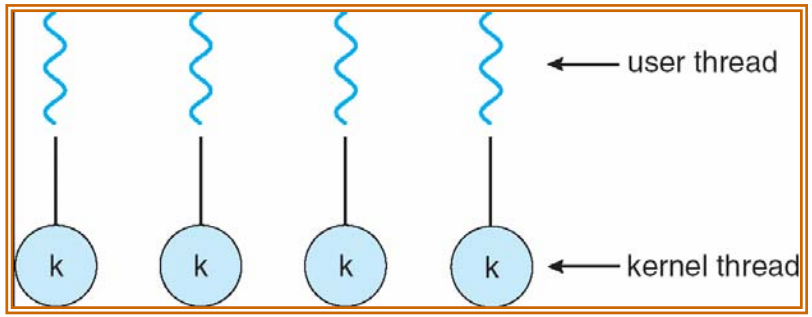
线程-多线程模型



- 多对一模型
  - 将许多用户级线程映射到一个内核线程
  - 线程管理在用户空间进行，效率较高
  - 处理机调度的单位仍然是进程
  - 缺点
    - 如果一个线程执行了阻塞系统调用，那么整个进程就会阻塞
    - 因为任何时刻只有一个线程访问内核，多个线程不能并行运行在多处理器上
- 说明
  - 在不支持内核级线程的操作系统上所实现的用户级线程库也使用多对一模型

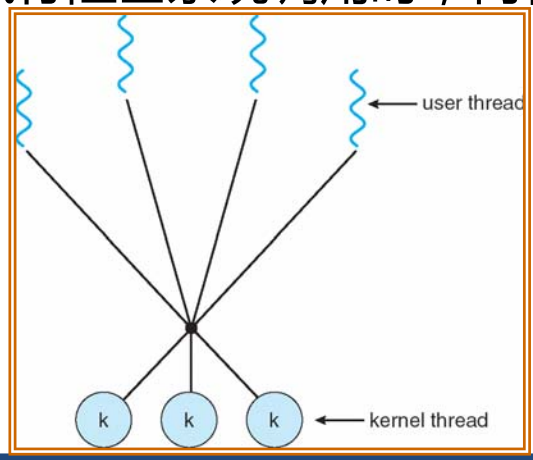
线程-多线程模型

- 一对一模型
  - 将每个用户线程映射到一个内核线程
  - 在一个线程执行阻塞时，允许另一个线程继续执行
  - 允许多个线程运行在多处理机系统上
  - 提供比多对一模型更好的并发功能
  - 缺点
    - 创建一个用户线程就需要创建一个相应内核线程
    - 创建内核线程的开销会影响应用程序的性能，这种模型的绝大多数实现限制系统所支持的线程数量



线程-多线程模型

- 多对多模型
  - 多路复用许多用户级线程到同样数量或更小数量的内核线程上
  - 内核线程的数量可能与特定应用程序或特定机器有关
  - 克服前两种模型的缺点
    - 开发人员可以创建任意多的必要的线程，并且相应内核线程能在多处理器系统上并行运行
    - 当一个线程执行阻塞系统调用时，内核能调度另一个线程来执行



主要内容

- 线程/进程基本概念
- 线程管理
- 线程属性控制
- 线程通信

pthread背景

- 早期各硬件厂商主要使用私有版本线程库，实现差异非常大，开发者难于开发可移植的线程应用
- 为能够最大限度的提高线程的性能，需要一个标准的编程接口
  - 对于UNIX系统，IEEE POSIX 1003.1c标准 (1995)定义了这样的接口
  - 遵从该标准实现的线程被称做POSIX线程，或pthreads
    - pthreads定义了一套C语言编程接口和函数调用
    - 包括一个pthread.h头文件和一个线程库

pthread介绍

- 基于POSIX标准的线程编程接口
  - 包括一个pthread.h头文件和一个线程库
  - 编译方法 gcc –g **.c -o *** –lpthread
- 功能
  - 线程管理
    - 支持线程创建/删除、分离/联合，设置/查询线程属性
  - 互斥
    - 处理同步，称为“mutex”
      - 创建/销毁、加锁/解锁互斥量，设置/修改互斥量属性
  - 条件变量
    - 支持基于共享互斥量的线程间通信
      - 建立/销毁、等待/触发特定条件变量，设置/查询条件变量属性

pthread类型

- 线程管理
  - 支持线程创建、分离、联合等，还包括线程属性的设置/查询
- 互斥
  - 处理同步，称为“mutex”
  - 提供创建、销毁、加锁和解锁互斥量
  - 也包括补充的修改互斥量属性功能，并用它去设置或者修改与互斥相关的属性
- 条件变量
  - 支持基于共享互斥量的线程间通信，以开发者的特定条件变量为基础。
  - 包括基于特定条件变量的建立、销毁、等待和信号触发
  - 设置/查询条件变量属性的功能也包括在内

pthread变量类型的命名规则

前缀形式	功能组
pthread_	线程自身及其他子例程
pthread_attr_	线程属性对象
pthread_mutex_	互斥变量
pthread_mutexattr_	互斥属性对象
pthread_cond_	条件变量
pthread_condattr_	条件属性对象
pthread_key_	特定线程数据键

线程管理

- 线程创建
- 线程终止
- 线程终止时的资源清理
- 说明
  - 头文件
    - #include <pthread.h>

线程创建

- 函数原型
  - int pthread_create(pthread_t *thread, const pthread_attr_t *attr，void *(*start_routine) (void*)，void * arg);
- 参数说明
  - thread：要创建的线程id指针
  - attr：创建线程时的线程属性
  - v(*start_routine)(void*)：返回值是void*类型的指针函数
  - arg：start_routine的参数
- 返回值
  - 成功返回0
  - 失败返回错误编号
    - EAGAIN：表示系统限制创建新的线程，如线程数目过多
    - EINVAL：代表线程属性值非法

线程示例

```
#include <pthread.h>
#include <stdio.h>
void *create(void *arg) {
    printf("new thread created ..... ");
}

int main(int argc, char *argv[]){
        pthread_t tidp;
        int error;
        error=pthread_create(&tidp, NULL, create, NULL);
          if(error != 0)
           {
              printf("pthread_create is not created ... ");
              return -1;
           }
          printf("prthread_create is created... ");
          return 0;
}
```

pthread_create()的参数传递问题

- 基本问题
  - int pthread_create(pthread_t *thread, const pthread_attr_t *attr , void *(*start_routine) (void*) , void * arg);
    - 仅允许传递一个参数给线程待执行的函数
    - 如何传递多个参数
- 解决途径
  - 构造一个包含所有参数的结构，将结构指针作为参数传递给pthread_create()
  - 所有参数必须利用(void *)来传递

pthread_create()参数传递—错误示例

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8

void *PrintHello(void *threadid){
   int *id_ptr, taskid;
   sleep(1);
   id_ptr = (int *) threadid;
   taskid = *id_ptr;
   printf("Hello from thread %d\n", taskid);
   pthread_exit(NULL);
}
```

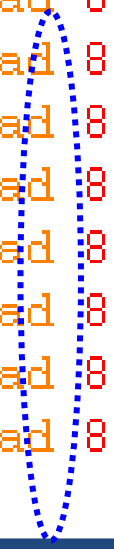
pthread_create()参数传递—错误示例

```
int main(int argc, char *argv[]){
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t<NUM_THREADS;t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL,
         PrintHello, (void *) &t);
      if (rc) {
         printf("ERROR; return code is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

pthread_create()参数传递—错误示例

- 运行结果

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
```



pthread_create()参数传递—正确示例1

- 传递一个简单整数

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8
char *messages[NUM_THREADS];
void *PrintHello(void *threadid){
   int *id_ptr, taskid;
   sleep(1);
   id_ptr = (int *) threadid;
   taskid = *id_ptr;
   printf("Thread %d: %s\n", taskid, messages[taskid]);
   pthread_exit(NULL);
   }
```

pthread_create()参数传递—正确示例1

```
int main(int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int *taskids[NUM_THREADS];
  int rc, t;
  messages[0] = "English: Hello World!";
  messages[1] = "French: Bonjour, le monde!";
  messages[2] = "Spanish: Hola al mundo";
  messages[3] = "Klingon: Nuq neH!";
  messages[4] = "German: Guten Tag, Welt!";
  messages[5] = "Russian: Zdravstvytye, mir!";
  messages[6] = "Japan: Sekai e konnichiwa!";
  messages[7] = "Latin: Orbis, te saluto!";
  for(t=0;t<NUM_THREADS;t++) {
    taskids[t] = (int *) malloc(sizeof(int));
    *taskids[t] = t;
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL,
      PrintHello, (void *) taskids[t]);
    if (rc) {
      printf("ERROR; return code is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);}
}
```

pthread_create()参数传递—正确示例1

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!
Thread 1: French: Bonjour, le monde!
Thread 2: Spanish: Hola al mundo
Thread 3: Klingon: Nuq neH!
Thread 4: German: Guten Tag, Welt!
Thread 5: Russian: Zdravstvytye, mir!
Thread 6: Japan: Sekai e konnichiwa!
Thread 7: Latin: Orbis, te saluto!
```

pthread_create()参数传递—正确示例2

- 向新建线程同时传入线程号/消息/线程号总和

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8
char *messages[NUM_THREADS];
struct thread_data{
   int   thread_id;
   int   sum;
   char *message;
};
struct thread_data thread_data_array[NUM_THREADS];
```

pthread_create() 参数传递—正确示例2

- 向新建线程同时传入线程号/消息/线程号总和

```
void *PrintHello(void *threadarg){
   int taskid, sum;
   char *hello_msg;
   struct thread_data *my_data;
   sleep(1);
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
   pthread_exit(NULL);
}
```

pthread_create()参数传递—正确示例2

- 向新建线程同时传入线程号/消息/线程号总和

```
int main(int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int *taskids[NUM_THREADS];
  int rc, t, sum;
  sum=0;
  messages[0] = "English: Hello World!";
  messages[1] = "French: Bonjour, le monde!";
  messages[2] = "Spanish: Hola al mundo";
  messages[3] = "Klingon: Nuq neH!";
  messages[4] = "German: Guten Tag, Welt!";
  messages[5] = "Russian: Zdravstvytye, mir!";
  messages[6] = "Japan: Sekai e konnichiwa!";
  messages[7] = "Latin: Orbis, te saluto!";
```

pthread_create()参数传递—正确示例2

- 向新建线程同时传入线程号/消息/线程号总和

```
  for(t=0;t<NUM_THREADS;t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
      (void *)&thread_data_array[t]);
    if (rc) {
      printf("ERROR; return code f is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

pthread_create()参数传递—正确示例2

- 向新建线程同时传入线程号/消息/线程号总和

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!  Sum=0
Thread 1: French: Bonjour, le monde!  Sum=1
Thread 2: Spanish: Hola al mundo  Sum=3
Thread 3: Klingon: Nuq neH!  Sum=6
Thread 4: German: Guten Tag, Welt!  Sum=10
hread 5: Russian: Zdravstvytye, mir!  Sum=15
Thread 6: Japan: Sekai e konnichiwa!  Sum=21
Thread 7: Latin: Orbis, te saluto!  Sum=28
```

线程ID的访问

- 获取线程自身的id
  - 函数原型　　pthread_t pthread_self(void);
  - 返回值　　调用线程的线程id
- 比较线程ID
  - 函数原型　　int pthread_equal(pthread_t tid1, pthread_t tid2);
  - 参数
    - tid1：线程1的id
    - tid2：线程2的id
  - 返回值
    - 相等返回非0值
    - 否则返回0

线程终止

- 正常终止
  - 方法1：线程自己调用pthread_exit()
    - void pthread_exit(void *rval_ptr);
    - rval_ptr：线程退出返回的指针，进程中其他线程可调用pthread_join（ ）访问到该指针
  - 方法2：在线程函数执行return
- 非正常终止
  - 其它线程的干预
  - 自身运行出错

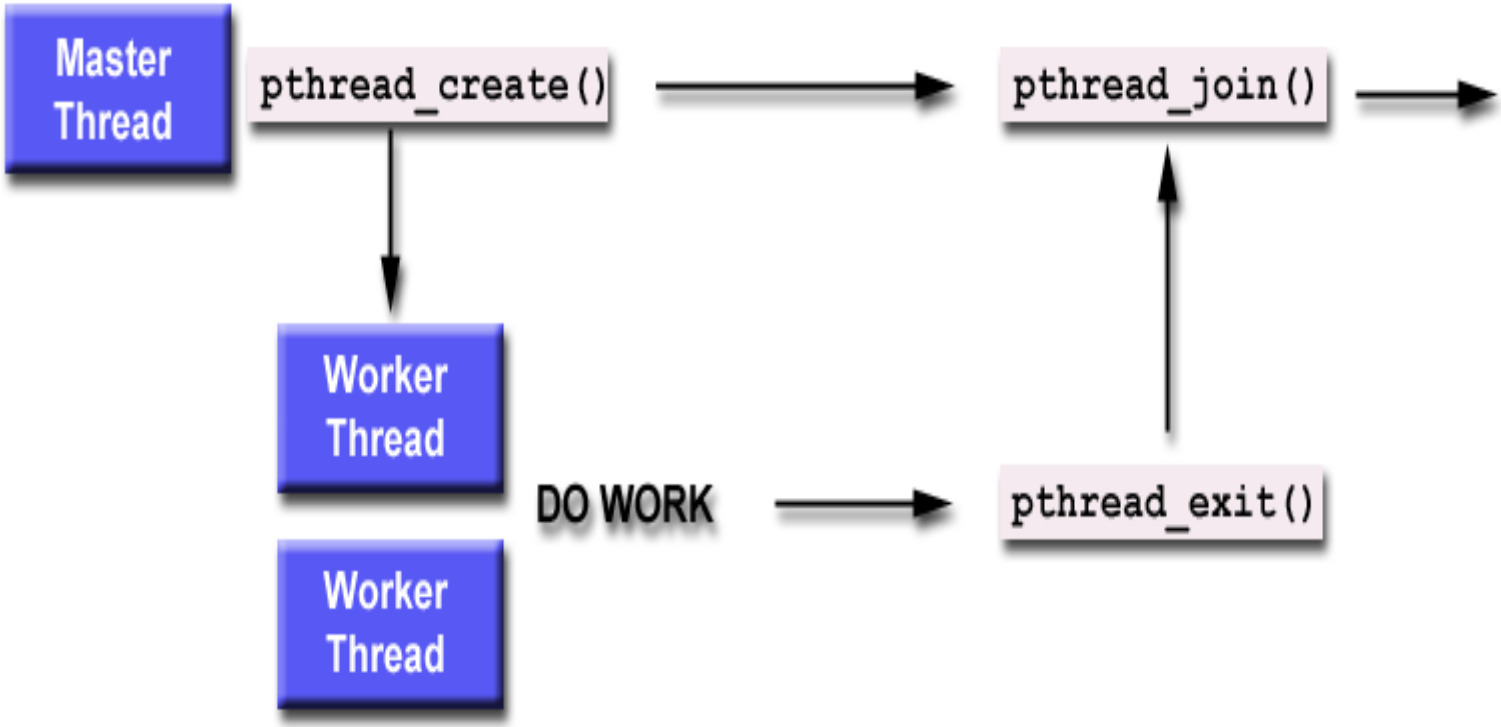
线程执行轨迹

- 同步方式（非分离状态）
  - 等待新创建线程结束
  - 只有当pthread_join()函数返回时，创建的线程才算终止，才可释放自己占用的系统资源
- 异步方式（分离状态）
  - 未被其他线程等待，自己运行结束即可终止线程并释放系统资源

线程同步终止

- 函数原型
  - int pthread_join(pthread_t thread, void ** rval_ptr);
- 功能
  - 调用者将挂起并等待新进程终止
  - 当新线程调用pthread_exit()退出或者return时，进程中的其他线程可通过pthread_join()获得进程的退出状态
- 使用约束
  - 一个新线程仅仅允许一个线程使用该函数等待它终止
  - 被等待线程应处于可join状态，即非DETACHED状态
- 返回值
  - 成功结束返回值为0,否则为错误编码
- 说明
  - 类似于waitpid()

线程同步终止



线程分离终止

- 函数原型
  - int pthread_detach(pthread_t thread)
- 功能
  - 执行该函数后线程处于DETACHED状态
  - 处于该状态的线程结束后自动释放内存资源，不能被pthread_join()同步
- 说明
  - 当线程被分离时，不能用pthread_join()等待其终止状态
  - 为避免内存泄漏，线程终止要么处于分离状态，要么处于同步状态

线程分离终止示例

```c
#include <stdio.h>
#include <pthread.h>

pthread_t tid[2];
void * func_1(void *arg){
  //为函数func_2创建线程
  pthread_create(&tid[1],NULL,func_2,NULL);
  //tid[0]将自己挂起，等待线程tid[1]的结束
  pthread_join(tid[1],NULL);
}
void * func_2(void *arg){
  //函数内容
}
int main(){
  //为函数func_1创建线程并将其分离
  pthread_create(&tid[0],NULL,func_1,NULL);
  pthread_detach(tid[0],NULL);
  //主进程不退出，但这两个线程的资源都可以释放掉
  while(1){
    sleep(10);
  }
  return 0;
}
```

线程取消

- 线程取消定义
  - 一般情况下，线程在其主体函数退出时自动终止，但也可因接收到另一个线程发来的终止（取消）请求而强制终止
- 线程取消方法
  - 向目标线程发CANCEL信号，但如何处理由目标线程决定
    - 忽略、立即终止或者继续运行至取消点
  - 线程接收到CANCEL信号的缺省处理（即pthread_create()创建线程的缺省状态）是继续运行至取消点
- 函数原型
  - int pthread_cancel(pthread_t thread);
  - 功能说明
    - 发送终止信号给thread线程，发送成功并不意味着thread会终止
  - 返回值
    - 若成功返回0，否则返回错误编号

线程取消相关函数

- int pthread_setcancelstate(int state, int *oldstate)
  - 设置本线程对CANCEL信号的反应
  - state
    - PTHREAD_CANCEL_ENABLE（缺省）：收到信号后设为CANCLED状态
    - PTHREAD_CANCEL_DISABLE：忽略CANCEL信号继续运行
  - old_state
    - 如果不为 NULL，则存入原来的Cancel状态以便恢复

线程取消相关函数(续)

- int pthread_setcanceltype(int type, int *oldtype)
  - 设置本线程取消动作的执行时机
  - type：仅当Cancel状态为Enable时该参数有效
    - PTHREAD_CANCEL_DEFFERED: 收到信号后继续运行至下一个取消点再退出
    - PTHREAD_CANCEL_ASYCHRONOUS: 立即执行取消动作（退出）
  - oldtype
    - 如果不为NULL则存入运来的取消动作类型值
- void pthread_testcancel(void)
  - 检查本线程是否处于Canceld状态
    - 如果是，则进行取消动作，否则直接返回

线程清理

- 线程可安排在退出时需要调用的函数，这样的函数称为线程清理处理程序
- 线程可建立多个清理处理程序（处理程序保存在栈中），即它们的执行顺序与它们注册时的顺序相反
- 函数原型
  - void pthread_cancel_push(void (*rtn)(void *)，void *arg);
  - void pthread_cancel_pop(int execute);
- 参数说明
  - rtn：处理程序入口地址
  - arg：传递给处理函数的参数
- 说明
  - 两个函数必须成对使用
  - 如果线程是通过从他的启动例程中返回而终止，它的处理程序就不会调用

线程清理示例

```
#include <pthread.h>
#include <stdio.h>
void cleanup(void *arg){
  printf("cleanup: %s\n",(char *)arg);
}
void *thr_fn(void *arg){ /*线程入口地址*/
  printf("thread start\n");
  /*设置第一个线程处理程序*/
  pthread_cleanup_push(cleanup,"thread first handler");
  /*设置第二个线程处理程序*/
  pthread_cleanup_push(cleanup,"thread second handler");
  printf("thread push complete\n");
  pthread_cleanup_pop(0); /*取消第一个线程处理程序*/
  pthread_cleanup_pop(0); /*取消第二个线程处理程序*/
}

int main(){
  pthread_t tid;
  void *tret;
  /*创建一个线程*/
  pthread_creat(&tid,NULL,thr_fn,(void *)1);
  pthread_join(tid,&tret); /*获得线程终止状态*/
  ptinrf("thread exit code %d\n",(int)tret);
}
```

主要内容

- 线程/进程基本概念
- 线程管理
- 线程属性控制
- 线程通信

线程属性

- 属性值不能直接设置，须使用相关函数进行操作
  - 初始化函数为pthread_attr_init，该函数必须在pthread_create函数之前调用

```
typedef struct{
    int  detachstate;  // 线程的分离状态
    int   scope;   // 线程绑定状态
    int  schedpolicy; // 线程调度策略
    struct sched_param schedparam; // 线程的调度参数
    int   inheritsched;  //线程的继承性
    size_t  guardsize; // 线程栈末尾的警戒缓冲区大小
    void *  stackaddr;  // 线程栈的位置
    size_t  stacksize;   // 线程栈的大小
}pthread_attr_t;
```

线程属性初始化

- 函数原型 int pthread_attr_init(pthread_attr_t *attr);

属性	值	说明
scope(线程域)	PTHREAD_SCOPE_PROCESS	线程是非绑定的，即不是永久的绑定在LWP上
Detachstate(分离状态)	PTHREAD_CREATE_JOINABLE	线程退出后，线程退出代码和线程都将暂时保留
Stackaddr(堆栈地址)	NULL	线程的堆栈由系统分配
Stacksize(堆栈大小)	1M bytes	线程的堆栈大小由系统决定
priority(优先级)		线程从父线程继承线程的优先级
Inheritsched(继承调度优先级)	PTHREAD_INHERIT_SCHED	线程继承父线程的调度优先级
schedpolicy（调度策略）	SCHED_OTHER	线程根据优先级调度，线程保持运行，直到更高优先级的线程强占了处理器资源，或是线程被阻塞，或是线程主动出让运行权

删除线程属性对象

- 函数原型
  - int pthread_attr_destroy(pthread_attr_t *attr);
- 返回值
  - 成功返回0
  - 出错，返回其他值
- 说明
  - 删除属性对象占用的内存
  - 调用这个函数后，相应的属性对象无效

线程分离状态(detachstate)

- 决定线程以何种方式终止自己
  - 非分离状态
    - 原有的线程等待创建的线程结束
    - 只有当pthread_join（ ）函数返回时，创建的线程才算终止，才能释放自己占用的系统资源
  - 分离线程
    - 没有被其他的线程所等待，自己运行结束后终止线程，马上释放系统资源
- 线程分离状态相关函数
  - pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)
    - detachstate选值
      - PTHREAD_CREATE_DETACHED（分离线程）
      - PTHREAD _CREATE_JOINABLE（非分离线程）
  - int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);

主要内容

- 线程/进程基本概念
- 线程管理
- 线程属性控制
- 线程通信

线程的互斥机制

- mutex变量就像一把锁，是线程同步和保护共享数据的主要方式
- mutex可以用来阻止竞争
- 在任何时候，只有一个线程能够获得mute
  - 尽管几个线程想获取一个mutex，但是只有一个线程能够成功
    - 其他线程需要等待，直到获取mutex的线程放弃mutex
    - 线程必须轮流访问需要保护的数据
- 线程经常利用mutex来加锁需要更新的全局变量，这也是几个线程需要同时更新全局变量时使用的安全方法
- 这样能保证在多线程环境下的全局变量的更新就如在单线程环境中一样
- 此全局变量的更新属于“临界区"

mutex的一般使用步骤

- 创建和初始化mutex
- 使用mutex
  - 各线程尝试获取mutex
    - 但仅有一个线程能够获取mutex并拥有它
  - 拥有mutex的线程执行需访问临界资源的特定处理例程
  - 拥有线程释放mutex
  - 其他线程阐述获取mutex
  - 重复上述步骤
- 销毁 mutex

mutex变量创建/销毁函数

- pthread_mutex_init(mutex,attr)
- pthread_mutex_destroy(mutex)
- pthread_mutexattr_init(attr)
- pthread_mutexattr_destroy(attr)

互斥锁创建

- 声明mutex变量：pthread_mutex_t类型
- 在使用前必须已经初始化（两种方式）
  - 静态方式
    - pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER
  - 动态方式
    - int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
  - 返回值
    - 成功返回0
    - 失败返回错误编号
  - 说明
    - mutex初始时是unlocked（未加锁的）的

互斥锁的类型

- 普通锁（PTHREAD_MUTEX_TIMED_NP）
  - 默认值，当一个线程加锁后，其余请求锁的线程形成一个等待队列，并在解锁后按优先级获得锁
- 嵌套锁（PTHREAD_MUTEX_RECURSIVE_NP）
  - 允许同一个线程对同一个锁成功获得多次，并通过多次unlock解锁
  - 如果是不同线程请求，则在加锁线程解锁时重新竞争
- 检错锁（PTHREAD_MUTEX_ERRORCHECK_NP）
  - 如果同一个线程请求同一个锁，则返回EDEADLK，否则与普通锁动作相同
- 适应锁（PTHREAD_MUTEX_ADAPTIVE_NP）
  - 动作最简单的锁类型，仅仅等待锁解锁后重新竞争

互斥锁的属性

- 用于设置mutex对象属性
  - 如果使用它，那么它一定是 pthread_mutexattr_t 类型 (可以设置NULL作为缺省的)。
- pthreads标准定义三种可选的mutex属性
  - protocol: 利用特定的协议来防止mutex优先级倒置
  - prioceiling: 特定的mutex优先级限制
  - process-shared: 特定进程共享mutex
- 说明
  - 并不是所有的应用提供所有这三种mutex属性
- 属性操作函数
  - pthread_mutexattr_init()：创建mutex属性对象
  - pthread_mutexattr_destroy()：销毁mutex属性对象

互斥锁的销毁

- 函数原型
  - int pthread_mutex_destroy(pthread_mutex_t *mutex)
    - 销毁一个互斥锁
    - 释放它锁占用的资源，且要求锁当前处于开放状态

互斥锁操作

- 加锁
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - 若mutex已被其他线程加锁，该调用会阻塞线程
- 尝试加锁
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
  - 若mutex已经被加锁，该调用会立即返回一个错误码
  - 利用此调用可以防止在优先级倒置所出现的死锁
- 解锁
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
  - 在下面的情况中，将返回一个错误
    - mutex已解锁
    - mutex被其他线程加锁

mutex变量示例1—修改前

```c
#include <stdio.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
void *thread_function(void *arg);
int run_now=1;  /*用run_now代表共享资源*/
void *thread_function(void *arg){
  int print_count2=0;
  while(print_count2++<5){
    if(run_now==2){ /*子线程：如果run_now为1就把它修改为1*/
      printf("function thread is run\n");
      run_now=1;
    }
    else {
      printf("function thread is sleep\n");
      sleep(1);
    }
  }
  pthread_exit(NULL);
}
```

mutex变量示例1—修改前

```
int main(){
  int print_count1=0; /*用于控制循环*/
  prhread_t a_thread;
  /*创建一个进程*/
  if(pthread_create(&a_thread,NULL, thread_function,NULL)!=0){
    perror("Thread createion failed");
    exit(1);
  }
  while(print_count1++<5){
    if(run_now==1){  /*主线程：如果run_now为1就把它修改为2*/
      printf("main thread is run\n");
      run_now=2;
    }
    else{
      printf("main thread is sleep\n");
      sleep(1);
    }
  }
  pthread_join(a_thread,NULL); /*等待子线程结束*/
  exit(0);
}
```

mutex变量示例1—修改前

- 运行结果
  - mai
    run

```
function thread is sleep
main thread is run
main thread is sleep
main thread is sleep
function thread is run
function thread is sleep
main thread is run
main thread is sleep
function thread is run
function thread is sleep
```

行，都可对

mutex变量示例1—修改后

```
#include <stdio.h>
#include <pthread.h>
#include <stdio.h>
void *thread_function(void *arg);
int run_now=1;   /*用run_now代表共享资源*/
pthread_mutex_t work_mutex;   /*定义互斥量*/

void *thread_function(void *arg){
   int print_count2=0;
   sleep(1);
   if(pthread_mutex_lock(&work_mutex)!=0){
      perror("Lock failed");
      exit(1);
   }
   else
      printf("function lock\n");
```

mutex变量示例1—修改后

```
    while(print_count2++<5){
      if(run_now==2){  /*分进程：如果run_now为1就把它修改为1*/
        printf("function thread is run\n");
        run_now=1;
      }
      else{
        printf("function thread is sleep\n");
        sleep(1);
      }
    }
    if(pthread_mutex_unlock(&work_mutex)!=0){   /*对互斥量解锁*/
      perror("unlock failed");
      exit(1);
    }
    else
      printf("function unlock\n");
    pthread_exit(NULL);
}
```

mutex变量示例1—修改后

```
int main(){
  int res;
  int print_count1=0;
  prhread_t a_thread;
  /*初始化互斥量*/
  if(pthread_mutex_init(&work_mutex,NULL)!=0){
    perror("Mutex init faied");
    exit(1);
  }
   /*创建新线程*/
  if(pthread_create(&a_thread,NULL,thread_function,NULL)!=0){
    perror("Thread createion failed");
    exit(1);
  }
  if(pthread_mutex_lock(&work_mutex)!=0){/*对互斥量加锁*/
    preeor("Lock failed");
    exit(1);
  }
  else
    printf("main lock\n");
```

mutex变量示例1—修改后

```
while(print_count1++<5){
   if(run_now==1){  /*主线程：如果run_now为1就把它修改为2*/
      printf("main thread is run\n");
      run_now=2;
   }
   else{
      printf("main thread is sleep\n");
      sleep(1);
   }
}
if(pthread_mutex_unlock(&work_mutex)!=0){ /*对互斥量解锁*/
   preeor("unlock failed");
   exit(1);
}
else
   printf("main unlock\n");
pthread_mutex_destroy(&work_mutex); /*收回互斥量资源*/
pthread_join(a_thread,NULL); /*等待子线程结束*/
exit(0);
}
```

mutex变量示例1—修改后

- 运行结果

```
main lock
main thread is run
main thread is sleep
main thread is sleep
main thread is sleep
main thread is sleep
main unlock
function lock
function thread is run
function thread is sleep
function thread is sleep
function thread is sleep
function thread is sleep
function unlock
```

mutex变量示例2—分析程序功能

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
typedef struct{
   double *a;
   double *b;
   double sum;
   int     veclen;
} DOTDATA;
#define NUMTHRDS 4
#define VECLEN 100
DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;
```

mutex变量示例2—分析程序功能

```
void *dotprod(void *arg){
  int i, start, end, offset, len ;
  double mysum, *x, *y;
  offset = (int)arg;
  len = dotstr.veclen;
  start = offset*len;
  end   = start + len;
  x = dotstr.a;
  y = dotstr.b;
  mysum = 0;
  for (i=start; i<end ; i++){
    mysum += (x[i] * y[i]);
  }
  pthread_mutex_lock (&mutexsum);
  dotstr.sum += mysum;
  pthread_mutex_unlock (&mutexsum);
  pthread_exit((void*) 0);
}
```

mutex变量示例2—分析程序功能

```
int main (int argc, char *argv[]){
  int i;
  double *a, *b;
  void *status;
  pthread_attr_t attr;
  a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
  b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
  for (i=0; i<VECLEN*NUMTHRDS; i++) {
    a[i]=1;
    b[i]=a[i];
  }
  dotstr.veclen = VECLEN;
  dotstr.a = a;
  dotstr.b = b;
  dotstr.sum=0;
  pthread_mutex_init(&mutexsum, NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

mutex变量示例2—分析程序功能

```
  for(i=0;i<NUMTHRDS;i++){
    pthread_create( &callThd[i], &attr, dotprod, (void *)i);
  }
  pthread_attr_destroy(&attr);
  for(i=0;i<NUMTHRDS;i++) {
    pthread_join( callThd[i], &status);
  }
  printf ("Sum =  %f  ", dotstr.sum);
  free (a);
  free (b);
  pthread_mutex_destroy(&mutexsum);
  pthread_exit(NULL);
}
```

条件变量

- 互斥锁的缺点
  - 通过控制存取数据来实现线程同步
  - 线程需不断轮询条件是否满足（忙等），消耗很多资源
- 条件变量
  - 利用线程间共享的全局变量实现同步
  - 条件变量使线程睡眠等待特定条件出现（无需轮询）
  - 使用方法
    - 通常条件变量和互斥锁同时使用
    - 一个线程因等待“条件变量的条件成立”而挂起
    - 另一个线程使"条件成立"（给出条件成立信号）

条件变量典型使用步骤

- 申明和初始化需要同步的全局数据/变量（如count）
- 申明和初始化一个条件变量对象
- 申明和初始化对应的mutex
- 创建若干进程并运行之

条件变量典型使用步骤

进程1	进程2
1、执行工作直到必须等待特定条件（如计数器必须满足特定值）； 2、对关联mutex上锁并检查全局变量的当前值; 3、调用ptread_cond_wait()进入阻塞等待，等待进程2发送的信号（说明： ptread_cond_wait()会自动对关联mutex解锁，以便可以被线程B使用）； 4、接收到信号后，将唤醒并自动对关联mutex上锁; 5、显式对关联mutex解锁; 6、继续执行。	1、执行自身工作; 2、对关联mutex上锁并修改进程1所等待的全局变量的值; 3、检查进程1所等待的全局变量的值，若满足期待条件，调用pthread_cond_signal()向进程1发送信号; 4、显式对关联mutex解锁; 5、继续执行。
主线程 (join/continue)	

条件变量检测

- 条件的检测是在互斥锁的保护下进行的
  - 如果条件为假，一个线程自动阻塞
  - 如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件

条件变量创建/销毁函数

- pthread_cond_init (condition,attr)
- pthread_cond_destroy(condition)
- pthread_condattr_init(attr)
- pthread_condattr_destroy(attr)

条件变量的初始化

- 声明条件变量：pthread_cond_t 类型
- 使用前必须初始化，有两种方法初始化方法
  - 静态方式
    - pthread_cond_t condition=PTHREAD_COND_INITIALIZER
  - 动态方式
    - int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
      - 被创建的条件变量ID通过 参数返回给调用线程
      - 该方法允许设置条件变量属性

条件变量的销毁

- 函数原型
  - int pthread_cond_destroy(pthread_cond_t *cond);
  - 销毁所指定的条件变量，同时将会释放所给它分配的资源
  - 调用该函数的线程并不要求等待在参数所指定的条件变量上

条件变量的等待

- 函数原型
  - int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  - int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t mytex, const struct timespec *abstime);
- 说明
  - 阻塞调用线程，直到满足特定的条件
    - 当该线程运行时，会被加锁，阻塞时会自动解锁
    - 当收到信号唤醒线程时，会被线程自动上锁当线程完成更新共享数据后，开发者有责任解锁
  - 这里的互斥锁必须是普通锁或者适应锁
  - 调用前必须由本线程加锁，激活前要保持锁是锁定状态

条件变量的激活

- 函数原型
  - int pthread_cond_signal(pthread_cond_t *cond);
  - int pthread_cond_broadcast(pthread_cond_t *cond);
- 说明
  - 用于通知（唤醒）等待在条件变量上的另一线程
  - 在被加锁后被调用，在完成pthread_cond_wait()运行后必须解锁
  - 二者区别
    - pthread_cond_signal()激活一个等待该条件的线程
    - pthread_cond_broadcast()激活所有等待的线程
    - 如果多于一个线程处于阻塞状态，应该用pthread_cond_broadcast()代替pthread_cond_signal()

条件变量等待与激活使用说明

- 如果在调用pthread_cond_wait()前先调用pthread_cond_signal()，将出现逻辑错误
- 当使用上述函数时，必须正确的加锁和解锁
  - 在调用pthread_cond_wait()之前没有成功加锁mutex会导致线程不会阻塞
  - 在调用 pthread_cond_signal()后没有成功解锁mutex，会导致pthread_cond_wait()一直运行 (保持线程阻塞)

条件变量示例

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int count = 0;
int thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

条件变量示例

```
void *inc_count(void *idp){
int j,i;
double result=0.0;
int *my_id = idp;
for (i=0; i<TCOUNT; i++) {
  pthread_mutex_lock(&count_mutex);
  count++;
  if (count == COUNT_LIMIT) {
    pthread_cond_signal(&count_threshold_cv);
    printf("inc_count(): thread %d, count = %d  Threshold reached.\n",
           *my_id, count);
  }
  printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
    *my_id, count);
  pthread_mutex_unlock(&count_mutex);
  for (j=0; j<1000; j++)
    result = result + (double)random();
  }
  pthread_exit(NULL);
}
```

条件变量示例

```
void *watch_count(void *idp){
  int *my_id = idp;
  printf("Starting watch_count(): thread %d\n", *my_id);
  pthread_mutex_lock(&count_mutex);
  if (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal
      received.\n", *my_id);
  }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

条件变量示例

```
int main (int argc, char *argv[]){
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, inc_count, (void *)&thread_ids[0]);
  pthread_create(&threads[1], &attr, inc_count, (void *)&thread_ids[1]);
  pthread_create(&threads[2], &attr, watch_count, (void *)&thread_ids[2]);
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
}
```

条件变量示例

```
inc_count(): thread 0, count = 1, unlocking
mutex
Starting watch_count(): thread 2
inc_count(): thread 1, count = 2, unlocking
mutex
inc_count(): thread 0, count = 3, unlocking
mutex
inc_count(): thread 1, count = 4, unlocking
mutex
inc_count(): thread 0, count = 5, unlocking
mutex
inc_count(): thread 0, count = 6, unlocking
mutex
inc_count(): thread 1, count = 7, unlocking
mutex
inc_count(): thread 0, count = 8, unlocking
mutex
```

生产者/消费者问题

- 采用多线程技术解决生产者/消费者问题
  - 也称有界缓冲区问题
    - 多个生产者线程向缓冲区中写数据
    - 多个消费者线程从缓冲区中读取数据
  - 生产者线程和消费者线程必须满足
    - 生产者写入缓冲区的数目不能超过缓冲区容量
    - 消费者读取的数目不能超过生产者写入的数目

生产者/消费者问题—PV原语操作

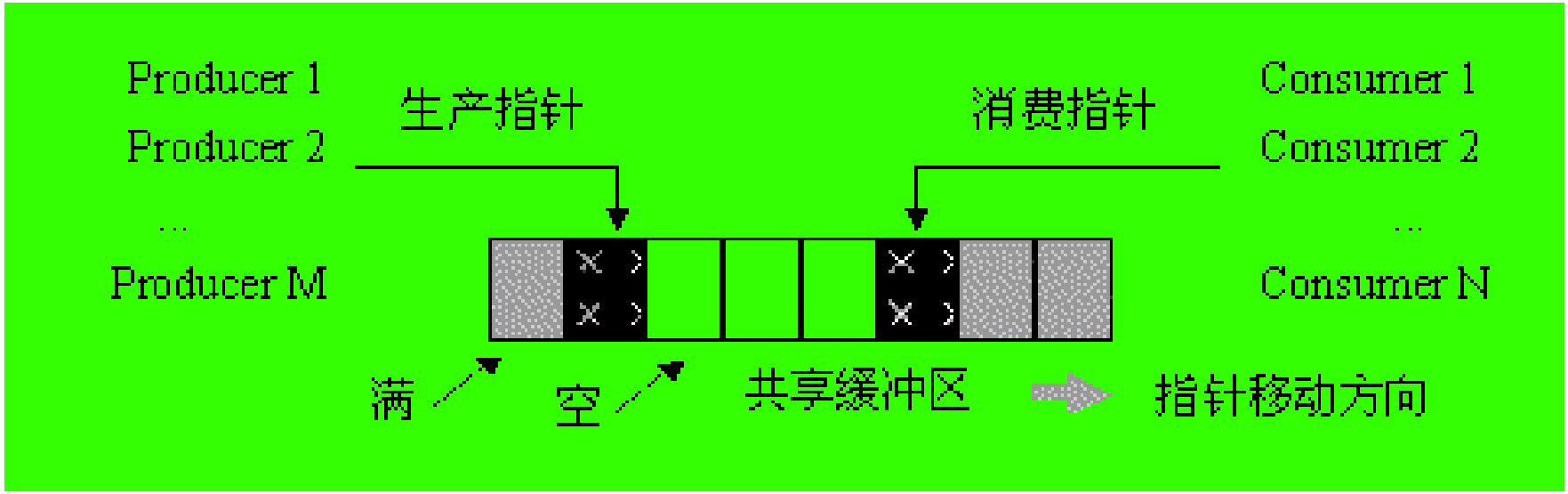
```
var B: array[0..k-1] of item;
empty:semaphore:=k;  /* 可以使用的空缓冲区数 */
full:semaphore:=0; /* 缓冲区内可以使用的产品数 */
mutex:semaphore:=1;
in , out:integer:= 0; /* 放入/取出缓冲区指针*/
cobegin
process producer_i                    process consumer_j
Begin                                 begin
  L1:produce a product;                 L2:P(full);
  P(empty);                             P(mutex);
  P(mutex);                             Product:= B[out];
  B[in] := product;                     out:=(out+1) mod k;
  in:=(in+1) mod k;                     V(mutex);
  V(mutex);                             V(empty);
  V(full);                              Consume a product;
  Goto L1;                              goto L2;
  end;                                end;
coend
```

生产者/消费者问题—问题分析

- 缓冲区须被生产者/消费者进程互斥访问
  - 生产者进程
    - 多个并发写进程互斥改变写指针
    - 写入条件：缓冲区非满
  - 消费者进程
    - 多个并发读进程互斥改变读指针
    - 读取条件：缓冲区非空
  - 读/写指针设计
    - 初始化时，读指针和写指针均为0
    - 如果读指针等于写指针，则缓冲区为空
    - 如果(写指针+ 1) % BUFFER_SIZE等于读指针，则缓冲区为满

生产者/消费者问题—问题分析(续)

- 缓冲区访问结构示意图



  - 如何利用线程互斥变量机制定义缓冲区结构
  - 如何利用线程同步通信机制协调生产者/消费者进程通信
- 解决途径
  - 线程条件变量通信机制

生产者/消费者问题—解题思路

- 缓冲区结构定义
  - 一个mutex变量：pthread_mutex_t
    - lock
  - 两个条件变量： pthread_cond_t
    - 分别控制缓存空/满状态指示

```
struct prodcons {
   int buffer[BUFSIZE];
   pthread_mutex_t lock;   //互斥LOCK
   int readpos , writepos;
   pthread_cond_t notempty; //缓冲区非空条件判断
   pthread_cond_t notfull;  //缓冲区未满条件判断
};
```

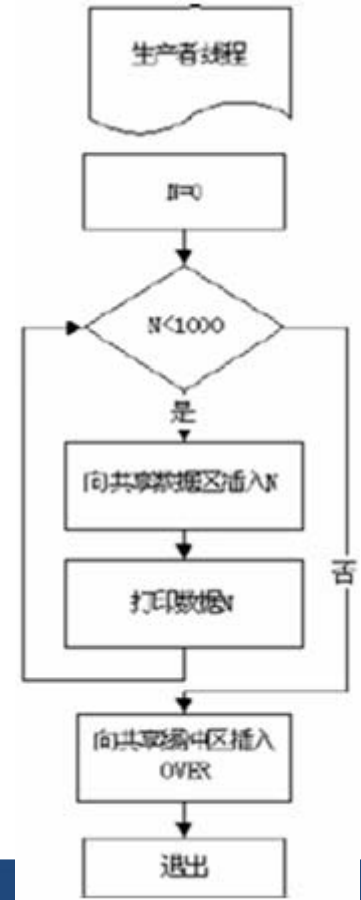
生产者/消费者问题—解题思路(续)

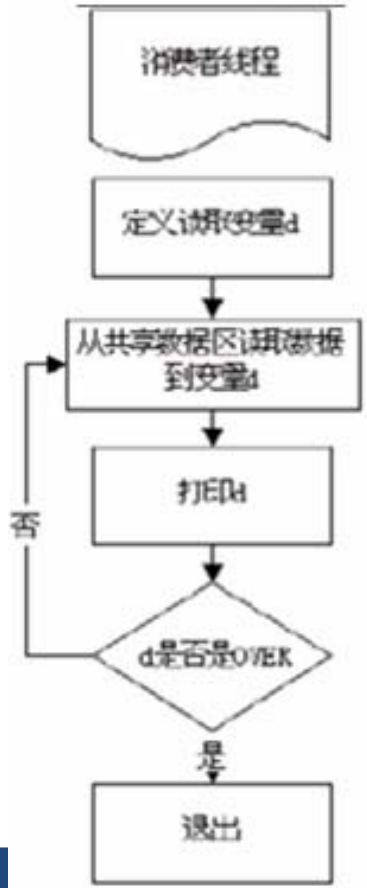
- 生产者/消费者进程之间的同步通信协调

生产者进程	消费者进程
1、调用pthread_mutex_lock()对lock上锁，并根据以下条件判断缓冲区是否已满； (writepos + 1) % BUFSIZE == readpos 2、若满，调用ptread_cond_wait()进入阻塞，等待notfull条件变量； 3、写入数据并移动写指针writepos; 4、调用pthread_cond_signal()向消费者信号通过notempty条件变量; 5、调用pthread_mutex_unlock()对mutex解锁。	1、调用pthread_mutex_lock()对lock上锁，并根据以下条件判断缓冲区是否为空； writepos == readpos 2、若空，调用ptread_cond_wait()进入阻塞，等待notempty条件变量； 3、读取数据并移动读指针readpos; 4、调用pthread_cond_signal()向消费者信号通过notfull条件变量; 5、调用pthread_mutex_unlock()对mutex解锁。

生产者/消费者问题—解决方案

- 主程序启动生产者/消费者线程
  - 生产者线程顺序地将0 到1000写入循环缓冲区
  - 消费者线程不断地从共享的循环缓冲区读取数据





生产者/消费者问题—关键函数

- 线程管理相关函数
  - int pthread_create( );
  - int pthread_join();
- 线程互斥控制相关函数
  - int pthread_mutex_init();
  - int pthread_mutex_lock();
  - int pthread_mutex_unlock();
- 线程条件变量控制相关函数
  - int pthread_cond_init();
  - int pthread_cond_wait();
  - int pthread_cond_signal();

生产者/消费者问题—程序代码

```
#include<stdio.h>
#include<pthread.h>
#define BUFSIZE 1000

struct prodcons {
   int buffer[BUFSIZE];
   pthread_mutex_t lock;   //互斥LOCK
   int readpos , writepos;
   pthread_cond_t notempty; //缓冲区非空条件判断
   pthread_cond_t notfull;  //缓冲区未满条件判断
};

void init(struct prodcons * b){
   pthread_mutex_init(&b->lock,NULL);
   pthread_cond_init(&b->notempty,NULL);
   pthread_cond_init(&b->notfull,NULL);
   b->readpos=0;
   b->writepos=0;
}
```

生产者/消费者问题—程序代码(续)

```
void put(struct prodcons* b,int data){
  //如果互斥锁mutex已经被上锁则挂起生产者线程,并返回
  pthread_mutex_lock(&b->lock);
  //等待缓冲区未满
  if((b->writepos + 1) % BUFSIZE == b->readpos){
    //缓冲区满,生产者将被挂起,直至重新被唤醒
    pthread_cond_wait(&b->notfull, &b->lock);
  }
  //写数据,并移动指针
  b->buffer[b->writepos]=data;
  b->writepos++;
  if(b->writepos >= BUFSIZE)
    b->writepos=0;
  //设置缓冲区非空的条件变量
  pthread_cond_signal(&b->notempty);
  pthread_mutex_unlock(&b->lock);
}
```

生产者/消费者问题—程序代码(续)

```
int get(struct prodcons *b){
  int data;
  pthread_mutex_lock(&b->lock);
  if(b->writepos == b->readpos){
    //等待缓冲区非空
    pthread_cond_wait(&b->notempty, &b->lock);
  }
  //读数据,移动读指针
  data = b->buffer[b->readpos];
  b->readpos++;
  if(b->readpos >= BUFSIZE)
    b->readpos=0;
  //设置缓冲区未满的条件变量
  pthread_cond_signal(&b->notfull);
  pthread_mutex_unlock(&b->lock);
  return data;
}
```

生产者/消费者问题—程序代码(续)

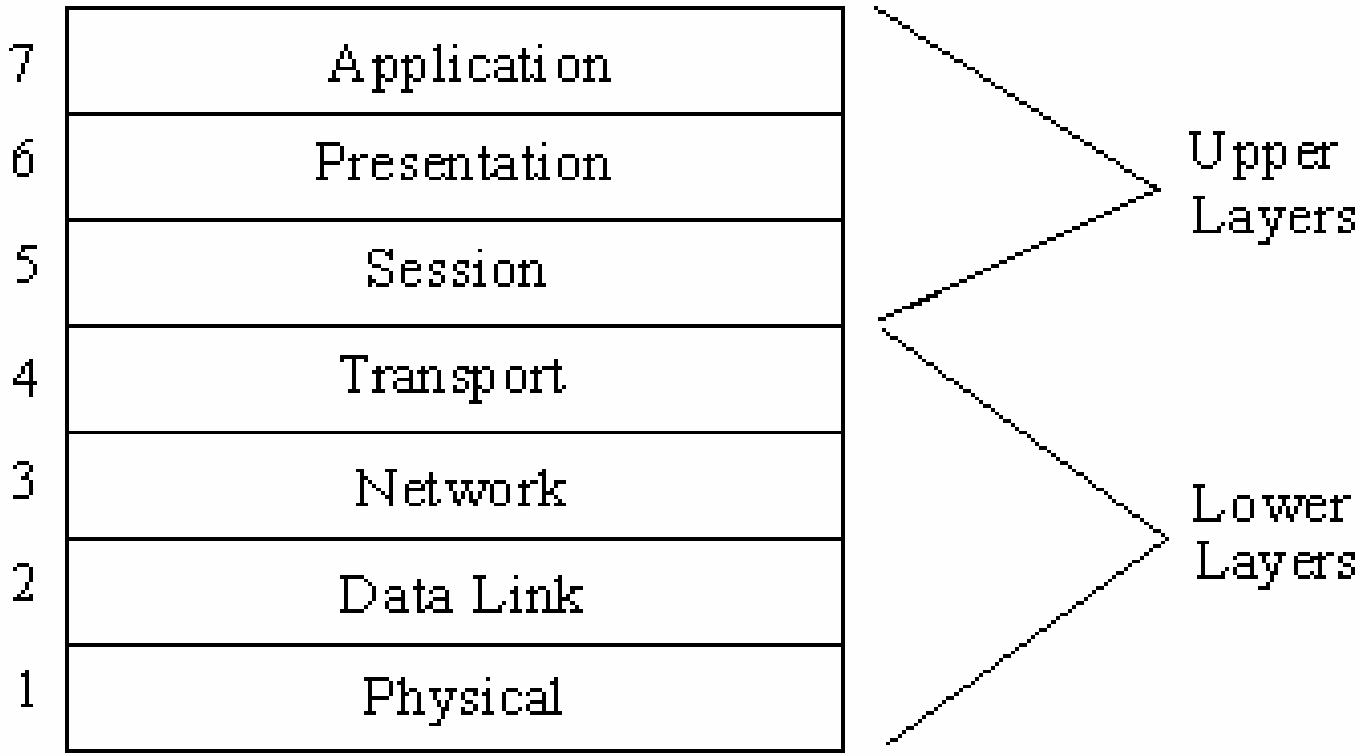
```
#define OVER (-1)
struct prodcons buffer;
void *producer(void *data){
   int n;
   for(n = 0; n < 10000; n++){
      printf("%d \n", n) ;
      put(&buffer, n);
   }
   put(&buffer, OVER);
   return NULL;
}
void *consumer(void * data){
   int d;
   while(1){
      d = get(&buffer);
      if(d == OVER)
         break;
      printf("%d\n", d);
   }
   return NULL;
}
```

生产者/消费者问题—程序代码(续)

```
int main(void){
  pthread_t th_a, th_b;
  void *retval;
  init(&buffer);
  pthread_create(&th_a, NULL, producer, 0);
  pthread_create(&th_b, NULL, consumer, 0);
  pthread_join(th_a, &retval);
  pthread_join(th_b, &retval);
  return 0;
}
```

网络编程

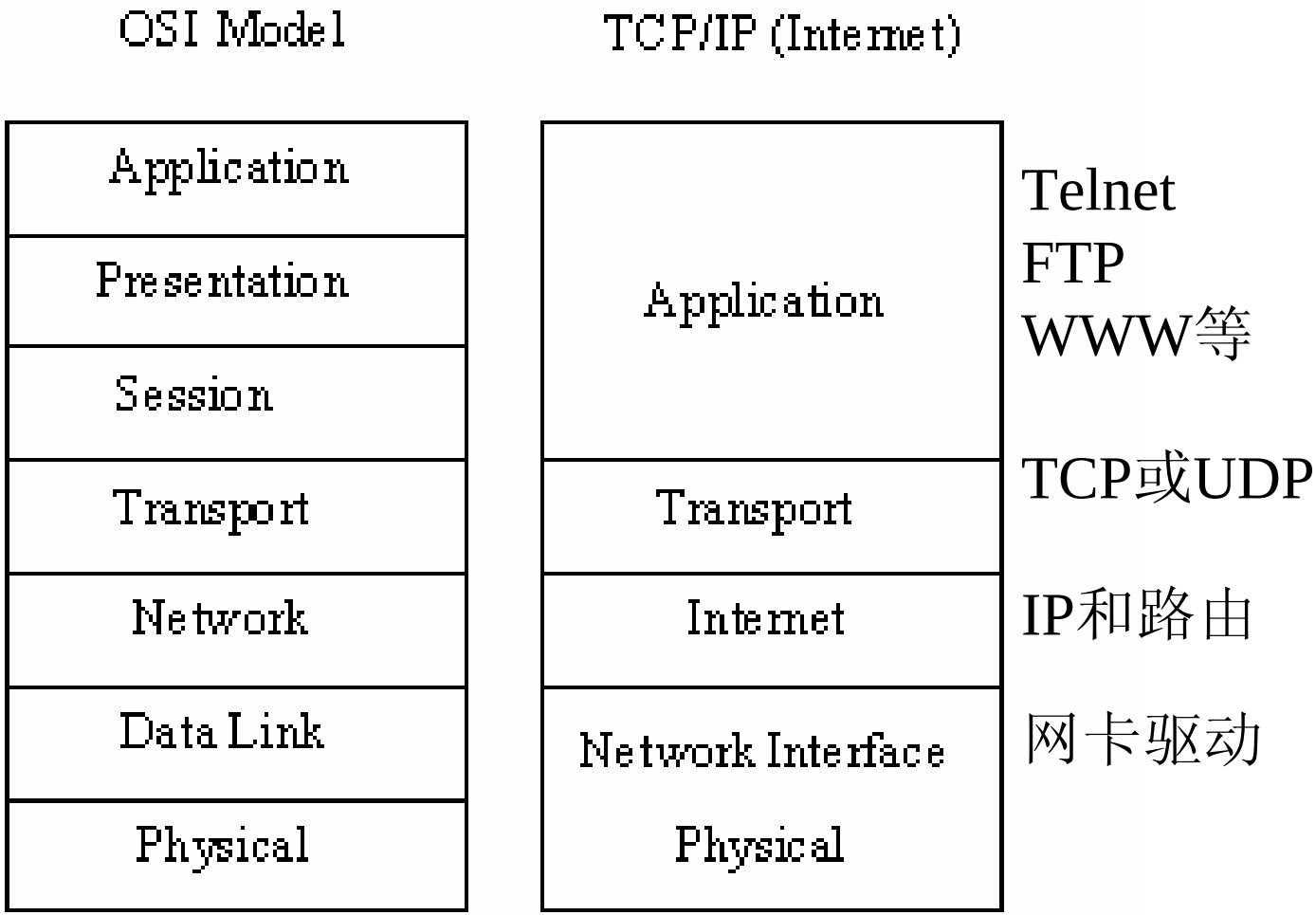
ISO/OSI七层协议模型



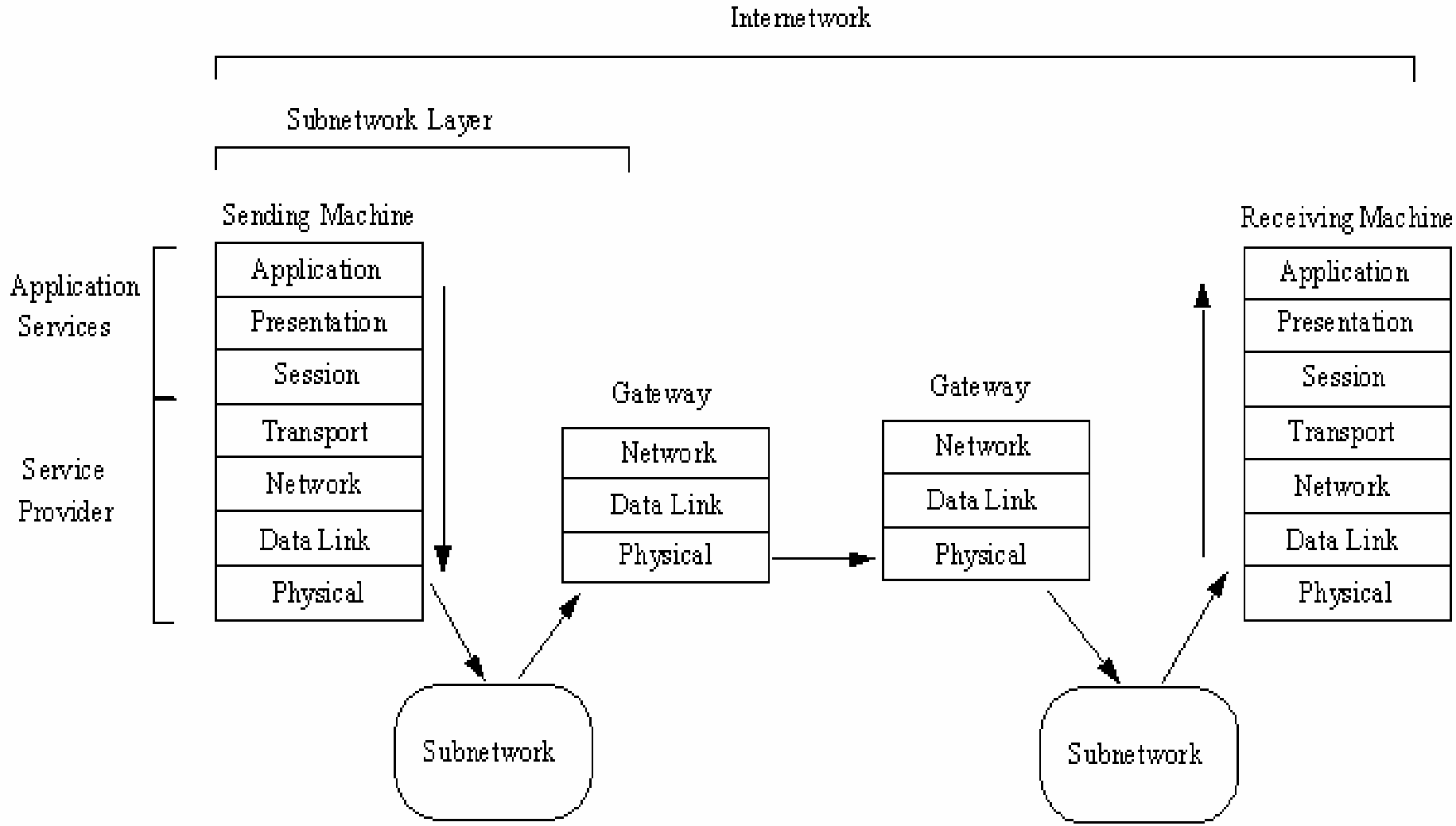
TCP/IP协议族

- TCP（Transmission Control Protocol）
  - 传输控制协议，基于连接的服务
- UDP（User Datagram Protocol）
  - 用户数据报协议，无连接的服务
- IP（Internet Protocol）
  - Internet协议，信息传递机制

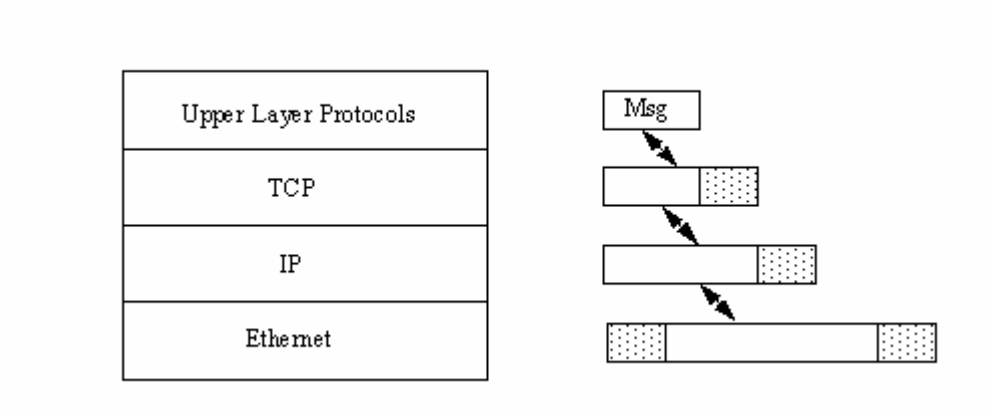
OSI模型与TCP/IP协议的对比



消息传递流


Internetwork
Subnetwork Layer
Sending Machine
Receiving Machine
Application Services
Service Provider
Application
Presentation
Session
Transport
Network
Data Link
Physical
Gateway
Network
Data Link
Physical
Gateway
Network
Data Link
Physical
Application
Presentation
Session
Transport
Network
Data Link
Physical
Subnetwork
Subnetwork

消息包的逐层递增



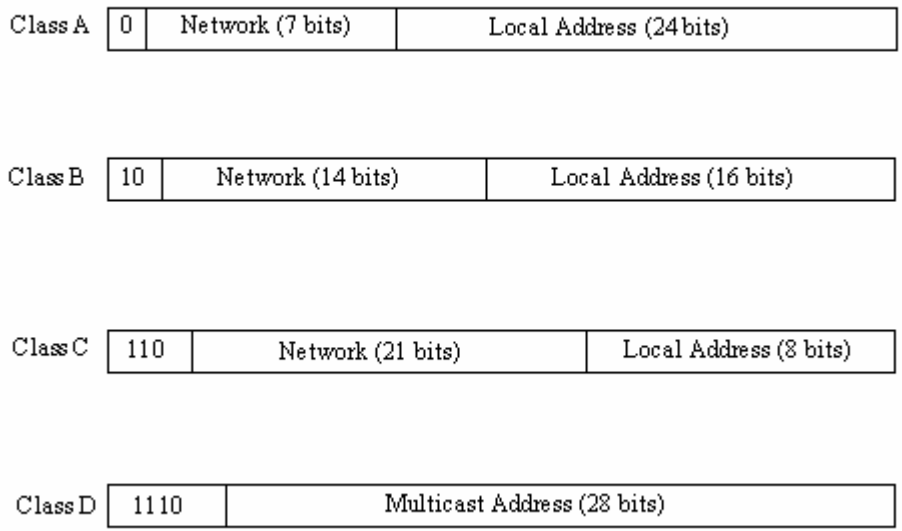
一些Socket编程的概念

- 流（Stream）
- 连接（Connection）
- 阻塞(Block)、非阻塞(Non-block)
- 同步(Synchronous)、异步(asynchronous)
- IP地址
- 字节顺序

IP地址

- IP地址是Internet中唯一的地址标识，
  - IP地址是一个32位长（正在扩充到128位）
  - 每个Internet包必须带有IP地址
- 点分十进制表示法
  - 将IP地址的4个字节的数字用十进制表示并用点隔开,如：202.112.58.200(0xCA703AC8)
- IP地址的分级
- 子网掩码（Subnet Mask）

四级IP地址



子网掩码

- 也用点分十进制表示
  - 例如：255.255.0.0
- 指明子网（局域网）的范围
  - Mask与IP地址进行与操作即可得出子网范围
  - 例如

    IP地址：166.111.160.1与166.111.161.45

    子网掩码：255.255.254.0

    即可得出这两个IP地址处于同一个子网内

MAC地址

- MAC地址是Ethernet协议使用的唯一地址
  - MAC地址是Ethernet NIC上自带的，48位长。
    - 如：00-88-CC-06-05-43
  - MAC地址作用范围是Ethernet（局域网）内
  - MAC地址存在于每一个Ethernet包中，是Ethernet包头的组成部分，Ethernet交换机根据Ethernet包头中的MAC源地址和MAC目的地址实现包的交换和传递
- MAC地址与IP地址无关

字节顺序

- 网络字节顺序（NBO，Network Byte Order）
  - 使用统一的字节顺序，避免兼容性问题
- 主机字节顺序（HBO，Host Byte Order）
  - 不同的机器HBO不相同，与CPU设计有关
  - Motorola 68k系列，HBO与NBO相同
  - Intel x86系列，HBO与NBO相反

SOCKET函数介绍

需要用到的头文件

- 数据类型：#include <sys/types.h>
- 函数定义：#include <sys/socket.h>

Berkeley Socket 常用函数列表

- 网络连接函数
- 获取/设置socket的参数或信息
- 转换函数

网络连接函数

- socket
- bind
- connect
- listen
- accept
- select
- recv, recvfrom
- send, sendto
- close, shutdown

获取/设置socket的参数或信息

- gethostbyaddr, gethostbyname
- gethostname
- getpeername
- getprotobyname, getprotobynumber
- getservbyname, getservbyport
- getsockname
- getsockopt, setsockopt
- ioctl

转换函数

- IP地址转换
  - inet_addr()
  - inet_ntoa()
- 字节顺序转换
  - htons()--"Host to Network Short"
  - htonl()--"Host to Network Long"
  - ntohs()--"Network to Host Short"
  - ntohl()--"Network to Host Long"

数据结构：sockaddr

```
struct sockaddr {
    unsigned short sa_family;  /* address family,
                                  AF_xxx */
    char sa_data[ 14];         /* 14 bytes of
                                  protocol address */
};
```

- 此数据结构用做bind、connect、recvfrom、sendto等函数的参数，指明地址信息

数据结构：sockaddr_in

```
struct sockaddr_in {
   short int sin_family;        /* Address family */
   unsigned short int sin_port; /* Port number */
   struct in_addr sin_addr;  /* Internet address */
   unsigned char sin_zero[ 8];      /* Same size as
                                       structsockaddr */
};
```

- 该结构与sockaddr兼容，供用户填入参数

数据结构：in_addr

```
struct in_addr {
    unsigned long s_addr;
};
```

- 这个数据结构是由于历史原因保留下来的，主要用作与以前的格式兼容。

程序中实际只填写sockaddr_in结构

```
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(3490); /* short,
                                   NBO*/
my_addr.sin_addr.s_addr =
   inet_addr("132.241.5.10");
bzero(&(my_addr.sin_zero), 8);
```

- 注意：sin_addr.s_addr填本机IP，如果此项填INADDR_ANY时，表示自动取本机IP填入该项(仅用于Server)

函数简介：socket

- Socket描述符与Linux中的文件描述符类似，也是一个int型的变量
- 函数调用：
  - int socket(int domain, int type, int protocol);
  - 函数返回Socket描述符，返回-1表示出错
  - domain参数只能取AF_INET, protocol参数一般取0
- 应用示例：
  - TCP：sockfd = socket(AF_INET,SOCK_STREAM,0);
  - UDP：sockfd =socket(AF_INET, SOCK_DGRAM,0);

函数简介：bind

- 作为Server程序，需要与一个端口绑定
  - int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
- bind函数返回-1表示出错，最常见的错误是该端口已经被其他程序绑定。
- 需要注意的一点：在Linux系统中，1024以下的端口只有拥有root权限的程序才能绑定

函数简介：connect

- 连接某个Server
  - int connect(int sockfd, struct sockaddr *servaddr, int addrlen);

- servaddr是事先填写好的结构，Server的IP和端口都在该数据结构中指定。

函数简介：listen

- 开始监听已经绑定的端口
  - 需要在此前调用bind()函数，否则由系统指定一个随机的端口
  - int listen(int sockfd, int queue_length);
- 接收队列
  - 一个新的Client的连接请求先被放在接收队列中，等待Server程序调用accept函数接受连接请求
  - queue_length指的就是接收队列的长度
  - 也就是在Server程序调用accept函数之前最大允许的连接请求数，多余的连接请求将被拒绝

函数简介：accept

- accept()函数将响应连接请求，建立连接
  - 产生一个新的socket描述符来描述该连接
  - 这个连接用来与特定的Client交换信息
- int accept(int sockfd,struct sockaddr *addr,int *addrlen);
  - addr将在函数调用后被填入连接对方的地址信息，如对方的IP、端口等。
- accept缺省是阻塞函数，阻塞直到有连接请求

accept()函数应用示例

```
struct sockaddr_in their_addr;  /* 用于存储连接对方的地址信息*/
int sin_size = sizeof(struct sockaddr_in);

... ... ( 依次调用socket(), bind(), listen()等函数 )
new_fd = accept(sockfd, &their_addr, &sin_size);
printf(" 对方地址: %s\n", inet_ntoa(their_addr.sin_addr));
... ...
```

函数简介：select

- 应用于多路同步I/O模式
  - int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
    - numfds是要多路选择的socket的最大值
    - 其中readfds, writefds, exceptfds都是socket集合，分别代表有数据可读、有数据要写、发生异常的socket集合。
    - timeout是select的时间限制
    - 返回值：在socket集合中准备好的socket个数

socket集合

- 集合变量类型：fd_set
- 集合变量运算宏：
  - FD_ZERO(*set)　　清空socket集合
  - FD_SET(s, *set)　　将s加入socket集合
  - FD_CLR(s, *set)　　从socket集合去掉s
  - FD_ISSET(s, *set)　判断s是否在socket集合中
- 常数FD_SETSIZE：集合元素的最多个数

Select应用举例

```
fd_set rset;                    /* 可读的socket集合 */

FD_ZERO(& rset);                /* 清空rset集合 */
FD_SET(sockfd, rset);           /* 将sockfd 加入rset集合 */

nready = select(maxfd+1, &rset, null, null, null);
    /* maxfd是已知的最大的socket */

if (FD_ISSET(sockfd, &rset)) {
    ……
}
```

函数简介：recv

- 用于TCP协议中接收信息
  - int recv(int sockfd, void *buf, int len, int flags);
    - buf，指向容纳接收信息的缓冲区的指针
    - len，缓冲区的大小
    - flags，接收标志
  - 函数返回实际接收的字节数，返回-1表示出错
  - recv缺省是阻塞函数，直到接收到信息或出错

函数简介：recvfrom

- 用于UDP协议中接收信息
  - int recvfrom(int sockfd,void *buf,int len,unsigned int flags, struct sockaddr *from, int *fromlen);
    - buf，指向容纳接收信息的缓冲区的指针
    - len，缓冲区的大小
    - flags，接收标志
    - from，指明接收数据的来源
  - 函数返回实际接收的字节数，返回-1表示出错
  - recvfrom是阻塞函数，直到接收到信息或出错

函数简介：send

- 用于TCP协议中发送信息
  - int send(int sockfd, const void *msg, int len, int flags);
    - msg，指向待发送信息的指针
    - len，待发送的字节数
    - flags，发送标志
  - 函数返回已发送的字节数，返回-1表示出错
  - send缺省是阻塞函数，直到发送完毕或出错
  - 注意：如果函数返回值与参数len不相等，则剩余的未发送信息需要再次发送

函数简介：sendto

- 用于UDP协议中发送信息
  - int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
    - buf，指向容纳接收信息的缓冲区的指针
    - len，缓冲区的大小
    - flags，接收标志
    - to，指明发送数据的目的地
  - 函数返回已发送的字节数，返回-1表示出错
  - sendto缺省是阻塞函数，直到发送完毕或出错
  - 注意：如果函数返回值与参数len不相等，则剩余的未发送信息需要再次发送

参数flags

- MSG_DONTROUTE
  - 对send, sendto有效
  - 表示不使用路由（一般不使用）
- MSG_PEEK
  - 对recv, recvfrom有效
  - 表示读出网络数据后不清除已读的数据
- MSG_OOB
  - 对发送接收都有效
  - 表示读/写带外数据(out-of-band data)

函数简介：close

- 关闭特定的socket连接
- 调用函数：int close(int sockfd);

- 关闭连接将中断对该socket的读写操作。
- 关闭用于listen()函数的socket将禁止其他Client的连接请求

函数简介：shutdown

- Shutdown()函数可以单方面的中断连接，即禁止某个方向的信息传递。
- 函数调用
  - int shutdown(int sockfd, int how);
  - 参数how：
    - 0 - 禁止接收信息
    - 1 - 禁止发送信息
    - 2 - 接收和发送都被禁止，与close()函数效果相同
  - 返回0表示调用成功，返回-1表示出错

函数简介：ioctl

- 获得或改变socket的I/O属性
  - int ioctl(int sockfd,long cmd,unsigned long* argp)
    - cmd：属性
    - argp：属性的参数（缓存区指针）
  - 常用的属性：
    - FIONREAD，返回socket缓冲区中未读数据的字节数
    - FIONBIO，argp为零时为阻塞模式，非零时为非阻塞模式
    - SIOCATMARK ，判断是否有未读的带外数据（仅用于TCP协议），返回true或false

函数简介：inet_addr, inet_ntoa

- unsigned long inet_addr (const char *cp);
  - inet_addr将一个点分十进制IP地址字符串转换成32位数字表示的IP地址
- char* inet_ntoa (struct in_addr in);
  - inet_ntoa将一个32位数字表示的IP地址转换成点分十进制IP地址字符串
- 这两个函数互为反函数

TCP：基于连接流的网络协议

- TCP相关的函数
- Server的例子
- Client的例子
- Server-Client结构图
- TCP编程的适用范围

Server程序的作用

- 程序初始化
- 持续监听一个固定的端口
- 收到Client的连接后建立一个socket连接
- 与Client进行通信和信息处理
  - 接收Client通过socket连接发送来的数据，进行相应处理并返回处理结果，如BBS Server
  - 通过socket连接向Client发送信息，如Time Server
- 通信结束后中断与Client的连接

完整的Server程序示例

一个简单的TCP SERVER

程序流程—

- 取得socket描述符：

```
int sockfd;
sockfd = socket(AF_INET, SOCK_STREAM,
   0)) ;
```

程序流程二

- 填写自身地址信息的sockaddr_in结构

```
struct sockaddr_in my_addr;    /* 自身的地址信息 */
my_addr.sin_family = AF_INET;
/* 网络字节顺序 */
my_addr.sin_port = htons(MYPORT);
/* 自动填本机IP */
my_addr.sin_addr.s_addr = INADDR_ANY;
/* 其余部分置0 */
bzero(&(my_addr.sin_zero), 8);
```

程序流程三

- 绑定端口

```
bind(sockfd, (struct sockaddr *)&my_addr,
  sizeof(struct sockaddr));
```

程序流程四

- 监听端口

```
#define BACKLOG 10
listen(sockfd, BACKLOG);
```

程序流程五

- 接受连接请求

```
int new_fd;                 /* 数据端口 */
struct sockaddr_in their_addr; /* 连接对方的
   地址信息 */
int sin_size;
sin_size = sizeof(struct sockaddr_in);
new_fd = accept(sockfd, (struct sockaddr
   *)&their_addr, &sin_size))
```

程序流程六

- 产生新进程（线程）处理读写socket

```
if (!fork()) {                    /* 子进程 */
    if (send(new_fd, "Hello, world!\ n", 14, 0) == -1)
      perror("send");
    close(new_fd);
    exit(0);
}
close(new_fd);
```

程序流程七

- 转程序流程五，继续等待其他Client的连接并处理

CLIENT部分

Client程序的作用

- 程序初始化
- 连接到某个Server上，建立socket连接
- 与Server进行通信和信息处理
  - 接收Server通过socket连接发送来的数据，进行相应处理
  - 通过socket连接向Server发送请求信息
- 通信结束后中断与Client的连接

程序流程一

- 取得socket描述符：

```
int sockfd;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

程序流程二

- 填写连接对方的地址信息的sockaddr_in结构

```
struct hostent *he;
struct sockaddr_in their_addr;  /* 对方的地址信息 */
he=gethostbyname( “whitehouse.gov” );
their_addr.sin_family = AF_INET;
their_addr.sin_port = htons(4000);          /* short,
   NBO */
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
bzero(&(their_addr.sin_zero), 8); /* 其余部分设置成0
   */
```

程序流程三

- 连接端口

```
connect(sockfd, (struct sockaddr
  *)&their_addr,     sizeof(struct sockaddr));
```

程序流程四

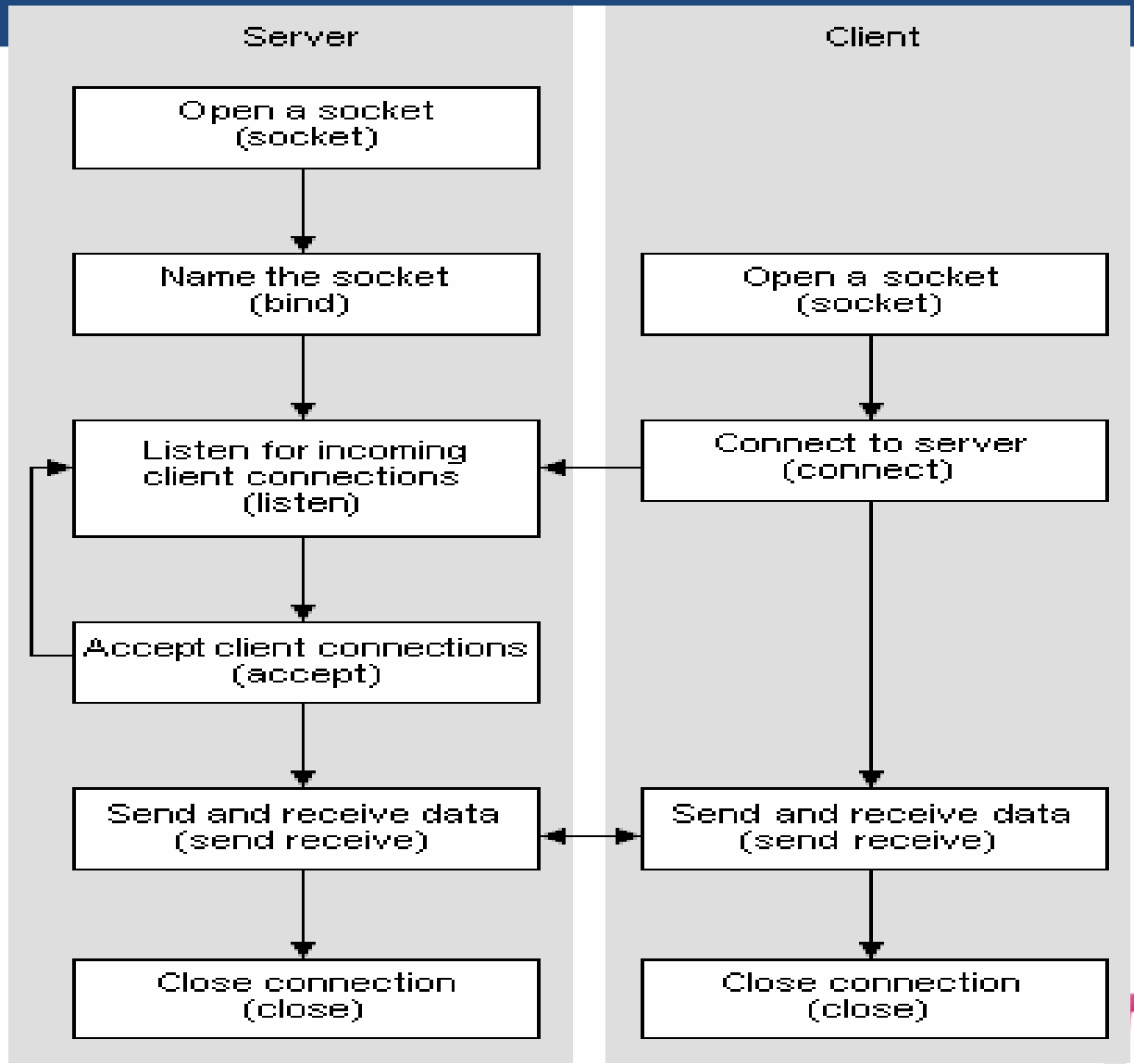
- 读写socket

程序流程五

- 关闭socket

  close(sockfd);

TCP, C/S结构图:



UDP：基于数据包的无连接协议

- UDP与TCP的区别
- Server的例子
- Client的例子
- Server-Client关系图
- UDP编程的适用范围

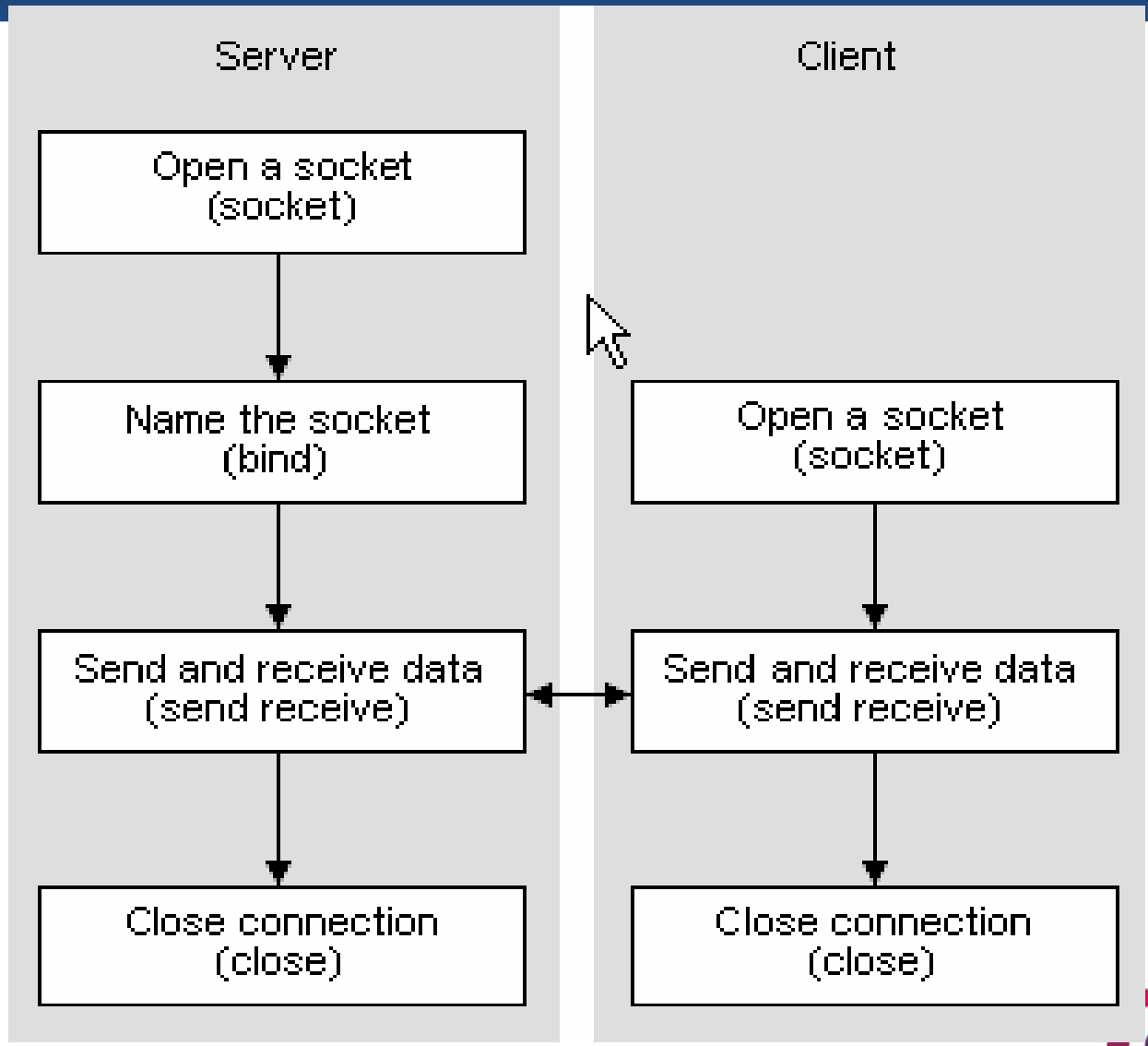
UDP与TCP的区别

- 基于连接与无连接
- 流模式与数据报模式
  - TCP保证数据正确性，UDP可能丢包
  - TCP保证数据顺序，UDP不保证
- 对系统资源的要求（TCP较多，UDP少）

具体编程时的区别

- socket()的参数不同
- UDP Server不需要调用listen和accept
- UDP收发数据用sendto/recvfrom函数
- TCP：地址信息在connect/accept时确定
  UDP：在sendto/recvfrom函数中每次均需指定地址信息
- UDP：shutdown函数无效

UDP Server-Client关系图



UDP应用举例

Server部分

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define MYPORT    3490              /* 监听端口 */

void main()
{
      int sockfd;                   /* 数据端口 */
      struct sockaddr_in my_addr;   /* 自身的地址信息 */
      struct sockaddr_in their_addr; /* 连接对方的地址信息 */
      int sin_size, retval;
      char buf[128];
```

```
if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
}

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);              /* 网络字节顺序 */
my_addr.sin_addr.s_addr = INADDR_ANY;      /* 自动填本机IP */
bzero(&(my_addr.sin_zero), 8);                 /* 其余部分置0 */

if  (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(my_addr)) == -1)
{
        perror("bind");
        exit(1);
}
```

```
    while(1) {                                  /* 主循环 */
        retval = recvfrom(sockfd, buf, 128, 0, (struct sockaddr *)&their_addr,
                                    &sin_size);
        printf("Received datagram from %s\n",inet_ntoa(their_addr.sin_addr));

        if (retval == 0) {
            perror (“recvfrom");
            close(sockfd);
            break;
        }

        retval = sendto(sockfd, buf, 128, 0,  (struct sockaddr *)&their_addr,
                                    sin_size);
    }
}
```

UDP编程的适用范围

- 部分满足以下几点要求时，应该用UDP
  - 面向数据报
  - 网络数据大多为短消息
  - 拥有大量Client
  - 对数据安全性无特殊要求
  - 网络负担非常重，但对响应速度要求高
- 例子：ICQ、ping

网络编程

- 协议设计
- Server程序的设计
  - 程序结构的考虑
  - 传输模式的考虑
  - 网络函数的考虑

设计网络协议

- 设计网络协议目的
- 设计网络协议时需要考虑的因素
  - 完备性
  - 正确性
  - 最简性（简便性）

示例：Chat协议

- 协议分成两组：
  - 客户端请求协议
  - 服务器通知协议
- 发送协议后对方会返回一个数字表示错误值，可以根据错误值判断是否完成操作和/或错误类型

客户端请求协议

- 普通用户命令：
  - 用户申请登录 0x80　　用户退出 0x81
  - 用户加入房间 0x82　　用户离开房间 0x83
  - 给用户发消息 0x84　　在房间中讲话 0x85
  - 取所有房间的信息 0x86　　取某一房间的信息 0x87
  - 取所有用户的信息 0x88　　取某一用户的信息 0x89

- 管理员命令：
  - 对所有用户广播 0x90
  - 踢出用户 0x91
  - 增加房间管理员 0x92

服务器通知协议

– 用户加入组	0xC0
– 用户离开组	0xC1
– 传递消息	0xC2
– 用户成为房间管理员	0xC3
– 用户被踢出房间	0xC4

对协议代码的返回值

- 0x7F：无错误（正确返回）。
- 0xF0：无效的房间名称。
- 0xF1：无效的用户名称。
- 0xF2：用户登录失败。
- 0xF3：当前用户没有此项操作的权力。
- 0xF4：达到最大房间数，无法新建房间。
- 0xF5：达到最大用户数，无法登录。
- 0xFE：无效协议代码
- 0xFF：其他错误

Server的设计

- 设计Server时需要考虑的因素
  - 响应速度（新建连接时、发送数据时）
  - 运行速度
  - I/O吞吐量
  - 其它：流量控制（QoS）、安全性
  - 针对特定协议的数据结构

设计Server的程序结构

- 程序结构的考虑
  - 多线程
  - 多进程
  - 单进程
- 网络函数的考虑
  - TCP流模式 或 UDP数据报模式
  - 阻塞函数 或 非阻塞函数

程序结构：多进程

- 在主进程调用accept()函数生成一个新的连接后，调用fork()产生一个子进程对这个新连接进行操作
- 在主进程结束前需要向所有子进程发中断信号并等待所有子进程执行完毕。
- 这种程序结构最简单，例子可以参照前面TCP Server的结构和代码。
- 主要应用于各连接操作相互独立的Server，可以保证各连接相互间的数据安全性, 如telnetd

程序结构：多线程

- 基本与多进程结构类似，但是在获得新连接时生成一个线程来对这个连接进行处理。
- 主要的优点：
  - 线程调度速度快，占用资源少
  - 线程可共享进程空间中的数据
- 主要应用于各个连接之间关系较紧密的Server，例如：BBS Server
- Server的响应速度和I/O吞吐量均较好，是最常用的程序结构

程序结构：单线程

- 通过select实现非阻塞的同步I/O模式
- 可以通过调用select函数得出需要读数据并处理的socket集合（也就是Client的集合），然后依次对每个socket读数据，处理并向socket写结果
- select得到的socket列表中有一个特殊的socket就是listen函数使用的socket，这个socket需要单独处理，调用accept生成新的socket连接并将这个新socket加入已有的socket集合。
- 该结构对算法效率要求较高，一般来说响应速度慢，但I/O处理速度最快。适用于连接数少、大数据吞吐量的Server

```
int listenfd, connfd, maxfd=0;
int nready;
fd_set rset, allset;
struct sockaddr_in cliaddr, servaddr;
int clilen;

listenfd = socket(AF_INET, SOCK_STREAM, 0);
if (listenfd > maxfd) maxfd = listenfd;

memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(4321);

bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

FD_ZERO(&allset);
FD_SET(listenfd, &allset);
listen(listenfd, 10);
```

```
while (1) {
        rset = allset;
        nready = select(maxfd+1, &rset, NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &rset)) {
                clilen = sizeof(cliaddr);
                connfd = accept(listenfd, (struct sockaddr*)&cliaddr,&clilen);

                if (client_num == FD_SETSIZE) {
                        fprintf(stderr, "too many clients\n");
                        exit(-1);
                }

                FD_SET(connfd, &allset);
                if (connfd > maxfd) maxfd = connfd;
                if (--nready <= 0) continue;
        }

        //以下依次判断FD_ISSET(某个socket, &rset) 并做相应处理
}
```

特殊的要求

- 用户线程池
  - 应用于多线程程序结构，提供用户线程的支持，减少管态/目态切换和线程调度时间，提高切换效率，并增大系统支持的线程数。
- 内存池
  - 一般用于Client连接个数极多且经常性变化的Server，它对内存的分配和管理要求较高，因此用独立的内存池来实现用户级的内存分配和管理，提高程序效率，且有利于程序调试。
- 缓冲池
  - 对网络数据先大批存入缓冲池再进行响应操作，有利于减少对设备I/O的访问次数，提高程序运行效率，同时对I/O吞吐量大的Server避免了丢包的问题

THE END

Technology Tarena 达内科技