**第一章 Linux基础知识**

常用的Linux发行版

fedora, CentOS, openEuler（华为）, ubuntu, linuxMint, deepin, archlinux, manjaro, slackware, openSUSE, gentoo linux（unix不是）

GNU

GNU(GNU’s Not Unix)致力于开发一个自由的类Unix的操作系统，是Linux发展的重要基础

创作了通用公共许可证GPL（General Public License）——免费得到源码永久使用权、任意修改、有义务公开修改后的代码

Linux的组成

内核

运行程序和管理硬件设备的核心

SHELL

用户与内核交互的一种接口，可输入命令等

文件系统

文件存放在磁盘等存储设备上的组织方法。支持ext4、NFS、ISO9660等

应用系统

应用程序的程序集，包括文本编辑器、编程语言、X Window、办公套件、Internet工具等

MBR分区

Master Boot Record 是硬盘的第一扇区，也即启动扇区。记录了硬盘分区信息等。

对于MBR分区的硬盘来说，容量、分区数量受限分区有几类：主分区、扩展分区、逻辑分区。要安装操作系统，硬盘必须要有一个主分区，每个硬盘最多只能分成4个主分区想要更多分区，可在硬盘4个主分区中的一个用作扩展分区，在扩展分区中可以建立多个逻辑分区。MBR分区的硬盘，逻辑分区全部从5开始编号

软件安装

source install（通用方式）

tar包：先解包，然后make命令安装

bin包：通常直接执行

Ubuntu：apt-get install XX

Linux目录结构

bin 存放常用命令的执行文件

sbin 存放系统管理命令。执行这些命令通常需要root权限

root 超级用户的主目录

home 用户目录

tmp 临时文件

mnt挂载点

lib存放系统动态链接共享库

文件类型与文件权限

权限查看和修改

查看权限：

ls：列出目录内容

-a不隐藏任何以“.”字符开始的条目-l使用长格式列出信息-i查看inode号ls filename表示只查看这个文件

stat：查看文件的inode信息包含了权限、inode号等全部信息

更改文件权限：

chmod [选项] [权限] 文件

-R 递归方式更改（更改目录中所有文件的权限）

u（所有者User）、g（组成员Group）、o（其他人Other）

+（添加权限）、-（取消权限）

r（读Read）、w（写Write）、x（运行eXecute）

chown:改变文件的所有者

chgrp:改变文件或目录的所属组

软链接与硬链接

硬链接（Hard Link）：多个文件名指向同一索引节点（inode），只有当最后一个链接被删除后，文件的数据块才会被释放，不能跨文件系统进行硬链接的创建。不能对目录进行创建，只可对文件创建。本质是一个普通文件，与原文件无差别。

软链接（符号链接Symbolic Link）：类似于Windows的快捷方式。实际上是一个特殊的文件，与源文件不是同一个inode。软链接的数据块存放的是另一文件的路径名指向。

Linux常用命令及相关命令参数

文件操作

ls 列出目录内容

cd 改变工作目录

mkdir 创建目录

rmdir 删除空目录

pwd 显示出当前工作目录的绝对路径

rm：remove 删除文件或目录

【命令格式】：rm [选项] 文件或目录

-f 强制删除文件或目录

-i 删除既有文件或目录之前先询问用户

-r 递归删除目录，-rf参数一定要谨慎

ln：link 链接文件或目录

【命令格式】：ln [选项] 源文件 目标链接

-d建立硬链接，ln默认建立硬链接

-s建立符号链接（软链接）

grep：文件匹配

【命令格式】 grep [选项] 参数 [文件]

grep include tcp.c ：显示含include的行

more/less：分页显示文件内容

【命令格式】 more [选项] 文件

+n 从第n行开始显示，例如more +3 tcp.c

可用于输出结果的分页显示。例如ls -l | more

less比more功能更强大，支持向前向后翻页，more仅能向后翻页；分页时支持搜索功能

chmod 改变文件权限

touch：改变文件时戳

文件的修改时间和访问时间的更新为当前时间

【命令格式】touch 文件名

如果文件不存在，创建文件

其它命令

su： 切换用户

【命令格式】：su [用户名]

若不加用户名，默认为切换到root

部分发行版需要使用sudo su或sudo -i才能切换到超级用户

注意：输密码看不到任何显示

sudo

当前命令临时使用root权限，如sudo reboot

**第二章 Linux编程基础**

gcc使用

编译流程

C/C++代码

预处理：对各种预处理命令进行处理，包括头文件的包含、宏定义的扩展、条件编译的选择等

编译：对预处理完的文件进行词法分析、语法分析、语义分析及优化后生成相应的汇编代码

汇编代码

汇编：将汇编代码转换成机器可以执行的指令

指令

链接：链接程序运行所需的所有目标文件、所依赖的库文件，最后生成可执行文件

可执行文件

-E、-S、-c、-o、-g的作用和使用

gcc [选项] 要编译的文件 [选项] [-o 文件名]

gcc默认生成的目标文件为a.out

-E：只预处理，不编译

-S：只编译，不汇编

-c：只汇编，不链接

缺省：链接

-o file：输出文件名

-g：在可执行程序中包含标准调试信息

Makefile

objects = main.o add.o dec.o mul.o div.o #输入文件

CC=gcc

CFLAGS=-Wall -O -g #编译选项，可省略

program: $(objects)

$(CC) $^ -o $@

clean:

rm \*.o program

执行方法、默认执行目标

命令make target，执行target指定的目标

如果省略target，执行第一条目标

gdb

编译程序： gcc -g source.c -o program

启动调试： gdb program

查看代码： list(l)

设置断点： break(b)

查看断点： info b

运行程序： run(r)

单步调试 next(n)或step(s)

查看变量 print(p)

继续运行 c

退出 q

**第三章 内核与启动管理**

Linux系统管理

典型引导过程

上电 BIOS自检 MBR/GPT GRUB 加载内核 Init 登录

Linux启动过程

运行级别 运行模式

3 完全多用户模式，服务器系统

5 带图形界面的多用户模式，桌面系统

Linux内核

五个主要子系统

进程调度 进程间通信 内存管理 虚拟文件系统 网络接口

内核的裁剪、编译（含模块编译）、安装的命令

内核裁剪的常用方式

make config 字符界面

make menuconfig 文本图形

make xconfig 图形界面

内核的编译和安装

make clean把环境给清理干净

make dep 编译相关依赖文件

make bzImage 编译命令，可以加上-jN

make modules 编译模块驱动程序

make modules install/make install

grub2-mkconfig -o /boot/grub2/grub.cfg更新引导

**第四章 文件系统**

文件的逻辑结构：应用程序（用户）所看到的文件结构

流式文件（无结构文件）

记录式文件（有结构文件）

文件的物理结构：文件在设备上的存储组织形式，即存储结构、文件分配方式

文件系统负责文件的逻辑结构与物理结构之间的映射

文件的物理结构（分配方式）有3种方式

连续文件（连续分配）

链式文件（链接分配）

索引文件（索引分配）

Linux的物理结构：流式文件

Linux的逻辑结构：索引文件

Linux文件系统

支持的主要文件系统

磁盘文件系统：ext3、ext4……

网络文件系统：nfs……

文件名与inode

索引节点（inode）指向数据块，文件名与inode关联

inode包含了文件的元信息，文件的创建者、创建日期、大小等等。

文件控制块（FCB）主部包含除文件名之外的全部信息，称为索引节点（index node）或i节点（inode）

次部只包含文件名和inode号

块和块组的概念

文件系统中存储的最小单位是块（Block）

引导块大小为1KB，数据块通常为4KB

包含一定数量的连续存储块组成块组

块位图、inode位图、inode表的作用和大小

需要用块位图来描述块组内哪些块被占用。块位图占一个块，其中的每个bit代表本块组中的一个块，这个bit为1表示该块已用，0表示该块空闲可用

若块大小4KB，每个bit描述一个块，块位图最多可描述32K个块，块组最大为128MB

inode位图（inode Bitmap）和块位图类似，本身占一个块，其中每个bit表示一个inode是否空闲可用

inode表（inode Table）

存放inode，占多个块

inode存储文件的信息，除文件名外，文件信息都存放在inode中

ext4中，每个inode占用256字节

每个文件都有一个inode，一个块组中的所有inode组成了inode表

系统定位一个文件的过程

根据文件名获取inode号

ls -i 获取inode号

已知inode号，确定该inode的实际位置

通过dumpe2fs命令查看块大小、每块组inode个数等信息

先计算inode所在的块组（块组号从0开始)

inode所在块组=(inode号-1)/每块组inode个数

再计算inode在该块组inode表中的序号

inode序号=(inode号-1)%每块组inode个数

确定inode在文件系统中的起始字节

某inode表项起始字节=inode起始块号\*每块所占字节数+该i节点在块组内的序号\*每个i节点表项所占字节数

inode起始块号是文件系统的已知信息，同样可以通过dumpe2fs得到

根据inode中的索引表找到数据块

在文件系统中找到inode并读出

通过hexdump可以查看

ext3多重索引原理

ext3采用多重索引，索引表的长度为15

索引表的每个索引项占4字节，共60字节

0-11为直接指针，表示数据块的编号

12、13、14为一、二、三级间接指针

间接指针所指向的块并非数据块，而是间接寻址块，用于存放索引项

ext4对ext3的改进

块号从32位扩展到48位

32位的块号最多描述4G个块，文件系统的容量为4KB\*4G =16TB

扩展到48位后，容量最高为2^48\*4KB=1EB

inode占用字节数扩展为256

ext4区段索引的基本原理

ext4会尽它自己的努力保持文件连续

使用区段替代ext3中的索引表

区段长度为60字节

每个区段12字节，一个inode中可以存放5个区段结构

其中第一个为extend header，实际能够使用的extend body为4个

文件相关命令

du、df、fdisk、dd

fdisk：建立和管理磁盘分区表

-l：列出指定设备的参数和分区表

df：统计文件系统的空间使用情况

du：统计目录和文件占用的磁盘空间

dd：建立文件

查找文件find、locate

find：搜索文件

【命令格式】find [路径] [表达式]

例如：find /tmp -name thread\*

locate：查找文件

建立一个数据库，从该数据库中索引，速度快

locate速度比find快得多，可先用locate查找，找不到再用find

查找命令whereis、which

whereis：定位命令和man page（帮助手册）

which：定位命令

mount：挂载文件系统

mount [选项] [设备] [挂载点目录]

-t：指定文件类型，-t auto为自动检测类型

-o：指定挂载方式，例如loop

umount：卸载文件系统

重定向：把显示信息输入到文件中

用0表示标准输入；1表示标准输出；2表示错误信息

ls -l 1>log：把标准输入重定向到log（默认标准输出重定向，1可以省略）

ls -l 2>log：把错误信息重定向到log

ls -l >log 2>&1：把标准输入和错误信息重定向到log

**第五章 进程与线程编程**

程序、进程的概念

程序：一组指令的有序集合。程序是静态的

进程：具有独立功能的程序的一次运行过程。进程是动态的，有生命周期。一个程序可以有多个进程

进程的标识、特点

进程ID可唯一标识进程

进程的特点：独立性 动态性 并发性

进程相关命令

ps：process status 查看进程

【命令格式】：ps [选项]

-e 显示所有进程

-f 全格式

kill：结束进程

killall：结束所有同名进程

进程的状态与转换

Linux支持前台进程和后台进程

命令后面加上&，可在后台运行

bg/fg：将作业放入后台或前台

常用的进程调度算法（抢占与非抢占）

先进先出（FIFO）：非抢占，先来先服务

最短进程优先（SJF）：非抢占，每次都将从调度队列中选择所需运行时间最短的就绪进程来运行

轮转调度（RR，时间片轮询）：抢占，给每个就绪进程分配一个时间片，如果进程在一个时间片用完后仍不能完成运行，系统会剥夺该进程使用CPU的权限，然后选择下一个要运行的就绪进程并将CPU分配给它

优先级调度：抢占，每个进程被赋予一个优先级，优先级最高的就绪进程先运行

完全公平调度法(CFS)：

每个进程都设有一个虚拟时钟，用于计量进程已消耗的CPU时间。优先级低的进程其虚拟时钟快，所有进程分到的虚拟运行时间是相同的。进程切换时，选择虚拟时钟最小的那个进程。

Linux的进程调度，在策略上实时进程优先级高于普通进程。

实时进程的调度、普通进程的调度

实时进程的调度：严格按优先级进行调度，同优先级采用先进先出（FIFO）或轮转法（Round Robin）。

普通进程的调度：兼顾相应速度、公平性、整体效率。新内核中采用完全公平调度法（CFS）

死锁

系统中若干进程相互“无知地”等待对方所占有地资源而无限地处于等待状态地一种僵持局面。

互斥与同步

同步：相关进程为协作完成同一任务而引起的制约关系

互斥：进程间因竞争系统资源而引起的间接制约关系

僵尸进程的概念与防范

子进程先结束。防范：父进程检测子进程结束。

多进程编程

fork、exec、waitpid的作用与用法

fork()函数得到的子进程是父进程的复制品

子进程所独有的只有它的进程号、资源使用和计时器等。

fork仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

在父进程中，fork返回新创建子进程的进程ID；

在子进程中，fork返回0；

如果出现错误，fork返回一个负值；

exec函数族

int execlp(const char \*file, const char \*arg, ...);

前两个参数为程序名，之后的参数是执行程序的参数，最后的参数是NULL

当进程调用一种exec函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行

exec并不创建新进程

进程间通信

进程间通信的方式

管道及有名管道

管道（无名管道）只能用于具有亲缘关系的进程（如父子进程、兄弟进程）。它不是普通的文件，不属于某种文件系统，而是单独构成一种文件系统，且只存在于内存中。

有名管道不是临时对象，而是在文件系统中作为一个特殊的设备文件而存在的实体。进程只要拥有适当的权限就可以自由使用FIFO管道，无需具有亲缘关系。

共享内存是最快的进程间通信方式。同一块物理内存被映射到进程A、B各自的进程地址空间。

其它进程间通信方式：

信号、信号量、消息队列、套接字（Socket）

共享内存最快、套接字可以跨机器通信

程序、进程、线程的区别和联系

进程是具有独立功能的程序的一次运行过程，线程是进程中的一个实体。

资源分配方面，进程是操作系统资源分配的基本单位

CPU调度方面，线程是调度执行的基本单位

多线程在效率方面优于多进程

多进程更为健壮

多线程编程

编译时需要使用-lpthread

同步机制：信号量与互斥锁

**第六章 Linux网络编程**

网络相关命令

ifconfig：网络配置

【命令格式】ifconfig 网络接口名称 IP地址 [netmask子网掩码]

默认子网掩码为255.255.255.0

ifconfig不跟参数：查看网络

ping：测试网络的联通性

【命令格式】ping [参数] 主机名或IP

默认一直执行

-c 次数 可以设定执行多少次

route：配置路由表

什么是Socket

套接字，实质上是一种文件描述符

通信五元组

协议类型、目的IP、源IP、目的端口、源端口

大端存储与小端存储

小端存储：低位低字节

大端存储：高位低字节

网络字节序为大端存储。

字节序转换的必要性及使用的函数

不同系统字节序处理方式不同。

htonl，htons，ntohl，ntohs

h代表host, n代表network。s代表short，l代表long。

Socket编程

TCP通信流程

服务器端：socket、bind、listen、accept、send、recv、close

客户端：socket、bind、connect、send、recv、close

socket：创建套接字。

bind：绑定本地地址，将socket与本地的IP地址、端口关联。

listen：将套接字设置为监听模式，以等待连接请求。需要在bind成功之后调用。

accept：进入监听模式后，使用accept函数来响应连接请求，建立连接并产生一个新的socket描述符来描述该连接，用来与特定的客户端交换信息。

connect：建立与服务器的连接。

send：向socket写入数据。

recv：从socket读出数据。

close：关闭当前的连接。

UDP通信流程

服务器端：socket、bind、recvfrom、sendto、close

客户端：socket、bind、sendto、recvfrom、close

sendto：类似于send，在无连接的UDP通信时使用。

recvfrom：类似于recv，同上

出错诊断

网络不通：ping测试网络连通性

端口被占用，导致bind失败：

netstat查看端口被占用情况和监听情况

-nap查看所有的服务端口、协议

tcpdump可用于网络抓包

-n 不把网络地址转换成名字

-vv 输出详细的报文信息

-i 指定监听的网络接口

-XX 显示16进制和ASCII码

-e 显示帧头信息

tcpdump表达式

类型net、host、port

方向src、dst

协议tcp、udp、arp等

**第七章 Shel**l

几种shell运行方式的区别：

方式1： source filename 或者 . filename

在当前shell环境下读取并执行

方式2： sh(或bash、ksh…) filename

新建一个子shell读取并执行filename中的命令

子shell继承父shell的环境变量，但子shell新建的、改变的变量不会被带回父shell

方式3：./filename

打开一个子shell来读取并执行filename中命令

必须有可执行的权限

常用环境变量、更改环境变量的方法，位置参数变量的用法：

常用环境变量：

HOME：当前用户的主目录

PATH：用冒号分隔的目录路径名

Shell将顺序搜索这些目录，找到的第一个与命令名称一致的可执行文件将被执行

PWD：当前工作目录的绝对路径名，该变量的取值随cd命令的使用而变化。

更改环境变量的方法：在声明或赋值时使用export

export 变量名 export 变量名=变量值

专用参数变量：

命令行参数相关

$\* 将所有位置参量看成一个字符串(以空格间隔)

$@ 将每个位置参量看成单独的字符串(以空格间隔)

$0 命令行上输入的Shell程序名

$# 表示命令行上参数的个数

进程状态相关

$? 表示上一条命令执行后的返回值

$$ 当前进程的进程号

$! 显示运行在后台的最后一个作业的 PID

常用内置命令

echo arg：在屏幕上显示出由arg指定的字串

export 变量[=变量值]：添加环境变量

eval arg1 [arg2] ...：对所有参数进行扫描和替换，将所有的参数连接成一个表达式，并计算或执行该表达式。参数中的任何变量都将被展开

read [-p "信息"] [var1 var2 ...]：从键盘输入内容为变量赋值，若缺省，则将输入的内容存入REPLY变量 （readonly：不能被清除或重新赋值的变量）

shift [n]：将位置参量列表依次左移n次，缺省为左移一次，最左边的参量删除

declare [选项] variable[=value]：用来声明变量

exit :退出脚本或当前Shell，返回值存在#?中，0表示成功退出，非0失败

“.”（点）：.Shell程序文件名，执行指定shell文件

数学计算相关命令：

let：用于算术运算（赋值符号和运算符两边不能有空格，引号内可以忽略空格）

expr：通用的表达式计算命令（可以只写表达式，乘法符号必须被转义）

printf：按照指定格式输出

shell特殊符号

一般通配符：

\* 匹配不限长度的多个字符

? 匹配任意一个字符

[] 匹配字符组所限定的任何一个字符

[!] 表示不在方括号中所列出的字符

三种引号：

（1）双引号

双引号内的字符，除$、`和\仍保留其特殊功能外，其余字符均作为普通字符对待

（$表示变量替换 倒引号表示命令替换 \为转义字符）

（2）单引号

由单引号括起来的字符都作为普通字符出现，即使是$、`和\

（3）倒引号

倒引号括起来的字符串被shell解释为命令行，在执行时，Shell会先执行该命令行

两种命令替换

`命令表` dir=`pwd`

$(命令表) dir=$(pwd)

括号：

()——命令组、结合$进行命令替换、初始化数组array=(a b c d)

[]——字符范围、数组编号、算术运算、条件判断

{}——变量引用等

(())——算术运算、for循环中的算术运算比较for((i=0;i<5;i++))

[[]]——条件判断等

条件测试

语句

格式1： test <测试表达式>

格式2： [ <测试表达式> ]

格式3： [[ <测试表达式> ]]

格式1 和 格式2 是等价的，格式3是扩展的 test 命令

在 [[ ]] 中可以使用通配符进行模式匹配

&&, ||, <, 和>能够正常存在于[[ ]]中，但不能在 [] 中出现

文件测试：

[ -f fname ] fname 存在且是普通文件时，返回真 ( 即返回 0 )

[ -L fname ] fname 存在且是链接文件时，返回真

[ -d fname ] fname 存在且是一个目录时，返回真

[ -e fname ] fname（文件或目录）存在时，返回真

字符串测试：

[ -z string ] 如果字符串string长度为0，返回真

[ -n string ] 如果字符串string长度不为0，返回真

[ str1 = str2 ] 两字符串相等（也可使用 == ）返回真

[ str1 != str2 ] 两字符串不等返回真

[[ str1 == str2 ]] 两字符串相同返回真

[[ str1 != str2 ]] 两字符串不相同返回真

[[ str1 =~ str2 ]] str2是str1的子串返回真

[[ str1 > str2 ]] str1大于str2返回真

字符串按从左到右对应字符的ASCII码进行比较

整数测试：

[ int1 -eq int2 ] int1 等于 int2 返回真

[ int1 -ne int2 ] int1 不等于 int2 返回真

[ int1 -gt int2 ] int1 大于 int2 返回真

[ int1 -ge int2 ] int1 大于或等于 int2 返回真

[ int1 -lt int2 ] int1 小于 int2 返回真

[ int1 -le int2 ] int1 小于或等于 int2 返回真

操作符两边必须留空格！ [ ]里面只能比较数字

逻辑测试：

[[ pattern1 && pattern2 ]] 逻辑与

[[ pattern1 || pattern2 ]] 逻辑或

[[ ! pattern ]] 逻辑非 （[[ ]]和(( ))用法相同）

[ expr1 -a expr2 ] 逻辑与，都为真时，结果为真

[ expr1 -o expr2 ] 逻辑或，有一个为真时，结果为真

[ ! expr ] 逻辑非

流程控制

if

if expr1 # 如果 expr1 为真(返回值为0)

Then # 那么

commands1 # 执行语句块 commands1

elif expr2 # 若 expr1 不真，而 expr2 为真

Then # 那么

commands2 # 执行语句块 commands2

... ... # 可以有多个 elif 语句

else # else 最多只能有一个

commands4 # 执行语句块 commands4

fi # if 语句必须以单词 fi 终止

case

case expr in # expr 为表达式，关键词 in 不要忘！

pattern1) # 若 expr 与 pattern1 匹配，注意括号

commands1 # 执行语句块 commands1

;; # 跳出 case 结构

pattern2) # 若 expr 与 pattern2 匹配

commands2 # 执行语句块 commands2

;; # 跳出 case 结构

... ... # 可以有任意多个模式匹配

\*) # 若 expr 与上面的模式都不匹配

commands # 执行语句块 commands

;; # 跳出 case 结构

Esac # case 语句必须以 esac 终止

for循环

for var in list # 每一次循环，依次把列表 list 中的一个值赋给循环变量

do # 循环体开始的标志

commands # 循环变量每取一次值，循环体就执行一遍

done # 循环结束的标志，返回循环顶部

While循环

while expr # 执行 expr

do # 若expr的退出状态为0，进入循环，否则退出while

commands # 循环体

done # 循环结束标志，返回循环顶部

until循环

until expr # 执行 expr

do # 若expr的退出状态非0，进入循环，否则退出until

commands # 循环体

done # 循环结束标志，返回循环顶部

break

用于强行退出当前循环

continue

用于忽略本次循环的剩余部分，回到循环的顶部，继续下一次循环

多进程实例（waitpid）

int main(void){

pid\_t pid, pid\_w;

int i;

pid = fork();

if(pid<0)

exit(1);

if(pid == 0) /\*\*\*\*\*\*\*\*子进程\*\*\*\*\*\*\*\*/

{

for(i=3;i>0;i--){

printf("This is the child\n");

sleep(2);

}

exit(0);

}

else{

do{

pid\_w=waitpid(pid,NULL,WNOHANG);

if(pid\_w==0){

printf(“child process is running\n");

sleep(1);

}

}while(pid\_w==0);

if(pid\_w==pid)

printf(“Child Process %d End!\n",pid\_w);

else

printf("some error occured.\n");

}

}

父子进程之间管道通信

int main()

{

int fd[2], nbytes;

pid\_t childpid;

char string[] = "Hello, world!\n";

char readbuffer[80];

if(pipe(fd)<0)

{ printf("创建失败\n");

return -1;

}

if((childpid = fork())== -1)

{ perror("fork");

exit(1);

}

if(childpid == 0)

{ close(fd[0]);

sleep(3);

write(fd[1], string, strlen(string));

close(fd[1]);

exit(0);

}

else

{ close(fd[1]);

nbytes = read(fd[0], readbuffer, sizeof(readbuffer));

printf("Received string: %s", readbuffer);

close(fd[0]);

}

return(0);

}

有名管道

只写端

int main()

{

char write\_fifo\_name[] = “fifo-pipe1”;

char read\_fifo\_name[] = "fifo-pipe2";

int write\_fd, read\_fd;

char buf[256];

int len;

int ret = mkfifo(write\_fifo\_name, S\_IRUSR |

S\_IWUSR);

write\_fd = open(write\_fifo\_name, O\_WRONLY);

while((read\_fd = open(read\_fifo\_name,O\_RDONLY))

== -1)

{

printf("Wait for %s...\n",read\_fifo\_name);

sleep(1);

}

while(1)

{

printf("Lucy: ");

fgets(buf, 256, stdin);

buf[strlen(buf)-1] = ‘\0’;

write(write\_fd, buf, strlen(buf));

len = read(read\_fd, buf, 256);

if( len > 0)

{

buf[len] = ‘\0’;

printf("Peter: %s\n", buf);

}

}

}

只读端

//\*\*\*\*\*fiforead.c\*\*\*

int main(void)

{

char write\_fifo\_name[] = “fifo-pipe2”;

char read\_fifo\_name[] = “fifo-pipe1”;

int write\_fd, read\_fd;

char buf[256];

int len;

int ret = mkfifo(write\_fifo\_name, S\_IRUSR | S\_IWUSR);

while((read\_fd = open(read\_fifo\_name, O\_RDONLY)) == -1)

{

sleep(1);

}

write\_fd = open(write\_fifo\_name, O\_WRONLY);

while(1)

{

len = read(read\_fd, buf, 256);

if(len > 0)

{

buf[len] = '\0';

printf("Lucy: %s\n",buf);

}

printf("Peter: ");

fgets(buf, 256, stdin);

buf[strlen(buf)-1] = '\0';

write(write\_fd, buf, strlen(buf));

}

}

共享内存通信

写端

/\*\*\*\*\* sharewrite.c \*\*\*\*\*\*\*/

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

{

int shm\_id,i;

key\_t key;

char temp;

char \* p\_map;

key = ftok(“.”, ’a’);

if(key==-1)

perror("ftok error");

shm\_id=shmget(key, 4096, IPC\_CREAT|0666);

if(shm\_id==-1)

{

perror("shmget error");

return;

}

p\_map=(char\*)shmat(shm\_id, NULL, 0);

temp='a';

for(i = 0;i<10;i++)

{

\*(p\_map+i)=temp;//write to shared memery

temp+=1;

}

if(shmdt(p\_map)==-1)

perror(" detach error ");

}

读端

/\*\*\*\*\*\*\*\*\*\* shareread.c \*\*\*\*\*\*\*\*\*\*\*\*/

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

{ int shm\_id, i;

key\_t key;

char \* p\_map;

key = ftok(“.”, ’a’);

if(key == -1)

perror("ftok error");

shm\_id = shmget(key, 4096, 0666);

if(shm\_id == -1)

{

perror("shmget error");

return;

}

p\_map = (char\*)shmat(shm\_id, NULL, 0);

for(i = 0; i<10; i++)

{

printf( “%c\n“, \*(p\_map+i) );

}

if(shmdt(p\_map) == -1)

perror(" detach error ");

}

多线程实例

void thread(void)

{

int i;

for(i=0;i<3;i++)

printf("This is a pthread.\n");

}

int main(void)

{

pthread\_t id;

int i,ret;

ret=pthread\_create(&id,NULL,(void \*) thread,NULL);

if(ret!=0){

printf ("Create pthread error!\n");

exit (1);

}

for(i=0;i<3;i++)

printf("This is the main process.\n");

pthread\_join(id,NULL);

return (0);

}

Tcp sever（服务端）

int main(int argc, char \*argv[])

{

int sockfd, new\_fd;

struct sockaddr\_in server\_addr, client\_addr;

int sin\_size, portnumber;

char input[1024];

short portnumber = atoi(argv[1]);

if(-1==(sockfd=socket(AF\_INET,SOCK\_STREAM,0)))

exit(1);

bzero(&server\_addr, sizeof(struct sockaddr\_in));

server\_addr.sin\_family=AF\_INET;

server\_addr.sin\_addr.s\_addr=htonl(INADDR\_ANY);

server\_addr.sin\_port=htons(portnumber);

bind(sockfd, (struct sockaddr \*)(&server\_addr),

sizeof(struct sockaddr) )

listen(sockfd,5)

while(1){

bzero(&client\_addr, sizeof(struct sockaddr\_in));

sin\_size=sizeof(client\_addr);

new\_fd=accept(sockfd,

(struct sockaddr \*)(&client\_addr), &sin\_size)

fgets(input,1000, stdin);

send(new\_fd, input, strlen(input), 0)

close(new\_fd);

}

close(sockfd);

exit(0);

}

Tcp client(客户端)

int main(int argc, char \*argv[])

int sockfd;

char buffer[1024];

struct sockaddr\_in server\_addr;

int nbytes;

portnumber=atoi(argv[2]);

sockfd=socket(AF\_INET, SOCK\_STREAM, 0)

bzero(&server\_addr, sizeof(struct sockaddr\_in));

server\_addr.sin\_family=AF\_INET;

server\_addr.sin\_port=htons(portnumber);

server\_addr.sin\_addr.s\_addr=inet\_addr(argv[1]);//IP

if(-1==connect(sockfd,(struct sockaddr \*)

(&server\_addr), sizeof(struct sockaddr)))

{

printf("Connect Error\n”);

exit(1);

}

if(-1==(nbytes=recv(sockfd,buffer,1024,0)))

printf("Read Error\n”);

exit(1);

}

buffer[nbytes]='\0';

printf("I have received:%s\n",buffer);

close(sockfd);

exit(0);

}

Udp服务端

int main(int argc, char\* argv[])

{

int sockfd;

struct sockaddr\_in servaddr, cliaddr;

sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(atoi(argv[1]));//指定端口号

bind(sockfd, (struct sockaddr \*) &servaddr,

sizeof(servaddr))

do\_echo(sockfd, (struct sockaddr \*) &cliaddr,

sizeof(cliaddr));

return 0;

}

void do\_echo(int sockfd, struct sockaddr \*pcliaddr,

socklen\_t clilen){

int n;

socklen\_t len = clilen;

char mesg[80];

for(;;){

n = recvfrom(sockfd, mesg, 80, 0, pcliaddr,&len);

mesg[n] = 0;

printf("\nClient Says: %s\n", mesg);

printf("Server says:");

fgets(mesg, 80, stdin);

sendto(sockfd,mesg,strlen(mesg),0,pcliaddr,len);

}

}

UDP客户端程序

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

{

int sockfd;

struct sockaddr\_in servaddr;

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port=htons(atoi(argv[2]));//第3参数为端口

servaddr.sin\_addr.s\_addr = inet\_addr(argv[1]);//IP

sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

do\_cli(stdin, sockfd, (struct sockaddr \*) &servaddr,

sizeof(servaddr));

return 0;

}

void do\_cli(FILE \*fp,int sockfd,struct sockaddr \*pservaddr,

socklen\_t servlen){

int n;

socklen\_t len=servlen;

char sendline[80], recvline[80+1];

while(1)

{

printf("Client Says:");

fgets(sendline, 80, stdin);

sendto(sockfd, sendline, strlen(sendline), 0,

pservaddr, len);

n=recvfrom(sockfd,recvline,80,0,pservaddr,&len);

recvline[n]=0;

printf("Server Reply:%s\n",recvline);

}

}