



Cortex-A9

嵌入式 Linux 移植及应用驱动

FS4412 部分

(V1.1)

华清远见教育集团 • 研发中心



170-9108-5953



2306275952/2668462267



<http://dev.hqyj.com>



support@farsight.com.cn

华清远见
dev.hqyj.com



前言

本书从嵌入式 Linux 的基础知识、系统环境搭建到综合应用，共分三个层次深入浅出地为读者拨开萦绕于嵌入式 Linux 这个概念的重重迷雾，引领读者渐渐步入嵌入式的世界，帮助探索者实实在在地把握第三次 IT 科技浪潮的方向。本书的特色如下：

(1) 重基础，适合教学。

(2) 重素质，全面讲解。本书在一般性教材的基础上，对嵌入式系统的软硬件开发环境进行了大量的讲解，可以让读者更进一步、更全面地了解嵌入式的开发过程。

(3) 重实践，与实际项目相结合。本书每个原理讲解都对应着丰富的实验内容，并且原理讲解和实验内容一一对应，使得每个知识都可以被深刻的理解运用，本书附带的光盘中给出了参考设计代码和文档。

(4) 重应用。书中的实例对时下经常使用的功能、设备、器材进行讲解和说明，力求教材所涉及的内容能紧跟行业实际应用的需要。

本书共分 19 个章节，每个章节下面包含一个对应此章节内容的实验。

第 1 章 嵌入式 Linux 主机开发环境搭建

第 2 章 嵌入式 Linux 主机调试环境搭建

第 3 章 交叉开发环境搭建

第 4 章 Linux 基础操作

第 5 章 Linux Shell 编程

第 6 章 GCC 编译

第 7 章 Linux 系统 Makefile 编写实验

第 8 章 嵌入式文件 I/O 编程实验

第 9 章 嵌入式 Linux 多任务编程实验

第 10 章 Linux 系统进程间通信实验

第 11 章 嵌入式 Linux 多线程编程实验

第 12 章 嵌入式 Linux 网络编程实验

第 13 章 BootLoader (Uboot) 移植实验

第 14 章 Linux 系统移植实验

第 15 章 Linux 文件系统构建实验

第 16 章 LED 驱动开发实验

第 17 章 PWM 驱动开发实验

第 18 章 按键驱动开发实验

第 19 章 ADC 驱动开发实验

第 20 章 I2C 设备驱动开发实验

第 19 章 其他驱动实验



华清远见
dev.hqyj.com



目 录

前 言	I
目 录	- 1 -
第 1 章 嵌入式 Linux 主机开发环境搭建	- 11 -
1.1 安装前准备	- 12 -
1.1.1 主机配置	- 12 -
1.1.2 Intel Virtualization Technology (32 位操作系统)	- 12 -
1.2 安装 VMware Player	- 14 -
1.3 运行开发环境	- 20 -
1.3.1 解压虚拟机镜像	- 20 -
1.3.2 打开虚拟机	- 22 -
1.3.3 配置优化虚拟机	- 24 -
1.3.4 启动虚拟机	- 29 -
1.3.5 设置 ROOT 密码	- 32 -
第 2 章 嵌入式 Linux 主机调试环境搭建	- 33 -
2.1 Linux 系统配置 TFTP 实验	- 33 -
2.1.1 实验目的	- 33 -
2.1.2 实验平台	- 33 -
2.1.3 实验原理	- 33 -
2.1.4 实验步骤	- 33 -
2.2 Linux 系统配置 NFS 实验	- 34 -
2.2.1 实验目的	- 34 -
2.2.2 实验平台	- 34 -
2.2.3 实验原理	- 34 -
2.2.4 实验步骤	- 34 -
第 3 章 交叉开发环境搭建	- 36 -
3.1 实验目的	- 36 -
3.2 实验平台	- 36 -
3.3 实验原理	- 36 -
3.4 实验步骤	- 36 -
3.4.1 配置开发环境网络	- 36 -
3.4.2 配置交叉工具链	- 37 -
3.4.3 拷贝文件	- 39 -
3.4.4 将共享目录中需要下载的文件拷贝到 tftp 目录中	- 39 -
3.4.5 解压文件系统	- 39 -



3.4.6	连接开发板.....	- 41 -
3.4.7	设置串口调试工具	- 41 -
3.4.8	启动开发板.....	- 43 -
3.4.9	烧写 uboot.....	- 46 -
3.4.10	NFS 挂载方式启动.....	- 47 -
3.4.11	EMMC 方式启动	- 50 -
3.4.12	制作 SD 卡启动盘（只需在开发板没有 UBoot 时做）	- 51 -
第 4 章	Linux 基础操作	- 58 -
4.1	Linux 基本命令	- 58 -
4.1.1	用户系统相关命令	- 59 -
4.1.2	文件目录相关命令	- 65 -
4.1.3	压缩打包相关命令	- 77 -
4.1.4	比较合并文件相关命令	- 80 -
4.1.5	网络相关命令	- 86 -
4.2	Linux 编辑器 vi 的使用.....	- 89 -
4.2.1	vi 的模式.....	- 89 -
4.2.2	vi 的基本流程	- 89 -
4.2.3	vi 的各模式功能键.....	- 91 -
第 5 章	Linux Shell 编程.....	- 94 -
5.1	实验原理	- 94 -
5.2	实验目的	- 94 -
5.3	实验平台	- 94 -
5.4	实验步骤	- 95 -
5.4.1	准备环境.....	- 95 -
5.4.2	拷贝代码.....	- 95 -
5.4.1	执行代码.....	- 96 -
5.4.2	实验结果.....	- 96 -
5.5	相关代码	- 96 -
第 6 章	GCC 编译.....	- 97 -
6.1	实验原理	- 97 -
6.2	实验目的	- 98 -
6.3	实验平台	- 98 -
6.4	实验步骤	- 98 -
6.4.1	准备环境.....	- 98 -
6.4.2	拷贝代码.....	- 99 -



6.4.3	编译代码	- 99 -
6.4.4	执行代码	- 99 -
6.4.5	实验结果	- 100 -
6.5	相关代码	- 100 -
第 7 章	Linux 系统 Makefile 编写实验	- 102 -
7.1	实验原理	- 102 -
7.1.1	Makefile 基本结构	- 102 -
7.1.2	Makefile 变量	- 103 -
7.1.3	Makefile 规则	- 106 -
7.1.4	Make 管理器的使用	- 107 -
7.2	实验目的	- 107 -
7.3	实验平台	- 107 -
7.4	实验步骤	- 108 -
7.4.1	环境准备	- 108 -
7.4.2	拷贝代码	- 108 -
7.4.3	执行代码	- 109 -
7.5	相关代码	- 111 -
第 8 章	嵌入式文件 I/O 编程实验	- 113 -
8.1	实验原理	- 113 -
8.1.1	系统调用	- 113 -
8.1.2	用户编程接口 (API)	- 113 -
8.1.3	系统命令	- 113 -
8.1.4	虚拟文件系统 (VFS)	- 114 -
8.1.5	Linux 中文件及文件描述符	- 115 -
8.1.6	标准 I/O 编程	- 115 -
8.2	标准 IO 实验	- 122 -
8.2.1	实验目的	- 122 -
8.2.2	实验平台	- 122 -
8.2.3	实验步骤	- 122 -
8.2.4	实验现象	- 122 -
8.2.5	实验代码	- 123 -
8.3	Linux 系统文件目录操作编程实验	- 123 -
8.3.1	实验目的	- 123 -
8.3.2	实验平台	- 123 -
8.3.3	实验步骤	- 123 -



8.3.4	实验现象	- 124 -
8.3.5	实验代码	- 124 -
第 9 章	嵌入式 Linux 多任务编程实验	- 129 -
9.1	使用 ps 命令查看进程信息	- 129 -
9.1.1	实验内容	- 129 -
9.1.2	实验平台	- 129 -
9.1.3	实验原理	- 129 -
9.1.4	实验步骤	- 129 -
9.2	使用 proc 文件系统查看进程信息	- 129 -
9.2.1	实验目的	- 129 -
9.2.2	实验平台	- 129 -
9.2.3	实验原理	- 129 -
9.2.4	实验内容	- 130 -
9.3	使用 fork、exit 和 exec 系统调用编写多进程程序	- 131 -
9.3.1	实验目的	- 131 -
9.3.2	实验平台	- 131 -
9.3.3	实验原理	- 131 -
9.3.4	实验步骤	- 131 -
9.3.5	实验现象	- 132 -
9.3.6	实验代码	- 132 -
9.4	Linux 系统守护进程实验	- 133 -
9.4.1	实验目的	- 133 -
9.4.2	实验平台	- 133 -
9.4.3	实验原理	- 133 -
9.4.4	实验步骤	- 134 -
9.4.5	实验现象	- 134 -
9.4.6	实验代码	- 135 -
第 10 章	Linux 系统进程间通信实验	- 137 -
10.1	实验原理	- 137 -
10.1.1	管道通信	- 138 -
10.1.2	无名管道系统调用	- 138 -
10.1.3	标准流管道	- 140 -
10.1.4	有名管道 (FIFO)	- 141 -
10.1.5	信号通信	- 142 -
10.1.6	信号量	- 149 -



10.1.7	共享内存	- 154 -
10.1.8	消息队列	- 155 -
10.2	Linux 系统无名管道通信实验	- 158 -
10.2.1	实验目的	- 158 -
10.2.2	实验平台	- 158 -
10.2.3	实验步骤	- 158 -
10.2.4	实验现象	- 158 -
10.2.5	实验代码	- 158 -
10.3	Linux 系统有名管道通信实验	- 160 -
10.3.1	实验目的	- 160 -
10.3.2	实验平台	- 160 -
10.3.3	实验内容	- 160 -
10.3.4	实验步骤	- 160 -
10.3.5	实验现象	- 161 -
10.3.6	实验代码	- 161 -
10.4	Linux 系统信号机制实验	- 165 -
10.4.1	实验目的	- 165 -
10.4.2	实验平台	- 165 -
10.4.3	实验内容	- 165 -
10.4.4	实验步骤	- 165 -
10.4.5	实验现象	- 166 -
10.4.6	实验代码	- 166 -
10.5	Linux 系统共享内存通信实验	- 168 -
10.5.1	实验目的	- 168 -
10.5.2	实验平台	- 168 -
10.5.3	实验内容	- 168 -
10.5.4	实验步骤	- 168 -
10.5.5	实验现象	- 169 -
10.5.6	实验代码	- 169 -
10.6	Linux 系统消息队列实验	- 173 -
10.6.1	实验目的	- 173 -
10.6.2	实验平台	- 173 -
10.6.3	实验内容	- 173 -
10.6.4	实验步骤	- 173 -
10.6.5	实验现象	- 174 -



10.6.6	实验代码	- 175 -
第 11 章	嵌入式 Linux 多线程编程实验	- 179 -
11.1	实验目的	- 179 -
11.2	实验平台	- 179 -
11.3	实验原理	- 179 -
11.3.1	线程基本编程	- 179 -
11.3.2	线程之间的同步与互斥	- 181 -
11.3.3	线程属性	- 184 -
11.4	实验步骤	- 186 -
11.4.1	实验现象	- 187 -
11.5	实验代码	- 187 -
第 12 章	嵌入式 Linux 网络编程实验	- 189 -
12.1	实验原理	- 189 -
12.1.1	TCP/IP 的分层模型	- 189 -
12.1.2	TCP/IP 分层模型特点	- 190 -
12.1.3	TCP/IP 核心协议	- 191 -
12.1.4	套接字 (socket) 概述	- 194 -
12.1.5	地址及顺序处理	- 195 -
12.1.6	套接字编程	- 200 -
12.2	Linux 系统 tcp 网络协议编程实验	- 205 -
12.2.1	实验目的	- 205 -
12.2.2	实验平台	- 205 -
12.2.3	实验内容	- 205 -
12.2.4	实验步骤	- 205 -
12.2.5	实验现象	- 206 -
12.2.6	实验代码	- 206 -
12.3	Linux 系统 udp 网络协议编程实验	- 210 -
12.3.1	实验目的	- 210 -
12.3.2	实验平台	- 210 -
12.3.3	实验内容	- 210 -
12.3.4	实验步骤	- 210 -
12.3.5	实验现象	- 211 -
12.3.6	实验代码	- 211 -
12.4	Linux 系统 select IO 复用实验	- 214 -
12.4.1	实验目的	- 214 -



12.4.2	实验平台	- 214 -
12.4.3	实验内容	- 214 -
12.4.4	实验步骤	- 214 -
12.4.5	实验现象	- 215 -
12.4.6	实验代码	- 215 -
第 13 章	BootLoader (Uboot) 移植实验	- 220 -
13.1	实验原理	- 220 -
13.1.1	概念	- 220 -
13.1.2	Bootloader 启动流程	- 220 -
13.1.3	Bootloader 的种类	- 221 -
13.1.4	U-Boot 概述	- 223 -
13.1.5	U-Boot 的常用命令	- 224 -
13.2	实验目的	- 226 -
13.3	实验平台	- 226 -
13.4	实验步骤	- 226 -
13.4.1	建立自己的平台	- 226 -
13.4.2	u-boot 移植	- 228 -
13.5	实验现象	- 252 -
第 14 章	Linux 系统移植实验	- 253 -
14.1	实验原理	- 253 -
14.2	内核的配置和编译	- 254 -
14.2.1	解压内核	- 254 -
14.2.2	修改内核顶层目录下的 Makefile	- 254 -
14.2.3	拷贝标准板配置文件	- 255 -
14.2.4	配置内核	- 255 -
14.2.5	编译内核	- 255 -
14.2.6	生成设备树	- 255 -
14.3	以太网卡驱动移植	- 256 -
14.3.1	实验目的	- 256 -
14.3.2	实验平台	- 256 -
14.3.3	实验步骤	- 256 -
14.3.4	实验现象	- 258 -
14.4	SD 卡驱动移植	- 258 -
14.4.1	实验目的	- 258 -
14.4.2	实验平台	- 259 -



14.4.3	实验步骤.....	- 259 -
14.4.4	实验现象.....	- 260 -
14.5	USB 驱动移植.....	- 260 -
14.5.1	实验目的.....	- 260 -
14.5.2	实验平台.....	- 260 -
14.5.3	实验步骤.....	- 260 -
14.6	LCD 驱动移植.....	- 264 -
14.6.1	实验目的.....	- 264 -
14.6.2	实验平台.....	- 264 -
14.6.3	实验步骤.....	- 264 -
14.6.4	实验现象.....	- 269 -
第 15 章	Linux 文件系统构建实验.....	- 270 -
15.1	实验原理.....	- 270 -
15.1.1	磁盘的物理组织.....	- 270 -
15.1.2	文件和目录.....	- 271 -
15.1.3	文件的分类.....	- 271 -
15.1.4	目录.....	- 272 -
15.1.5	文件系统.....	- 273 -
15.1.6	虚拟文件系统.....	- 273 -
15.1.7	虚拟文件系统概述.....	- 273 -
15.2	根文件系统开发实验.....	- 274 -
15.2.1	实验目的.....	- 274 -
15.2.2	实验平台.....	- 274 -
15.2.3	实验步骤.....	- 274 -
15.2.4	实验现象.....	- 277 -
15.3	Ramdisk 文件系统制作实验.....	- 278 -
15.3.1	实验目的.....	- 278 -
15.3.2	实验平台.....	- 278 -
15.3.3	实验步骤.....	- 278 -
第 16 章	LED 驱动开发实验.....	- 280 -
16.1	实验原理.....	- 280 -
16.2	实验目的.....	- 280 -
16.3	实验平台.....	- 280 -
16.4	实验步骤.....	- 281 -
16.4.1	环境准备.....	- 281 -



16.4.2	代码准备	- 281 -
16.4.3	编译代码	- 282 -
16.4.4	执行代码	- 284 -
16.4.5	实验现象	- 285 -
第 17 章	PWM 驱动开发实验	- 286 -
17.1	实验原理	- 286 -
17.2	实验目的	- 287 -
17.3	实验平台	- 287 -
17.4	实验步骤	- 287 -
17.4.1	环境准备	- 287 -
17.4.2	代码准备	- 287 -
17.4.3	编译代码	- 287 -
17.4.4	执行代码	- 288 -
17.4.5	实验现象	- 289 -
第 18 章	按键驱动开发实验	- 290 -
18.1	实验原理	- 290 -
18.2	实验目的	- 290 -
18.3	实验平台	- 290 -
18.4	实验步骤	- 290 -
18.4.1	环境准备	- 290 -
18.4.2	代码准备	- 290 -
18.4.3	编译代码	- 290 -
18.4.4	修改设备树文件	- 291 -
18.4.5	执行代码	- 292 -
18.4.6	实验现象	- 292 -
第 19 章	ADC 驱动开发实验	- 293 -
19.1	实验原理	- 293 -
19.2	实验目的	- 294 -
19.3	实验平台	- 294 -
19.4	实验步骤	- 294 -
19.4.1	环境准备	- 294 -
19.4.2	代码准备	- 294 -
19.4.3	编译代码	- 294 -
19.4.4	编译应用程序	- 295 -
19.4.5	修改设备树文件	- 295 -



19.4.6	执行代码	- 295 -
19.4.7	实验现象	- 296 -
第 20 章	I2C 设备驱动开发实验	- 297 -
20.1	实验原理	- 297 -
20.2	实验目的	- 297 -
20.3	实验平台	- 297 -
20.4	实验步骤	- 297 -
20.4.1	环境准备	- 297 -
20.4.2	代码准备	- 297 -
20.4.3	编译代码	- 297 -
20.4.4	编译应用程序	- 298 -
20.4.5	修改设备树文件	- 298 -
20.4.6	执行代码	- 299 -
20.4.7	实验现象	- 300 -



第 1 章 嵌入式 Linux 主机开发环境搭建

华清远见开发环境是基于 Ubuntu 12.04 LTS 64-bit 操作系统搭建的，使用 [VMware Player](#)（**免费版**）作为虚拟机工具软件（用户也可以使用 VMware 公司所提供的**付费版**虚拟机软件 [VMware Workstation](#) 代替 VMware Player）。本开发环境可用作嵌入式 Linux 和 Android 的编译与开发。

本开发环境在 Ubuntu 12.04 64-bit LTS 基础上，安装了编译调试 Bootloader、Linux、Android 系统所需要的工具和依赖的库，用户可以在无需额外操作的基础上，直接使用本开发环境进行嵌入式的学习和工作。

本开发环境在 Ubuntu 12.04 64-bit 基础上，安装配置了如下工具：

- 将 GCC、G++ 编译器版本从 4.6 降至 4.4；
- 安装了 Android 编译所需要的工具和库（source.android.com）；
- 安装 SUN JAVA JDK 6；
- 安装内核编译所依赖的工具包；
- 解决了 libncurses 32 位和 64 位不能同时安装导致编译 Android 和配置内核软件冲突的问题；
- 安装制作安卓文件系统 yaffs2 格式 mkyaffs 工具；
- 添加了常用的 arm-linux 交叉工具链，版本号为 4.3.2、4.4.6、4.5.1；
- 安装 Vim、Ctags；
- 安装 Vim 常用插件；
- 安装配置 TFTP；
- 安装配置 NFS 网络文件系统服务；
- 安装 SSH 工具网络服务程序；
- 安装 Krmit 串口调试工具；
- 安装 Sogou 输入法；
- 关闭 Ubuntu 更新提示；

说明 1：Ubuntu 用户名为“linux”，主机名为“ubuntu64”，默认密码为“1”

说明 2：以上安装过的软件和库，用户在不明确的前提下，切勿再次安装！（比如 source.android.com 要求的、tftp 服务等等），如果因为重复安装导致源码编译错误，请重新解压开发环境镜像。

下图是使用华清远见开发环境编译源码：

```

1 #
2 # (C) Copyright 2006-2008
3 # Wolfgang Denk, DENX Software Engineering, wd@denx.de.
4 #
5 # See file CREDITS for list of people who contributed to this
6 # project.
7 #
8 # This program is free software; you can redistribute it and/or
9 # modify it under the terms of the GNU General Public License as
10 # published by the Free Software Foundation; either version 2 of
11 # the License, or (at your option) any later version.
12 #
13 # This program is distributed in the hope that it will be useful,
14 # but WITHOUT ANY WARRANTY; without even the implied warranty of
15 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16 # GNU General Public License for more details.
17 #
18 # You should have received a copy of the GNU General Public License
19 # along with this program; if not, write to the Free Software
20 # Foundation, Inc., 59 Temple Place, Suite 330, Boston,
21 # MA 02111-1307 USA
22 #
23
24 VERSION = 1
25 PATCHLEVEL = 3
26 SUBLEVEL = 4
27 EXTRAVERSION =
28 U_BOOT_VERSION = $(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)
29 VERSION_FILE = $(obj)include/version_autogenerated.h
30
31 HOSTARCH := $(shell uname -n | \
32 sed -e s/\.86/L386/ \
33 -e s/sun4u/sparc64/ \
34 -e s/arm-/arm/ \
35 -e s/sa110/arm/ \
36 -e s/powerpc/ppc/ \

```



1.1 安装前准备

1.1.1 主机配置

华清远见开发环境是基于 Ubuntu 12.04 LTS 64-bit 操作系统搭建的，使用 VMware Player **免费版** 作为虚拟机工具软件。用作 Linux 和 Android 的编译与开发。所以建议开发主机硬件配置越高越好，配置越高则开发效率则越高。

配置	参数
CPU	Intel/AMD CPU; 主频 2GHz 或者更多; 双核或者更多; 支持 EM64T (INTEL)或者 X86_64(AMD)指令; 支持 Intel Virtualization Technology (32 位操作系统必须)
内存	内存 2G 以上 (Windows XP / 7 / 8 32 位); 内存 4G 以上 (Windows 7 / 8 64 位);
硬盘	虚拟机存放硬盘分区剩余空间 80G 以上; 推荐使用虚拟机挂载物理 ext 分区 (无 Linux 基础者慎用)
USB	支持 USB2.0 或者更高; 2 路或者更多;
网卡	至少一路以太网卡;
串口	RS232 串口 (也可以用 USB 转串口替代)
操作系统	Windows XP 或者更新; 32 位/64 位操作系统;

1.1.2 Intel Virtualization Technology (32 位操作系统)

此选项为用户**主机操作系统为 32 位必须具备**，如果主机操作系统为 64 位，则可忽略此部。

华清远见开发环境是基于 Ubuntu 12.04 LTS 64-bit 操作系统搭建的，使用 VMware Player 作为虚拟机工具软件。所以在如果主机为 32 位操作系统，那么系统必须支持英特尔虚拟化技术 (Intel Virtualization Technology) 才可以通过 VMware Player 工具使用 64 位操作系统。



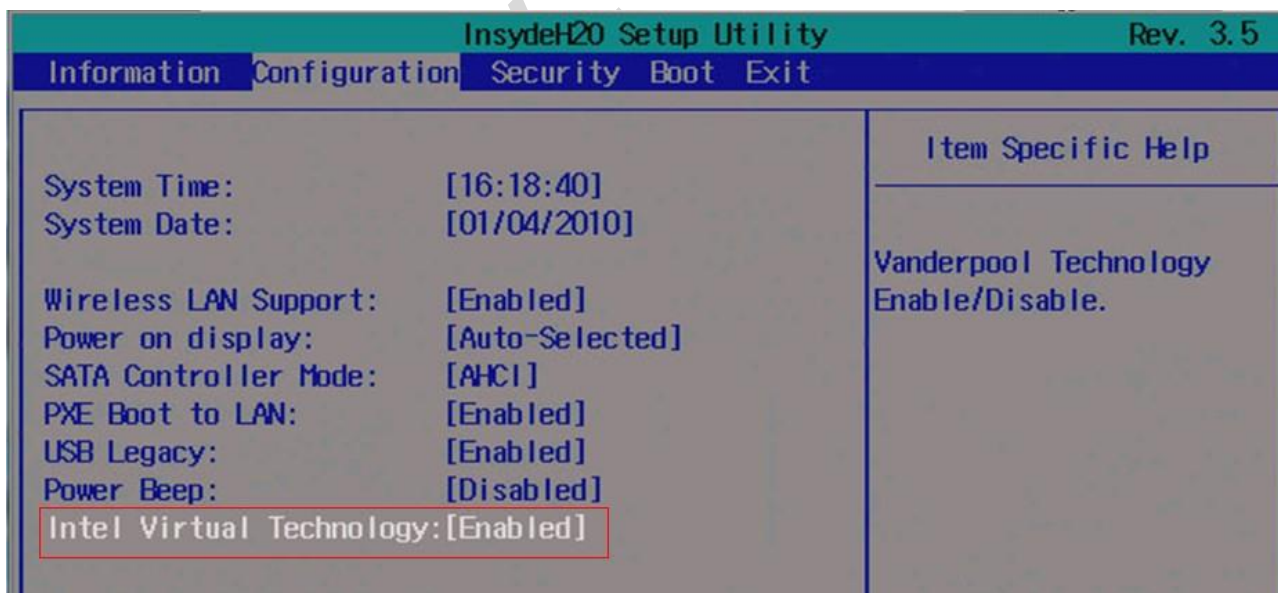
【检查主机 CPU 是否支持 Intel Virtualization Technology】

如下图所示，用户可以使用光盘目录下中的 CPU-Z 软件检查自己的 CPU 是否支持虚拟机化技术。



【在 BIOS 中打开 Intel Virtualization Technology】

如果 CPU 支持虚拟化技术，请在 BIOS 设置里面打开（不同型号的主板 BIOS 界面可能不同，如果找不到可以自行搜索“机型+BIOS+VT”的关键字）。





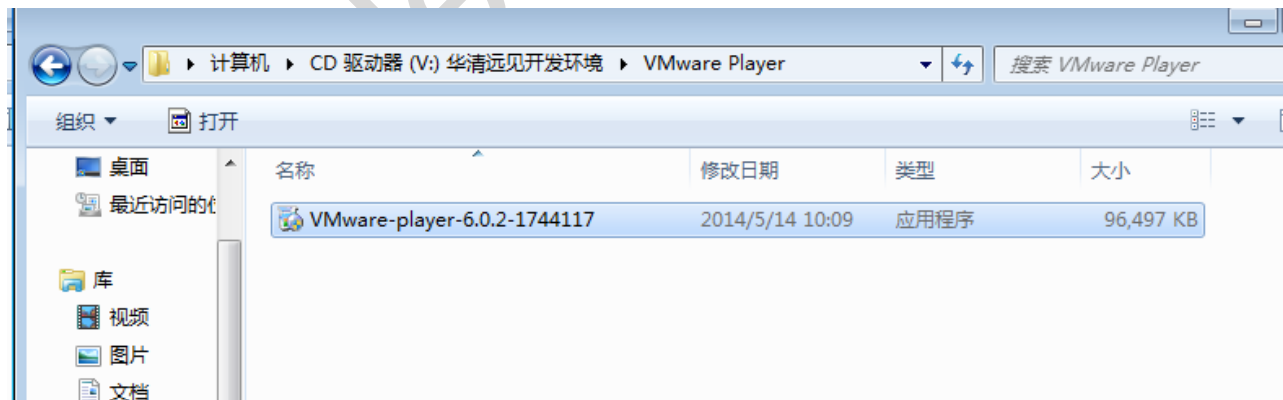
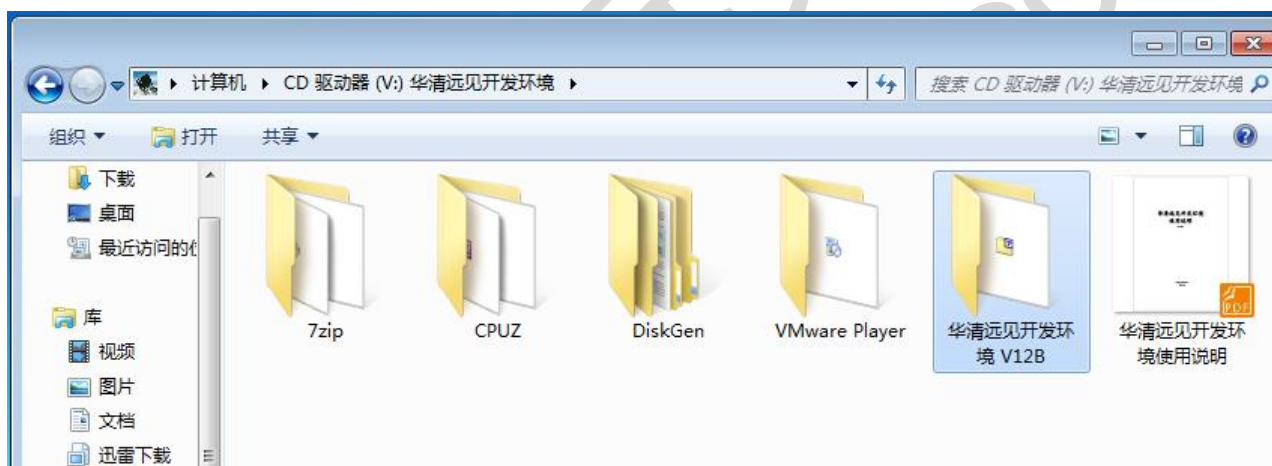
1.2 安装 VMware Player

VMware Player 从 6.0 版本之后默认支持中文，所以华清远见开发环境 V12B 使用当前最新版的 VMware Player（版本号为 6.0.2 build-1744117），如要正常使用此开发环境，必须保证 VMware Player 版本号大于等于当前给出的版本号，否则可能会出现因为 VMware Tools 版本过高引起虚拟机无法正常启动的情况。

（如果用户使用 VMware Workstation，版本号应该大于等于（10.0.1-1379776），否则可能会出现因为 VMware Tools 版本过高引起虚拟机无法正常启动的情况。）

【打开 VMware Player 安装程序】

打开【华清远见开发环境\VMware Player】目录下的 VMware Player 安装程序（VMware-player-6.0.2-1744117.exe）。





【VMware Player 安装程序初始化】

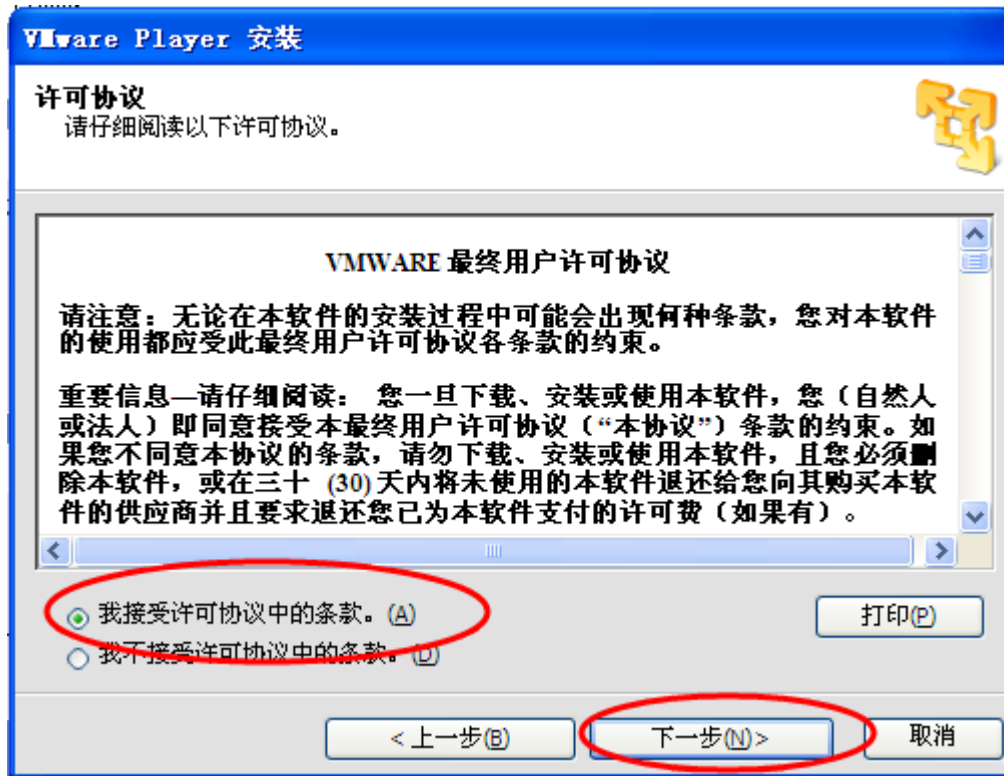


【VMware Player 安装向导】

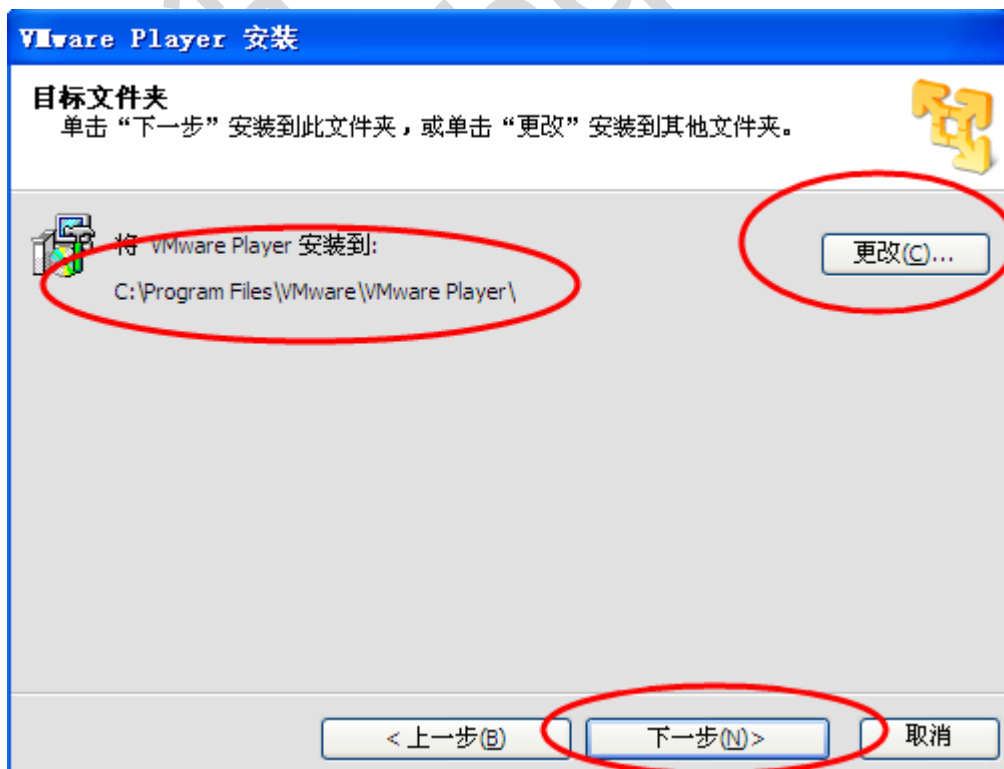




【VMware Player 许可协议】

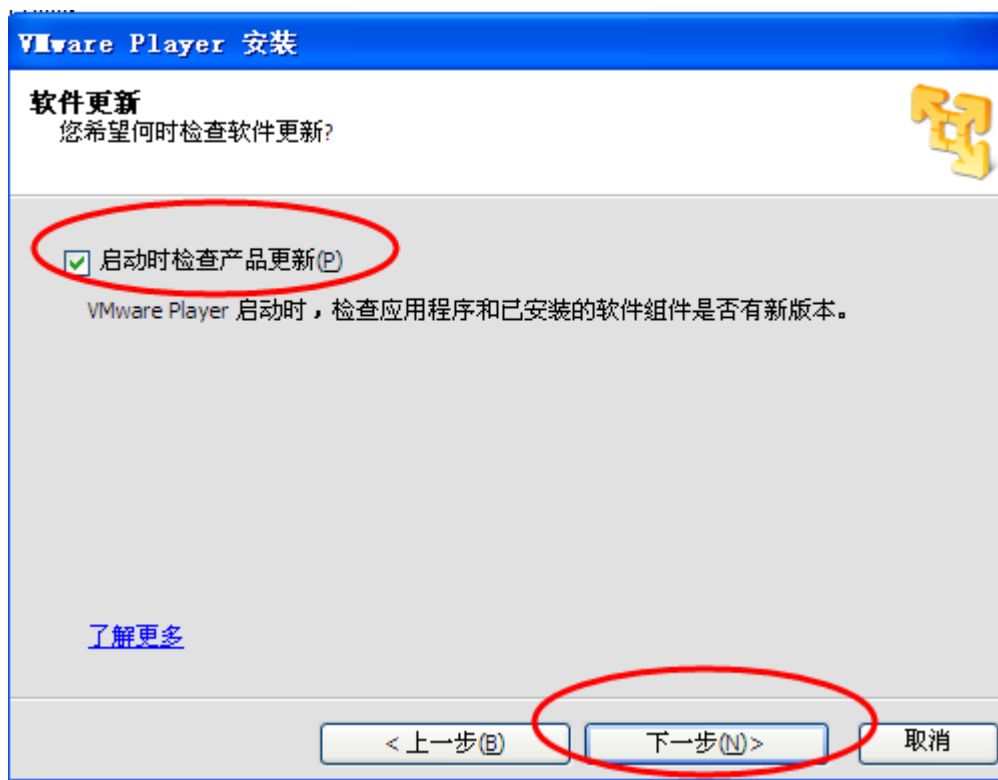


【更改安装路径】



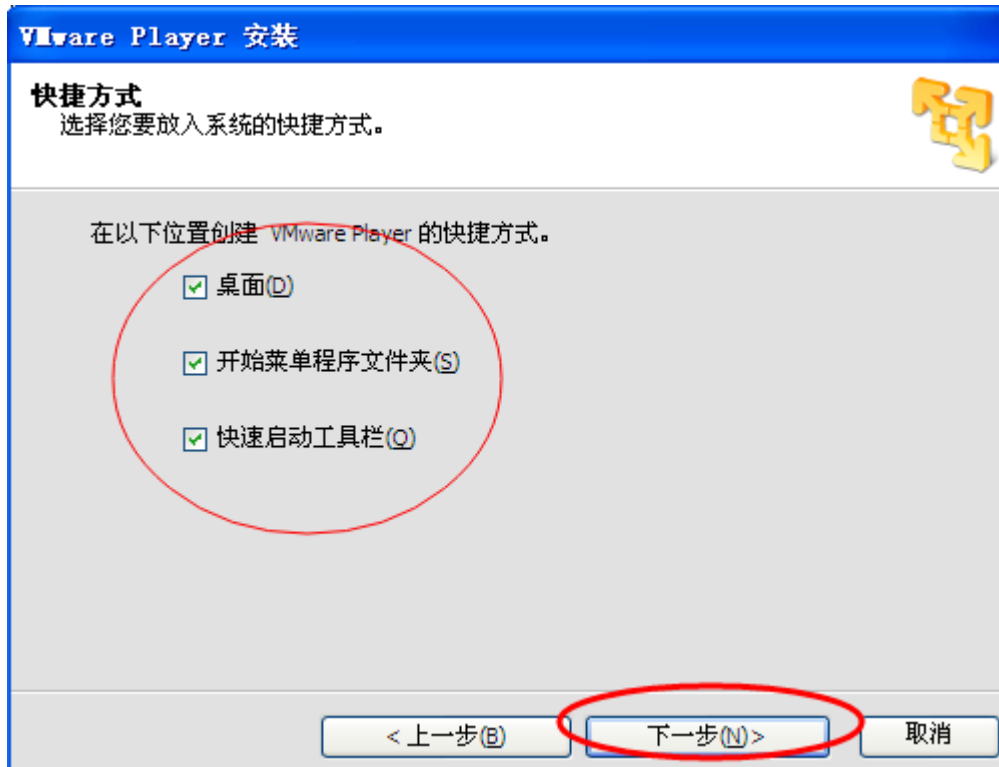


【选择是否检查更新，加入用户体验改进计划】





【创建快捷方式】

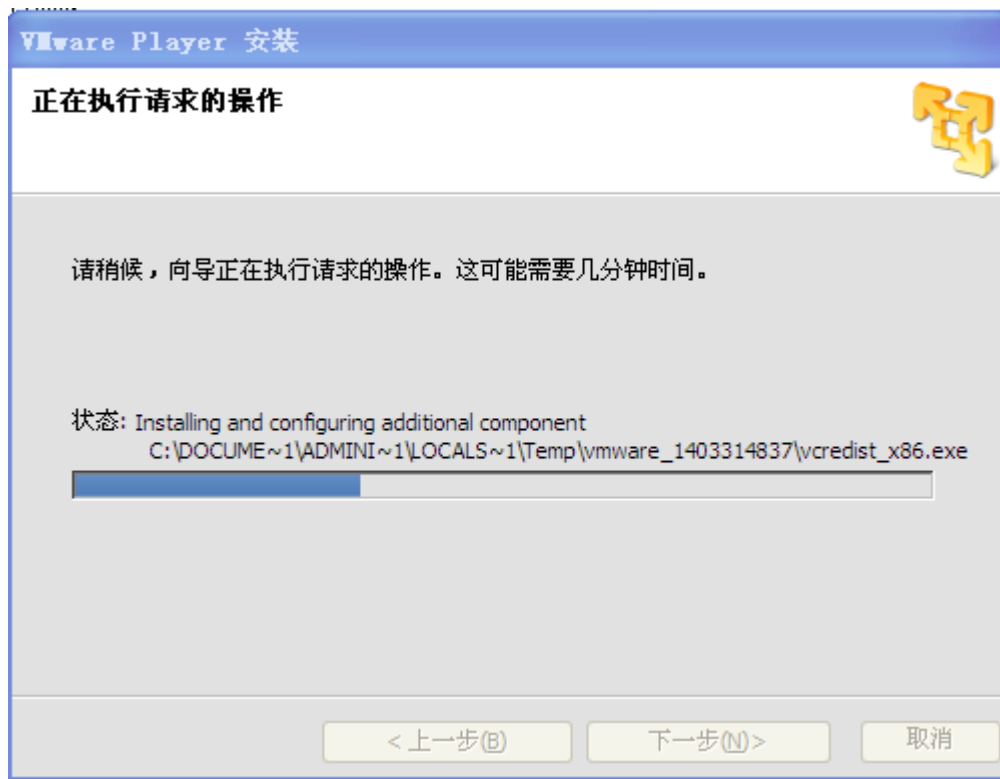


【确认安装】





【正在安装】



【安装完成】

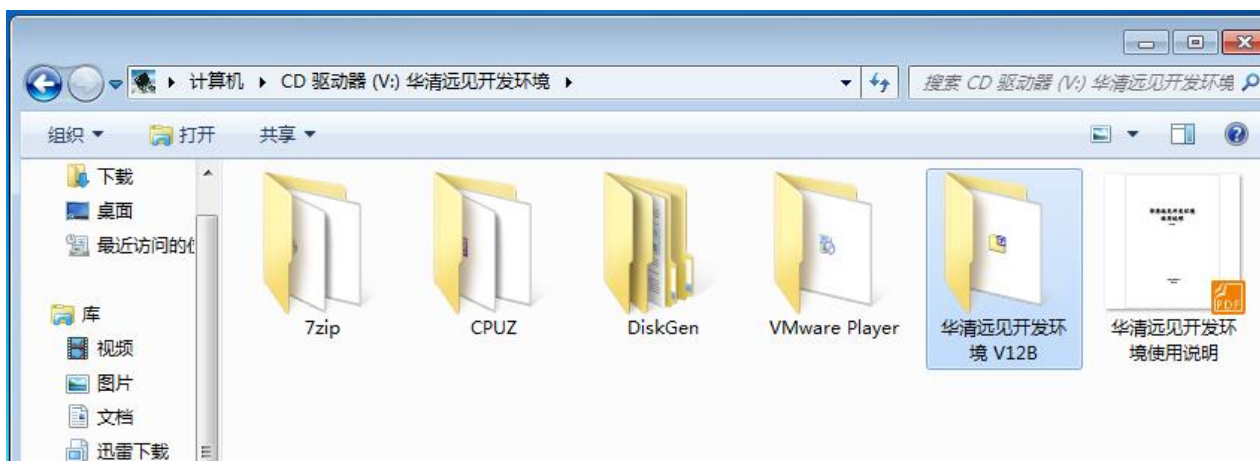




1.3 运行开发环境

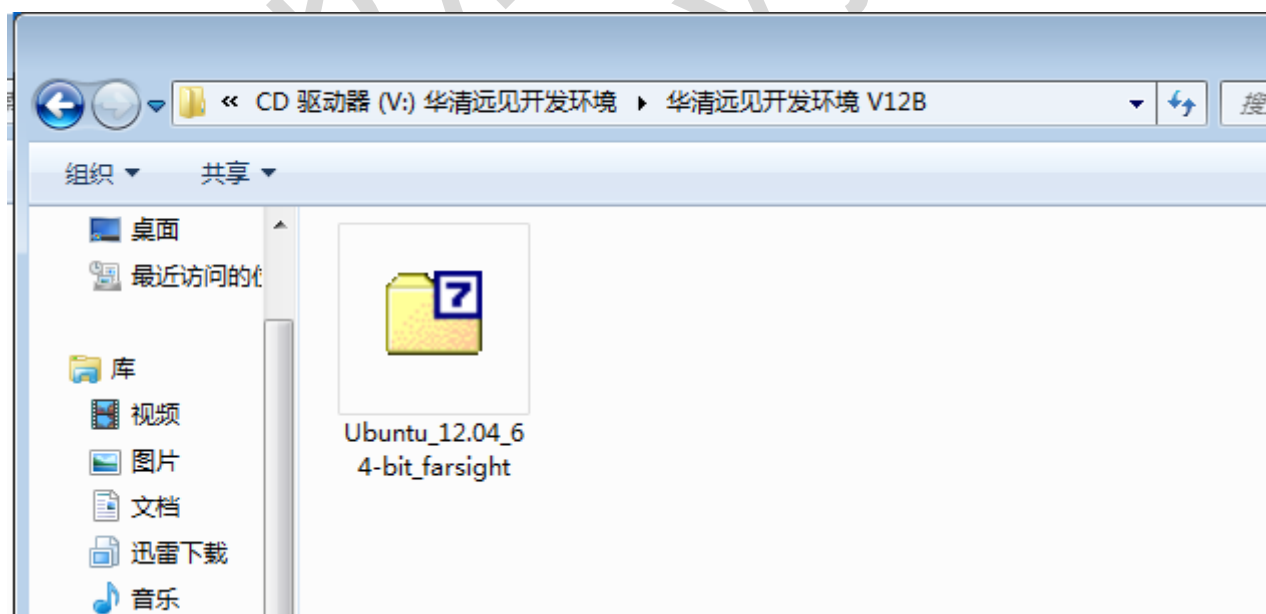
1.3.1 解压虚拟机镜像

打开光盘【华清远见开发环境 V12B】目录。

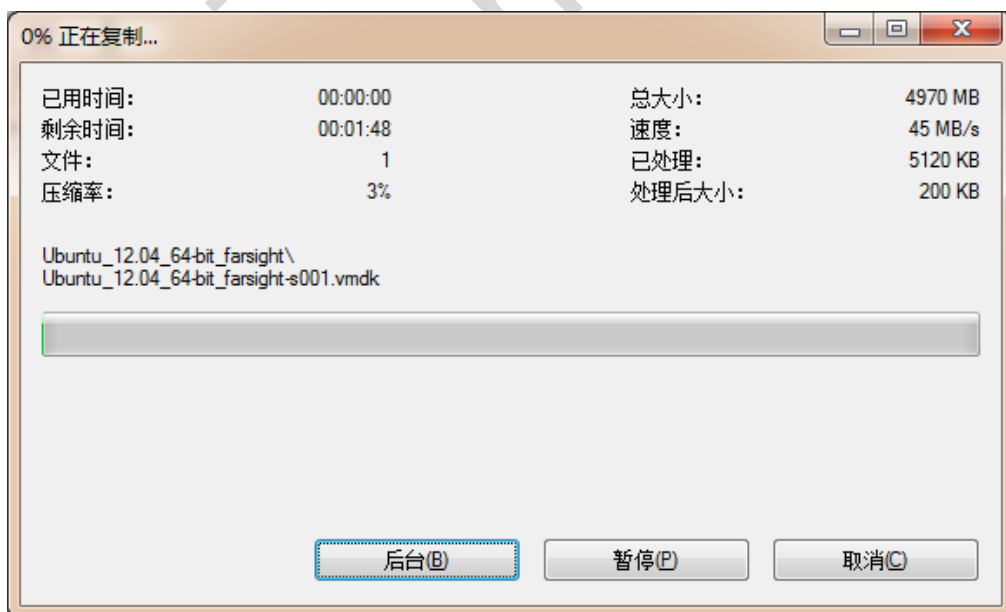
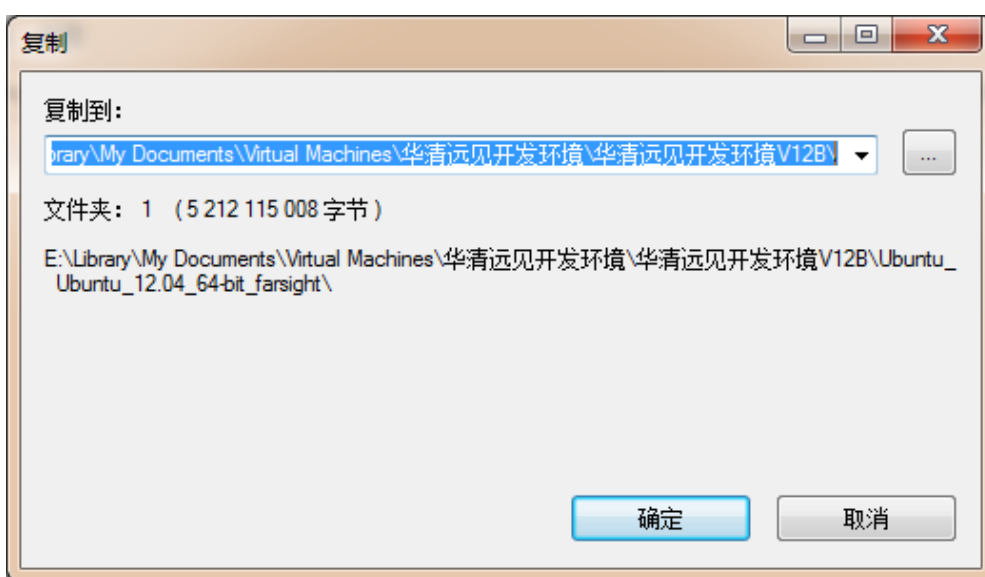
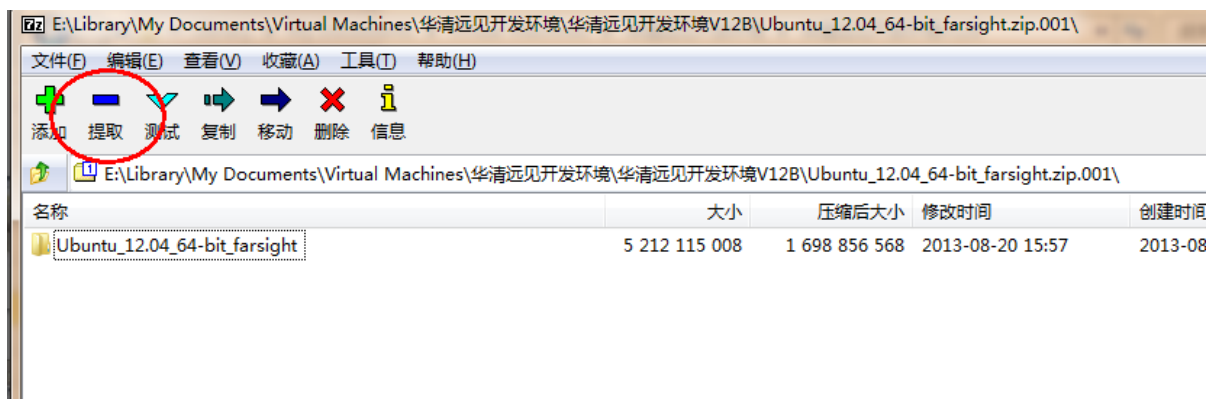


【打开华清远见开发环境压缩文件】

此处建议使用光盘下的 7zip 压缩软件解压。



【解压华清远见开发环境】



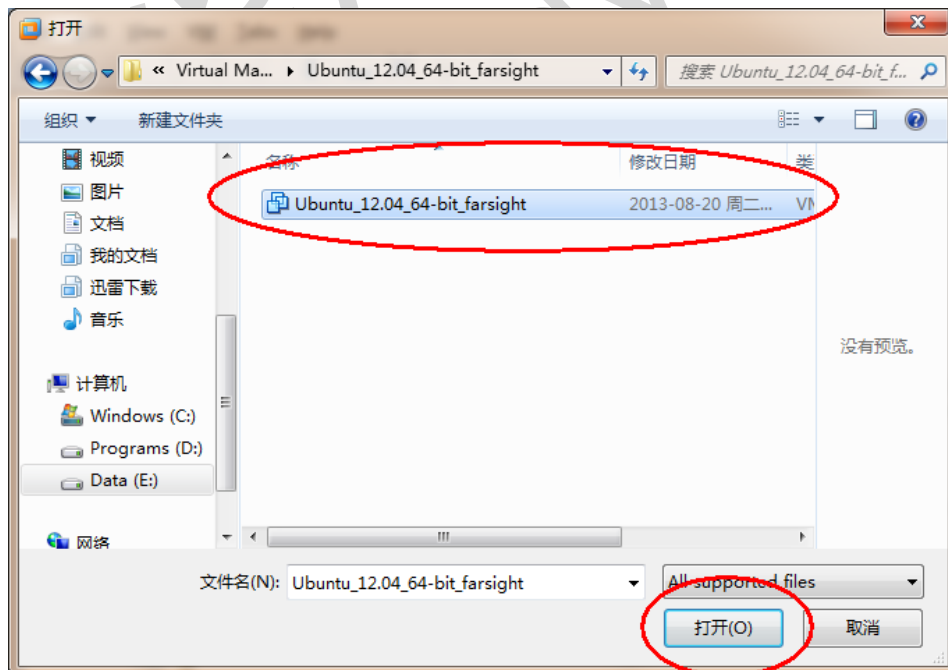
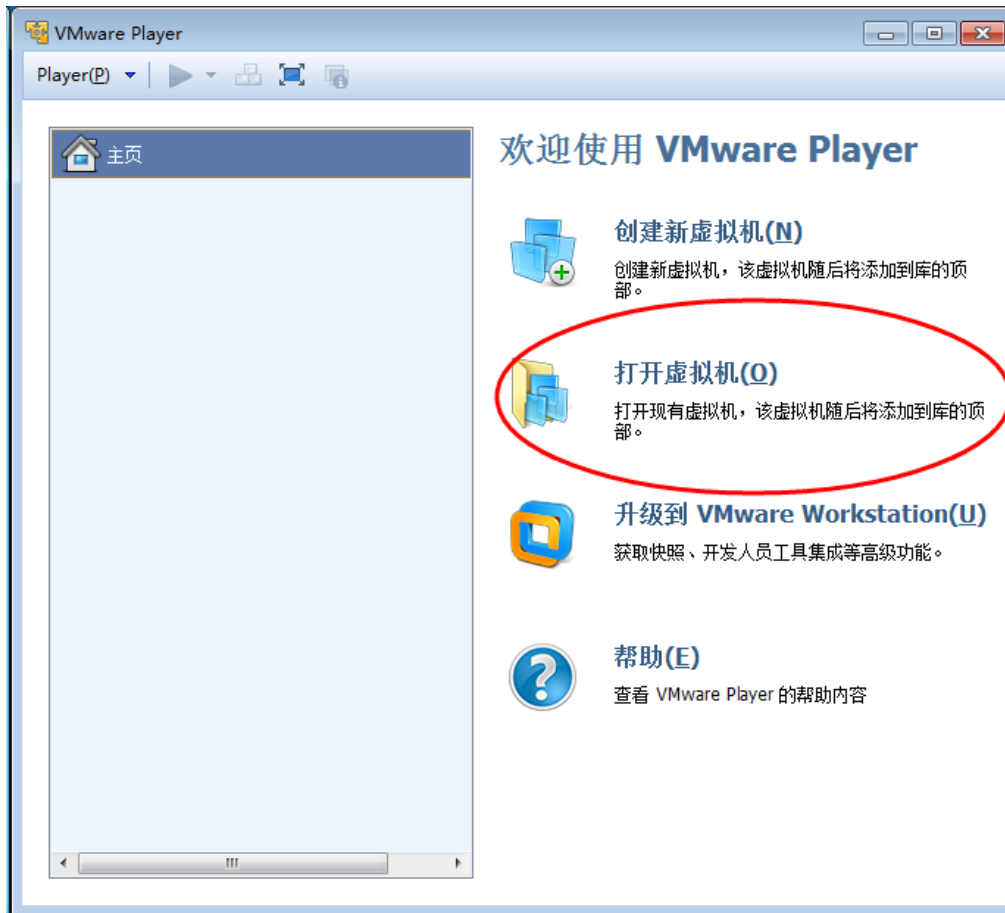


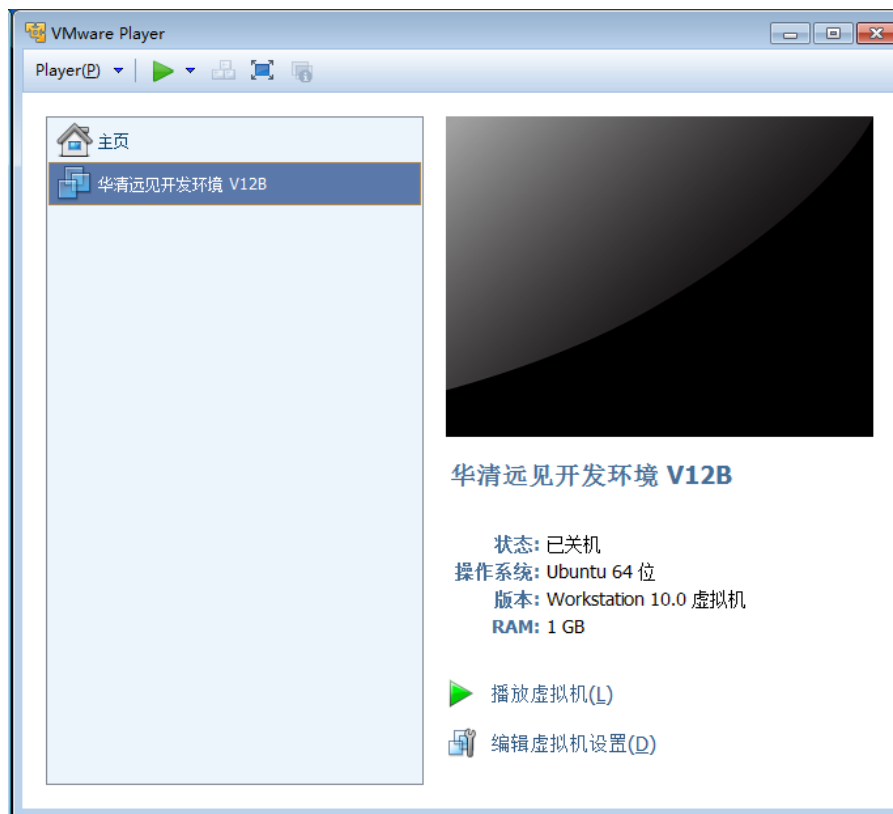
1.3.2 打开虚拟机

【打开 VMware Player】

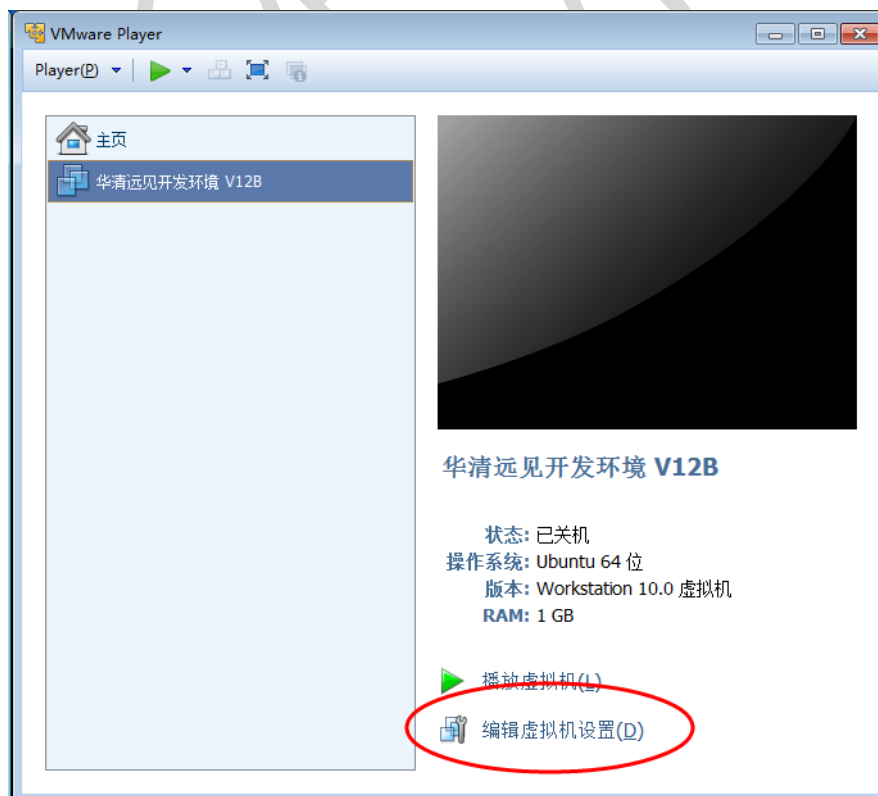


【打开虚拟机镜像】





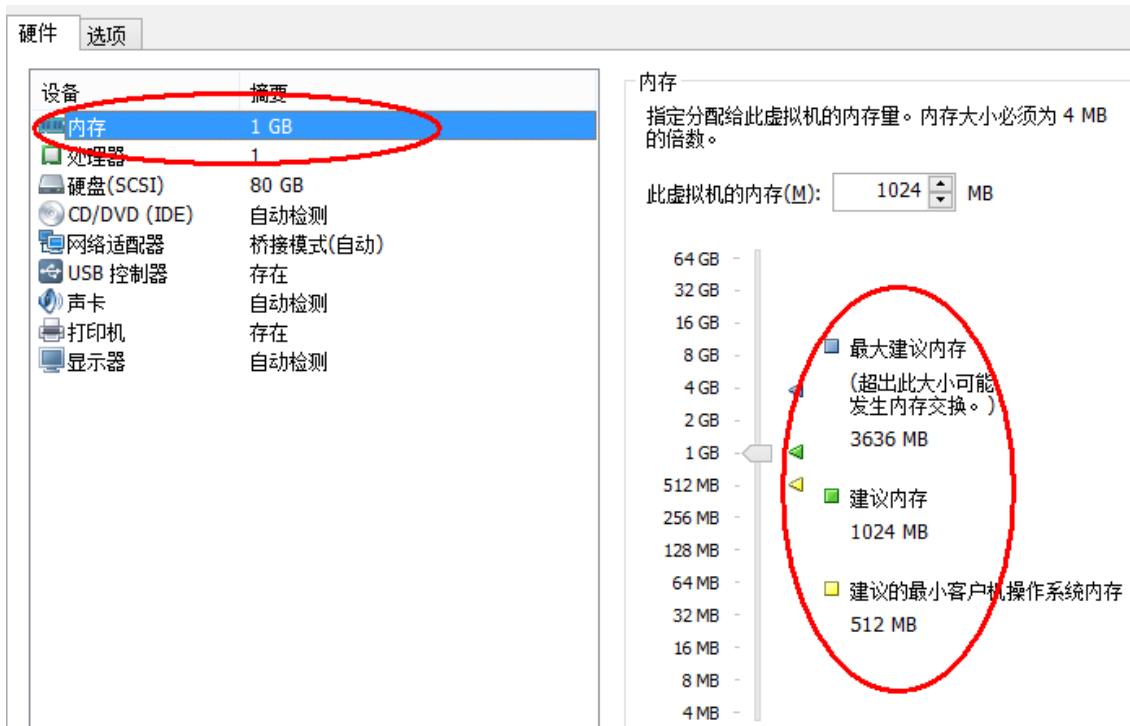
1.3.3 配置优化虚拟机





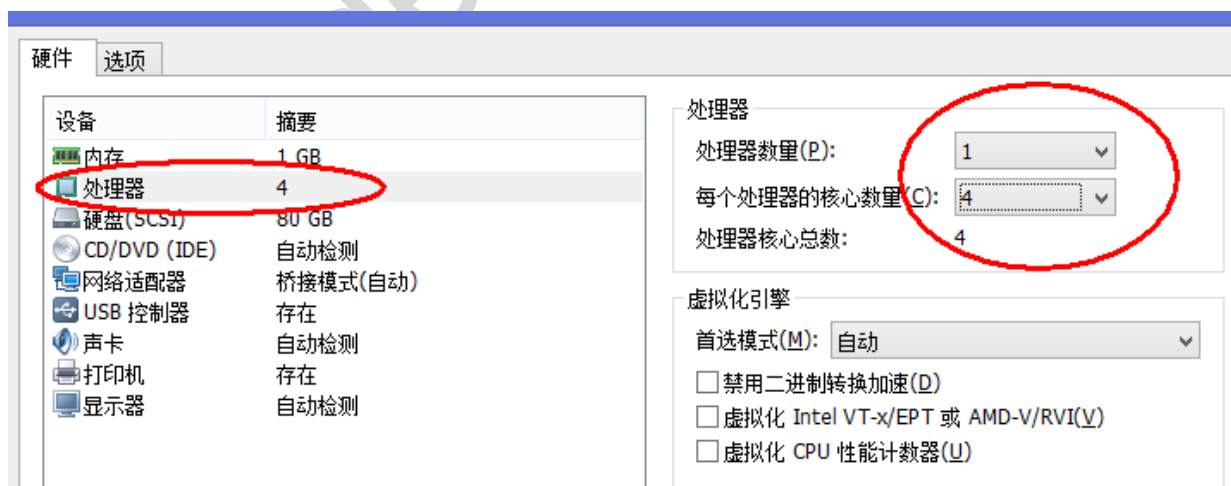
【增加内存大小】

根据主机配置修改虚拟机内存大小。例如主机内存 1G，那分配虚拟机的内存大小应该小于 512M，否则物理机操作系统运行会卡；如果主机内存大于 4G（足够大），那可以根据 VMware Player 的提示和自己的需求修改内存大小。**注意：如果需要编译 Android，那内存大小最好大于 1G。**



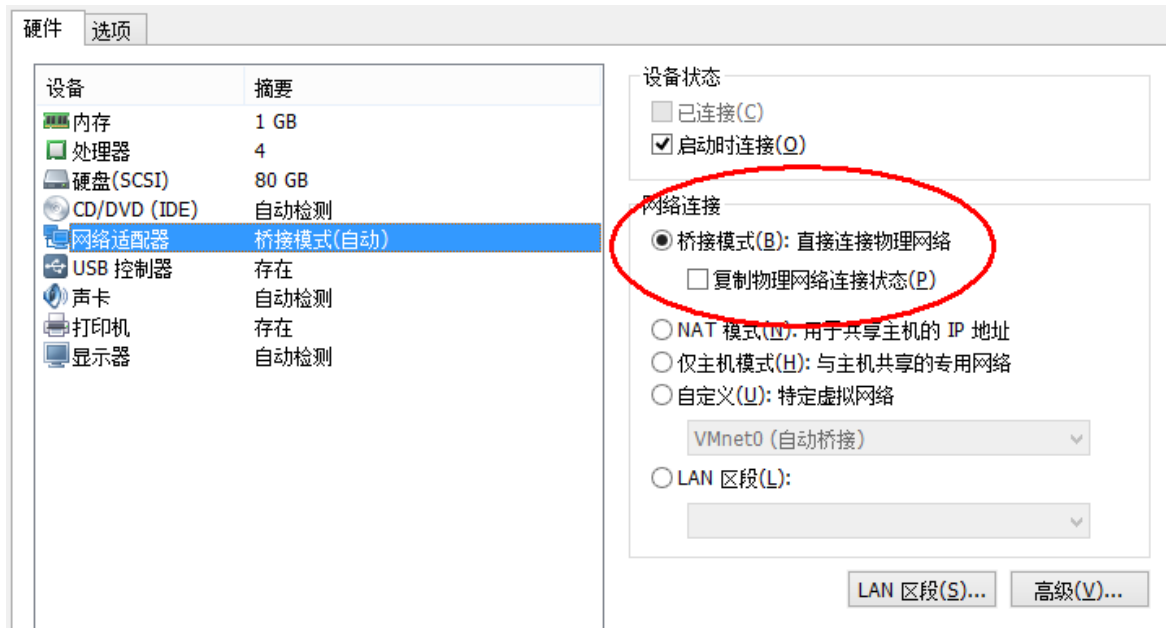
【修改 CPU 数量】

根据主机 CPU 配置修改虚拟机 CPU 数量。例如笔者 CPU 为 Intel Core-i3 M380（双核四线程），那处理器数量设置为 1,每个处理器的核心数量设置为 4。**注意：如果设置的总核心数不要超过 CPU 的核心数。**



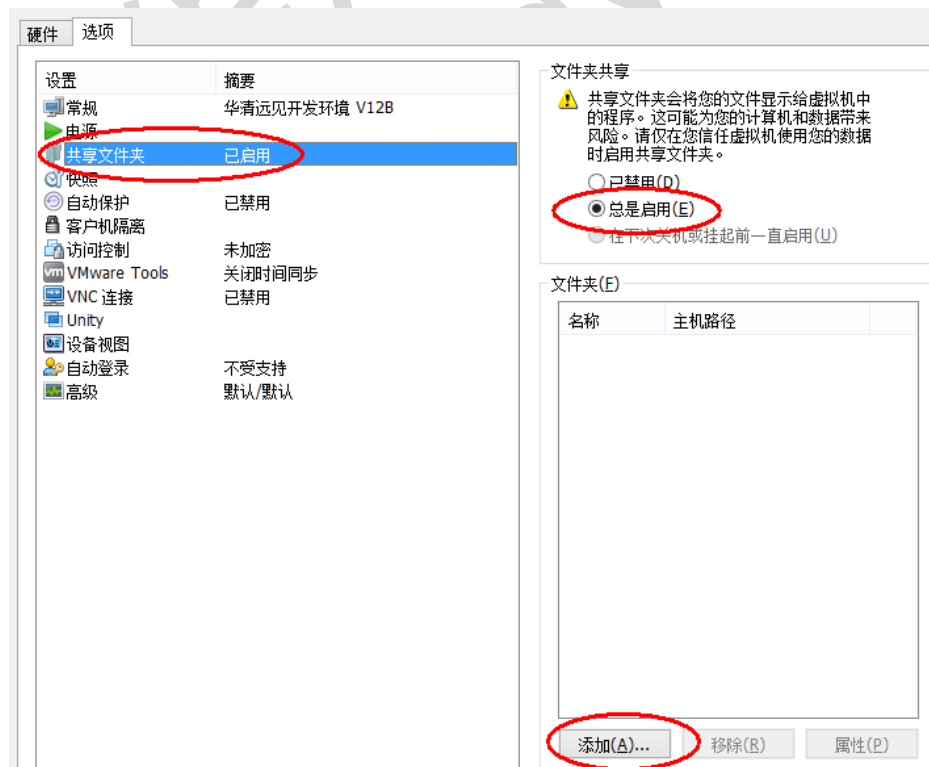


【确保网络连接为桥接模式】



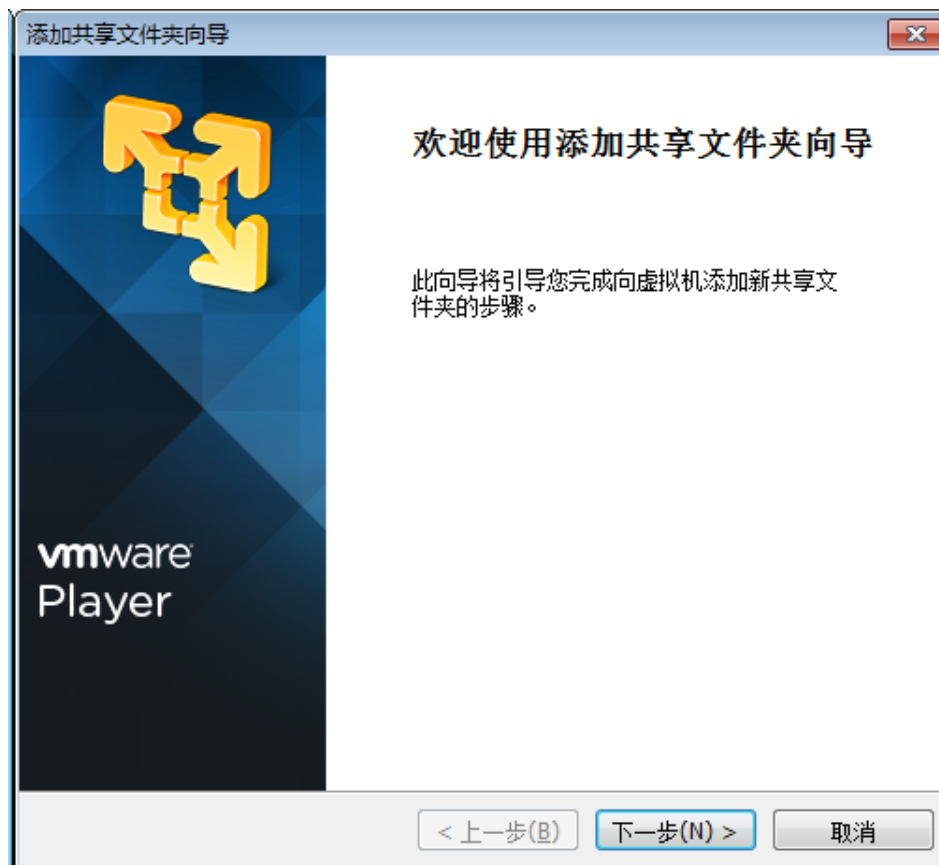
【增加共享目录】

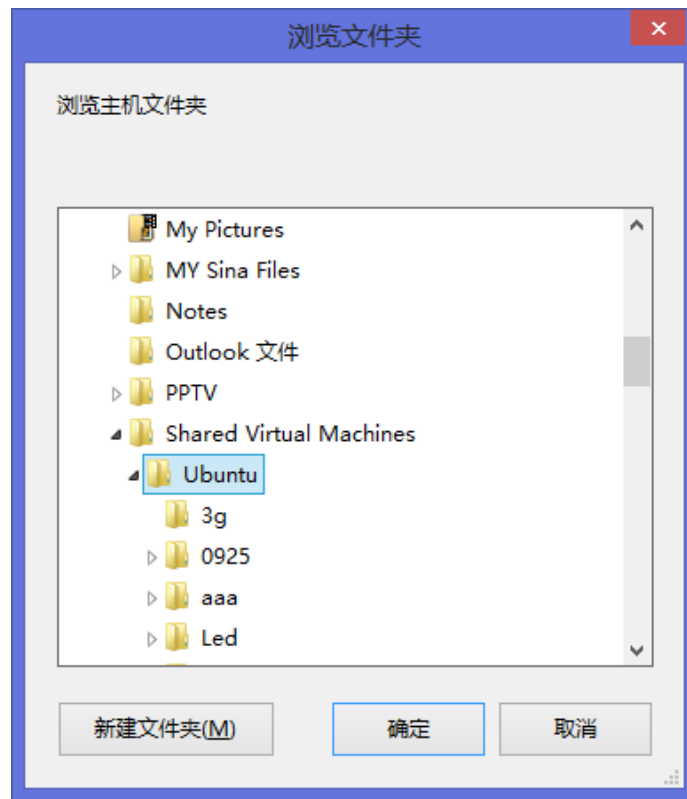
共享目录可以在虚拟机访问物理硬盘分区的内容，也可以将虚拟机里的文件拷贝至物理机，是虚拟机和物理机很好的交流桥梁。



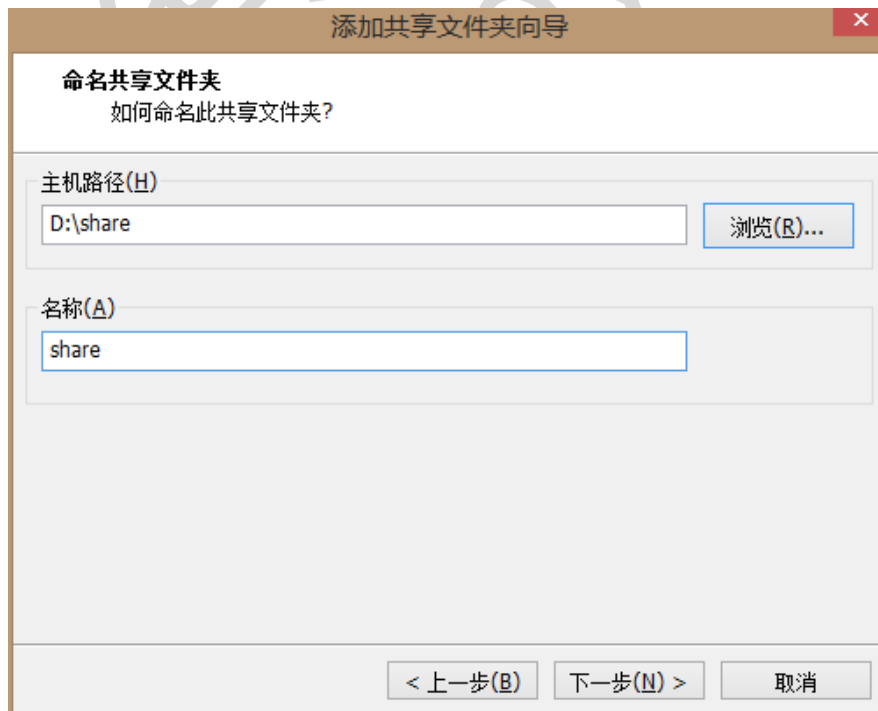


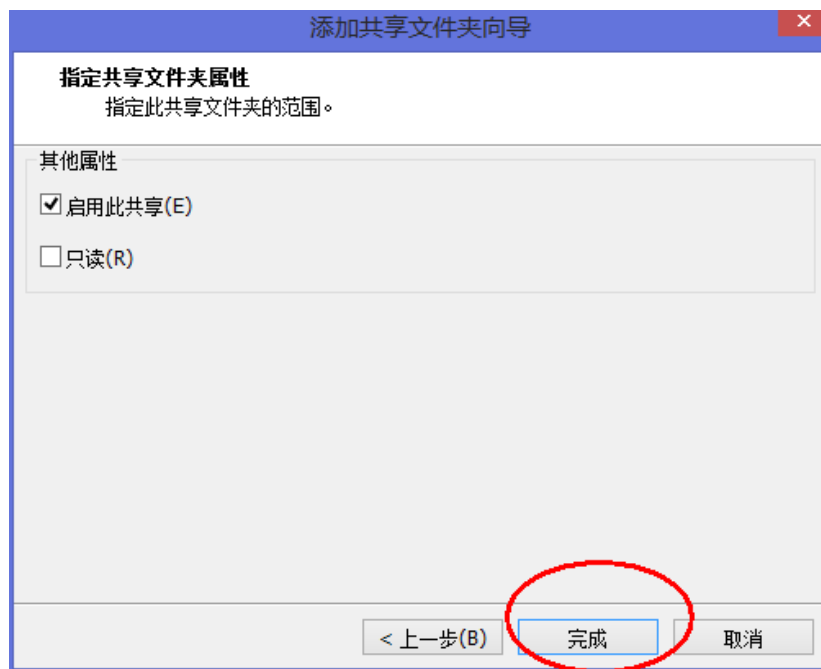
【点击上图“添加”】



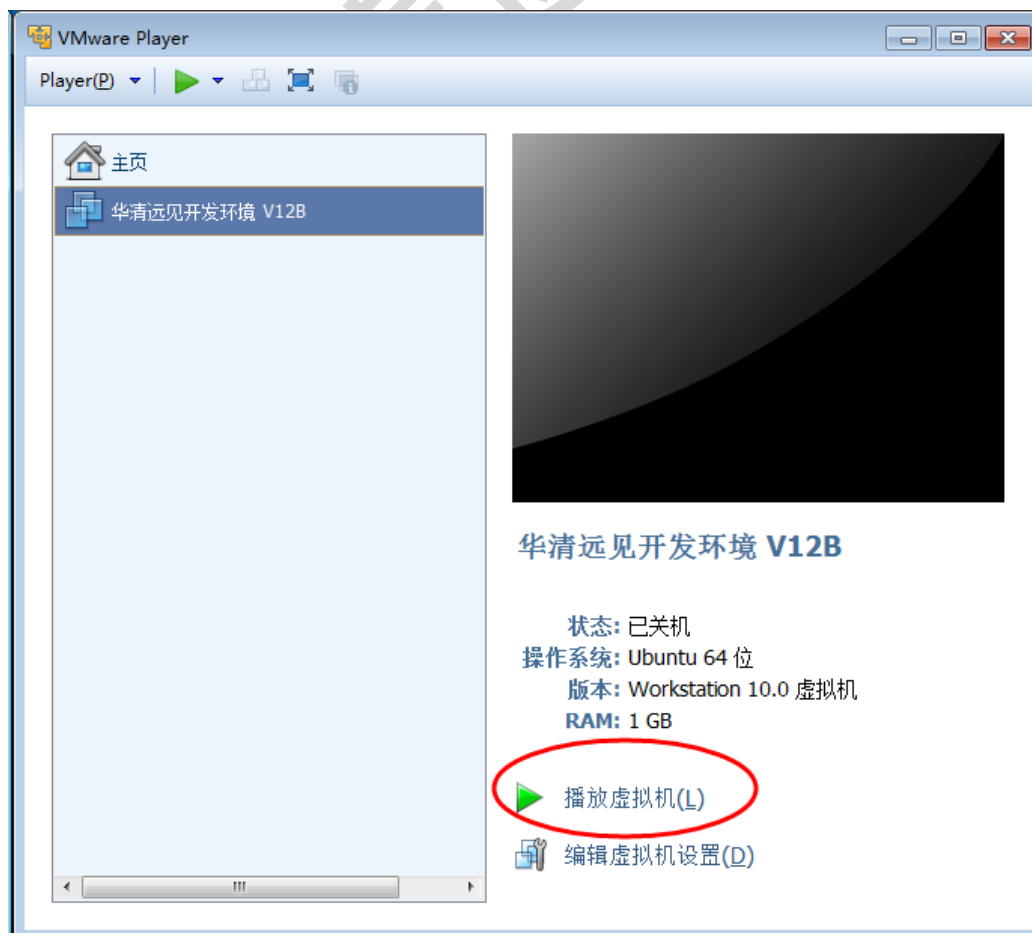


【修改在虚拟机内看到物理磁盘目录的名字】



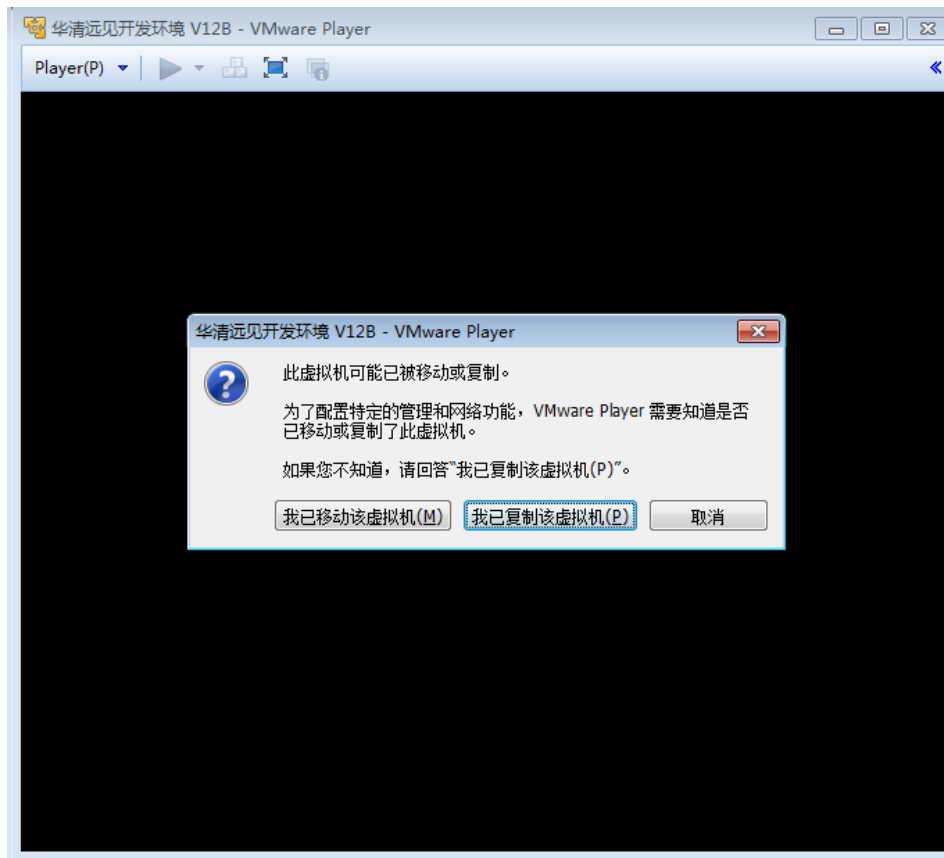


1.3.4 启动虚拟机

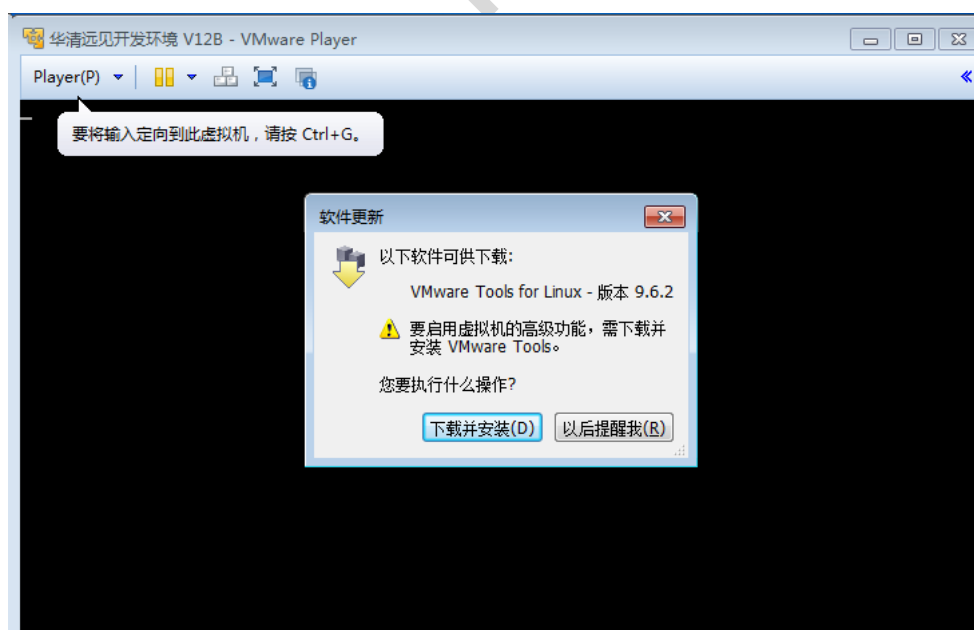


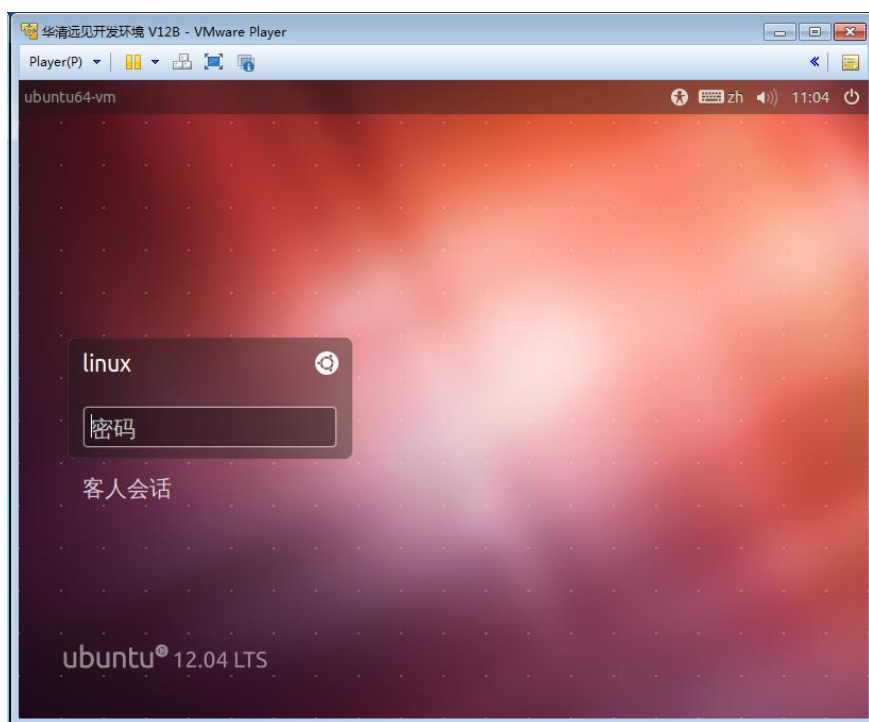


【启动虚拟机，选择“我已复制该虚拟机”】

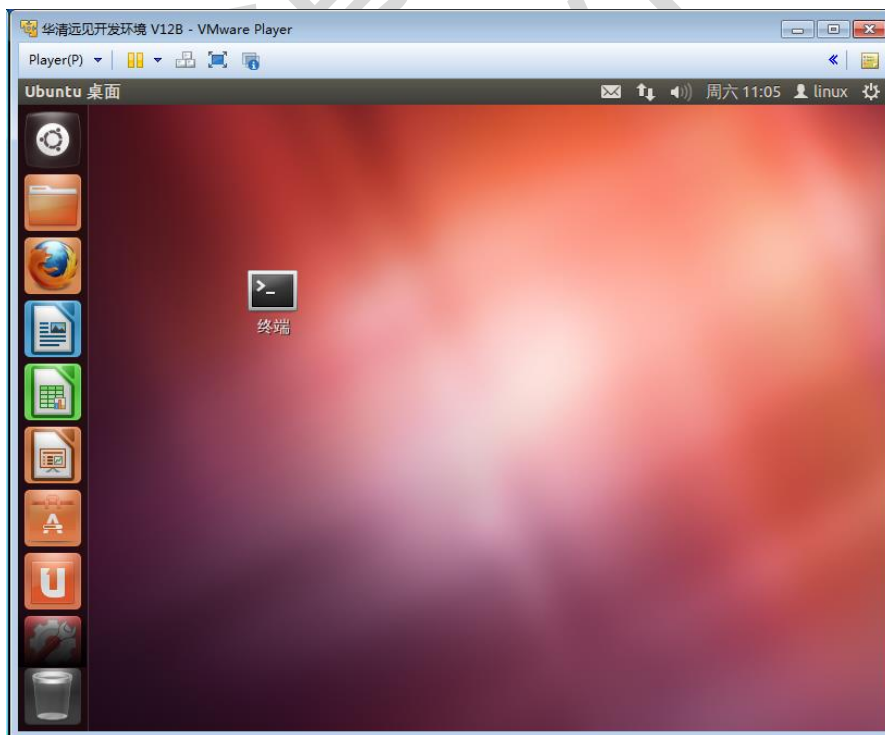


【在有网络的情况下，会提示有软件更新可以下载，可以选择下载，或者以后提醒】





【输入密码，密码为1】



嵌入式 Linux 开发环境至此搭建完毕。



1.3.5 设置 ROOT 密码

华清远见开发环境默认没有设置 Linux Root 管理员密码，**建议一般用户使用普通用户模式下操作，需要管理员操作时使用 `sudo` 命令获取临时管理员权限。**

设置 Linux Root 管理员密码步骤如下：

- 使用 `passwd` 命令生成 Root 密码（Linux 用户默认密码为 1）

```
$ sudo passwd
```

输入两次新的 UNIX 密码，即为 ROOT 用户密码

```
linux@ubuntu64-vm: ~  
linux@ubuntu64-vm:~$ ls  
examples.desktop  README  workdir  模板  图片  下载  桌面  
lx.leesheen@gmail.com  toolchain  公共的  视频  文档  音乐  
linux@ubuntu64-vm:~$  
linux@ubuntu64-vm:~$ sudo passwd  
[sudo] password for linux:  
输入新的 UNIX 密码 :  
重新输入新的 UNIX 密码 :  
passwd : 已成功更新密码  
linux@ubuntu64-vm:~$
```

- 使用 `su` 命令切换至 Root 用户模式下，键入使用 `passwd` 的密码

```
root@ubuntu64-vm: /home/linux  
linux@ubuntu64-vm:~$ ls  
examples.desktop  README  workdir  模板  图片  下载  桌面  
lx.leesheen@gmail.com  toolchain  公共的  视频  文档  音乐  
linux@ubuntu64-vm:~$  
linux@ubuntu64-vm:~$ sudo passwd  
[sudo] password for linux:  
输入新的 UNIX 密码 :  
重新输入新的 UNIX 密码 :  
passwd : 已成功更新密码  
linux@ubuntu64-vm:~$  
linux@ubuntu64-vm:~$ su  
密码 :  
root@ubuntu64-vm: /home/linux#
```




第 2 章 嵌入式 Linux 主机调试环境搭建

TFTP (Trivial File Transfer Protocol, 简单文件传输协议) 是 TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议, 常被用于开发测试使用。

NFS 方式是开发板通过 NFS 挂载放在主机 (PC) 上的根文件系统。此时在主机在文件系统中进行的操作同步反映在开发板上; 反之, 在开发板上进行的操作同步反映在主机中的根文件系统上。

实际工作中, 我们经常使用 TFTP 方式来调试内核, NFS 方式挂载文件系统, 这种方式对于系统的调试非常方便。

2.1 Linux 系统配置 TFTP 实验

2.1.1 实验目的

熟悉 Linux TFTP 配置, 为后续 Linux 底层开发做准备 (后面会用 tftp 从宿主机传输镜像到 FS_4412 开发板)。

2.1.2 实验平台

华清远见开发环境。

2.1.3 实验原理

TFTP 协议是简单文件传输协议, 基于 UDP 协议, 没有文件管理、用户控制功能。TFTP 分为服务器端程序和客户端程序, 在主机上通常同时配置有 TFTP 服务端和客户端。

2.1.4 实验步骤

打开虚拟机, 运行 Ubuntu 12.04 系统, 打开命令行终端 (系统桌面上默认有)

华清远见开发环境中已经包含 tftp 服务, 安装不必操作。可以进行此实验的测试部分。

```
$ cd /tftpboot  
$ ls
```

```
linux@ubuntu64-vm:~$ cd /tftpboot/  
linux@ubuntu64-vm:/tftpboot$ ls  
test  
linux@ubuntu64-vm:/tftpboot$ cat test  
  
this is a test file!  
  
linux@ubuntu64-vm:/tftpboot$
```

回到家目录。



\$ cd ~ //Linux 下波浪线【~】代表用户的 home 目录，我们俗称主目录或者家目录

```
$ tftp 127.0.0.1
> get test
```

```
linux@ubuntu64-vm:/tftpboot$ cd ~
linux@ubuntu64-vm:~$ tftp 127.0.0.1
tftp> get test
Received 26 bytes in 0.0 seconds
tftp> q
linux@ubuntu64-vm:~$ ls
examples.desktop  README  workdir  模板  图片  下载  桌面
lx.leesheen@gmail.com  test  公共的  视频  文档  音乐
linux@ubuntu64-vm:~$ cat test

this is a test file!

linux@ubuntu64-vm:~$
```

如上图所示，没有出现错误代码，且在家目录（/home/linux）下出现 test 文件，内容相同，则证明 tftp 服务建立成功。

2.2 Linux 系统配置 NFS 实验

2.2.1 实验目的

通过实验熟悉 Linux NFS 文件系统的配置过程，为后续 Linux 底层开发实验做准备。

2.2.2 实验平台

华清远见开发环境。

2.2.3 实验原理

NFS 方式是开发板通过 NFS 挂载放在主机（PC）上的根文件系统。此时在主机在文件系统中进行的操作同步反映在开发板上；反之，在开发板上进行的操作同步反映在主机中的根文件系统中。实际工作中，我们经常使用 NFS 方式挂载系统，这种方式对于系统的调试非常方便。详见“第 1 章搭建嵌入式 Linux 开发环境-主机交叉开发环境配置”。

2.2.4 实验步骤

打开虚拟机，运行 Ubuntu 12.04 系统，打开命令行终端。

配置/etc/exports 说明：（sudo 获取权限。输入密码，默认为 1；如不会使用 vim，命令行 vim 字段用 gedit 代替，下面省略此说明。）

```
$ sudo vim /etc/exports
```



NFS 允许挂载的目录及权限在文件/etc/exports 中进行了定义。例如，我们要将/source/rootfs 目录共享出来，那么我们需要在/etc/exports 文件末尾添加如下一行：

```
/source/rootfs *(rw,sync,no_root_squash,no_subtree_check)
```

```
linux@ubuntu64-vm: ~
1 # /etc/exports: the access control list for filesystems which may be exported
2 #   to NFS clients.  See exports(5).
3 #
4 # Example for NFSv2 and NFSv3:
5 # /srv/homes hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_
6 #
7 # Example for NFSv4:
8 # /srv/nfs4 gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
9 # /srv/nfs4/homes gss/krb5i(rw,sync,no_subtree_check)
10 #
11
12 /source/rootfs *(rw,sync,no_root_squash,no_subtree_check)
13
```

其中：/source/rootfs 是要共享的目录，*代表允许所有的网络段访问，rw 是可读写权限,sync 是资料同步写入内存和硬盘，no_root_squash 是 NFS 客户端分享目录使用者的权限，如果客户端使用的是 root 用户，那么对于该共享目录而言，该客户端就具有 root 权限。

重启服务：

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

重启服务成功如下显示。（如果设置的路径没有相应内容，会提示错误，可以先忽略这个问题）

```
linux@ubuntu64-vm:~$ sudo /etc/init.d/nfs-kernel-server restart
* Stopping NFS kernel daemon [ OK ]
* Unexporting directories for NFS kernel daemon... [ OK ]
* Exporting directories for NFS kernel daemon...
exportfs: Failed to stat /source/rootfs: No such file or directory [ OK ]
* Starting NFS kernel daemon [ OK ]
linux@ubuntu64-vm:~$
```



第 3 章 交叉开发环境搭建

所谓交叉开发是指先在一台通用 PC 上进行软件的编辑、编译与连接，然后下载到嵌入式设备中运行调试的开发过程。通用 PC 成为宿主机，嵌入式设备成为目标机。

3.1 实验目的

熟悉嵌入式 Linux 交叉开发环境的搭建与使用。

3.2 实验平台

华清远见开发环境；FS4412 开发板

3.3 实验原理

使用 tftp 的方式下载内核，运行到开发板上；使用 nfs 方式挂载文件系统，为后续的开发做准备。

3.4 实验步骤

3.4.1 配置开发环境网络

虚拟机网络方式为桥接模式，此状态下虚拟机下的操作系统和主机操作系统为平级状态。为了调试方便，我们可以给虚拟机下的 Ubuntu 一个静态的 IP 地址。

假设我们使用的网络地址为 192.168.100.x 段的，那么我可以给 Ubuntu 分配一个 IP 为 192.168.100.192，配置过程如下所示。

配置虚拟机网络环境。

```
$ sudo vim /etc/network/interfaces
```

修改文件如下图所示。保存退出（环境默认是加【#】注释掉的内容，修改前删除 4-11 行每行的注释）。

```
1 auto lo
2 iface lo inet loopback
3
4 auto eth0
5 iface eth0 inet static
6 address 192.168.100.192
7 netmask 255.255.255.0
8 gateway 192.168.100.1
9 network 192.168.100.0
10 broadcast 192.168.100.255
11 dns-nameservers 192.168.100.1
12
```

应用网络修改。

```
$ sudo /etc/init.d/networking restart
```



```
[sudo] password for linux:
linux@ubuntu64-vm:~$ sudo /etc/init.d/networking restart
* Running /etc/init.d/networking restart is deprecated because it may not enable
  again some interfaces
* Reconfiguring network interfaces...
ssh stop/waiting
ssh start/running, process 3252
[ OK ]
linux@ubuntu64-vm:~$
```

如上图所示表明 IP 修改成功。使用【ifconfig】命令查看我们修改的结果。如果没有修改成功，重复上述步骤，或者重新启动虚拟机 Ubuntu 系统即可。

```
linux@ubuntu64-vm:~$ ifconfig
eth0      Link encap:以太网 硬件地址 00:0c:29:eb:c8:ce
          inet 地址:192.168.100.192 广播:192.168.100.255 掩码:255.255.255.0
          inet6 地址: fe80::20c:29ff:feeb:c8ce/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:708 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:256 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:90508 (90.5 KB)  发送字节:28463 (28.4 KB)

lo        Link encap:本地环回
          inet 地址:127.0.0.1 掩码:255.0.0.0
          inet6 地址: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  跃点数:1
          接收数据包:186 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:186 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:17566 (17.5 KB)  发送字节:17566 (17.5 KB)

linux@ubuntu64-vm:~$
```

3.4.2 配置交叉工具链

华清远见开发环境包含了 3 个版本的交叉工具链，路径在/usr/local/toolchain/下（旧版本在/home/linux/toolchain/下），不必再次解压，直接指向路径即可。下列步骤均按照新版本的路径介绍，旧版本的用户可以选择更新或者自行修改路径即可。

一般情况下，当我们输入一个 Linux 命令，比如【ls】可以直接执行功能，但我当前的目录并没有【ls】这个文件。【ls】这个命令文件在【/bin】这个目录中，我们可以执行这是因为这些命令所在的路径包含在了用户的环境变量中。我们为了使用方便也将我们经常使用的交叉工具链添加到环境变量中。

```
linux@ubuntu64-vm:~$ cd /usr/local/toolchain/
linux@ubuntu64-vm:/usr/local/toolchain$ ls
toolchain-4.3.2  toolchain-4.4.6  toolchain-4.5.1
linux@ubuntu64-vm:/usr/local/toolchain$
```

修改文件~/.bashrc，添加如下内容

```
$ vim /etc/bash.bashrc // 注意 bashrc 前面有句点
```

添加下面一行代码到文件的末尾。



```
export PATH=$PATH:/usr/local/toolchain/toolchain-4.4.6/bin/
```

```
55         return $?
56         elif [ -x /usr/share/command-not-found/command-not-found ]; then
57             /usr/bin/python /usr/share/command-not-found/command-not-found -- "$1"
58             return $?
59     else
60         printf "%s: command not found\n" "$1" >&2
61         return 127
62     fi
63 }
64 fi
65
66 export PATH=$PATH:/usr/local/toolchain/toolchain-4.6.4/bin/
/etc/bash.bashrc [R0]
```

重启配置文件

```
$ source /etc/bash.bashrc
```

工具链的测试

```
$ arm-none-linux-gnueabi-gcc -v
```

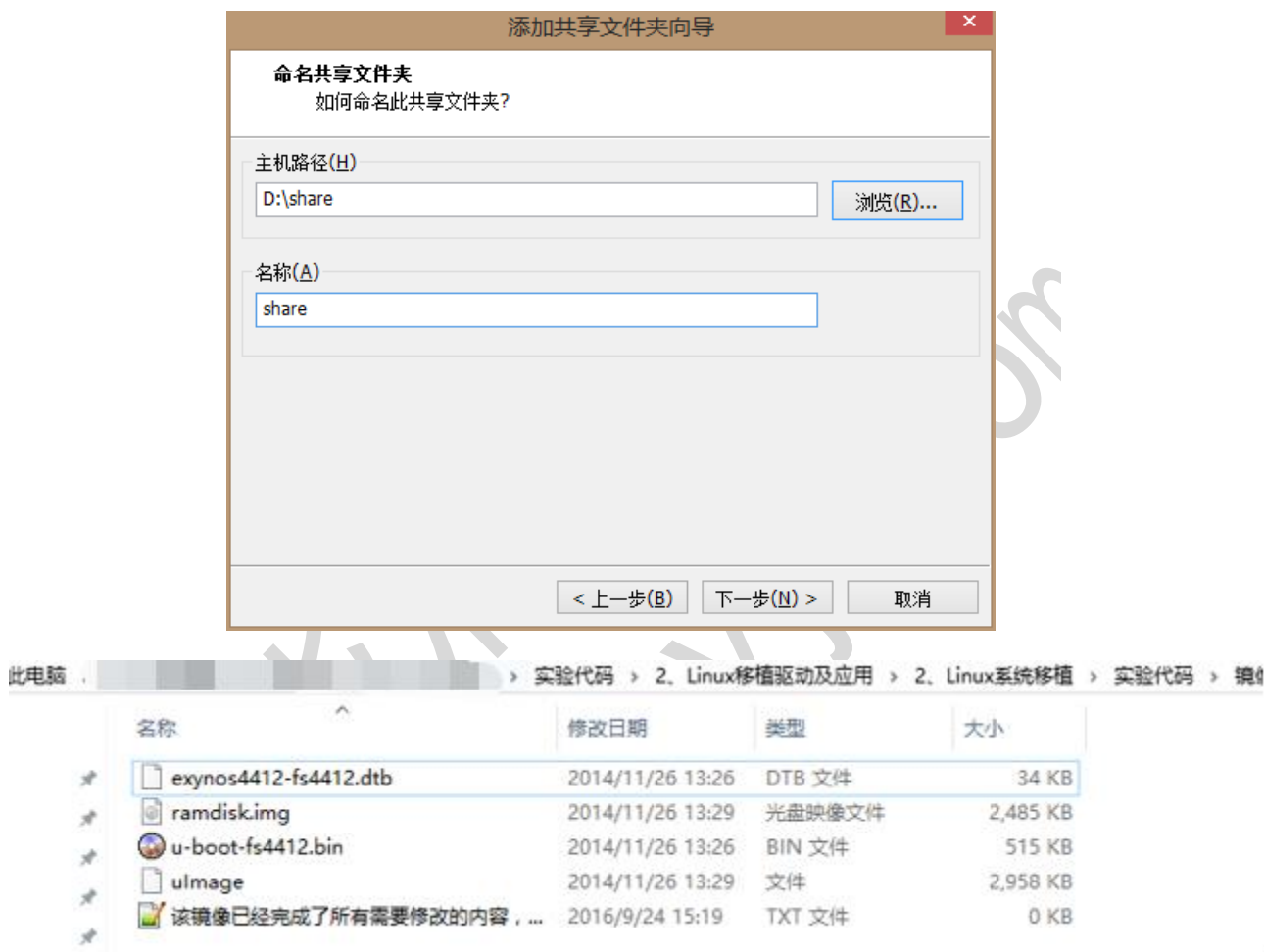
```
linux@ubuntu64-vm: ~
linux@ubuntu64-vm:~$ arm-none-linux-gnueabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-none-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/home/linux/toolchain/toolchain-4.6.4/bin/../libexec/gcc/arm-arm1176jzfssf-linux-gnueabi/4.6.4/lto-wrapper
Target: arm-arm1176jzfssf-linux-gnueabi
Configured with: /work/builddir/src/gcc-4.6.4/configure --build=i686-build_pc-linux-gnu --host=i686-build_pc-linux-gnu --target=arm-arm1176jzfssf-linux-gnueabi --prefix=/opt/TuxamitoSoftToolchains/arm-arm1176jzfssf-linux-gnueabi/gcc-4.6.4 --with-sysroot=/opt/TuxamitoSoftToolchains/arm-arm1176jzfssf-linux-gnueabi/gcc-4.6.4/arm-arm1176jzfssf-linux-gnueabi/sysroot --enable-languages=c,c++ --with-arch=armv6zk --with-cpu=arm1176jzf-s --with-tune=arm1176jzf-s --with-fpu=vfp --with-float=softfp --with-pkgversion='crosstool-NG hg+default-2685dfa9de14 - tc0002' --disable-sjlj-exceptions --enable-__cxa_atexit --disable-libmudflap --disable-libgomp --disable-libssp --disable-libquadmath --disable-libquadmath-support --with-gmp=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-mpfr=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-mpc=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-ppl=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-cloog=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-libelf=/work/builddir/arm-arm1176jzfssf-linux-gnueabi/buildtools --with-host-libstdcxx='-static-libgcc -Wl,-Bstatic,-lstdc++,-Bdynamic -lm' --enable-threads=posix --enable-target-optspace --without-long-double-128 --disable-nls --disable-multilib --with-local-prefix=/opt/TuxamitoSoftToolchains/arm-arm1176jzfssf-linux-gnueabi/gcc-4.6.4/arm-arm1176jzfssf-linux-gnueabi/sysroot --enable-c99 --enable-long-long
Thread model: posix
gcc version 4.6.4 (crosstool-NG hg+default-2685dfa9de14 - tc0002)
linux@ubuntu64-vm:~$
```



直接输入命令，即可执行，这样说明我们的交叉工具链就安装好了

3.4.3 拷贝文件

拷贝【华清远见-CORTEXA9 资料：\程序源码\Linux 移植实验源码\镜像】中的所有文件拷贝到 Ubuntu 共享目录【D:\share】下（1.3.3 节设置的位置）。通过共享目录



3.4.4 将共享目录中需要下载的文件拷贝到 tftp 目录中

拷贝 u-boot-fs4412.bin、uImage、exynos4412-fs4412.dtb 文件到虚拟机 Ubuntu 下的/tftpboot 目录下。

```
$ cp /mnt/hgfs/share/u-boot-fs4412.bin /mnt/hgfs/share/uImage /mnt/hgfs/share/exynos4412-fs4412.dtb /tftpboot/
```

```
linux@ubuntu64-vm:~$ cp /mnt/hgfs/share/u-boot-fs4412.bin /mnt/hgfs/share/uImage /mnt/hgfs/share/exynos4412-fs4412.dtb /tftpboot/
linux@ubuntu64-vm:~$ ls /tftpboot/
exynos4412-fs4412.dtb test u-boot-fs4412.bin uImage
linux@ubuntu64-vm:~$
```

3.4.5 解压文件系统

拷贝【华清远见-CORTEXA9 资料：\程序源码\Linux 移植实验源码\移植后的源码\rootfs.tar.xz】文件



到虚拟机 Ubuntu 下的/source 目录下。

```
$ cp /mnt/hgfs/share/rootfs.tar.xz /source/
```

```
linux@ubuntu64-vm:~$ cp /mnt/hgfs/share/rootfs.tar.xz /source/  
linux@ubuntu64-vm:~$ ls /source/  
rootfs.tar.xz  
linux@ubuntu64-vm:~$
```

```
$ cd /source
```

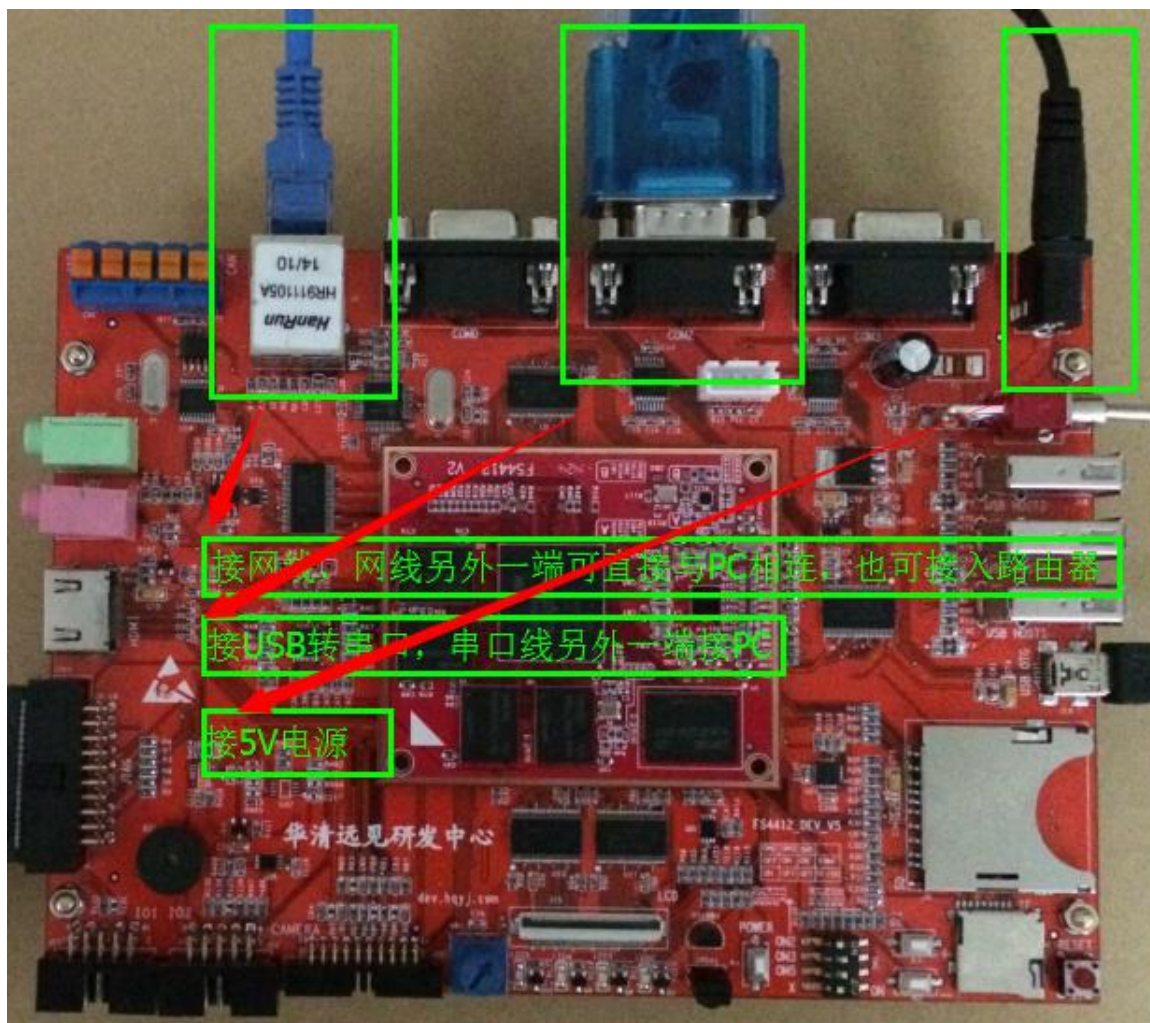
```
$ tar xvf rootfs.tar.xz
```

```
linux@ubuntu64-vm:~$ cd /source/  
linux@ubuntu64-vm:/source$ ls  
rootfs.tar.xz  
linux@ubuntu64-vm:/source$ tar xvf rootfs.tar.xz
```

```
rootfs/bin/ps  
rootfs/bin/busybox  
rootfs/bin/kill  
rootfs/bin/cp  
rootfs/bin/gunzip  
rootfs/bin/df  
rootfs/bin/sleep  
rootfs/bin/pwd  
rootfs/bin/adduser  
rootfs/bin/ttyhack  
rootfs/bin/getopt  
rootfs/bin/sed  
rootfs/bin/delgroup  
rootfs/bin/ionice  
rootfs/bin/chgrp  
rootfs/bin/chown  
rootfs/bin/watch  
rootfs/bin/ping6  
rootfs/bin/sync  
rootfs/bin/mt  
rootfs/bin/run-parts  
rootfs/bin/rpm  
rootfs/bin/su  
rootfs/bin/mount  
rootfs/bin/catv  
rootfs/bin/tar  
rootfs/proc/  
rootfs/sys/  
linux@ubuntu64-vm:/source$
```

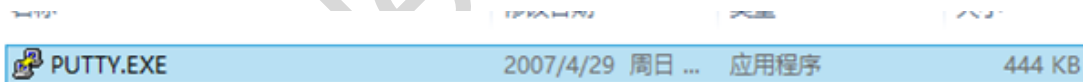


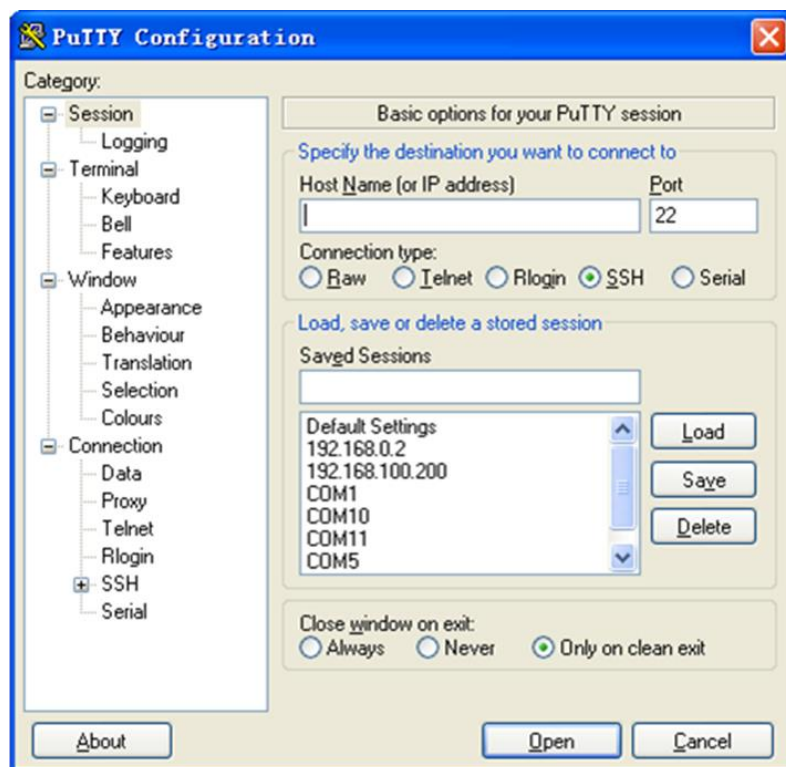

3.4.6 连接开发板



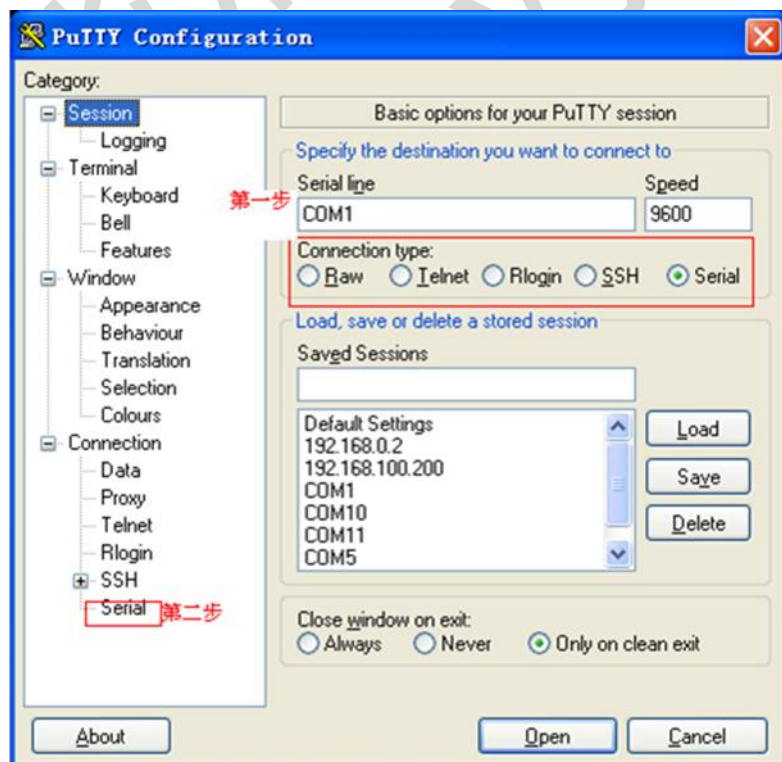
3.4.7 设置串口调试工具

打开【华清远见-CORTEXA9 资料\工具软件\Windows\串口调试工具\putty.exe】文件。

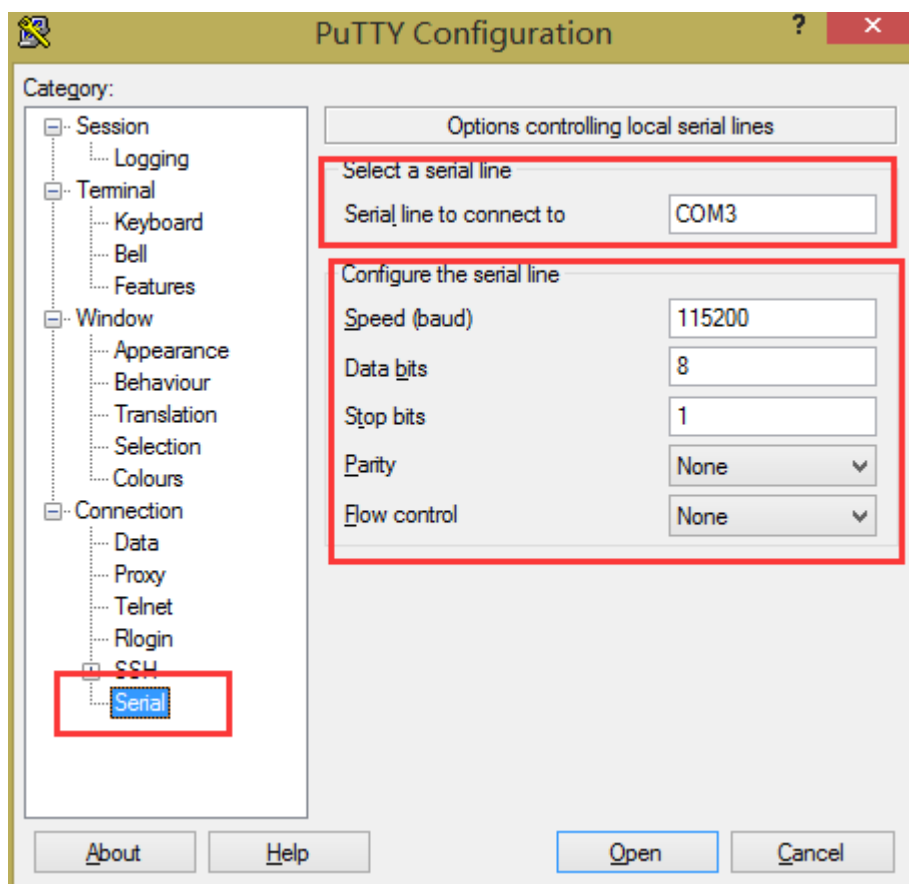




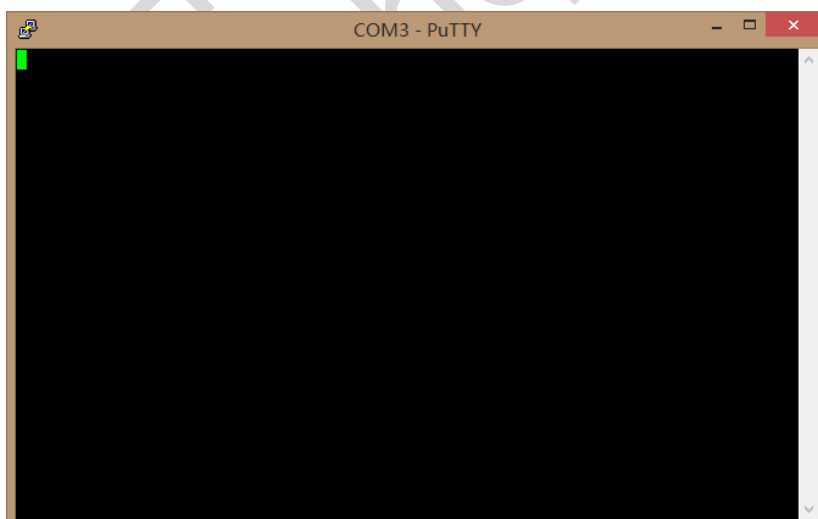
选择串口（Serial）连接方式：



选中第一步方框内的 Serial,再点击第二步中的 Serial,进入串口设置的对话框：

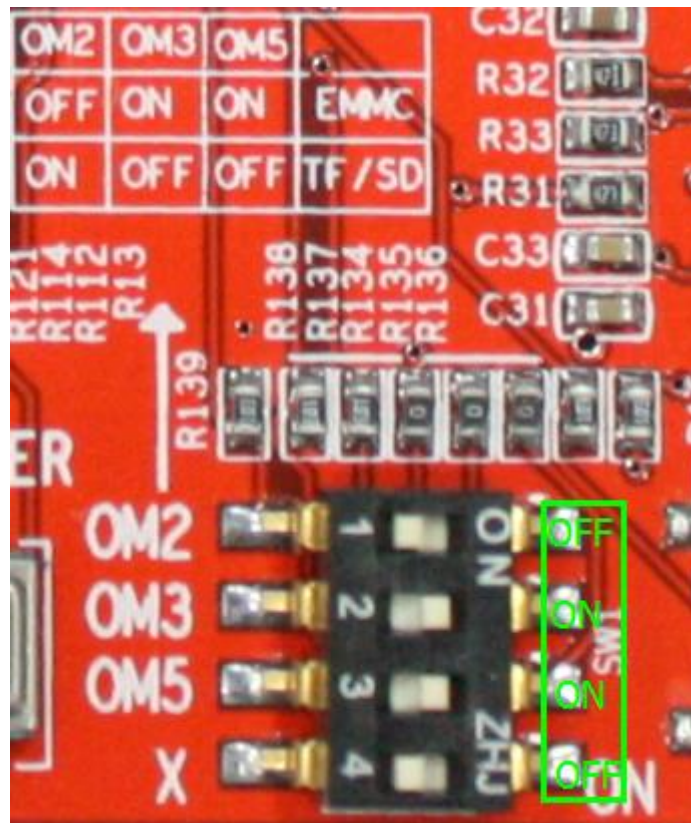


点击 open 打开串口。



3.4.8 启动开发板

拨码开关拨至 0110，如下图所示。



启动开发板，在倒计时结束前，按任意键停止在 Uboot 处，串口终端显示图下图所示。

（注意下图的操作是需要 **在 uboot 版本为 2013.01 下操作**，如果开发板中 **uboot 版本为 2010**，则无法烧写成功。如果此处 **uboot 版本不正确**，可以参考 3.4.12 节，使用 SD 卡方式烧写 UBoot）

```
COM6 - PuTTY
U-Boot 2013.01 (Aug 24 2014 - 12:01:19) for FS4412

CPU:      Exynos4412@1000MHz

Board: FS4412
DRAM:  1 GiB
WARNING: Caches not enabled
MMC:  MMC0:  3728 MB
In:    serial
Out:   serial
Err:   serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK
eMMC CLOSE Success.!!

Checking Boot Mode ... EMMC4.41
Net:   dm9000
Hit any key to stop autoboot:  0
FS4412 #
```




修改开发板环境变量（命令框内开头为【#】一般是需要是在串口终端对开发板进行的操作，【\$】一般是在虚拟机下对 Ubuntu 进行的操作，下面省略此说明）。尽量手动输入，不去复制 pdf，避免非法字符。

```
# setenv serverip 192.168.100.192 //主机的 IP 地址，和 3.4.1 节，设置的 Ubuntu IP 地址一致
# setenv ipaddr 192.168.100.191 //板子的 IP，不要和 Windows 或 Ubuntu 冲突
# saveenv //保存环境变量
```

使用【print】命令查看修改后的环境变量。

```
COM6 - PuTTY

Checking Boot Mode ... EMMC4.41
Net:   dm9000
Hit any key to stop autoboot:  0
FS4412 # print
baudrate=115200
boardname=fs4412
bootargs=root=/dev/nfs nfsroot=192.168.3.3:/source/rootfs rw con
sole=ttySAC2,115200 init=/linuxrc ip=192.168.3.8
bootcmd=tftp 41000000 uImage;tftp 42000000 exynos4412-fs4412.dtb
;bootm 41000000 - 42000000
bootdelay=3
date=2014-08-25
ethact=dm9000
ethaddr=11:22:33:44:55:66
fileaddr=41000000
filesize=80B00
gatewayip=192.168.100.1
ipaddr=192.168.100.191
netmask=255.255.255.0
serverip=192.168.100.192
stderr=serial
stdin=serial
stdout=serial

Environment size: 508/16380 bytes
FS4412 #
```

使用 ping 命令尝试 ping 一下 Ubuntu 主机，如下图所示，表示网络已经联通。

```
# ping 192.168.100.192
```



```
COM6 - PuTTY
boardname=fs4412
bootargs=root=/dev/nfs nfsroot=192.168.3.3:/source/rootfs rw con
sole=ttySAC2,115200 init=/linuxrc ip=192.168.3.8
bootcmd=tftp 41000000 uImage;tftp 42000000 exynos4412-fs4412.dtb
;bootm 41000000 - 42000000
bootdelay=3
date=2014-08-25
ethact=dm9000
ethaddr=11:22:33:44:55:66
fileaddr=41000000
filesize=80B00
gatewayip=192.168.100.1
ipaddr=192.168.100.191
netmask=255.255.255.0
serverip=192.168.100.192
stderr=serial
stdin=serial
stdout=serial

Environment size: 508/16380 bytes
FS4412 # ping 192.168.100.192
dm9000 i/o: 0x5000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
operating at 100M full duplex mode
Using dm9000 device
host 192.168.100.192 is alive
FS4412 #
```

3.4.9 烧写 uboot

在 uboot 命令行下，执行命令：

```
# tftp 40008000 u-boot-fs4412.bin
# movi write u-boot 40008000
```



```
COM6 - PuTTY
Checking Boot Mode ... EMMC4.41
Net: dm9000
Hit any key to stop autoboot: 0
FS4412 # tftp 40008000 u-boot-fs4412.bin
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
operating at 100M full duplex mode
Using dm9000 device
TFTP from server 192.168.100.192; our IP address is 192.168.100.191
Filename 'u-boot-fs4412.bin'.
Load address: 0x40008000
Loading: #####
#####
397.5 KiB/s
done
Bytes transferred = 527104 (80b00 hex)
FS4412 # mwi write u-boot 40008000
writing bootloader.. 0, 1038
MMC write: dev # 0, block # 0, count 1038. 1038 blocks write finished
1038 blocks verify1: OK
eMMC CLOSE Success.!!
completed
FS4412 #
```

重启开发板，如下图所示。

```
COM6 - PuTTY
MMC write: dev # 0, block # 0, count 1038. 1038 blocks write finished
1038 blocks verify1: OK
eMMC CLOSE Success.!!
completed
FS4412 #
U-Boot 2013.01 (Aug 24 2014 - 12:01:19) for FS4412

CPU: Exynos4412@1000MHz

Board: FS4412
DRAM: 1 GiB
WARNING: Caches not enabled
MMC: MMC0: 3728 MB
In: serial
Out: serial
Err: serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK
eMMC CLOSE Success.!!

Checking Boot Mode ... EMMC4.41
Net: dm9000
Hit any key to stop autoboot: 0
FS4412 #
```

3.4.10 NFS 挂载方式启动

修改开发板环境变量。

```
# setenv serverip 192.168.100.192
# setenv ipaddr 192.168.100.191
# setenv gatewayip 192.168.100.1
```



```
# setenv bootcmd tftp 41000000 uImage;tftp 42000000 exynos4412-fs4412.dtb; bootm
41000000 - 42000000 //此处和上文是一行内容，避免复制遗漏，结果参见下图

# setenv bootargs root=/dev/nfs nfsroot=192.168.100.192:/source/rootfs rw ip=192.168.100.191
init=/linuxrc console=ttySAC2,115200 //此处和上文是一行内容，避免复制遗漏，结果参见下图

# saveenv
```

用户在设定完这些环境变量后，如果需要重新启动 android 系统，首先需要将开发板的 uboot 版本变为 2010.03 版本（**移植所需要的 uboot 版本为 2013.01，用户需要特别注意**），其次在烧写完 android 系统镜像后，需要按照 FS4412 的使用手册中第六章中的相关操作修改环境变量。

```
COM6 - PuTTY
source/rootfs rw ip=192.168.100.191 init=/linuxrc console=ttySAC
2,115200
FS4412 # save
Saving Environment to MMC...
Writing to MMC(0)... .done
FS4412 # print
baudrate=115200
boardname=fs4412
bootargs=root=/dev/nfs nfsroot=192.168.100.192:/source/rootfs rw
ip=192.168.100.191 init=/linuxrc console=ttySAC2,115200
bootcmd=tftp 41000000 uImage;tftp 42000000 exynos4412-fs4412.dtb
;bootm 41000000 - 42000000
bootdelay=3
date=2014-08-25
ethact=dm9000
ethaddr=11:22:33:44:55:66
fileaddr=41000000
filesize=80B00
gatewayip=192.168.100.1
ipaddr=192.168.100.191
netmask=255.255.255.0
serverip=192.168.100.192
stderr=serial
stdin=serial
stdout=serial

Environment size: 516/16380 bytes
FS4412 #
```

重启开发板，如下图启动，表明内核从主机的/tftpboot处下载，文件系统为 nfs 网络文件系统，位置为主机的/source/rootfs/处。


```
COM6 - PuTTY
[ 1.595000] drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
[ 1.620000] dm9000 5000000.ethernet eth0: link down
[ 1.660000] mmc0: new high speed SDHC card at address 619b
[ 1.665000] mmcblk1: mmc0:619b SE08G 7.28 GiB
[ 1.670000]   mmcblk1: p1
[ 1.870000] dm9000 5000000.ethernet eth0: link up, 100Mbps, full-duplex
, lpa 0xCDE1
[ 1.875000] IP-Config: Guessing netmask 255.255.255.0
[ 1.875000] IP-Config: Complete:
[ 1.875000]       device=eth0, hwaddr=00:0a:2d:a6:55:a2, ipaddr=192.168.
100.191, mask=255.255.255.0, gw=255.255.255.255
[ 1.875000]       host=192.168.100.191, domain=, nis-domain=(none)
[ 1.875000]       bootserver=255.255.255.255, rootserver=192.168.100.192
, rootpath=
[ 1.875000] clk: Not disabling unused clocks
[ 1.905000] usb 1-3: new high-speed USB device number 2 using exynos-eh
ci
[ 1.930000] VFS: Mounted root (nfs filesystem) on device 0:10.
[ 1.935000] devtmpfs: mounted
[ 1.935000] Freeing unused kernel memory: 228K (c0530000 - c0569000)
[ 2.070000] hub 1-3:1.0: USB hub found
[ 2.070000] hub 1-3:1.0: 3 ports detected
[root@farsight]#
```



3.4.11 EMMC 方式启动

将【华清远见-CORTEXA9 资料:\程序源码\Linux 移植实验源码\镜像】下的 U-boot (u-boot-fs4412.bin)、内核镜像 (uImage)、设备树 (exynos4412-fs4412.dtb)、文件系统 (ramdisk.img) 拷贝到虚拟机/tftpboot 目录下:

若用户已经拷贝这些文件到虚拟机/tftpboot 目录下, 则不需要再次拷贝。

此电脑 > 实验代码 > 2、Linux移植驱动及应用 > 2、Linux系统移植 > 实验代码 > 镜像

名称	修改日期	类型	大小
exynos4412-fs4412.dtb	2014/11/26 13:26	DTB 文件	34 KB
ramdisk.img	2014/11/26 13:29	光盘映像文件	2,485 KB
u-boot-fs4412.bin	2014/11/26 13:26	BIN 文件	515 KB
ulmage	2014/11/26 13:29	文件	2,958 KB
该镜像已经完成了所有需要修改的内容, ...	2016/9/24 15:19	TXT 文件	0 KB

a) 烧写 U-boot 到 EMMC 上

```
# tftp 40008000 u-boot-fs4412.bin

# movi write u-boot 40008000
```

b) 烧写内核镜像到 EMMC 上

```
# tftp 41000000 uImage

# movi write kernel 41000000
```

c) 烧写设备树文件到 EMMC 上

```
# tftp 41000000 exynos4412-fs4412.dtb

# movi write dtb 41000000
```

d) 烧写文件系统镜像到 EMMC 上

```
# tftp 41000000 ramdisk.img

# movi write rootfs 41000000 300000 (烧写大小)
```

e) 设置启动参数(从 EMMC 模式自启动 完全脱离服务器)

```
# setenv bootcmd movi read kernel 41000000\;movi read dtb 42000000\;movi read rootfs 43000000
300000\;bootm 41000000 43000000 42000000

#setenv bootargs

# saveenv
```



```
COM6 - PuTTY
[ 1.580000] mmcblk0boot1: mmc1:0001 4YMD3R partition 2 4.00 MiB
[ 1.580000] mmcblk0boot0: mmc1:0001 4YMD3R partition 3 512 KiB
[ 1.580000] mmcblk0: p1 p2 p3 p4
[ 1.585000] mmcblk0boot1: unknown partition table
[ 1.585000] mmcblk0boot0: unknown partition table
[ 1.640000] hub 1-0:1.0: 3 ports detected
[ 1.645000] drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
[ 1.670000] dm9000 5000000.ethernet eth0: link down
[ 1.705000] mmc0: new high speed SDHC card at address cd6d
[ 1.710000] mmcblk1: mmc0:cd6d SE08G 7.28 GiB
[ 1.715000] mmcblk1: p1
[ 1.955000] usb 1-3: new high-speed USB device number 2 using exynos-ehci
[ 2.090000] hub 1-3:1.0: USB hub found
[ 2.090000] hub 1-3:1.0: 3 ports detected
[ 3.295000] IP-Config: Guessing netmask 255.255.255.0
[ 3.295000] IP-Config: Complete:
[ 3.300000] dm9000 5000000.ethernet eth0: link up, 100Mbps, full-duplex, lpa
[ 3.305000] device=eth0, hwaddr=00:0a:2d:a6:55:a2, ipaddr=192.168.9.233,
[ 3.320000] host=192.168.9.233, domain=, nis-domain=(none)
[ 3.325000] bootserver=255.255.255.255, rootserver=192.168.9.120, rootpa
[ 3.330000] clk: Not disabling unused clocks
[ 3.335000] RAMDISK: gzip image found at block 0
[ 3.600000] VFS: Mounted root (ext2 filesystem) on device 1:0.
[root@farsight]#
```

3.4.12 制作 SD 卡启动盘（只需在开发板没有 UBoot 时做）

SD 启动盘制作：

```
$ cd ~
```

将【华清远见-CORTEXA9 资料:程序源码\Linux 移植实验源码\制作 SD 卡启动盘工具】目录下的 sdfuse_q 拷贝到虚拟机 Ubuntu 的共享目录下。

名称	修改日期	类型	大小
sdfuse_q	2016/9/24 18:20	文件夹	

```
$ cp /mnt/hgfs/share/sdfuse_q/ ~ -a
```

```
linux@ubuntu64-vm:~$ cp /mnt/hgfs/share/sdfuse_q/ ~ -a
linux@ubuntu64-vm:~$ ls
examples.desktop  README  toolchain  公共的  视频  文档  音乐
lx.leesheen@gmail.com  sdfuse_q  workdir  模板  图片  下载  桌面
linux@ubuntu64-vm:~$
```

```
$ cd sdfuse_q //进入 sdfuse_q 目录
```

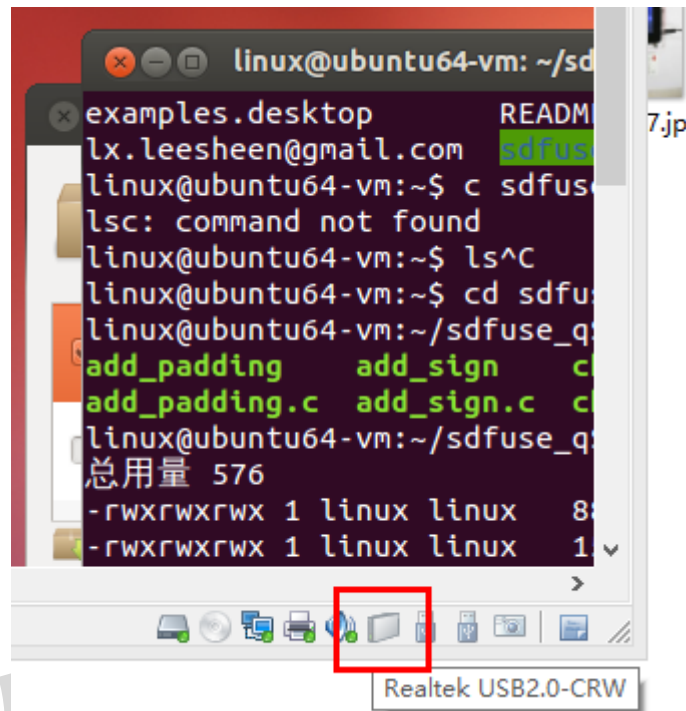
```
$ make //执行编译命令
```

```
$ chmod 777 *.sh
```

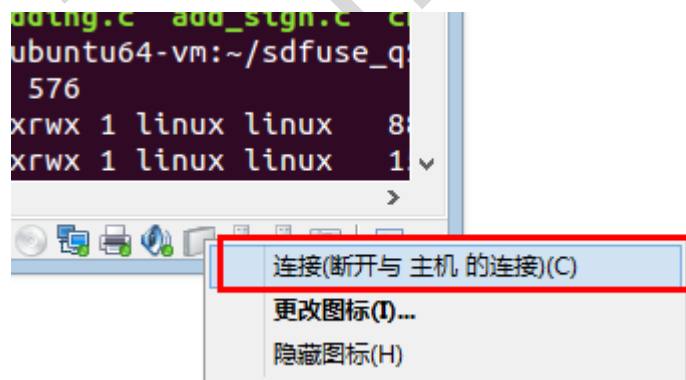


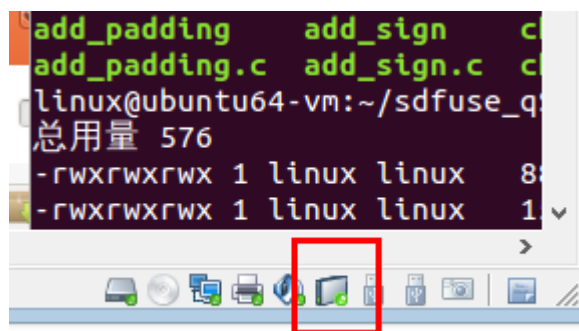
```
linux@ubuntu64-vm:~/sdfuse_q$ ls
add_padding  add_sign  chksum  Makefile  sd_fusing_exynos4x12.sh
add_padding.c add_sign.c chksum.c  mkuboot.sh u-boot-fs4412.bin
linux@ubuntu64-vm:~/sdfuse_q$ make
gcc -o chksum chksum.c
gcc -o add_sign add_sign.c
gcc -o add_padding add_padding.c
linux@ubuntu64-vm:~/sdfuse_q$
```

用读卡器将 SD 卡插入电脑，虚拟机识别到 SD 读卡器。



右键点击图标，选择【连接】





查看生成的设备节点，笔者 SD 卡在 Ubuntu 系统中的设备节点是/dev/sdb，这里提供一种方式查看设备节点，首先输入 ls /dev/sd* 【*代表匹配所有符合 sd 的选项】，sd*最后的设备为 sdb。

```
linux@ubuntu64-vm:~/workdir/fs4412/mksduboot/sdfuse_q$ ls /dev/sd*
/dev/sda /dev/sda1 /dev/sda2 /dev/sda5 /dev/sdb /dev/sdb1
linux@ubuntu64-vm:~/workdir/fs4412/mksduboot/sdfuse_q$
```

使用 df -Th 命令，下图就是整个 SD 卡被 ubuntu 识别之后所产生的设备节点。(从容量来说和 SD 卡容量对等，从挂载点来说符合一般的 SD 卡挂载点)。

```
linux@ubuntu64-vm:~/workdir/fs4412/mksduboot/sdfuse_q$ df -Th
文件系统      类型      容量  已用  可用 已用% 挂载点
/dev/sda1      ext4       78G   9.0G   65G   13%  /
udev           devtmpfs   486M   4.0K   486M    1%  /dev
tmpfs          tmpfs     198M   808K   197M    1%  /run
none           tmpfs     5.0M    0    5.0M    0%  /run/lock
none           tmpfs     495M   200K   495M    1%  /run/shm
:host:/        vmhgfs    126G   99G   27G   79%  /mnt/hgfs
/dev/sdb1      vfat       7.5G   12K   7.5G    1%  /media/FA09-B19E
linux@ubuntu64-vm:~/workdir/fs4412/mksduboot/sdfuse_q$
```

NOTE: 关于 SD 卡或者 U 盘在 ubuntu 下识别顺序的问题，有如下的规则：在插入的 SD 卡或者 U 盘设备被 ubuntu 识别之后，会依次识别成 b, c, d……如果在插入需要制作的 SD 卡后没有其他的 SD 卡或者 U 盘设备插入，那么插入的 SD 卡会被识别成为/dev/sd*下的最后一个纯字母的设备(如下图中即被识别成为 sdb)，即类似于下图：

```
linux@ubuntu64-vm:~/workdir/fs4412/mksduboot/sdfuse_q$ ls /dev/sd*
/dev/sda /dev/sda1 /dev/sda2 /dev/sda5 /dev/sdb /dev/sdb1
linux@ubuntu64-vm:~/workdir/fs4412/mksduboot/sdfuse_q$
```

在确定了设备节点之后，使用如下的命令制作 SD 卡，如果识别的节点不是 sdb，则需要更换为识别的节点。

```
$ sudo ./mkuboot.sh /dev/sdb //将 uboot 烧写到 sd 卡中
```



```
linux@ubuntu64-vm:~/workdir/fs4412/mksduboot/sdfuse_q$ sudo ./mkuboot.sh /dev/sdb
Fuse FS4412 trustzone uboot file into SD card
/dev/sdb reader is identified.
u-boot-fs4412.bin fusing...
记录了1029+1 的读入
记录了1029+1 的写出
527104字节(527 kB)已复制, 16.8736 秒, 31.2 kB/秒
u-boot-fs4412.bin image has been fused successfully.
Eject SD card
```

重新插入 SD 卡，如果提示需格式化，格式化即可。在 SD 卡目录下创建目录 sdupdate，并将【华清远见-CORTEXA9 资料:\程序源码\Linux 移植实验源码\镜像】下的 u-boot-fs4412.bin（图一）拷贝到 sdupdate（图二）目录下，这个操作在 windows 下或 Linux 下做都可以。

名称	修改日期	类型	大小
exynos4412-fs4412.dtb	2014/11/26 13:26	DTB 文件	34 KB
ramdisk.img	2014/11/26 13:29	光盘映像文件	2,485 KB
u-boot-fs4412.bin	2014/11/26 13:26	BIN 文件	515 KB
ulimage	2014/11/26 13:29	文件	2,958 KB
该镜像已经完成了所有需要修改的内容，...	2016/9/24 15:19	TXT 文件	0 KB

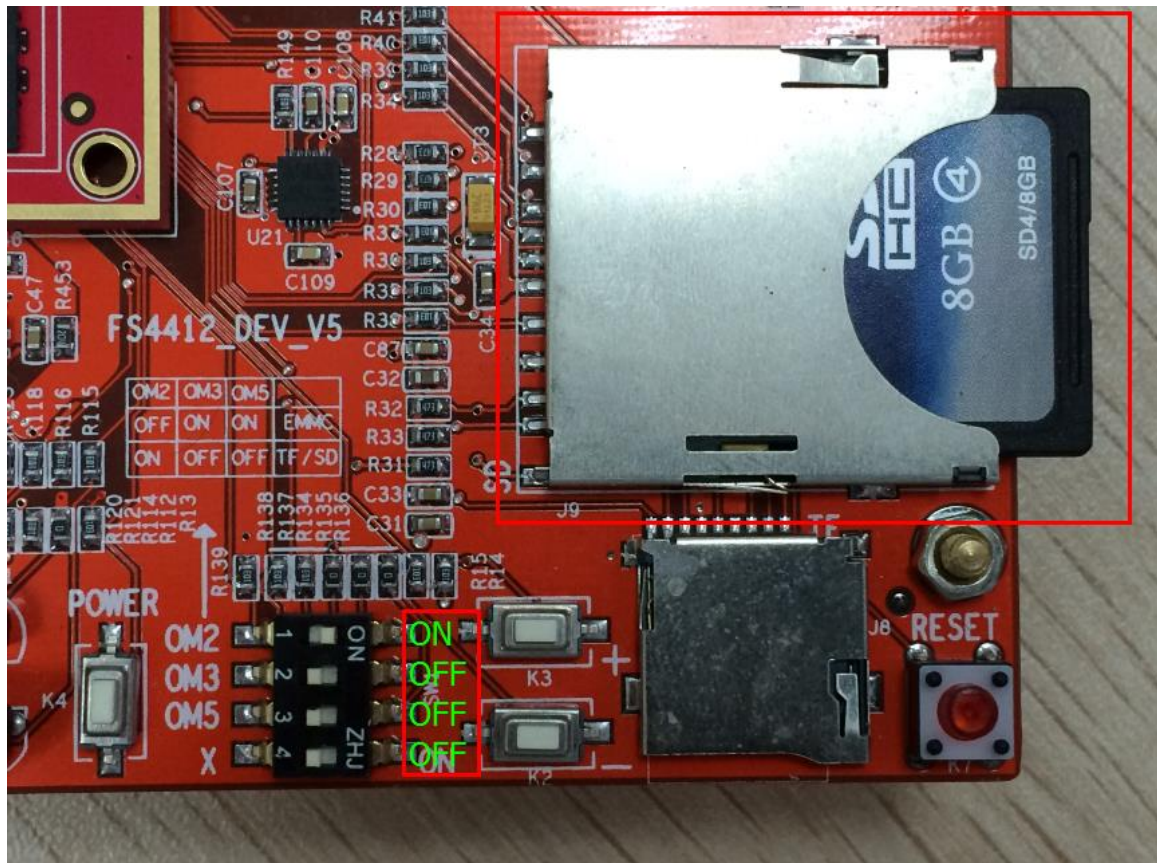
图一

这台电脑 > 可移动磁盘 (L:) > sdupdate

名称	修改日期	类型	大小
u-boot-fs4412.bin	2014/11/26 13:26	BIN 文件	515 KB

图二

将 SD 卡插入开发板 SD 卡槽内，拨码拨至 1000，按照 3.4.6 节步骤连接开发板，连接完成后，按照 3.4.7 节设置串口调试助手，设置完毕启动开发板。



```
COM6 - PuTTY
U-Boot 2010.03 (Aug 25 2014 - 02:40:14) For FarSight FS4412 eMMC

        APLL = 1000MHz, MPLL = 800MHz
        ARM_CLOCK = 1000MHz
PMIC:    S5M8767 (VER5.0)
Board:   FS4412
DRAM:    1 GB
MMCO:    3728 MB
MMC1:    7460 MB
*** Warning - using default environment

In:      serial
Out:     serial
Err:     serial

Checking Boot Mode ... SDMMC
Net:     dm9000
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000:  running in 16 bit mode
MAC:     11:22:33:44:55:66
Hit any key to stop autoboot:  0
FS4412 #
```

在倒计时时按任意键即可看到上图所示，即为 SD 卡启动成功。



在 uboot 命令行下，执行命令：

```
# sdfuse flashall
```

COM6 - PuTTY

```
*** Warning - using default environment

In:      serial
Out:     serial
Err:     serial

Checking Boot Mode ... SDMMC
Net:      dm9000
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot:  0
FS4412 # sdfuse flashall
[Fusing Image from SD Card.]
.fdisk is completed

partition #    size(MB)    block start #    block count    partition_Id
1             2064         3378474         4228686        0x0C
2              302          37290          619014        0x83
3             1026         656304         2103156        0x83
4              302        2759460          619014        0x83

>>>part_type : 2
```

将拨码开关拨至 0110，重启开发板，如下图所示。



```
COM6 - PuTTY
U-Boot 2013.01 (Aug 24 2014 - 12:01:19) for FS4412

CPU:      Exynos4412@1000MHz

Board: FS4412
DRAM: 1 GiB
WARNING: Caches not enabled
MMC:  MMC0:  3728 MB
In:     serial
Out:    serial
Err:    serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK
eMMC CLOSE Success.!!

Checking Boot Mode ... EMMC4.41
Net:  dm9000
Hit any key to stop autoboot:  0
FS4412 # █
```

用户再根据 3.4.10 的步骤设置环境参数即可。



第 4 章 Linux 基础操作

Linux 是个高可靠、高性能的系统，而所有这些优越性只有在直接使用 Linux 命令行（Shell 环境）才能充分地体现出来。本章内容让大家掌握一些基本的 Linux 命令并能够独立定制 Linux 中的系统服务。

4.1 Linux 基本命令

在安装完 Linux 再次启动之后，就可以进入到与 Windows 类似的图形化界面了。这个界面就是 Linux 图形化界面 X 窗口系统（简称 X）的一部分。要注意的是，X 窗口系统仅仅是 Linux 上面的一个软件（或者也可称为服务），它不是 Linux 自身的一部分。虽然现在的 X 窗口系统已经与 Linux 整合地相当好了，但毕竟还不能保证绝对的可靠性。另外，X 窗口系统是一个相当耗费系统资源的软件，它会大大地降低 Linux 的系统性能。因此，若是希望更好地享受 Linux 所带来的高效及高稳定性，建议读者尽可能地使用 Linux 的命令行界面，也就是 Shell 环境。

当用户在命令行下工作时，不是直接同操作系统内核交互信息的，而是由命令解释器接受命令，分析后再传给相关的程序。Shell 是一种 Linux 中的命令行解释程序，就如同 Command.com 是 DOS 下的命令解释程序一样，为用户提供使用操作系统的接口。它们之间的关系如图 2.1 所示。用户在提示符下输入的命令都由 Shell 先解释然后传给 Linux 内核。

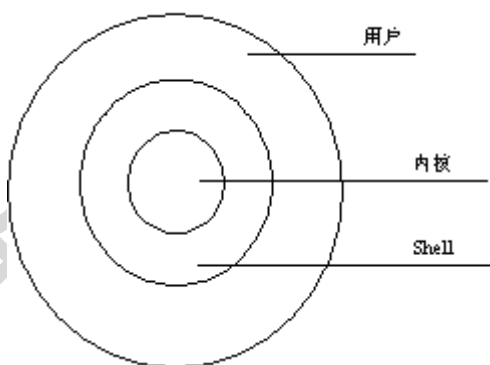


图 2.1 内核、Shell 和用户的关系

Linux 中运行 Shell 的环境是“系统工具”下的“终端”，读者可以单击“终端”以启动 Shell 环境。这时屏幕上显示类似“[linux@ubuntu64]\$”的信息，其中，linux 是指系统用户，而 home 是指当前所在的目录。

由于 Linux 中的命令非常多，要全部介绍几乎不可能。因此，在本书中按照命令的用途进行分类讲解，并且对每一类中最常用的命令详细讲解，同时列出同一类中的其他命令。由于同一类的命令都有很大的相似性，因此，读者通过学习本书中所列命令，可以很快地掌握其他命令。

命令格式说明。

- 格式中带[]的表明为可选项，其他为必选项。
- 选项可以多个连带写入。
- 本章后面选项参数列表表中加粗的含义是：该选项是非常常用的选项。



4.1.1 用户系统相关命令

Linux 是一个多用户的操作系统，每个用户又可以属于不同的用户组，下面，首先来熟悉一下 Linux 中的用户切换和用户管理的相关命令。

1. 用户切换 (su)

(1) 作用

变更为其它使用者的身份，主要用于将普通用户身份转变为超级用户，而且需输入相应用户密码。

(2) 格式

`su [选项] [使用者]`

其中的使用者为要变更的对应使用者。

(3) 常见参数

主要选项参数见表 2.1 所示。

表 2.1 su 命令常见参数列表

选 项	参 数 含 义
<code>-, -l, --login</code>	为该使用者重新登录，大部分环境变量（如 HOME、SHELL 和 USER 等）和工作目录都是以该使用者（USER）为主。若没有指定 USER，缺省情况是 root
<code>-m, -p</code>	执行 su 时不改变环境变量
<code>-c, --command</code>	变更账号为 USER 的使用者，并执行指令（command）后再变回原来使用者

(4) 使用示例

```
[linuxlinux@www linux]$ su - root
```

```
Password:
```

```
[root@www root]#
```

示例通过 su 命令将普通用户变更为 root 用户，并使用选项“-”携带 root 环境变量。

(5) 使用说明

- 在将普通用户变更为 root 用户时建议使用“-”选项，这样可以将 root 的环境变量和工作目录同时带入，否则在以后的使用中可能会由于环境变量的原因而出错。

- 在转变为 root 权限后，提示符变为#。

2. 用户管理 (useradd 和 passwd)

Linux 中常见用户管理命令如表 2.2 所示，本书仅以 useradd 和 passwd 为例进行详细讲解，其他命令类似，请读者自行学习使用。



表 2.2 Linux 常见用户管理命令

命 令	命 令 含 义	格 式
useradd	添加用户账号	useradd [选项] 用户名
usermod	设置用户账号属性	usermod [选项] 属性值
userdel	删除对应用户账号	userdel [选项] 用户名
groupadd	添加组账号	groupadd [选项] 组账号
groupmod	设置组账号属性	groupmod [选项] 属性值
groupdel	删除对应组账号	groupdel [选项] 组账号
passwd	设置账号密码	passwd [对应账号]
id	显示用户 ID、组 ID 和用户所属的组列表	id [用户名]
groups	显示用户所属的组	groups [组账号]
who	显示登录到系统的所有用户	who

(1) 作用

- ① useradd: 添加用户账号。
- ② passwd: 更改对应用户账号密码。

(2) 格式

- ① useradd: useradd [选项] 用户名。
- ② passwd: passwd [选项] [用户名]。

其中的用户名为修改账号密码的用户，若不带用户名，缺省为更改当前使用者账号密码。

(3) 常用参数

- ① useradd 主要选项参数见表 2.3 所示。

表 2.3 useradd 命令常见参数列表

选 项	参 数 含 义
-g	指定用户所属的群组
-m	自动建立用户的登入目录
-n	取消建立以用户名称为名的群组

- ② passwd: 一般很少使用选项参数。

(4) 使用实例

```
[root@www root]# useradd ycw
```

```
[root@www root]# passwd ycw
```

New password:

Retype new password:



```
passwd: all authentication tokens updated successfully
[root@www root]# su - ycw
[ycwycw@www ycw]$
[ycw@www ycw]$ pwd (查看当前目录)
/home/ycw
```

实例中先添加了用户名为 ycw 的用户，接着又为该用户设置了账号密码。并从 su 的命令可以看出，该用户添加成功，其工作目录为“/home/ycw”。

(5) 使用说明

- 在使用添加用户时，这两个命令是一起使用的，其中，useradd 必须用 root 的权限。而且 useradd 指令所建立的账号，实际上是保存在“/etc/passwd”文本文件中，文件中每一行包含一个账号信息。
- 在缺省情况下，useradd 所做的初始化操作包括在“/home”目录下为对应账号建立一个名为同名的主目录，并且还为该用户单独建立一个与用户名同名的组。
- adduser 只是 useradd 的符号链接（关于符号链接的概念在本节后面会有介绍），两者是相同的。
- passwd 还可用于普通用户修改账号密码，Linux 并不采用类似 windows 的密码回显（显示为*号），所以输入的这些字符用户是看不见的。密码最好包括字母、数字和特殊符号，并且设成 6 位以上。

3. 系统管理命令（ps 和 kill）

Linux 中常见的系统管理命令如表 2.4 所示，本书以 ps 和 kill 为例进行讲解。

表 2.4 Linux 常见系统管理命令

命 令	命 令 含 义	格 式
ps	显示当前系统中由该用户运行的进程列表	ps [选项]
top	动态显示系统中运行的程序（一般为每隔 5s）	top
kill	输出特定的信号给指定 PID（进程号）的进程	kill [选项] 进程号 (PID)
uname	显示系统的信息（可加选项-a）	uname [选项]
setup	系统图形化界面配置	setup
crontab	循环执行例行性命令	crontab [选项]
shutdown	关闭或重启 Linux 系统	shutdown [选项] [时间]
uptime	显示系统已经运行了多长时间	uptime
clear	清除屏幕上的信息	clear

(1) 作用

- ① ps：显示当前系统中由该用户运行的进程列表。
- ② kill：输出特定的信号给指定 PID（进程号）的进程，并根据该信号而完成指定的行为。其中可能的信号有进程挂起、进程等待、进程终止等。

(2) 格式



① ps: ps [选项]。

② kill: kill [选项] 进程号 (PID)。

kill 命令中的进程号为信号输出的指定进程的进程号，当选项是缺省时为输出终止信号给该进程。

(3) 常见参数

① ps 主要选项参数见表 2.5 所示。

表 2.5 ps 命令常见参数列表

选 项	参 数 含 义
-ef	查看所有进程及其 PID (进程号)、系统时间、命令详细目录、执行者等
-aux	除可显示 -ef 所有内容外，还可显示 CPU 及内存占用率、进程状态
-w	显示加宽并且可以显示较多的信息

② kill 主要选项参数见表 2.6 所示。

表 2.6 kill 命令常见参数列表

选 项	参 数 含 义
-s	根据指定信号发送给进程
-p	打印出进程号 (PID)，但并不送出信号
-l	列出所有可用的信号名称

(4) 使用实例

```
[root@www root]# ps -ef
```

```

UID      PID  PPID  C STIME TTY          TIME CMD
root      1    0  0  2005 ?        00:00:05 init
root      2    1  0  2005 ?        00:00:00 [keventd]
root      3    0  0  2005 ?        00:00:00 [ksoftirqd_CPU0]
root      4    0  0  2005 ?        00:00:00 [ksoftirqd_CPU1]
root    7421    1  0  2005 ?        00:00:00 /usr/local/bin/ntpd -c /etc/ntp.
root    21787 21739  0 17:16 pts/1    00:00:00 grep ntp
```

```
[root@www root]# kill 7421
```

```
[root@www root]# ps -ef|grep ntp
```

```

root    21789 21739  0 17:16 pts/1    00:00:00 grep ntp
```

该实例中首先查看所有进程，并终止进程号为 7421 的 ntp 进程，之后再次查看时已经没有该进程号的进程。



(5) 使用说明

- ps 在使用中通常可以与其他一些命令结合起来使用，主要作用是提高效率。
- ps 选项中的参数 w 可以写多次，通常最多写 3 次，它的含义表示加宽 3 次，这足以显示很长的命令行了。例如：ps -auxwww。

4. 磁盘相关命令 (fdisk)

Linux 中与磁盘相关的命令如表 2.7 所示，本书仅以 fdisk 为例进行讲解。

表 2.7 Linux 常见系统管理命令

选 项	参 数 含 义	格 式
free	查看当前系统内存的使用情况	free [选项]
df	查看文件系统的磁盘空间占用情况	df [选项]
du	统计目录（或文件）所占磁盘空间的大小	du [选项]
fdisk	查看硬盘分区情况及对硬盘进行分区管理	fdisk [-l]

(1) 作用

fdisk 可以查看硬盘分区情况，并可对硬盘进行分区管理，这里主要向读者介绍查看硬盘分区情况，另外，fdisk 也是一个非常好的硬盘分区工具，感兴趣的读者可以另外查找资料学习使用 fdisk 进行硬盘分区。

(2) 格式

fdisk [-l]

(3) 使用实例

```
[root@linux ~]# fdisk -l
```

```
Disk /dev/hda: 40.0 GB, 40007761920 bytes
```

```
240 heads, 63 sectors/track, 5168 cylinders
```

```
Units = cylinders of 15120 * 512 = 7741440 bytes
```

Device	Boot	Start	End	Blocks	Id	System
/dev/hda1	*	1	1084	8195008+	c	W95 FAT32 (LBA)
/dev/hda2		1085	5167	30867480	f	W95 Ext'd (LBA)
/dev/hda5		1085	2439	10243768+	b	W95 FAT32
/dev/hda6		2440	4064	12284968+	b	W95 FAT32
/dev/hda7		4065	5096	7799526	83	Linux
/dev/hda8		5096	5165	522081	82	Linux swap

可以看出，使用“fdisk -l”列出了文件系统的分区情况。



(4) 使用说明

- 使用 fdisk 必须拥有 root 权限。
- IDE 硬盘对应的设备名称分别为 hda、hdb、hdc 和 hdd，SCSI 硬盘对应的设备名称则为 sda、sdb、…此外，hda1 代表 hda 的第一个硬盘分区，hda2 代表 hda 的第二个分区，依此类推。
- 通过查看/var/log/messages 文件，可以找到 Linux 系统已辨认出来的设备代号。

5. 磁盘挂载命令 (mount)

(1) 作用

挂载文件系统，它的使用权限是超级用户或/etc/fstab 中允许的使用者。正如 1.2.1 节中所述，挂载是指把分区和目录对应的过程，而挂载点是指挂载在文件树中的位置。mount 命令就可以把文件系统挂载到相应的目录下，并且由于 Linux 中把设备都当作文件一样使用，因此，mount 命令也可以挂载不同的设备。

通常，在 Linux 下“/mnt”目录是专门用于挂载不同的文件系统的，它可以在该目录下新建不同的子目录来挂载不同的设备文件系统。

(2) 格式

mount [选项] [类型] 设备文件名 挂载点目录

其中的类型是指设备文件的类型。

(3) 常见参数

mount 常见参数如表 2.8 所示。

表 2.8 mount 命令选项常见参数列表

选 项	参 数 含 义
-a	依照/etc/fstab 的内容装载所有相关的硬盘
-l	列出当前已挂载的设备、文件系统名称和挂载点
-t 类型	将后面的设备以指定类型的文件格式装载到挂载点上。常见的类型有前面介绍过的几种：vfat、ext3、ext2、iso9660、nfs 等
-f	通常用于除错。它会使 mount 不执行实际挂上的动作，而是模拟整个挂上的过程，通常会和-v 一起使用

(4) 使用实例

使用 mount 命令主要通过以下几个步骤。

① 确认是否为 Linux 可以识别的文件系统，Linux 可识别的文件系统只要是以下几种。

- Windows95/98 常用的 FAT32 文件系统：vfat。
- WinNT/2000 的文件系统：ntfs。
- OS/2 用的文件系统：hpfs。
- Linux 用的文件系统：ext2、ext3、nfs。
- CD-ROM 光盘用的文件系统：iso9660。



② 确定设备的名称，确定设备名称可通过使用命令“fdisk -l”查看。

③ 查找挂载点。

必须确定挂载点已经存在，也就是在“/mnt”下的相应子目录已经存在，一般建议在“/mnt”下新建几个如“/mnt/windows”，“/mnt/usb”的子目录，现在有些新版本的 Linux 都可自动挂载文件系统。

④ 挂载文件系统如下所示。

```
[root@linux mnt]# mount -t vfat /dev/hda1 /mnt/c
[root@linux mnt]# cd /mnt/c
24.s03e01.pdtv.xvid-sfm.rm vb Documents and Settings Program Files
24.s03e02.pdtv.xvid-sfm.rm vb Downloads Recycled
...
```

C 盘是原先笔者 Windows 系统的启动盘。可见，在挂载了 C 盘之后，可直接访问 Windows 下的 C 盘的内容。

⑤ 在使用完该设备文件后可使用命令 umount 将其卸载。

```
[root@linux mnt]# umount /mnt/c
[root@linux mnt]# cd /mnt/c
[root@linux c]# ls
```

可见，此时目录“/mnt/c”下为空。Windows 下的 C 盘成功卸载。

4.1.2 文件目录相关命令

由于 Linux 中有关文件目录的操作非常重要，也非常常用，因此在本节中，作者将基本所有的文件操作命令都进行了讲解。

1. cd

(1) 作用

改变工作目录。

(2) 格式

cd [路径]

其中的路径为要改变的工作目录，可为相对路径或绝对路径。

(3) 使用实例

```
[root@www uclinux]# cd /home/linux/
[root@www linux]# pwd
[root@www linux]# /home/linux/
```



该实例中变更工作目录为“/home/linux/”，在后面的 pwd（显示当前目录）的结果中可以看出。

（4）使用说明

• 该命令将当前目录改变至指定路径的目录。若没有指定路径，则回到用户的主目录。为了改变到指定目录，用户必须拥有对指定目录的执行和读权限。

- 该命令可以使用通配符。
- 可使用“cd -”可以回到前次工作目录。
- “./”代表当前目录，“../”代表上级目录。

2. ls

（1）作用

列出目录的内容。

（2）格式：ls [选项] [文件]

其中文件选项为指定查看指定文件的相关内容，若未指定文件，默认查看当前目录下的所有文件。

（3）常见参数

ls 主要选项参数见表 2.9 所示

表 2.9

ls 命令常见参数列表

选 项	参 数 含 义
-l, --format=single-column	一行输出一个文件（单列输出）
-a, -all	列出目录中所有文件，包括以“.”开头的文件
-d	将目录名和其他文件一样列出，而不是列出目录的内容
-l, --format=long, --format=verbose	除每个文件名外，增加显示文件类型、权限、硬链接数、所有者名、组名、大小（Byte）及时间信息（如未指明是其他时间即指修改时间）
-f	不排序目录内容，按它们在磁盘上存储的顺序列出

（4）使用实例

```
[ycwing@www /]$ ls -l
total 220
drwxr-xr-x  2 root  root    4096 Mar 31  2005 bin
drwxr-xr-x  3 root  root    4096 Apr  3  2005 boot
-rw-r--r--  1 root  root         0 Apr 24  2002 test.run
...
```

该实例查看当前目录下的所有文件，并通过选项“-l”显示出详细信息。

显示格式说明如下。

文件类型与权限 链接数 文件属主 文件属组 文件大小 修改的时间 名字



(5) 使用说明

- 在 ls 的常见参数中，-l（长文件名显示格式）的选项是最为常见的。可以详细显示出各种信息。
- 若想显示出所有“.”开头的文件，可以使用-a，这在嵌入式的开发中很常用。

3. mkdir

(1) 作用

创建一个目录。

(2) 格式

mkdir [选项] 路径

(3) 常见参数

mkdir 主要选项参数如表 2.10 所示

表 2.10

mkdir 命令常见参数列表

选 项	参 数 含 义
-m	对新建目录设置存取权限，也可以用 chmod 命令（在本节后面会有详细说明）设置
-p	可以是一个路径名称。此时若此路径中的某些目录尚不存在，在加上此选项后，系统将自动建立好那些尚不存在的目录，即一次可以建立多个目录

(4) 使用实例

```
[root@www linux]# mkdir -p ./hello/my
[root@www my]# pwd (查看当前目录命令)
/home/linux/hello/my
```

该实例使用选项“-p”一次创建了./hello/my 多级目录。

```
[root@www my]# mkdir -m 777 ./why
[root@www my]# ls -l
total 4
drwxrwxrwx  2 root  root    4096 Jan 14 09:24 why
```

该实例使用改选项“-m”创建了相应权限的目录。对于“777”的权限在本节后面会有详细的说明。

(5) 使用说明

该命令要求创建目录的用户在创建路径的上级目录中具有写权限，并且路径名不能是当前目录中已有的目录或文件名称。

4. cat

(1) 作用

连接并显示指定的一个和多个文件的有关信息。



(2) 格式

cat[选项]文件 1 文件 2...

其中的文件 1、文件 2 为要显示的多个文件。

(3) 常见参数

cat 命令的常见参数如表 2.11 所示。

表 2.11

cat 命令常见参数列表

选 项	参 数 含 义
-n	由第一行开始对所有输出的行数编号
-b	和-n 相似，只不过对于空白行不编号

(4) 使用实例

```
[ycw@www ycw]$ cat -n hello1.c hello2.c
1  #include <stdio.h>
2  void main()
3  {
4      printf("Hello!This is my home!\n");
5  }
6  #include <stdio.h>
7  void main()
8  {
9      printf("Hello!This is your home!\n");
10 }
```

在该实例中，指定对 hello1.c 和 hello2.c 进行输出，并指定行号。

5. cp、mv 和 rm

(1) 作用

- ① **cp**: 将给出的文件或目录复制到另一文件或目录中。
- ② **mv**: 为文件或目录改名或将文件由一个目录移入另一个目录中。
- ③ **rm**: 删除一个目录中的一个或多个文件或目录。

(2) 格式

- ① **cp**: cp [选项] 源文件或目录 目标文件或目录。
- ② **mv**: mv [选项] 源文件或目录 目标文件或目录。
- ③ **rm**: rm [选项] 文件或目录。

(3) 常见参数

- ① **cp** 主要选项参数见表 2.12 所示。



表 2.12

cp 命令常见参数列表

选 项	参 数 含 义
-a	保留链接、文件属性，并复制其子目录，其作用等于 dpr 选项的组合
-d	拷贝时保留链接
-f	删除已经存在的目标文件而不提示
-i	在覆盖目标文件之前将给出提示要求用户确认。回答 y 时目标文件将被覆盖，而且是交互式拷贝
-p	此时 cp 除复制源文件的内容外，还将把其修改时间和访问权限也复制到新文件中
-r	若给出的源文件是一目录文件，此时 cp 将递归复制该目录下所有的子目录和文件。此时目标文件必须为一个目录名

② **mv** 主要选项参数如表 2.13 所示。

表 2.13

mv 命令常见参数列表

选 项	参 数 含 义
-i	若 mv 操作将导致对已存在的目标文件的覆盖，此时系统询问是否重写，并要求用户回答 y 或 n，这样可以避免误覆盖文件
-f	禁止交互操作。在 mv 操作要覆盖某已有的目标文件时不给任何指示，在指定此选项后，i 选项将不再起作用

③ **rm** 主要选项参数如表 2.14 所示。

表 2.14

rm 命令常见参数列表

选 项	参 数 含 义
-i	进行交互式删除
-f	忽略不存在的文件，但从不给提示
-r	指示 rm 将参数中列出的全部目录和子目录均递归地删除

(4) 使用实例

① **cp**

```
[root@www hello]# cp -a ./my/why/ ./
[root@www hello]# ls
my  why
```

该实例使用 -a 选项将 “/my/why” 目录下的所有文件复制到当前目录下。而此时在原先目录下还有原有的文件。



② mv

```
[root@www hello]# mv -i ./my/why/ ./
[root@www hello]# ls
my  why
```

该实例中把“/my/why”目录下的所有文件移至当前目录，则原目录下文件被自动删除。

③ rm

```
[root@www hello]# rm -r -i ./why
rm: descend into directory './why'? y
rm: remove './why/my.c'? y
rm: remove directory './why'? y
```

该实例使用“-r”选项删除“./why”目录下所有内容，系统会进行确认是否删除。

(5) 使用说明

① cp: 该命令把指定的源文件复制到目标文件或把多个源文件复制到目标目录中。

② mv:

- 该命令根据命令中第二个参数类型的不同（是目标文件还是目标目录）来判断是重命名还是移动文件，当第二个参数类型是文件时，mv 命令完成文件重命名，此时，它将所给的源文件或目录重命名为给定的目标文件名；

- 当第二个参数是已存在的目录名称时，mv 命令将各参数指定的源文件均移至目标目录中；
- 在跨文件系统移动文件时，mv 先复制，再将原有文件删除，而链至该文件的链接也将丢失。

③ rm:

- 如果没有使用-r 选项，则 rm 不会删除目录；
- 使用该命令时一旦文件被删除，它是不能被恢复的，所以最好使用-i 参数。

6. chown 和 chgrp

(1) 作用

① chown: 修改文件所有者和组别。

② chgrp: 改变文件的组所有权。

(2) 格式

① chown: chown [选项]...文件所有者[所有者组名] 文件

其中的文件所有者为修改后的文件所有者。

② chgrp: chgrp [选项]... 文件所有组 文件

其中的文件所有组为改变后的文件组拥有者。

(3) 常见参数

chown 和 chgrp 的常见参数意义相同，其主要选项参数如表 2.15 所示。



表 2.15

chown 和 chgrp 命令常见参数列表

选 项	参 数 含 义
-c, -changes	详尽地描述每个 file 实际改变了哪些所有权
-f, --silent,--quiet	不打印文件所有权就不能修改的报错信息

(4) 使用实例

在笔者的系统中一个文件的所有者原先是这样的。

```
[root@www linux]# ls -l
-rwxr-xr-x  15 apectel  linux      4096  6月  4  2005 uClinux-dist.tar
```

可以看出，这是一个文件，它的文件拥有者是 apectel，具有可读写和执行的权限，它所属的用户组是 linux，具有可读和执行的权限，但没有可写的全权，同样，系统其他用户对其也只有可读和执行的权限。

首先使用 chown 将文件所有者改为 root。

```
[root@www linux]# chown root uClinux-dist.tar
[root@www linux]# ls -l
-rwxr-xr-x  15 root    linux      4096  6月  4  2005 uClinux-dist.tar
```

可以看出，此时，该文件拥有者变为了 root，它所属文件用户组不变。

接着使用 chgrp 将文件用户组变为 root。

```
[root@www linux]# chgrp root uClinux-dist.tar
[root@www linux]# ls -l
-rwxr-xr-x  15 root    root        4096  6月  4  2005 uClinux-dist.tar
```

(5) 使用说明

- 使用 chown 和 chgrp 必须拥有 root 权限。

7. chmod

(1) 作用

改变文件的访问权限。

(2) 格式

chmod 可使用符号标记进行更改和八进制数指定更改两种方式，因此它的格式也有两种不同的形式。

- ① 符号标记：chmod [选项]…符号权限[符号权限]…文件



其中的符号权限可以指定为多个，也就是说，可以指定多个用户级别的权限，但它们中间要用逗号分开表示，若没有显示指出则表示不作更改。

② 八进制数：`chmod [选项] ...八进制权限 文件...`

其中的八进制权限是指要更改后的文件权限。

(3) 选项参数

`chmod` 主要选项参数如表 2.16 所示。

表 2.16

`chmod` 命令常见参数列表

选 项	参 数 含 义
<code>-c</code>	若该文件权限确实已经更改，才显示其更改动作
<code>-f</code>	若该文件权限无法被更改也不要显示错误信息
<code>-v</code>	显示权限变更的详细资料

(4) 使用实例

`chmod` 涉及文件的访问权限，在此对相关的概念进行简单的回顾。

在 1.6.1 节中已经提到，文件的访问权限可表示成：`- rwx rwx rwx`。在此设有三种不同的访问权限：读（r）、写（w）和运行（x）。三个不同的用户级别：文件拥有者（u）、所属的用户组（g）和系统里的其他用户（o）。在此，可增加一个用户级别 a（all）来表示所有这三个不同的用户级别。

① 对于第一种符号连接方式的 `chmod` 命令中，用加号“+”代表增加权限，用减号“-”删除权限，等于号“=”设置权限。

例如原先笔者系统中有文件 `uClinux20031103.tgz`，其权限如下所示。

```
[root@www linux]# ls -l
-rw-r--r-- 1 root root 79708616 Mar 24 2005 uClinux20031103.tgz

[root@www linux]# chmod a+rx,u+w uClinux20031103.tgz

[root@www linux]# ls -l
-rwxr-xr-x 1 root root 79708616 Mar 24 2005 uClinux20031103.tgz
```

可见，在执行了 `chmod` 之后，文件拥有者除拥有所有用户都有的可读和执行的权限外，还有可写的权限。

② 对于第二种八进制数指定的方式，将文件权限字符代表的有效位设为“1”，即“rw-”、“rw-”和“r--”的八进制表示为“110”、“110”、“100”，把这个 2 进制串转换成对应的 8 进制数就是 6、6、4，也就是说该文件的权限为 664（三位八进制数）。这样对于转化后 8 进制数、2 进制及对应权限的关系如表 2.17 所示。

表 2.17

转化后 8 进制数、2 进制及对应权限的关系

转换后 8 进制数	2 进制	对 应 权 限	转换后 8 进制数	2 进制	对 应 权 限
0	000	没有任何权限	1	001	只能执行



2	010	只写	3	011	只写和执行
4	100	只读	5	101	只读和执行
6	110	读和写	7	111	读，写和执行

同上例，原先笔者系统中有文件 `genromfs-0.5.1.tar.gz`，其权限如下所示。

```
[root@www linux]# ls -l
-rw-rw-r-- 1 linux linux 20543 Dec 29 2004 genromfs-0.5.1.tar.gz
[root@www linux]# chmod 765 genromfs-0.5.1.tar.gz
[root@www linux]# ls -l
-rwxrw-r-x 1 linux linux 20543 Dec 29 2004 genromfs-0.5.1.tar.gz
```

可见，在执行了 `chmod 765` 之后，该文件的拥有者权限、文件组权限和其他用户权限都恰当地对应了。

(5) 使用说明

- 使用 `chmod` 必须具有 `root` 权限。

8. grep

(1) 作用

在指定文件中搜索特定的内容，并将含有这些内容的行标准输出。

(2) 格式

`grep [选项] 格式 [文件及路径]`

其中的格式是指要搜索的内容格式，若缺省“文件及路径”则默认表示在当前目录下搜索。

(3) 常见参数

`grep` 主要选项参数如表 2.18 所示。

表 2.18

`grep` 命令常见参数列表

选 项	参 数 含 义
<code>-c</code>	只输出匹配行的计数
<code>-I</code>	不区分大小写（只适用于单字符）
<code>-h</code>	查询多文件时不显示文件名
<code>-l</code>	查询多文件时只输出包含匹配字符的文件名
<code>-n</code>	显示匹配行及行号
<code>-s</code>	不显示不存在或无匹配文本的错误信息



-v

显示不包含匹配文本的所有行

(4) 使用实例

```
[root@www linux]# grep "hello" / -r
Binary file ./iscit2005/备份/iscit2004.sql matches
./ARM_TOOLS/uClinux-Samsung/linux-2.4.x/Documentation/s390/Debugging390.txt:hello
world$2 = 0
...
```

该本例中，“hello”是要搜索的内容，“/ -r”是指定文件，表示搜索根目录下的所有文件。

(5) 使用说明

- 在缺省情况下，“grep”只搜索当前目录。如果此目录下有许多子目录，“grep”会以如下形式列出：“grep:sound:Is a directory”这会使“grep”的输出难于阅读。但有两种解决的方法：
 - ① 明确要求搜索子目录：grep -r（正如上例中所示）；
 - ② 忽略子目录：grep -d skip。
- 当预料到有许多输出，可以通过管道将其转到“less”（分页器）上阅读：如 grep "h" ./ -r |less 分页阅读。

- grep 特殊用法：

grep pattern1|pattern2 files: 显示匹配 pattern1 或 pattern2 的行；

grep pattern1 files|grep pattern2: 显示既匹配 pattern1 又匹配 pattern2 的行；

9. find

(1) 作用

在指定目录中搜索文件，它的使用权限是所有用户。

(2) 格式

find [路径][选项][描述]

其中的路径为文件搜索路径，系统开始沿着此目录树向下查找文件。它是一个路径列表，相互用空格分离。若缺省路径，那么默认为当前目录。

其中的描述是匹配表达式，是 find 命令接受的表达式。

(3) 常见参数

[选项]主要参数如表 2.19 所示。

表 2.19

find 选项常见参数列表

选 项	参 数 含 义
-depth	使用深度级别的查找过程方式，在某层指定目录中优先查找文件内容
-mount	不在其他文件系统（如 Msdos、Vfat 等）的目录和文件中查找



[描述]主要参数如表 2.20 所示。

表 2.20

find 描述常见参数列表

选 项	参 数 含 义
-name	支持通配符*和?
-user	用户名：搜索文件属主为用户名（ID 或名称）的文件
-print	输出搜索结果，并且打印

(4) 使用实例

```
[root@www linux]# find ./ -name qiong*.c
./qiong1.c
./iscit2005/qiong.c
```

在该实例中使用了 -name 的选项支持通配符。

(5) 使用说明

- 若使用目录路径为 “/”，通常需要查找较多的时间，可以指定更为确切的路径以减少查找时间。
- find 命令可以使用混合查找的方法，例如，想在/etc 目录中查找大于 500000 字节，并且在 24 小时内修改的某个文件，则可以使用 -and（与）把两个查找参数链接起来组合成一个混合的查找方式，如 “find /etc -size +500000c -and -mtime +1”。

10. locate

(1) 作用

用于查找文件。其方法是先建立一个包括系统内所有文件名称及路径的数据库，之后当寻找时就只需查询这个数据库，而不必实际深入档案系统之中了。因此其速度比 find 快很多。

(2) 格式

locate [选项]

(3) locate 主要选项参数如表 2.21 所示。

表 2.21

locate 命令常见参数列表

选 项	参 数 含 义
-u	从根目录开始建立数据库
-U	指定开始的位置建立数据库
-f	将特定的文件系统排除在数据库外，例如 proc 文件系统中的文件
-r	使用正则运算式做寻找的条件
-o	指定数据库的名称

(4) 使用实例

```
[root@www linux]# locate issue -U ./
```



```
[root@www linux]# updatedb
[root@www linux]# locate -r issue*
./ARM_TOOLS/uClinux-Samsung/lib/libpam/doc/modules/pam_issue.sgml
./ARM_TOOLS/uClinux-Samsung/lib/libpam/modules/pam_issue
./ARM_TOOLS/uClinux-Samsung/lib/libpam/modules/pam_issue/Makefile
./ARM_TOOLS/uClinux-Samsung/lib/libpam/modules/pam_issue/pam_issue.c
...
```

示例中首先在当前目录下建立了一个数据库，并且在更新了数据库之后进行正则匹配查找。通过运行可以发现 `locate` 的运行速度非常快。

(5) 使用说明

`locate` 命令所查询的数据库由 `updatedb` 程序来更新的，而 `updatedb` 是由 `cron daemon` 周期性建立的，但若所找到的档案是最近才建立或刚更名的，可能会找不到，因为 `updatedb` 默认每天运行一次，用户可以由修改 `crontab` (`etc/crontab`) 来更新周期值。

11. ln

(1) 作用

为某一个文件在另外一个位置建立一个符号链接。当需要在不同的目录用到相同的文件时，Linux 允许用户不用在每一个需要的目录下都存放一个相同的文件，而只需将其他目录下文件用 `ln` 命令链接即可，这样就不必重复地占用磁盘空间。

(2) 格式

`ln[选项] 目标 目录`

(3) 常见参数

➤ `-s` 建立符号链接（这也是通常惟一使用的参数）。

(4) 使用实例

```
[root@www uclinux]# ln -s ../genromfs-0.5.1.tar.gz ./hello
[root@www uclinux]# ls -l
total 77948
lrwxrwxrwx 1 root root 24 Jan 14 00:25 hello -> ../genromfs-0.5.1.tar.gz
```

该实例建立了当前目录的 `hello` 文件与上级目录之间的符号连接，可以看见，在 `hello` 的 `ls -l` 中的第一位为“1”，表示符号链接，同时还显示了链接的源文件。

(5) 使用说明

- `ln` 命令会保持每一处链接文件的同步性，也就是说，不论改动了哪一处，其他的文件都会发生相同的变化。

- `ln` 的链接又软链接和硬链接两种：



软链接就是上面所说的 `ln -s ***`，它只会在用户选定的位置上生成一个文件的镜像，不会重复占用磁盘空间，平时使用较多的都是软链接；

硬链接是不带参数的 `ln ***`，它会在用户选定的位置上生成一个和源文件大小相同的文件，无论是软链接还是硬链接，文件都保持同步变化。

4.1.3 压缩打包相关命令

Linux 中打包压缩的如表 2.22 所示，本书以 `gzip` 和 `tar` 为例进行讲解。

表 2.22

Linux 常见系统管理命令

命令	命令含义	格 式
<code>bzip2</code>	.bz2 文件的压缩（或解压）程序	<code>bzip2[选项]</code> 压缩（解压缩）的文件名
<code>bunzip2</code>	.bz2 文件的解压缩程序	<code>bunzip2[选项]</code> .bz2 压缩文件
<code>bzip2recover</code>	用来修复损坏的.bz2 文件	<code>bzip2recover</code> .bz2 压缩文件
<code>gzip</code>	.gz 文件的压缩程序	<code>gzip [选项]</code> 压缩（解压缩）的文件名
<code>gunzip</code>	解压被 <code>gzip</code> 压缩过的文件	<code>gunzip [选项]</code> .gz 文件名
<code>unzip</code>	解压 <code>winzip</code> 压缩的.zip 文件	<code>unzip [选项]</code> .zip 压缩文件
<code>compress</code>	早期的压缩或解压程序（压缩后文件名为.Z）	<code>compress [选项]</code> 文件
<code>tar</code>	对文件目录进行打包或解包	<code>tar [选项] [打包后文件名]</code> 文件目录列表

1. gzip

（1）作用

对文件进行压缩和解压缩，而且 `gzip` 根据文件类型可自动识别压缩或解压。

（2）格式

`gzip [选项]` 压缩（解压缩）的文件名。

（3）常见参数

`gzip` 主要选项参数如表 2.23 所示。

表 2.23

gzip 命令常见参数列表

选 项	参 数 含 义
<code>-c</code>	将输出信息写到标准输出上，并保留原有文件
<code>-d</code>	将压缩文件解压
<code>-l</code>	对每个压缩文件，显示下列字段：压缩文件的大小、未压缩文件的大小、压缩比、未压缩文件的名字
<code>-r</code>	查找指定目录并压缩或解压缩其中的所有文件



-t	测试，检查压缩文件是否完整
-v	对每一个压缩和解压的文件，显示文件名和压缩比

(4) 使用实例

```
[root@www my]# gzip hello.c

[root@www my]# ls
hello.c.gz

[root@www my]# gzip -l hello.c

      compressed      uncompressed  ratio uncompressed_name
61                39.3% hello.c
```

该实例将目录下的“hello.c”文件进行压缩，选项“-l”列出了压缩比。

(5) 使用说明

- 使用 gzip 压缩只能压缩单个文件，而不能压缩目录，其选项“-d”是将该目录下的所有文件逐个进行压缩，而不是压缩成一个文件。

2. tar

(1) 作用

对文件目录进行打包或解包。

在此需要对打包和压缩这两个概念进行区分。打包是指将一些文件或目录变成一个总的文件，而压缩则是将一个大的文件通过一些压缩算法变成一个小文件。为什么要区分这两个概念呢？这是由于在 Linux 中的很多压缩程序（如前面介绍的 gzip）只能针对一个文件进行压缩，这样当想要压缩较多文件时，就要借助它的工具将这些堆文件先打成一个包，然后再用原来的压缩程序进行压缩。

(2) 格式

tar [选项] [打包后文件名]文件目录列表。

tar 可自动根据文件名识别打包或解包动作，其中打包后文件名为用户自定义的打包后文件名称，文件目录列表可以是要进行打包备份的文件目录列表，也可以是进行解包的文件目录列表。

(3) 主要参数

tar 主要选项参数如表 2.24 所示。

表 2.24

tar 命令常见参数列表

选 项	参 数 含 义
-c	建立新的打包文件
-r	向打包文件末尾追加文件
-x	从打包文件中解出文件
-o	将文件解开到标准输出
-v	处理过程中输出相关信息



-f	对普通文件操作
-z	调用 gzip 来压缩打包文件，与-x 联用时调用 gzip 完成解压缩
-j	调用 bzip2 来压缩打包文件，与-x 联用时调用 bzip2 完成解压缩
-Z	调用 compress 来压缩打包文件，与-x 联用时调用 compress 完成解压缩

(4) 使用实例

```
[root@www home]# tar -cvf ycw.tar ./ycw
./ycw/
./ycw/.bash_logout
./ycw/.bash_profile
./ycw/.bashrc
./ycw/.bash_history
./ycw/my/
./ycw/my/1.c.gz
./ycw/my/my.c.gz
./ycw/my/hello.c.gz
./ycw/my/why.c.gz
[root@www home]# ls -l ycw.tar
-rw-r--r--  1 root    root      10240 Jan 14 15:01 ycw.tar
```

该实例将“./ycw”目录下的文件加以打包，其中选项“-v”在屏幕上输出了打包的具体过程。

```
[root@www linux]# tar -zxvf linux-2.6.11.tar.gz
linux-2.6.11/
linux-2.6.11/drivers/
linux-2.6.11/drivers/video/
linux-2.6.11/drivers/video/aty/
...
```

该实例用选项“-z”调用 **gzip**，并-x 联用时完成解压缩。

(5) 使用说明

tar 命令除了用于常规的打包之外，使用更为频繁的是用选项“-z”或“-j”调用 **gzip** 或 **bzip2**（Linux 中另一种解压工具）完成对各种不同文件的解压。

表 2.25 对 Linux 中常见类型的文件解压命令做一总结。

表 2.25

Linux 常见类型的文件解压命令一览表

文件后缀	解压命令	示例
------	------	----



.a	tar xv	tar xv hello.a
.z	Uncompress	uncompress hello.Z
.gz	Gunzip	gunzip hello.gz
.tar.Z	tar xvZf	tar xvZf hello.tar.Z
.tar.gz/.tgz	tar xvzf	tar xvzf hello.tar.gz
tar.bz2	tar jxvf	tar jxvf hello.tar.bz2
.rpm	安装: rpm -i	安装: rpm -i hello.rpm
	解压: rpm2cpio	解压: rpm2cpio hello.rpm
.deb (Debian 中的 文件格式)	安装: dpkg -i	安装: dpkg -i hello.deb
	解压: dpkg-deb --fsys- tarfile	解压: dpkg-deb --fsys-tarhello hello.deb
.zip	Unzip	unzip hello.zip

4.1.4 比较合并文件相关命令

1. diff

(1) 作用

比较两个不同的文件或不同目录下的两个同名文件功能，并生成补丁文件。

(2) 格式

diff[选项] 文件 1 文件 2

diff 比较文件 1 和文件 2 的不同之处，并按照选项所指定的格式加以输出。diff 的格式分为命令格式和上下文格式，其中上下文格式又包括了旧版上下文格式和新版上下文格式，命令格式分为标准命令格式、简单命令格式及混合命令格式，它们之间的区别会在使用实例中进行详细地讲解。当选项缺省时，diff 默认使用混合命令格式。

(3) 主要参数

diff 主要选项参数如表 2.26 所示。

表 2.26

diff 命令常见参数列表

选 项	参 数 含 义
-r	对目录进行递归处理
-q	只报告文件是否有不同，不输出结果
-e, -ed	命令格式
-f	RCS (修订控制系统) 命令简单格式
-c, --context	旧版上下文格式
-u, --unified	新版上下文格式



-Z	调用 compress 来压缩归档文件，与-x 联用时调用 compress 完成解压缩
----	--

(4) 使用实例

以下有两个文件 hello1.c 和 hello2.c。

```
//hello1.c

#include <stdio.h>

void main()

{

    printf("Hello!This is my home!\n");

}

//hello2.c

#include <stdio.h>

void main()

{

    printf("Hello!This is your home!\n");

}
```

以下实例主要讲解了各种不同格式的比较和补丁文件的创建方法。

① 主要格式比较

首先使用旧版上下文格式进行比较。

```
[root@www ycw]# diff -c hello1.c hello2.c

*** hello1.c      Sat Jan 14 16:24:51 2006

--- hello2.c      Sat Jan 14 16:54:41 2006

*****

*** 1,5 ****
```



```
#include <stdio.h>

void main()

{

!       printf("Hello!This is my home!\n");

}

--- 1,5 ----

#include <stdio.h>

void main()

{

!       printf("Hello!This is your home!\n");

}
```

可以看出，用旧版上下文格式进行输出时，在显示每个有差别行的同时还显示该行的上下三行，区别的地方用“!”加以标出，由于示例程序较短，上下三行已经包含了全部代码。

接着使用新版的上下文格式进行比较。

```
[root@www ycw]# diff -u hello1.c hello2.c

--- hello1.c      Sat Jan 14 16:24:51 2006

+++ hello2.c      Sat Jan 14 16:54:41 2006

@@ -1,5 +1,5 @@

#include <stdio.h>

void main()

{

-       printf("Hello!This is my home!\n");

+       printf("Hello!This is your home!\n");

}
```



可以看出，在新版上下文格式输出时，仅把两个文件的不同之处分别列出，而相同之处没有重复列出，这样大大方便了用户的阅读。

接下来使用**命令格式**进行比较。

```
[root@www ycw]# diff -e hello1.c hello2.c

4c

    printf("Hello!This is your home!\n");
```

可以看出，命令符格式输出时仅输出了不同的行，其中命令符“4c”中的数字表示行数，字母的含义为 **a**——添加，**b**——删除，**c**——更改。因此，-e 选项的命令符表示：若要把 hello1.c 变为 hello2.c，就需要把 hello1.c 的第四行改为显示出的“printf (“Hello!This is your home!\n”);”即可。

选项“-f”和选项“-e”显示的内容基本相同，就是数字和字母的顺序相交换了，从以下的输出结果可以看出。

```
[root@www ycw]# diff -f hello1.c hello2.c

c4

    printf("Hello!This is your home!\n");
```

在 diff 选项缺省的情况下，输出结果如下所示。

```
[root@www ycw]# diff hello1.c hello2.c

4c4

<     printf("Hello!This is my home!\n");
---
>     printf("Hello!This is your home!\n");
```

可以看出，diff 缺省情况下的输出格式充分显示了如何将 hello1.c 转化为 hello2.c 的方法，即通过“4c4”实现。

② 创建补丁文件（也就是差异文件）是 diff 的功能之一，不同的选项格式可以生成与之相对应的补



丁文件。见下例。

```
[root@www ycw]# diff hello1.c hello2.c >hello.patch

[root@www ycw]# vi hello.patch

4c4

<     printf("Hello!This is my home!\n");

---

>     printf("Hello!This is your home!\n");
```

可以看出，使用缺省选项创建补丁文件的内容和前面使用缺省选项的输出内容是一样的。

2. patch

(1) 作用

命令跟 `diff` 配合使用，把生成的补丁文件应用到现有代码上。

(2) 格式

`patch [选项] [待 patch 的文件[patch 文件]]`。

常用的格式为：`patch -pnum [patch 文件]`，其中的 `-pnum` 是选项参数，在后面会详细介绍。

(3) 常见参数

`patch` 主要选项参数如表 2.27 所示。

表 2.27

`patch` 命令常见参数列表

选 项	参 数 含 义
<code>-b</code>	生成备份文件
<code>-d</code>	把 <code>dir</code> 设置为解释补丁文件名的当前目录
<code>-e</code>	把输入的补丁文件看作是 <code>ed</code> 脚本
<code>-pnum</code>	剥离文件名中的前 <code>NUM</code> 个目录成分
<code>-t</code>	在执行过程中不要求任何输入
<code>-v</code>	显示 <code>patch</code> 的版本号

以下对 `-pnum` 选项进行说明。

首先查看以下示例（对分别位于 `xc.orig/config/cf/Makefile` 和 `xc.bsd/config/cf/Makefile` 的文件使用 `patch` 命令）。



```
diff -ruNa xc.orig/config/cf/Makefile xc.bsd/config/cf/Makefile
```

以下是 patch 文件的头标记。

```
--- xc.orig/config/cf/Imake.cf Fri Jul 30 12:45:47 1999  
+++ xc.new/config/cf/Imake.cf Fri Jan 21 13:48:44 2000
```

这个 patch 如果直接应用，那么它会去找“xc.orig/config/cf”目录下的 Makefile 文件，假如用户源码树的根目录是缺省的 xc 而不是 xc.orig，则除了可以把 xc.orig 移到 xc 处之外，还有什么简单的方法应用此 patch 吗？NUM 就是为此而设的：patch 会把目标路径名剥去 NUM 个“/”，也就是说，在此例中，-p1 的结果是 config/cf/Makefile，-p2 的结果是 cf/Makefile。因此，在此例中就可以用命令 cd xc; patch _p1 < /pathname/xxx.patch 完成操作。

(4) 使用实例

```
[root@www ycw]# diff hello1.c hello2.c >hello1.patch  
  
[root@www ycw]# patch ./hello1.c < hello1.patch  
  
patching file ./hello1.c  
  
[root@www ycw] ]# vi hello1.c  
  
#include <stdio.h>  
  
void main()  
  
{  
  
    printf("Hello!This is your home!\n");  
  
}
```

在该实例中，由于 patch 文件和源文件在同一目录下，因此直接给出了目标文件的目录，在应用了 patch 之后，hello1.c 的内容变为了 hello2.c 的内容。

(5) 使用说明



- 如果 patch 失败，patch 命令会把成功的 patch 行补上其差异，同时（无条件）生成备份文件和一个 .rej 文件。rej 文件里是没有成功提交的 patch 行，需要手工打上补丁。这种情况在原码升级的时候有可能会发生。

- 在多数情况下，patch 程序可以确定补丁文件的格式，当它不能识别时，可以使用 -c、-e、-n 或者 -u 选项来指定输入的补丁文件的格式。由于只有 GNU patch 可以创建和读取新版上下文格式的 patch 文件，因此，除非能够确定补丁所面向的只是那些使用 GNU 工具的用户，否则应该使用旧版上下文格式来生成补丁文件。

- 为了使 patch 程序能够正常工作，需要上下文的行数至少是 2 行（即至少是有一处差别的文件）。

4.1.5 网络相关命令

Linux 下网络相关的常见命令如下表 2.28 所示，本书仅以 ifconfig 和 ftp 为例进行说明。

表 2.28

Linux 下网络相关命令

选 项	参 数 含 义	常见选项格式
netstat	显示网络连接、路由表和网络接口信息	netstat [-an]
nslookup	查询一台机器的 IP 地址和其对应的域名	Nslookup [IP 地址/域名]
finger	查询用户的信息	finger [选项] [使用者] [用户@主机]
ping	用于查看网络上的主机是否在工作	ping [选项] 主机名/IP 地址
ifconfig	查看和配置网络接口的参数	ifconfig [选项] [网络接口]
ftp	利用 ftp 协议上传和下载文件	在本节中会详细讲述
telnet	利用 telnet 协议浏览信息	telnet [选项] [IP 地址/域名]
ssh	利用 ssh 登录对方主机	ssh [选项] [IP 地址]

1. ifconfig

（1）作用

用于查看和配置网络接口的地址和参数，包括 IP 地址、网络掩码、广播地址，它的使用权限是超级用户。

（2）格式

ifconfig 有两种使用格式，分别用于查看和更改网络接口。

① ifconfig [选项] [网络接口]：用来查看当前系统的网络配置情况。

② ifconfig 网络接口 [选项] 地址：用来配置指定接口（如 eth0，eth1）的 IP 地址、网络掩码、广播地址等。

（3）常见参数

ifconfig 第二种格式常见选项参数如表 2.29 所示。



表 2.29

ftp 命令选项常见参数列表

选 项	参 数 含 义
-interface	指定的网络接口名，如 eth0 和 eth1
up	激活指定的网络接口卡
down	关闭指定的网络接口
broadcast address	设置接口的广播地址
poin to point	启用点对点方式
address	设置指定接口设备的 IP 地址
netmask address	设置接口的子网掩码

(4) 使用实例

首先，在本例中使用 ifconfig 的第一种格式来查看网口配置情况。

```
[root@linux workplace]# ifconfig

eth0      Link encap:Ethernet  HWaddr 00:08:02:E0:C1:8A

          inet addr:59.64.205.70  Bcast:59.64.207.255  Mask:255.255.252.0
          inet6 addr: fe80::208:2ff:fee0:c18a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:26931 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3209 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:6669382 (6.3 MiB)  TX bytes:321302 (313.7 KiB)
          Interrupt:11

lo        Link encap:Local Loopback

          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:2537 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2537 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2093403 (1.9 MiB)  TX bytes:2093403 (1.9 MiB)
```

可以看出，使用 ifconfig 的显示结果中详细列出了所有活跃接口的 IP 地址、硬件地址、广播地址、子网掩码、回环地址等。

```
[root@linux workplace]# ifconfig eth0

eth0      Link encap:Ethernet  HWaddr 00:08:02:E0:C1:8A
```



```
inet addr:59.64.205.70 Bcast:59.64.207.255 Mask:255.255.252.0
inet6 addr: fe80::208:2ff:fee0:c18a/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:27269 errors:0 dropped:0 overruns:0 frame:0
TX packets:3212 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:6698832 (6.3 MiB) TX bytes:322488 (314.9 KiB)

Interrupt:11
```

在此例中，通过指定接口显示出对应接口的详细信息。另外，用户还可以通过指定参数“-a”来查看所有接口（包括非活跃接口）的信息。

接下来的示例指出了如何使用 `ifconfig` 的第二种格式来改变指定接口的网络参数配置。

```
[root@linux ~]# ifconfig eth0 down
[root@linux ~]# ifconfig
lo          Link encap:Local Loopback
            inet addr:127.0.0.1 Mask:255.0.0.0
            inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING MTU:16436 Metric:1
            RX packets:1931 errors:0 dropped:0 overruns:0 frame:0
            TX packets:1931 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:2517080 (2.4 MiB) TX bytes:2517080 (2.4 MiB)
```

在此例中，通过将指定接口的状态设置为 **DOWN**，暂时暂停该接口的工作。

```
[root@linux workplace]# ifconfig eth0 210.25.132.142 netmask 255.255.255.0
[root@linux workplace]# ifconfig
eth0       Link encap:Ethernet HWaddr 00:08:02:E0:C1:8A
            inet addr:210.25.132.142 Bcast:210.25.132.255 Mask:255.255.255.0
            inet6 addr: fe80::208:2ff:fee0:c18a/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:1722 errors:0 dropped:0 overruns:0 frame:0
            TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:147382 (143.9 KiB) TX bytes:398 (398.0 b)

Interrupt:11

...
```




从上例可以看出，`ifconfig` 改变了接口 `eth0` 的 IP 地址、子网掩码等，在之后的 `ifconfig` 查看中可以看出确实发生了变化。

(5) 使用说明

用 `ifconfig` 命令配置的网络设备参数不需重启就可生效，但在机器重新启动以后将会失效。

4.2 Linux 编辑器 vi 的使用

Linux 系统提供了一个完整的编辑器家族系列，如 `ed`、`ex`、`vi` 和 `emacs` 等。按功能它们可以分为两大类：行编辑器（`ed`、`ex`）和全屏幕编辑器（`vi`、`emacs`）。行编辑器每次只能对单行进行操作，使用起来很不方便。而全屏幕编辑器可以对整个屏幕进行编辑，用户编辑的文件直接显示在屏幕上，从而克服了行编辑的那种不直观的操作方式，便于用户学习和使用，具有强大的功能。

`vi` 是 Linux 系统的第一个全屏幕交互式编辑程序，它从诞生至今一直得到广大用户的青睐，历经数十年仍然是人们主要使用的文本编辑工具，足以见其生命力之强，而强大的生命力是其强大的功能带来的。由于大多数读者在此之前都已经用惯了 Windows 的 `word` 等编辑器，因此，在刚刚接触时总会或多或少不适应，但只要习惯之后，就能感受到它的方便与快捷。

4.2.1 vi 的模式

`vi` 有 3 种模式，分别为命令行模式、插入模式及底行模式各模式的功能，下面具体进行介绍。

(1) 命令行模式

用户在用 `vi` 编辑文件时，最初进入的为一般模式。在该模式中可以通过上下移动光标进行“删除字符”或“整行删除”等操作，也可以进行“复制”、“粘贴”等操作，但无法编辑文字。

(2) 插入模式

只有在该模式下，用户才能进行文字编辑输入，用户可按 `[ESC]` 键回到命令行模式。

(3) 底行模式

在该模式下，光标位于屏幕的底行。用户可以进行文件保存或退出操作，也可以设置编辑环境，如寻找字符串、列出行号等。

4.2.2 vi 的基本流程

(1) 进入 `vi`，即在命令行下键入 `vi hello`（文件名）。此时进入的是命令行模式，光标位于屏幕的上方，如图 3.1 所示。

(2) 在命令行模式下键入 `i` 进入到插入模式，如图 3.2 所示。可以看出，在屏幕底部显示有“插入”表示插入模式，在该模式下可以输入文字信息。

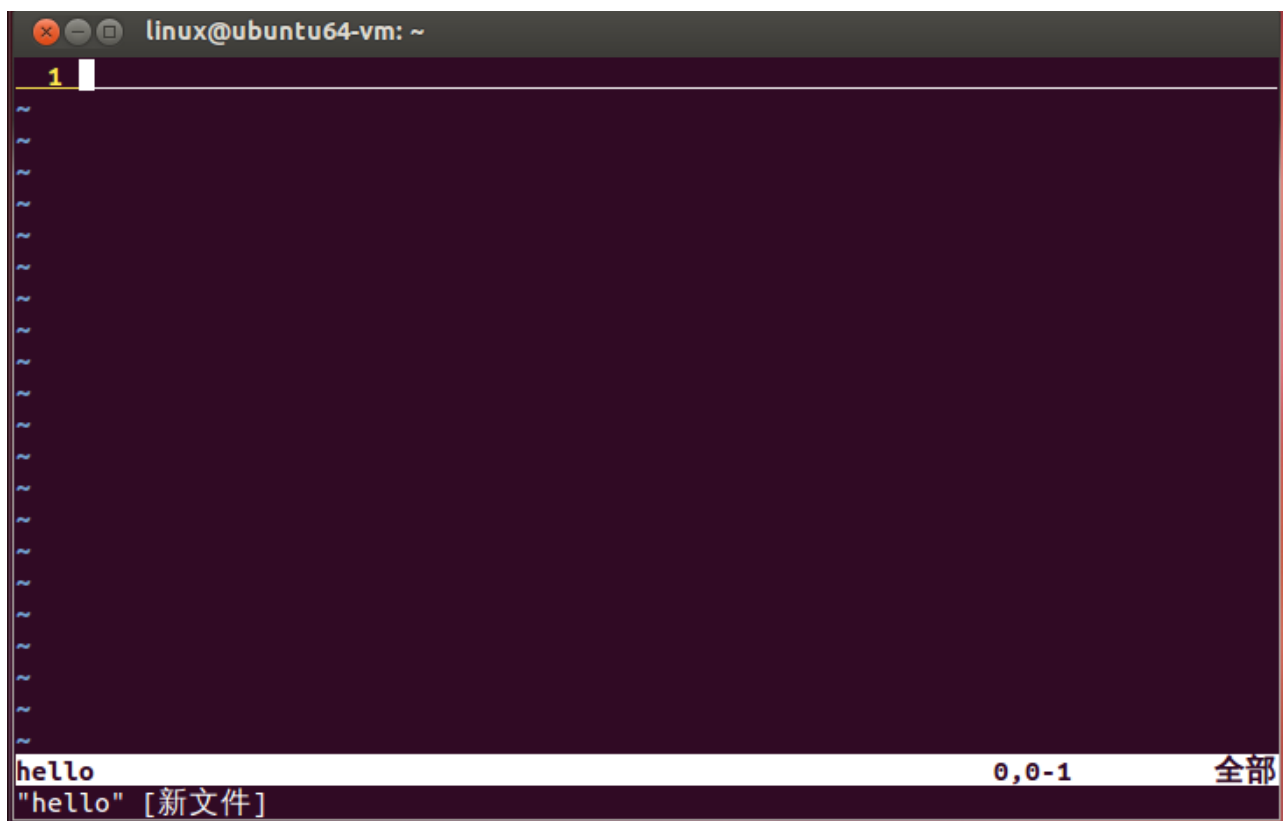


图 3.1 进入 vi 命令行模式

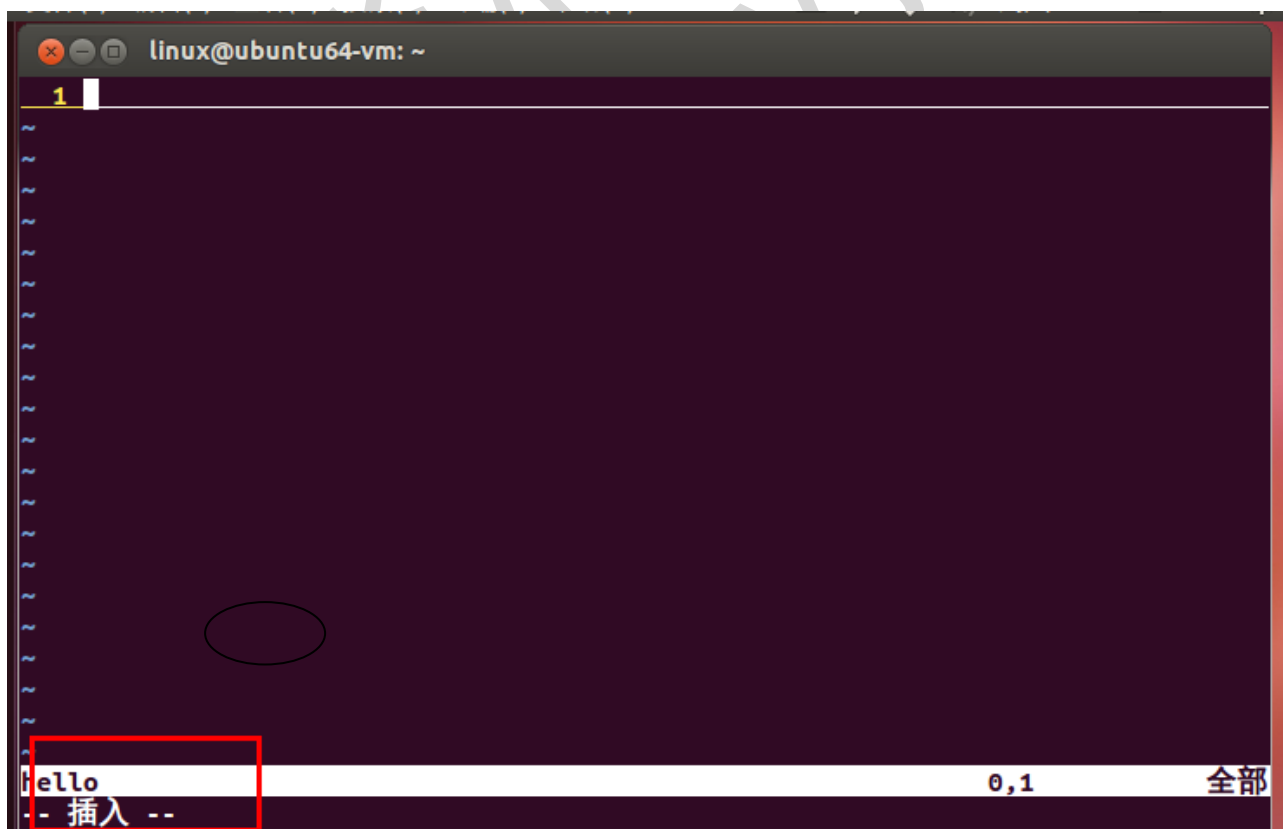


图 3.2 进入 vi 插入模式



(3) 最后，在插入模式中，输入“Esc”，则当前模式转入命令行模式，并在底行行中输入“:wq”（存盘退出）进入底行模式，如图 3.3 所示。

这样，就完成了简单的 vi 操作流程：命令行模式→插入模式→底行模式。由于 vi 在不同的模式下有不同的操作功能，因此，读者一定要时刻注意屏幕最下方的提示，分清所在的模式。

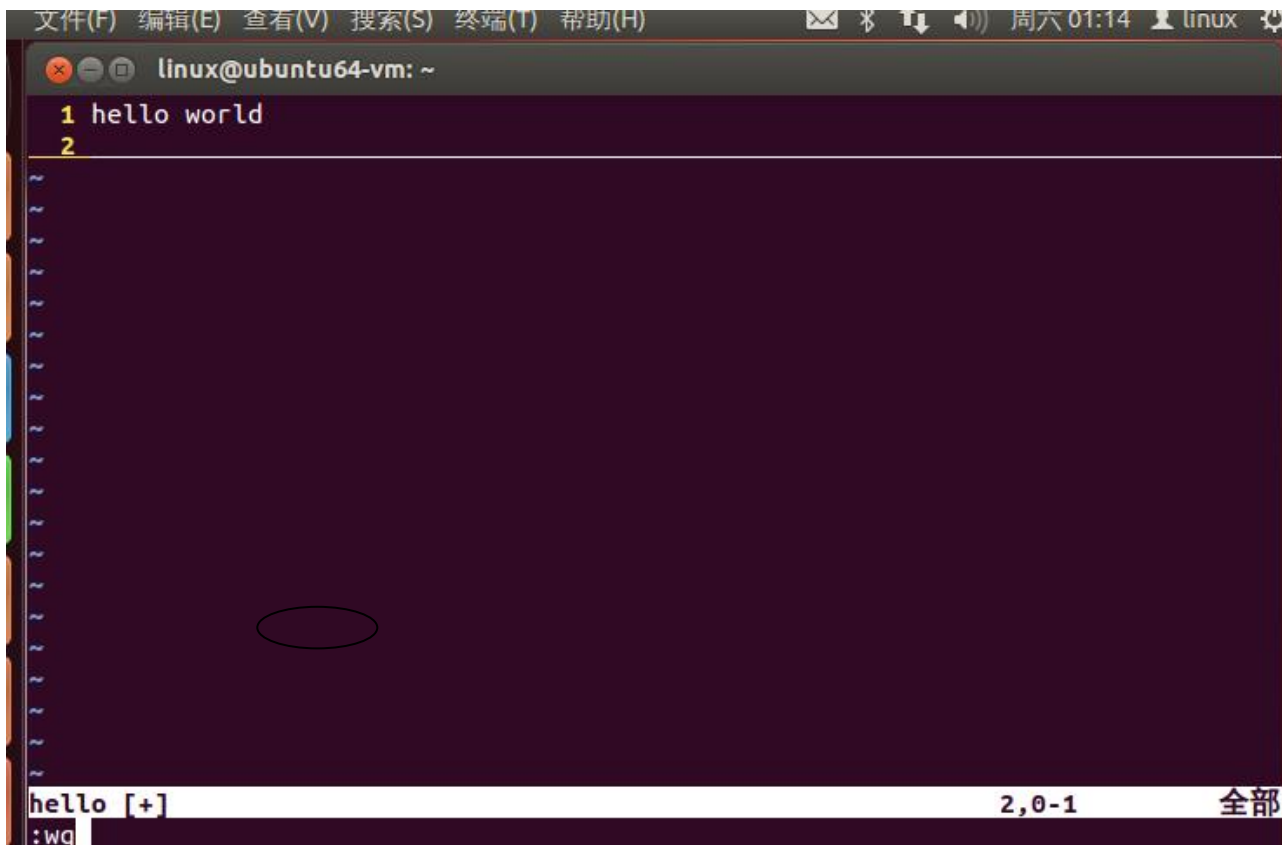


图 3.3 进入 Vi 底行模式

4.2.3 vi 的各模式功能键

(1) 命令行模式常见功能键如表 3.1 所示。

表 3.1 vi 命令行模式功能键

目 录	目 录 内 容
I	切换到插入模式，此时光标当于开始输入文件处
A	切换到插入模式，并从目前光标所在位置的下一个位置开始输入文字
O	切换到插入模式，且从行首开始插入新的一行
[ctrl]+[b]	屏幕往“后”翻动一页
[ctrl]+[f]	屏幕往“前”翻动一页



[ctrl]+[u]	屏幕往“后”翻动半页
[ctrl]+[d]	屏幕往“前”翻动半页
0 (数字 0)	光标移到本行的开头
G	光标移动到文章的最后
nG	光标移动到第 n 行
\$	移动到光标所在行的“行尾”
n<Enter>	光标向下移动 n 行
/name	在光标之后查找一个名为 name 的字符串
?name	在光标之前查找一个名为 name 的字符串
x	删除光标所在位置的一个字符

续表

目 录	目 录 内 容
X	删除光标所在位置的“前面”一个字符
dd	删除光标所在行
ndd	从光标所在行开始向下删除 n 行
yy	复制光标所在行
nyy	复制光标所在行开始的向下 n 行
p	将缓冲区内的字符粘贴到光标所在位置（与 yy 搭配）
u	恢复前一个动作

(2) 插入模式的功能键只有一个，也就是 Esc 退出到命令行模式。

(3) 底行模式常见功能键如表 3.2 所示。

表 3.2 vi 底行模式功能键

目 录	目 录 内 容
:w	将编辑的文件保存到磁盘中



:q	退出 Vi（系统对做过修改的文件会给出提示）
:q!	强制退出 Vi（对修改过的文件不作保存）
:wq	存盘后退出
:w [filename]	另存一个名为 filename 的文件
:set nu	显示行号，设定之后，会在每一行的前面显示对应行号
:set nonu	取消行号显示

华清远见
dev.hqyj.com



第 5 章 Linux Shell 编程

5.1 实验原理

Shell 是系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。

实际上 Shell 是一个命令解释器，它解释由用户输入的命令并且把它们送到内核。不仅如此，Shell 有自己的编程语言用于对命令的编辑，它允许用户编写由 shell 命令组成的程序。Shell 编程语言具有普通编程语言的很多特点，比如它也有循环结构和分支控制结构等，用这种编程语言编写的 Shell 程序与其他应用程序具有同样的效果。

Linux 提供了像 Microsoft Windows 那样的可视的命令输入界面--X Window 的图形用户界面（GUI）。它提供了很多桌面环境系统，其操作就像 Windows 一样，有窗口、图标和菜单，所有的管理都是通过鼠标控制。

每个 Linux 系统的用户可以拥有他自己的用户界面或 Shell，用以满足他们自己专门的 Shell 需要。

同 Linux 本身一样，Shell 也有多种不同的版本。主要有下列版本的 Shell：

Bourne Shell：是贝尔实验室开发的。

BASH：是 GNU 的 Bourne Again Shell，是 GNU 操作系统上默认的 shell。

Korn Shell：是对 Bourne Shell 的发展，在大部分内容上与 Bourne Shell 兼容。

C Shell：是 SUN 公司 Shell 的 BSD 版本。

Z Shell：The last shell you'll ever need! Z 是最后一个字母，也就是终极 Shell。它集成了 bash、ksh 的重要特性，同时又增加了自己独有的特性。

不论是哪一种 Shell，它最主要的功用都是解译使用者在命令列提示符号下输入的指令。Shell 语法分析命令列，把它分解成以空白区分开的符号（token），在此空白包括了跳位键（tab）、空白和换行（New Line）。如果这些字包含了 metacharacter，shell 将会评估（evaluate）它们的正确用法。另外，shell 还管理档案输入输出及幕后处理（background processing）。在处理命令列之后，shell 会寻找命令并开始执行它们。

5.2 实验目的

熟悉 Linux shell 脚本的编写方法

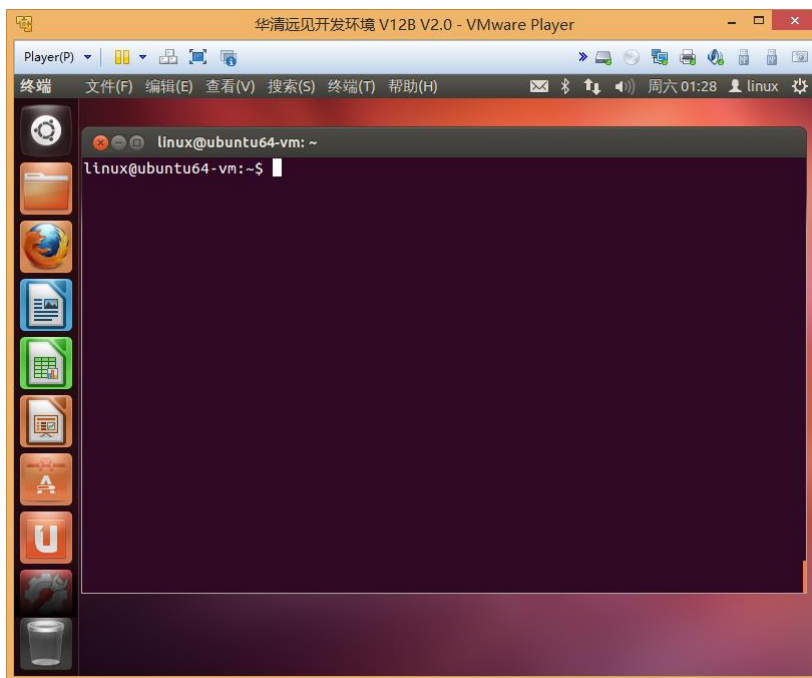
5.3 实验平台

华清远见开发环境



5.4 实验步骤

5.4.1 准备环境



5.4.2 拷贝代码

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\02. Linux 系统 shell 程序设计实验\实验代码】目录拷贝到虚拟机共享目录下。

(注意：此步骤已经将下文中 Linux 应用部分实验代码都拷贝到共享目录下，下文则省略此部分说明)

明)

1、建立相关目录；

```
$ mkdir workdir/linux/  
$ mkdir workdir/linux/application  
$ cd workdir/linux/application
```



```
$ mkdir 5-shell
```

2、将代码从共享目录拷入虚拟机 Linux 操作系统下；（可使用【ctrl+空格】切换输入法）

```
$ mkdir 5-shell  
$ cp /mnt/hgfs/share/实验代码/02.Linux 系统 shell 程序设计实验/实验代码/* 5-shell/ -a  
$ cd 5-shell/
```



```
linux@ubuntu64-vm:~/workdir/linux/application$ cp /mnt/hgfs/share/实验代码/02.\
Linux系统shell程序设计实验/实验代码/* 5-shell/ -a
linux@ubuntu64-vm:~/workdir/linux/application$ cd 5-shell/
linux@ubuntu64-vm:~/workdir/linux/application/5-shell$ ls
case_and_if.sh file_or_dir.sh simple unload.sh
```

5.4.1 执行代码

```
$ chmod 777 file_or_dir.sh //给脚本添加可执行权限
```

```
$ ./file_or_dir.sh //执行脚本，脚本的作用是判断该目录下的文件时目录还是普通文
件
```

5.4.2 实验结果

```
linux@ubuntu64-vm:~/workdir/linux/application/5-shell$ chmod 777 file_or_dir.sh
linux@ubuntu64-vm:~/workdir/linux/application/5-shell$ ./file_or_dir.sh
case_and_if.sh is a file
file_or_dir.sh is a file
simple is a directory
unload.sh is a file
linux@ubuntu64-vm:~/workdir/linux/application/5-shell$
```

5.5 相关代码

```
#!/bin/sh
for i in *
do
    if [ -f $i ]
    then
        echo "$i is a file"
    elif [ -d $i ]
    then
        echo "$i is a directory"
    fi
done
```




第 6 章 GCC 编译

6.1 实验原理

GNU CC（简称为 Gcc）是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译用 C、C++ 和 Object C 等语言编写的程序。gcc 不仅功能强大，而且可以编译如 C、C++、Object C、Java、Fortran、Pascal、Modula-3 和 Ada 等多种语言，而且 gcc 又是一个交叉平台编译器，它能够在当前 CPU 平台上为多种不同体系结构的硬件平台开发软件，因此尤其适合在嵌入式领域的开发编译。

下表 3.3 是 gcc 支持编译源文件的后缀及其解释。

表 3.3 Gcc 所支持后缀名解释

后 缀 名	所对应的语言	后 缀 名	所对应的语言
.c	C 原始程序	.s/.S	汇编语言原始程序
.C/.cc/.cxx	C++原始程序	.h	预处理文件（头文件）
.m	Objective-C 原始程序	.o	目标文件
.i	已经过预处理的 C 原始程序	.a/.so	编译后的库文件
.ii	已经过预处理的 C++原始程序		

如本章开头提到的，gcc 的编译流程分为了 4 个步骤，分别为：

- 预处理（Pre-Processing）；
- 编译（Compiling）；
- 汇编（Assembling）；
- 链接（Linking）。

gcc 有超过 100 个的可用选项，主要包括总体选项、告警和出错选项、优化选项和体系结构相关选项。以下对每一类中最常用的选项进行讲解。

（1）总体选项

gcc 的总结选项如表 3.4 所示，很多在前面的示例中已经有所涉及。

表 3.4 gcc 总体选项列表

后 缀 名	所对应的语言
-c	只是编译不链接，生成目标文件 “.o”
-S	只是编译不汇编，生成汇编代码
-E	只进行预编译，不做其他处理
-g	在可执行程序中包含标准调试信息



-o file	把输出文件输出到 file 里
-v	打印出编译器内部编译各过程的命令行信息和编译器的版本
-I dir	在头文件的搜索路径列表中添加 dir 目录
-L dir	在库文件的搜索路径列表中添加 dir 目录
-static	链接静态库
-llibrary	连接名为 library 的库文件

对于“-c”、“-E”、“-o”、“-S”选项在前一小节中已经讲解了其使用方法，在此主要讲解另外两个非常常用的库依赖选项“-I dir”和“-L dir”。

6.2 实验目的

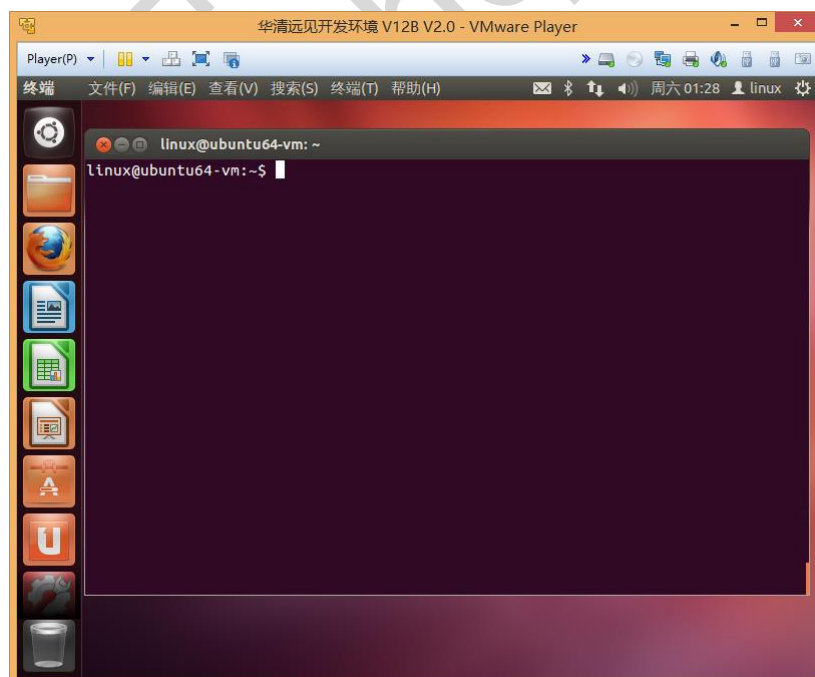
通过实验学习 gcc 编译器编译 c 程序的方法，熟悉 gcc 编译程序的各个阶段。

6.3 实验平台

华清远见开发环境，FS4412 开发板

6.4 实验步骤

6.4.1 准备环境





6.4.2 拷贝代码

将【华清远见-CORTEXA9 资料：程序源码\Linux 应用实验源码\03. Linux 系统 GCC 编译器的使用实验\实验代码】目录拷贝到虚拟机共享目录下。

(注意：此步骤已经在 5.4.1 中完成，如果没有完成请参照此章节。省略此部分说明)

将 gcc 部分代码拷贝到虚拟机 Linux 下。

1、建立相关目录；

```
$ cd workdir/linux/application
$ mkdir 6-gcc
```

2、将代码从共享目录拷入虚拟机 Linux 操作系统下；（可使用【ctrl+空格】切换输入法）

```
$ cp /mnt/hgfs/share/实验代码/03.Linux系统GCC编译器的使用实验/实验代码/* 6-gcc/ -a
$ cd 6-gcc/
```

```
linux@ubuntu64-vm:~/workdir/linux/application$ cp /mnt/hgfs/share/实验代码/03.
Linux系统GCC编译器的使用实验/实验代码/* 6-gcc/ -a
linux@ubuntu64-vm:~/workdir/linux/application$ cd 6-gcc/
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$ ls
hello helloworld.c
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$
```

6.4.3 编译代码

```
$ arm-none-linux-gnueabi-gcc helloworld.c -o hello
$ mkdir /source/rootfs/app
$ cp hello /source/rootfs/app/
```

```
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$ arm-cortex_a8-linux-gnueabi-gcc helloworld.c -o hello
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$ ls
hello helloworld.c
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$
```

```
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$ mkdir /source/rootfs/app
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$ cp hello /source/rootfs/app/
linux@ubuntu64-vm:~/workdir/linux/application/6-gcc$
```

6.4.4 执行代码

按照 3.4.10 章节的步骤，通过 tftp 下载内核，nfs 挂载文件系统，启动开发板。



```
COM4 - PuTTY
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright (c) Pierre Ossman
s3c-sdhci s3c-sdhci.0: clock source 0: hsmmc (133400000 Hz)
s3c-sdhci s3c-sdhci.0: clock source 2: sclk_mmc (96000000 Hz)
mmc0: SDHCI controller on samsung-hsmmc [s3c-sdhci.0] using ADMA
s3c-sdhci s3c-sdhci.1: clock source 0: hsmmc (133400000 Hz)
s3c-sdhci s3c-sdhci.1: clock source 2: sclk_mmc (96000000 Hz)
mmc1: SDHCI controller on samsung-hsmmc [s3c-sdhci.1] using ADMA
s3c-sdhci s3c-sdhci.2: clock source 0: hsmmc (133400000 Hz)
s3c-sdhci s3c-sdhci.2: clock source 2: sclk_mmc (96000000 Hz)
mmc2: SDHCI controller on samsung-hsmmc [s3c-sdhci.2] using ADMA
s3c-sdhci s3c-sdhci.3: clock source 0: hsmmc (133400000 Hz)
s3c-sdhci s3c-sdhci.3: clock source 2: sclk_mmc (96000000 Hz)
mmc3: SDHCI controller on samsung-hsmmc [s3c-sdhci.3] using ADMA
TCP cubic registered
NET: Registered protocol family 17
VFP support v0.3: implementor 41 architecture 3 part 30 variant c rev 2
dm9000 dm9000: eth0: link down
mmc0: new high speed SDHC card at address 1234
mmcblk0: mmc0:1234 SA04G 3.63 GiB
mmcblk0: p1 p2 p3 p4
usb 1-1: new full-speed USB device number 2 using s5p-ohci
hub 1-1:1.0: USB hub found
hub 1-1:1.0: 4 ports detected
IP-Config: Guessing netmask 255.255.255.0
IP-Config: Complete:
    device=eth0, addr=192.168.1.191, mask=255.255.255.0, gw=255.255.255.255,
    host=192.168.1.191, domain=, nis-domain=(none),
    bootserver=255.255.255.255, rootserver=192.168.1.192, rootpath=
dm9000 dm9000: eth0: link up, 100Mbps, full-duplex, lpa 0x4DE1
VFS: Mounted root (nfs filesystem) on device 0:10.
Freeing init memory: 164K
[root@farsight /]#
```

在开发板串口终端执行应用程序。

```
# cd /app
# ./hello
```

6.4.5 实验结果

```
VFS: Mounted root (nfs filesystem) on device 0:10.
Freeing init memory: 164K
[root@farsight /]# ls
app      dev      lib      mnt      root     sys      usr
bin      etc      linuxrc  proc     sbin     tmp      var
[root@farsight /]# cd app/
[root@farsight /app]# ls
hello
[root@farsight /app]# ./hello
hello,world!
[root@farsight /app]#
```

6.5 相关代码

C++ Code



```
1  #include <stdio.h>
2
3  int main (int argc, char **argv)
4  {
5      printf("hello,world!\n");
6
7      return 0;
8  }
9
```

华清远见
dev.hqyj.com



第 7 章 Linux 系统 Makefile 编写实验

7.1 实验原理

到此为止，读者已经了解了如何在 Linux 下使用编辑器编写代码，如何使用 gcc 把代码编译成可执行文件，那么，所有的工作看似已经完成了，为什么还需要 Make 这个工程管理器呢？

所谓工程管理器，顾名思义，是指管理较多的文件的。读者可以试想一下，有一个上百个文件的代码构成的项目，如果其中只有一个或少数几个文件进行了修改，按照之前所学的 gcc 编译工具，就不得不把这所有的文件重新编译一遍，因为编译器并不知道哪些文件是最近更新的，而只知道需要包含这些文件才能把源代码编译成可执行文件，于是，程序员就不能不再重新输入数目如此庞大的文件名以完成最后的编译工作。

人们就希望有一个工程管理器能够自动识别更新了的文件代码，同时又不需要重复输入冗长的命令行，这样，Make 工程管理器也就应运而生了。

实际上，Make 工程管理器也就是个“自动编译管理器”，这里的“自动”是指它能够根据文件时间戳自动发现更新过的文件而减少编译的工作量，同时，它通过读入 Makefile 文件的内容来执行大量的编译工作。用户只需编写一次简单的编译语句就可以了。它大大提高了实际项目的工作效率，而且几乎所有 Linux 下的项目编程均会涉及它，希望读者能够认真学习本节内容。

7.1.1 Makefile 基本结构

Makefile 是 Make 读入的惟一配置文件，因此本节的内容实际就是讲述 Makefile 的编写规则。在一个 Makefile 中通常包含如下内容：

- 需要由 make 工具创建的目标体 (target)，通常是目标文件或可执行文件；
- 要创建的目标体所依赖的文件 (dependency_file)；
- 创建每个目标体时需要运行的命令 (command)。

它的格式为：

```
target: dependency_files
      command
```

例如，有两个文件分别为 hello.c 和 hello.h，创建的目标体为 hello.o，执行的命令为 gcc 编译指令：gcc -c hello.c，那么，对应的 Makefile 就可以写为：

```
#The simplest example
hello.o: hello.c hello.h
      gcc -c hello.c -o hello.o
```

接着就可以使用 make 了。使用 make 的格式为：make target，这样 make 就会自动读入 Makefile（也可以是首字母小写 makefile）并执行对应 target 的 command 语句，并会找到相应的依赖文件。如下所示：

```
[root@localhost makefile]# make hello.o
gcc -c hello.c -o hello.o
[root@localhost makefile]# ls
hello.c hello.h hello.o Makefile
```



可以看到，Makefile 执行了“hello.o”对应的命令语句，并生成了“hello.o”目标体。

7.1.2 Makefile 变量

上面示例的 Makefile 在实际中是几乎不存在的，因为它过于简单，仅包含两个文件和一个命令，在这种情况下完全不必要编写 Makefile 而只需在 Shell 中直接输入即可，在实际中使用的 Makefile 往往是包含很多的文件和命令的，这也是 Makefile 产生的原因。下面就给出稍微复杂一些的 Makefile 进行讲解：

```
sunq:kang.o yul.o
Gcc kang.o bar.o -o myprog
kang.o : kang.c kang.h head.h
Gcc -Wall -O -g -c kang.c -o kang.o
yul.o : bar.c head.h
Gcc -Wall -O -g -c yul.c -o yul.o
```

在这个 Makefile 中有 3 个目标体（target），分别为 sunq、kang.o 和 yul.o，其中第一个目标体的依赖文件就是后两个目标体。如果用户使用命令“make sunq”，则 make 管理器就是找到 sunq 目标体开始执行。

这时，make 会自动检查相关文件的时间戳。首先，在检查“kang.o”、“yul.o”和“sunq”3 个文件的时间戳之前，它会向下查找那些把“kang.o”或“yul.o”作为目标文件的时间戳。比如，“kang.o”的依赖文件为“kang.c”、“kang.h”、“head.h”。如果这些文件中任何一个的时间戳比“kang.o”新，则命令“gcc -Wall -O -g -c kang.c -o kang.o”将会执行，从而更新文件“kang.o”。在更新完“kang.o”或“yul.o”之后，make 会检查最初的“kang.o”、“yul.o”和“sunq”3 个文件，只要文件“kang.o”或“yul.o”中的任比文件时间戳比“sunq”新，则第二行命令就会被执行。这样，make 就完成了自动检查时间戳的工作，开始执行编译工作。这也就是 Make 工作的基本流程。

接下来，为了进一步简化编辑和维护 Makefile，make 允许在 Makefile 中创建和使用变量。变量是在 Makefile 中定义的名字，用来代替一个文本字符串，该文本字符串称为该变量的值。在具体要求下，这些值可以代替目标体、依赖文件、命令以及 makefile 文件中其他部分。在 Makefile 中的变量定义有两种方式：一种是递归展开方式，另一种是简单方式。

递归展开方式定义的变量是在引用在该变量时进行替换的，即如果该变量包含了对其他变量的应用，则在引用该变量时一次性将内嵌的变量全部展开，虽然这种类型的变量能够很好地完成用户的指令，但是它也有严重的缺点，如不能在变量后追加内容（因为语句：CFLAGS=\$(CFLAGS) -O 在变量扩展过程中可能导致无穷循环）。

为了避免上述问题，简单扩展型变量的值在定义处展开，并且只展开一次，因此它不包含任何对其他变量的引用，从而消除变量的嵌套引用。

递归展开方式的定义格式为：VAR=var。

简单扩展方式的定义格式为：VAR:=var。

Make 中的变量使用均使用格式为：\$(VAR)。

下面给出了上例中用变量替换修改后的 Makefile，这里用 OBJS 代替 kang.o 和 yul.o，用 CC 代替 gcc，用 CFLAGS 代替“-Wall -O -g”。这样在以后修改时，就可以只修改变量定义，而不需要修改下面的定义实体，从而大大简化了 Makefile 维护的工作量。

经变量替换后的 Makefile 如下所示：

```
OBJS = kang.o yul.o
CC = gcc
```



```
CFLAGS = -Wall -O -g
sung : $(OBJS)
    $(CC) $(OBJS) -o sung
kang.o : kang.c kang.h
    $(CC) $(CFLAGS) -c kang.c -o kang.o
yul.o : yul.c yul.h
    $(CC) $(CFLAGS) -c yul.c -o yul.o
```

可以看到，此处变量是以递归展开方式定义的。

Makefile 中的变量分为用户自定义变量、预定义变量、自动变量及环境变量。如上例中的 **OBJS** 就是用户自定义变量，自定义变量的值由用户自行设定，而预定义变量和自动变量为通常在 **Makefile** 都会出现的变量，其中部分有默认值，也就是常见的设定值，当然用户可以对其进行修改。

预定义变量包含了常见编译器、汇编器的名称及其编译选项。表 3.11 列出了 **Makefile** 中常见预定义变量及其部分默认值。

表 3.11 Makefile 中常见预定义变量

命 令 格 式	含 义
AR	库文件维护程序的名称，默认值为 ar
AS	汇编程序的名称，默认值为 as
CC	C 编译器的名称，默认值为 cc
CPP	C 预编译器的名称，默认值为 \$(CC) -E
CXX	C++编译器的名称，默认值为 g++
FC	FORTTRAN 编译器的名称，默认值为 f77
RM	文件删除程序的名称，默认值为 rm -f
ARFLAGS	库文件维护程序的选项，无默认值
ASFLAGS	汇编程序的选项，无默认值
CFLAGS	C 编译器的选项，无默认值
CPPFLAGS	C 预编译的选项，无默认值
CXXFLAGS	C++编译器的选项，无默认值
FFLAGS	FORTTRAN 编译器的选项，无默认值



可以看出，上例中的 CC 和 CFLAGS 是预定义变量，其中由于 CC 没有采用默认值，因此，需要把“CC=gcc”明确列出来。

由于常见的 gcc 编译语句中通常包含了目标文件和依赖文件，而这些文件在 Makefile 文件中目标体的一行已经有所体现，因此，为了进一步简化 Makefile 的编写，就引入了自动变量。自动变量通常可以代表编译语句中出现目标文件和依赖文件等，并且具有本地含义（即下一语句中出现的相同变量代表的是下一语句的目标文件和依赖文件）。表 3.12 列出了 Makefile 中常见自动变量。

表 3.12 Makefile 中常见自动变量

命令格式	含 义
\$*	不包含扩展名的目标文件名称
\$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
\$<	第一个依赖文件的名称
\$?	所有时间戳比目标文件晚的依赖文件，并以空格分开
续表	
命令格式	含 义
\$@	目标文件的完整名称
\$^	所有不重复的依赖文件，以空格分开
%	如果目标是归档成员，则该变量表示目标的归档成员名称

自动变量的书写比较难记，但是在熟练了之后会非常的方便，请读者结合下例中的自动变量改写的 Makefile 进行记忆。

```
OBJS = kang.o yul.o
CC = Gcc
CFLAGS = -Wall -O -g
sunq : $(OBJS)
    $(CC) $^ -o $@
kang.o : kang.c kang.h
    $(CC) $(CFLAGS) -c $< -o $@
yul.o : yul.c yul.h
    $(CC) $(CFLAGS) -c $< -o $@
```

另外，在 Makefile 中还可以使用环境变量。使用环境变量的方法相对比较简单，make 在启动时会自动读取系统当前已经定义了的环境变量，并且会创建与之具有相同名称和数值的变量。但是，如果用户在 Makefile 中定义了相同名称的变量，那么用户自定义变量将会覆盖同名的环境变量。



7.1.3 Makefile 规则

Makefile 的规则是 Make 进行处理的依据，它包括了目标体、依赖文件及其之间的命令语句。一般的，Makefile 中的一条语句就是一个规则。在上面的例子中，都显示地指出了 Makefile 中的规则关系，如 “\$(CC) \$(CFLAGS) -c \$< -o \$@”，但为了简化 Makefile 的编写，make 还定义了隐式规则和模式规则，下面就分别对其进行讲解。

1. 隐式规则

隐含规则能够告诉 make 怎样使用传统的技术完成任务，这样，当用户使用它们时就不必详细指定编译的具体细节，而只需把目标文件列出即可。Make 会自动搜索隐式规则目录来确定如何生成目标文件。如上例就可以写成：

```
OBJS = kang.o yul.o
CC = Gcc
CFLAGS = -Wall -O -g
suno : $(OBJS)
        $(CC) $^ -o $@
```

为什么可以省略后两句呢？因为 Make 的隐式规则指出：所有 “.o” 文件都可自动由 “.c” 文件使用命令 “\$(CC) \$(CPPFLAGS) \$(CFLAGS) -c file.c -o file.o” 生成。这样 “kang.o” 和 “yul.o” 就会分别调用 “\$(CC) \$(CFLAGS) -c kang.c -o kang.o” 和 “\$(CC) \$(CFLAGS) -c yul.c -o yul.o” 生成。

表 3.13 给出了常见的隐式规则目录：

表 3.13 Makefile 中常见隐式规则目录

对应语言后缀名	规 则
C 编译: .c 变为.o	\$(CC) -c \$(CPPFLAGS) \$(CFLAGS)
C++编译: .cc 或.C 变为.o	\$(CXX) -c \$(CPPFLAGS) \$(CXXFLAGS)
Pascal 编译: .p 变为.o	\$(PC) -c \$(PFLAGS)
Fortran 编译: .f 变为.o	\$(FC) -c \$(FFLAGS)

2. 模式规则

模式规则是用来定义相同处理规则的多个文件的。它不同于隐式规则，隐式规则仅仅能够用 make 默认的变量来进行操作，而模式规则还能引入用户自定义变量，为多个文件建立相同的规则，从而简化 Makefile 的编写。

模式规则的格式类似于普通规则，这个规则中的相关文件前必须用 “%” 标明。使用模式规则修改后的 Makefile 的编写如下：

```
OBJS = kang.o yul.o
CC = Gcc
CFLAGS = -Wall -O -g
suno : $(OBJS)
```



```
$ (CC) $^ -o $@
%.o : %.c
$ (CC) $ (CFLAGS) -c $< -o $@
```

7.1.4 Make 管理器的使用

使用 Make 管理器非常简单，只需在 make 命令的后面键入目标名即可建立指定的目标，如果直接运行 make，则建立 Makefile 中的第一个目标。

此外 make 还有丰富的命令行选项，可以完成各种不同的功能。下表 3.17 列出了常用的 make 命令行选项。

表 3.14 make 的命令行选项

命令格式	含 义
-C dir	读入指定目录下的 Makefile
-f file	读入当前目录下的 file 文件作为 Makefile
命令格式	含 义
-i	忽略所有的命令执行错误
-I dir	指定被包含的 Makefile 所在目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 make 变量数据库和隐含规则
-s	在执行命令时不显示命令
-w	如果 make 在执行过程中改变目录，则打印当前目录名

续表

7.2 实验目的

通过对包含多文件的 Makefile 的编写，熟悉各种形式的 Makefile，并且进一步加深对 Makefile 中用户自定义变量、自动变量及预定义变量的理解。

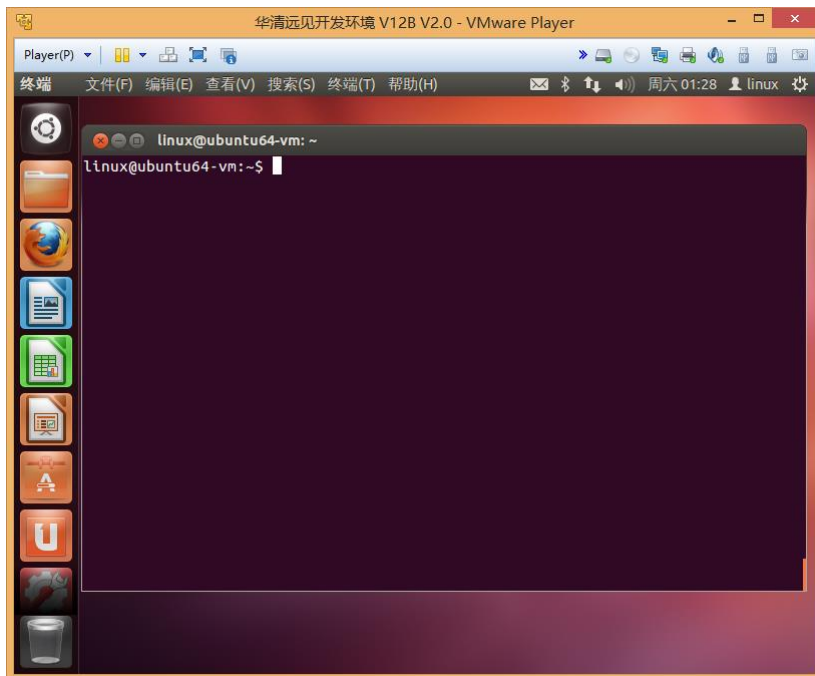
7.3 实验平台

华清远见开发环境，FS4412 开发板



7.4 实验步骤

7.4.1 环境准备



7.4.2 拷贝代码

将【华清远见-CORTEXA9 资料：程序源码\Linux 应用实验源码\05. Linux 系统 Makefile 编写实验/实验代码/5.2/makefileTest】目录拷贝到虚拟机共享目录下。

(注意：此步骤已经在 5.4.1 中完成，如果没有完成请参照此章节。省略此部分说明)

将 Makefile 部分代码拷贝到虚拟机 Linux 下。

1、建立相关目录；

```
$ cd workdir/linux/application
$ mkdir 7-Makefile
```

2、将代码从共享目录拷入虚拟机 Linux 操作系统下；（可使用【ctrl+空格】切换输入法）

```
$ cp /mnt/hgfs/share/实验代码/05.Linux 系统 Makefile 编写实验/实验代码/5.2/makefileTest/* 7-
Makefile/ -a
$ cd 7-Makefile/
```

```
linux@ubuntu64-vm:~/workdir/linux/application$ cd ..
linux@ubuntu64-vm:~/workdir/linux/application$ cp /mnt/hgfs/share/2. Linux系统部分/05. Linux系统Makefile编写实验/实验代码/makefileTest
/ 7-Makefile/ -a
linux@ubuntu64-vm:~/workdir/linux/application$
```



7.4.3 执行代码

进入 makefileTest 目录，执行 make。

```
$ make CLEAN
```

```
$ make
```

会出现如下信息：

```
linux@ubuntu64-vm:~/workdir/linux/application/7-Makefile/makefileTest$ make
mkdir -p bin
f1 f2 main obj
begin compile
make -C f1
make[1]: 正在进入目录 `/home/linux/workdir/linux/application/7-Makefile/makefileTest/f1'
make[1]: “../obj/f1.o”是最新的。
make[1]:正在离开目录 `/home/linux/workdir/linux/application/7-Makefile/makefileTest/f1'
make -C f2
make[1]: 正在进入目录 `/home/linux/workdir/linux/application/7-Makefile/makefileTest/f2'
make[1]: “../obj/f2.o”是最新的。
make[1]:正在离开目录 `/home/linux/workdir/linux/application/7-Makefile/makefileTest/f2'
make -C main
make[1]: 正在进入目录 `/home/linux/workdir/linux/application/7-Makefile/makefileTest/main'
make[1]: “../obj/main.o”是最新的。
make[1]:正在离开目录 `/home/linux/workdir/linux/application/7-Makefile/makefileTest/main'
make -C obj
make[1]: 正在进入目录 `/home/linux/workdir/linux/application/7-Makefile/makefileTest/obj'
make[1]: “../bin/myapp”是最新的。
make[1]:正在离开目录 `/home/linux/workdir/linux/application/7-Makefile/makefileTest/obj'
cp bin/myapp /source/rootfs/app
linux@ubuntu64-vm:~/workdir/linux/application/7-Makefile/makefileTest$ ls /source/rootfs/app/
hello  myapp
```

目录树结构如下：

```
-- Makefile
-- bin
--   |-- myapp
-- f1
--   |-- Makefile
--   |-- f1.c
-- f2
--   |-- Makefile
--   |-- f2.c
-- include
--   |-- myinclude.h
-- main
--   |-- Makefile
--   |-- main.c
-- obj
--   |-- Makefile
--   |-- f1.o
--   |-- f2.o
--   |-- main.o

6 directories, 13 files
```

我们看到在 bin 目录下生成了我们的目标文件 myapp，并且拷贝到了 /source/rootfs/app 下，在 obj 目录下生成了.o 的中间文件。让我们在开发板运行下 myapp 看下结果吧。



按照 3.4.10 章节的步骤，通过 tftp 下载内核，nfs 挂载文件系统，启动开发板。

```
COM4 - PuTTY
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
s3c-sdhci s3c-sdhci.0: clock source 0: hsmmc (133400000 Hz)
s3c-sdhci s3c-sdhci.0: clock source 2: sclk_mmc (96000000 Hz)
mmc0: SDHCI controller on samsung-hsmmc [s3c-sdhci.0] using ADMA
s3c-sdhci s3c-sdhci.1: clock source 0: hsmmc (133400000 Hz)
s3c-sdhci s3c-sdhci.1: clock source 2: sclk_mmc (96000000 Hz)
mmc1: SDHCI controller on samsung-hsmmc [s3c-sdhci.1] using ADMA
s3c-sdhci s3c-sdhci.2: clock source 0: hsmmc (133400000 Hz)
s3c-sdhci s3c-sdhci.2: clock source 2: sclk_mmc (96000000 Hz)
mmc2: SDHCI controller on samsung-hsmmc [s3c-sdhci.2] using ADMA
s3c-sdhci s3c-sdhci.3: clock source 0: hsmmc (133400000 Hz)
s3c-sdhci s3c-sdhci.3: clock source 2: sclk_mmc (96000000 Hz)
mmc3: SDHCI controller on samsung-hsmmc [s3c-sdhci.3] using ADMA
TCP cubic registered
NET: Registered protocol family 17
VFP support v0.3: implementor 41 architecture 3 part 30 variant c rev 2
dm9000 dm9000: eth0: link down
mmc0: new high speed SDHC card at address 1234
mmcblk0: mmc0:1234 SA04G 3.63 GiB
  mmcblk0: p1 p2 p3 p4
usb 1-1: new full-speed USB device number 2 using s5p-ohci
hub 1-1:1.0: USB hub found
hub 1-1:1.0: 4 ports detected
IP-Config: Guessing netmask 255.255.255.0
IP-Config: Complete:
    device=eth0, addr=192.168.1.191, mask=255.255.255.0, gw=255.255.255.255,
    host=192.168.1.191, domain=, nis-domain=(none),
    bootserver=255.255.255.255, rootserver=192.168.1.192, rootpath=
dm9000 dm9000: eth0: link up, 100Mbps, full-duplex, lpa 0x4DE1
VFS: Mounted root (nfs filesystem) on device 0:10.
Freeing init memory: 164K
[root@farsight /]#
```

在开发板串口终端执行应用程序。

```
# cd app
# ./myapp
```

```
Message from f1.c...
Message from f2.c...
[root@farsight /app]# ./myapp
Message from f1.c...
Message from f2.c...
[root@farsight /app]#
```

我们也可以用如下命令清除中间文件和目标文件，恢复 make 之前的状态，在虚拟机下输入：

```
$ make CLEAN
```



```
-- Makefile
-- f1
|-- Makefile
|-- f1.c
-- f2
|-- Makefile
|-- f2.c
-- include
|-- myinclude.h
-- main
|-- Makefile
|-- main.c
-- obj
|-- Makefile
```

5 directories, 9 files

我们可以看到已经变为 make 之前的目录状态了。

7.5 相关代码

```
CC = arm-none-linux-gnueabi-gcc
SUBDIRS = f1 \
        f2 \
        main \
        obj

OBJS = f1.o f2.o main.o
BIN = myapp
OBJS_DIR = obj
BIN_DIR = bin
export CC OBJS BIN OBJS_DIR BIN_DIR

all : CHECK_DIR $(SUBDIRS)
    cp bin/myapp /source/rootfs/app
CHECK_DIR :
    mkdir -p $(BIN_DIR)
$(SUBDIRS) : ECHO
    make -C $@
ECHO:
    @echo $(SUBDIRS)
    @echo begin compile
CLEAN :
```



```
@$(RM) $(OBS_DIR)/*.o
```

```
@rm -rf $(BIN_DIR)
```

华清远见
dev.hqyj.com



第 8 章 嵌入式文件 I/O 编程实验

在 Linux 系统中，大部分机制都会抽象成一个文件，这样对它们的操作就像对文件的操作一样。在嵌入式应用开发中，文件 I/O 编程是最常用的，也是最基本的内容，希望读者好好掌握。

8.1 实验原理

8.1.1 系统调用

所谓系统调用是指操作系统提供给用户程序调用的一组“特殊”接口，用户程序可以通过这组“特殊”接口获得操作系统内核提供的服务。例如用户可以通过进程控制相关的系统调用来创建进程、实现进程之间的通信等。

在这里，为什么用户程序不能直接访问系统内核提供的服务呢？这是由于在 Linux 中，为了更好地保护内核空间，将程序的运行空间分为内核空间和用户空间（也就是常称的内核态和用户态），它们分别运行在不同的级别上，逻辑上是相互隔离的。因此，用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。

但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供给用户的“特殊接口”——系统调用规定用户进程进入内核空间的具体位置。进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回到用户空间。

Linux 系统调用非常精简（只有 250 个左右），它继承了 Unix 系统调用中最基本和最有用的部分。这些系统调用按照功能逻辑大致可分为进程控制、进程间通信、文件系统控制、存储管理、网络管理、套接字控制、用户管理等几类。

8.1.2 用户编程接口（API）

前面讲到的系统调用并不直接与程序员进行交互，它仅仅是一个通过软中断机制向内核提交请求以获取内核服务的接口。实际使用中程序员调用的通常是用户编程接口——API。

例如创建进程的 API 函数 `fork()` 对应于内核空间的 `sys_fork()` 系统调用。但并不是所有的函数都对应一个系统调用，有时，一个 API 函数会需要几个系统调用来共同完成函数的功能，甚至还有一些 API 函数不需要调用相应的系统调用（因此它所完成的不是内核提供的服务）。

在 Linux 中，用户编程接口（API）遵循了在 Unix 中最流行的应用编程界面标准——POSIX 标准。POSIX 标准是由 IEEE 和 ISO/IEC 共同开发的标准系统。该标准基于当时现有的 Unix 实践和经验，描述了操作系统的系统调用编程接口（实际上就是 API），用于保证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。这些系统调用编程接口主要是通过 C 库（libc）实现的。

8.1.3 系统命令

系统命令相对 API 更高了一层，它实际上一个可执行程序，它的内部引用了用户编程接口（API）来实现相应的功能。它们之间的关系如下图 4.1 所示。

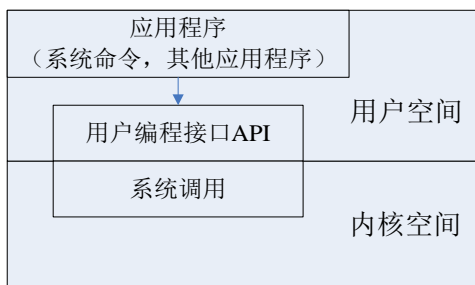


图 4.1 系统调用、API 及系统命令之间的关系

8.1.4 虚拟文件系统 (VFS)

Linux 系统成功的关键因素之一就是具有与其他操作系统和谐共存的能力。Linux 的文件系统由两层结构构建。第一层是虚拟文件系统 (VFS)，第二层是各种不同的具体的文件系统。

VFS 就是把各种具体的文件系统的公共部分抽取出来，形成一个抽象层，是系统内核的一部分。它位于用户程序和具体的文件系统之间。它对用户程序提供了标准的文件系统调用接口，对具体的文件系统（比如：Ext2，FAT32 等），它通过一系列的对不同文件系统公用的函数指针来实际调用具体的文件系统函数，完成实际的各有差异的操作。任何使用文件系统的程序必须经过这层接口来使用它。通过这样的方式，VFS 就对用户屏蔽了底层文件系统的实现细节和差异。

VFS 不仅可以对具体文件系统的数据结构进行抽象，以一种统一的数据结构进行管理，并且还可以接受用户层的系统调用，例如：open()、read()、write()、stat()、link()等。此外，它还支持多种具体文件系统之间的相互访问，接受内核其他子系统的操作请求，例如内存管理和进程调度。VFS 在 Linux 系统中的位置如图 4.2 所示。

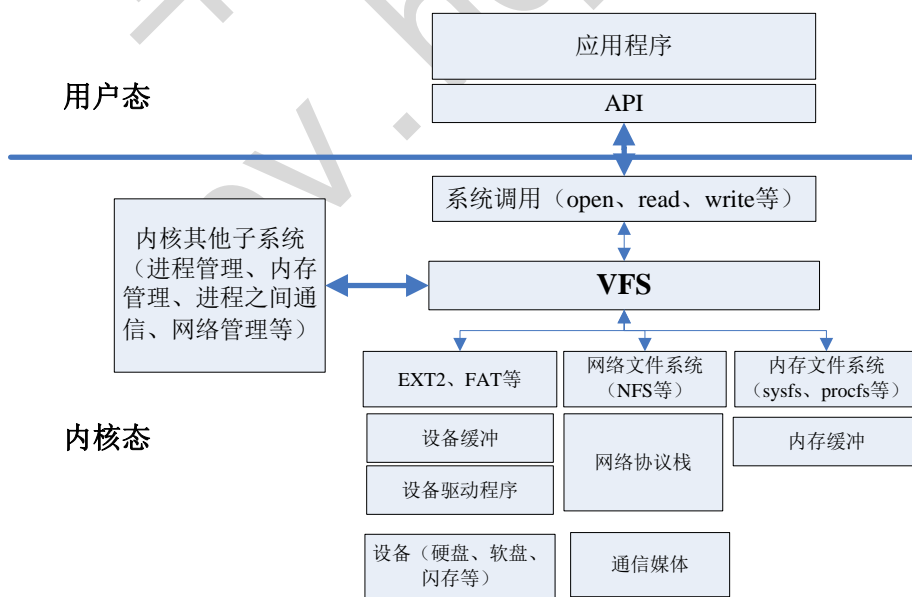


图 4.2 vfs 在 Linux 系统中的位置

通过以下命令可以查看系统中支持哪些文件系统：



```
$ cat /proc/filesystems
```

```
nodev    sysfs
nodev    rootfs
.....
nodev    tmpfs
nodev    pipefs
.....
        ext2
nodev    ramfs
nodev    hugetlbfs
        iso9660
nodev    mqueue
nodev    selinuxfs
        ext3
nodev    rpc_pipefs
.....
```

8.1.5 Linux 中文件及文件描述符

Linux 操作系统都是基于文件概念的。文件是以字符序列而构成的信息载体。根据这一点，可以把 I/O 设备当作文件来处理。因此，与磁盘上的普通文件进行交互所用的同一系统调用可以直接用于 I/O 设备。这样大大简化了系统对不同设备的处理，提高了效率。Linux 中的文件主要分为 4 种：普通文件、目录文件、链接文件和设备文件。

那么，内核如何区分和引用特定的文件呢？这里用到了一个重要的概念——文件描述符。对于 Linux 而言，所有对设备和文件的操作都是使用文件描述符来进行的。文件描述符是一个非负的整数，它是一个索引值，并指向在内核中每个进程打开文件的记录表。当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符；当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数。

通常，一个进程启动时，都会打开 3 个文件：标准输入、标准输出和标准出错处理。这 3 个文件分别对应文件描述符为 0、1 和 2（也就是宏替换 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`，鼓励读者使用这些宏替换）。

基于文件描述符的 I/O 操作虽然不能直接移植到类 Linux 以外的系统上去（如 Windows），但它往往是实现某些 I/O 操作的惟一途径，如 Linux 中低层文件操作函数、多路 I/O、TCP/IP 套接字编程接口等。同时，它们也很好地兼容 POSIX 标准，因此，可以很方便地移植到任何 POSIX 平台上。基于文件描述符的 I/O 操作是 Linux 中最常用的操作之一，希望读者能够很好地掌握。

8.1.6 标准 I/O 编程

本章前面几节所述的文件及 I/O 读写都是基于文件描述符的。这些都是基本的 I/O 控制，是不带缓存



的。而本节所要讨论的 I/O 操作都是基于流缓冲的，它是符合 ANSI C 的标准 I/O 处理，这里有很多函数读者已经非常熟悉了（如 `printf()`、`scanf()` 函数等），因此本节中仅简要介绍最主要的函数。

前面讲述的系统调用是操作系统直接提供的函数接口。因为运行系统调用时，Linux 必须从用户态切换到内核态，执行相应的请求，然后再返回到用户态，所以应该尽量减少系统调用的次数，从而提高程序的效率。

标准 I/O 提供流缓冲的目的是尽可能减少使用 `read()` 和 `write()` 等系统调用的数量。标准 I/O 提供了 3 种类型的缓冲存储。

- 全缓冲：在这种情况下，当填满标准 I/O 缓存后才进行实际 I/O 操作。对于存放在磁盘上的文件通常是由标准 I/O 库实施全缓冲的。标准 I/O 尽量多读写文件到缓冲区，当缓冲区已满或手动 `flush` 时才会进行磁盘操作。

- 行缓冲：在这种情况下，当在输入和输出中遇到行结束符时，标准 I/O 库执行 I/O 操作。这允许我们一次输出一个字符（如 `fputc()` 函数），但只有写了一行之后才进行实际 I/O 操作。标准输入和标准输出就是使用行缓冲的典型例子。

- 不带缓冲：标准 I/O 库不对字符进行缓冲。如果用标准 I/O 函数写若干字符到不带缓冲的流中，则相当于用系统调用 `write()` 函数将这些字符全写到被打开的文件上。标准出错 `stderr` 通常是不带缓存的，这就使得出错信息可以尽快显示出来，而不管它们是否含有一个行结束符。

在下面讨论具体函数时，请读者注意区分以上的三种不同情况。

1. 打开文件

打开文件有三个标准函数，分别为：`fopen()`、`fdopen()` 和 `freopen()`。它们可以以不同的模式打开，但都返回一个指向 `FILE` 的指针，该指针指向对应的 I/O 流。此后，对文件的读写都是通过这个 `FILE` 指针来进行。其中 `fopen()` 可以指定打开文件的路径和模式，`fdopen()` 可以指定打开的文件描述符和模式，而 `freopen()` 除可指定打开的文件、模式外，还可指定特定的 I/O 流。

`fopen()` 函数格式如表 4.16 所示。

表 4.16 `fopen()` 函数语法要点

所需头文件	<code>#include <stdio.h></code>
函数原型	<code>FILE * fopen(const char * path, const char * mode)</code>
函数传入值	<code>path</code> : 包含要打开的文件路径及文件名
	<code>mode</code> : 文件打开状态，详细信息参考表 2.17
函数返回值	成功: 指向 <code>FILE</code> 的指针
	失败: <code>NULL</code>

其中，`mode` 类似于 `open()` 函数中的 `flag`，可以定义打开文件的访问权限等，表 4.17 说明了 `fopen()` 中 `mode` 的各种取值。

表 4.17 `mode` 取值说明



r 或 rb	打开只读文件，该文件必须存在
r+ 或 r +b	打开可读写的文件，该文件必须存在
w 或 wb	打开只写文件，若文件存在则文件长度清为 0，即会擦写文件以前的内容。若文件不存在则建立该文件
w+ 或 w +b	打开可读写文件，若文件存在则文件长度清为 0，即会擦写文件以前的内容。若文件不存在则建立该文件
a 或 ab	以附加的方式打开只写文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留
a+ 或 a +b	以附加方式打开可读写的文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留

注意在每个选项中加入 b 字符用来告诉函数库打开的文件为二进制文件，而非纯文本文件。不过在 Linux 系统中会自动识别不同类型的文件而将此符号可以忽略。

fdopen()函数格式如表 4.18 所示。

表 4.18 fdopen()函数语法要点

所需头文件	#include <stdio.h>
函数原型	FILE * fdopen(int fd, const char * mode)
函数传入值	fd: 要打开的文件描述符
	mode: 文件打开状态, , 详细信息参考表 2.17
函数返回值	成功: 指向 FILE 的指针
	失败: NULL

fopen()函数格式如表 4.19 所示。

表 4.19 fopen()函数语法要点

所需头文件	#include <stdio.h>
函数原型	FILE * fopen(const char *path, const char * mode, FILE * stream)
函数传入值	path: 包含要打开的文件路径及文件名
	mode: 文件打开状态, , 详细信息参考表 2.17
	stream: 已打开的文件指针
函数返回值	成功: 指向 FILE 的指针
	失败: NULL

2. 关闭文件



关闭标准流文件的函数为 `fclose()`，该函数将缓冲区内的数据全部写入到文件中，并释放系统所提供的文件资源。

`fclose()`函数格式如表 4.20 所示。

表 4.20 `fclose()`函数语法要点

所需头文件	<code>#include <stdio.h></code>
函数原型	<code>int fclose(FILE * stream)</code>
函数传入值	stream: 已打开的文件指针
函数返回值	成功: 0 失败: EOF

3. 读文件

在文件流被打开之后，可对文件流进行读写等操作，其中读操作的函数为 `fread()`。

`fread()`函数格式如表 2.21 所示。

表 4.21 `fread()`函数语法要点

所需头文件	<code>#include <stdio.h></code>
函数原型	<code>size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream)</code>
函数传入值	ptr: 存放读入记录的缓冲区
	size: 读取的记录大小
	nmemb: 读取的记录数
	stream: 要读取的文件流
函数返回值	成功: 返回实际读取到的 <code>nmemb</code> 数目 失败: EOF

4. 写文件

`fwrite()`函数是用于对指定的文件流进行写操作。`fwrite()`函数格式如表 4.22 所示。

表 4.22 `fwrite()`函数语法要点

所需头文件	<code>#include <stdio.h></code>
函数原型	<code>size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream)</code>
函数传入值	ptr: 存放写入记录的缓冲区
	size: 写入的记录大小
	nmemb: 写入的记录数
	stream: 要写入的文件流



函数返回值	成功：返回实际写入到的 nmemb 数目 失败：EOF
-------	--------------------------------

文件打开之后，根据一次读写文件中字符的数目可分为字符输入输出、行输入输出和格式化输入输出，下面分别对这 3 种不同的方式进行讨论。

1. 字符输入输出

字符输入输出函数一次仅读写一个字符。其中字符输入输出函数如表 4.23 和表 4.24 所示。

表 4.23 字符输出函数语法要点

所需头文件	#include <stdio.h>
函数原型	int getc(FILE * stream) int fgetc(FILE * stream) int getchar(void)
函数传入值	stream: 要输入的文件流
函数返回值	成功：下一个字符 失败：EOF

表 4.24 字符输入函数语法要点

所需头文件	#include <stdio.h>
函数原型	int putc(int c, FILE * stream) int fputc(int c, FILE * stream) int putchar(int c)
函数返回值	成功：字符 c 失败：EOF

这几个函数功能类似，其区别仅在于 getc()和 putc()通常被实现为宏，而 fgetc()和 fputc()不能实现为宏，因此，函数的实现时间会有所差别。

下面这个实例结合 fputc()和 fgetc()，将标准输入复制到标准输出中去。

```
/*fput.c*/
#include<stdio.h>
main()
{
    int c;
    /*把 fgetc()的结果作为 fputc()的输入*/
    fputc(fgetc(stdin), stdout);
}
```

运行结果如下所示：



\$./fput

w (用户输入)

w (屏幕输出)

2. 行输入输出

行输入输出函数一次操作一行。其中行输入输出函数如表 4.25 和表 4.26 所示。

表 4.25 行输出函数语法要点

所需头文件	#include <stdio.h>
函数原型	char * gets(char *s) char fgets(char * s, int size, FILE * stream)
函数传入值	s: 要输入的字符串 size: 输入的字符串长度 stream: 对应的文件流
函数返回值	成功: s 失败: NULL

表 4.26 行输入函数语法要点

所需头文件	#include <stdio.h>
函数原型	int puts(const char *s) int fputs(const char * s, FILE * stream)
函数传入值	s: 要输出的字符串 stream: 对应的文件流
函数返回值	成功: s 失败: NULL

这里以 gets()和 puts()为例进行说明，本实例将标准输入复制到标准输出，如下所示：

```
/*gets.c*/
#include<stdio.h>
main()
{
    char s[80];
    fputs(fgets(s, 80, stdin), stdout);
}
```

运行该程序，结果如下所示：



```
$ ./gets
```

This is stdin（用户输入）

This is stdin（屏幕输出）

3. 格式化输入输出

格式化输入输出函数可以指定输入输出的具体格式，这里有读者已经非常熟悉的 printf()、scanf() 等函数，这里就简要介绍一下它们的格式。如下表 4.25～表 4.27 所示。

表 4.25 格式化输出函数 1

所需头文件	#include <stdio.h>
函数原型	int printf(const char *format,...) int fprintf(FILE *fp, const char *format,...) int sprintf(char *buf, const char *format,...)
函数传入值	format: 记录输出格式 fp: 文件描述符 buf: 记录输出缓冲区
函数返回值	成功: 输出字符数（sprintf 返回存入数组中的字符数） 失败: NULL

表 4.26 格式化输出函数 2

所需头文件	#include <stdarg.h> #include <stdio.h>
函数原型	int vprintf(const char *format, va_list arg) int vfprintf(FILE *fp, const char *format, va_list arg) int vsprintf(char *buf, const char *format, va_list arg)
函数传入值	format: 记录输出格式 fp: 文件描述符 arg: 相关命令参数
函数返回值	成功: 存入数组的字符数 失败: NULL

表 4.27 格式化输入函数

所需头文件	#include <stdio.h>
函数原型	int scanf(const char *format,...) int fscanf(FILE *fp, const char *format,...) int sscanf(char *buf, const char *format,...)



函数传入值	format: 记录输出格式 fp: 文件描述符 buf: 记录输入缓冲区
函数返回值	成功: 输出字符数 (sprintf 返回存入数组中的字符数) 失败: NULL

8.2 标准 IO 实验

8.2.1 实验目的

通过实验学习标准 IO 的编程方法。

8.2.2 实验平台

华清原价开发环境

8.2.3 实验步骤

本实验通过一个简单的程序计算默认缓冲区的大小，理解标准 I/O 提供的三种类型的缓存，这三种缓存类型分别是：

全缓冲：当填满 I/O 缓存后才进行实际的 I/O 操作；

行缓冲：当在输入和输出中遇到新换行符(‘\n’)时，进行 I/O 操作；

不带缓冲：标准 I/O 库不对字符进行缓冲，例如 stderr；

本实验验证全缓存类型。

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\06. Linux 系统标准 IO 实验\实验代码】目录下的“61.c”拷贝到【~/workdir/linux/application/8-io】目录下并编译：

```
$ gcc 61.c -o io_test
```

```
$ ./io_test
```

8.2.4 实验现象

可以看到终端屏幕共输出了 1024 字节大小的内容，多余的内容没有输出，这就验证了全缓存的特点：当填满 I/O 缓存后才进行实际的 I/O 操作。

```
linux@ubuntu64-vm:~/workdir/linux/application/8-io$ gcc 61.c -o io_test
linux@ubuntu64-vm:~/workdir/linux/application/8-io$ ./io_test
0000010020030040050060070080090100110120130140150160170180190200210220230240250260270280290300310320330340350360
3703803904004104204304404504604704804905005105205305405505605705805906006106206306406506606706806907007107207307
4075076077078079080081082083084085086087088089090091092093094095096097098099100101102103104105106107108109110111
1121131141151161171181191201211221231241251261271281291301311321331341351361371381391401411421431441451461471481
4915015115215315415515615715815916016116216316416516616716816917017117217317417517617717817918018118218318418518
6187188189190191192193194195196197198199200201202203204205206207208209210211212213214215216217218219220221222223
2242252262272282292302312322332342352362372382392402412422432442452462472482492502512522532542552562572582592602
6126226326426526626726826927027127227327427527627727827928028128228328428528628728828929029129229329429529629729
8299300301302303304305306307308309310311312313314315316317318319320321322323324325326327328329330331332333334335
3363373383393403
```



8.2.5 实验代码

```
#include <stdio.h>

int main()
{
    int i=0;
    for(i=0;i<379;i++)//每次向缓冲区内写三个字符
    {
        if(i>=100)
            fprintf(stdout,"%d",i);
        else if (i>=10)
            fprintf(stdout,"0%d",i);
        else if (i>=0)
            fprintf(stdout,"00%d",i);
    }
    while(1);//强制执行，如果取消，程序结束时将会输出所有字符，看不到效果了。
}
```

8.3 Linux 系统文件目录操作编程实验

8.3.1 实验目的

通过实验学习用 Linux C 操作文件目录的方法。

8.3.2 实验平台

华清远见开发环境

8.3.3 实验步骤

运行 ubuntu 系统，打开命令行终端。

执行命令,创建实验目录:

```
$ cd ~/workdir/linux/application/8-io
$ mkdir -p dir/example
$ cd dir/example
$ touch 1.c 2.c //在 example 目录下创建两个文件，后面程序会读这两个文件
$ cd ../
```

将【华清远见-CORTEXA9 资料： \程序源码\Linux 应用实验源码\07. Linux 系统文件目录操作编程实验\实验代码\7.2】目录下的“mys.c”文件拷贝到该目录下。

执行命令:



```
$ gcc myls.c -o myls
```

```
$/mysls example
```

8.3.4 实验现象

程序执行后会列出 example 目录下的两个文件。

```
mysls.c:114: 警告：格式 '%d' 需要类型 'int'，但实参 2 的类型为 '__nlink_t'
linux@ubuntu64-vm:~/workdir/linux/application/8-io/dir$ ./mysls example/
2.c 1.c
linux@ubuntu64-vm:~/workdir/linux/application/8-io/dir$ a
```

8.3.5 实验代码

mysls.c

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <sys/types.h>
4#include <sys/stat.h>
5#include <errno.h>
6#include <unistd.h>
7#include <string.h>
8#include <time.h>
9#include <pwd.h>
10#include <grp.h>
11#include <dirent.h>
12
13#define N 128
14
15int aflag = 0, lflag = 0;
16
17void displayfile(char *name, char *showname);
18void displaydir(const char *name);
19
20int main(int argc, char *argv[])
21{
22    struct stat buf;
23    int ch, i;
24
25    if (argc < 2)
26    {
27        fprintf(stdout, "usage: %s filename\n", argv[0]);
28        exit(-1);
29    }
30
31    while ((ch = getopt(argc, argv, "al")) != -1)
```



```
32 {
33     switch(ch)
34     {
35         case 'a':
36             aflag = 1;
37             break;
38         case 'l':
39             lflag = 1;
40             break;
41         default:
42             printf("wrong option %c\n", optopt);
43     }
44 }
45
46 // printf("aflag=%d lflag=%d\n", aflag, lflag);
47
48 for (i = optind; i < argc; i++)
49 {
50     if (stat(argv[i], &buf) == -1)
51     {
52         perror("stat");
53         exit(-1);
54     }
55
56     if (!S_ISDIR(buf.st_mode))
57         displayfile(argv[i], argv[i]);
58     else
59         displaydir(argv[i]);
60 }
61 if (lflag == 0)
62     printf("\n");
63 return 0;
64 }
65
66 void displayfile(char *name, char *showname)
67 {
68     struct stat buf;
69     struct tm *p;
70     int i = 8;
71
72     if (lflag == 0)
73     {
74         printf("%s ", showname);
75         return;
76     }
```



```
76     }
77
78     if (stat(name, &buf) == -1)
79     {
80         perror("stat");
81         exit(-1);
82     }
83
84     switch (buf.st_mode & S_IFMT)
85     {
86     case S_IFSOCK:
87         printf("s");
88         break;
89     case S_IFLNK:
90         printf("l");
91         break;
92     case S_IFREG:
93         printf("-");
94         break;
95     case S_IFBLK:
96         printf("b");
97         break;
98     case S_IFDIR:
99         printf("d");
100        break;
101    case S_IFCHR:
102        printf("c");
103        break;
104    case S_IFIFO:
105        printf("p");
106        break;
107    default:
108        printf("?");
109    }
110    while (i >= 0)
111    {
112        if ((buf.st_mode) >> i & 1)
113        {
114            switch (i % 3)
115            {
116            case 2:
117                printf("r");
118                break;
119            case 1:
```



```
120         printf("w");
121         break;
122     case 0:
123         printf("x");
124         break;
125     }
126 }
127 else
128 {
129     printf("-");
130 }
131 i--;
132 }
133
134 printf(" ");
135
136 /* hard links */
137 printf("%d", buf.st_nlink);
138
139 /* uid gid */
140 printf(" %s %s ", getpwuid(buf.st_uid)->pw_name, getgrgid(buf.st_gid)->gr_name);
141
142 printf("%ld ", buf.st_size);
143
144 p = localtime(&buf.st_mtime);
145 printf("%d-%d-%d %d:%d ", p->tm_year + 1900, p->tm_mon + 1,
146         p->tm_mday, p->tm_hour, p->tm_min);
147 printf("%s\n", showname);
148 }
149 void displaydir(const char *name)
150 {
151     DIR *dir;
152     struct dirent *ditem;
153     char str[N];
154
155     dir = opendir(name);
156     while ((ditem = readdir(dir)) != NULL)
157     {
158         if (strcmp(".", ditem->d_name, 1) == 0 && aflag == 0)
159             continue;
160         sprintf(str, "./%s", ditem->d_name);
161         displayfile(str, ditem->d_name);
162     }
163 }
```



华清远见
dev.hqyj.com



第 9 章 嵌入式 Linux 多任务编程实验

9.1 使用 ps 命令查看进程信息

9.1.1 实验内容

ps 是基本的 Linux 命令，通过本实验，不仅要熟悉 ps 命令方法，更重要的是可以了解 Linux 进程的组成。

9.1.2 实验平台

华清远见开发环境

9.1.3 实验原理

ps: 查看系统中的进程，Linux 中可以使用 ps -aux 查看所有进程。其中 PID 代表进程 ID，TTY 是该进程是由哪个控制台启动的，CMD 则是命令。

如果想列出更详细的信息，则可使用命令：“ps -auxw”。参数 w 表示加宽显示的命令行，参数 w 可以写多次，通常最多写 3 次，表示加宽 3 次，这足以显示很长的命令行了。

9.1.4 实验步骤

在 shell 提示符下输入如下命令，并解释输出的结果：

```
$ ps
```

终端显示：

```
PID TTY TIME CMD
```

```
16767 pts/1 0:00 ps
```

```
18029 pts/1 0:00 bash
```

```
$ ps aux
```

```
PID TTY TIME CMD
```

9.2 使用 proc 文件系统查看进程信息

9.2.1 实验目的

本实验将指导学员了解 proc 文件系统，通过 proc 文件系统查询进程信息，可以扩展到修改系统参数。

9.2.2 实验平台

华清远见开发环境

9.2.3 实验原理

/proc 文件系统是一个虚拟文件系统，通过文件系统接口实现对内核的访问，输出系统运行状态。它以文件系统的形式，为操作系统本身和应用进程之间的通信提供了一个界面，使应用程序能够安全，方



便的获得系统当前的运行状况和内核的内部数据信息，并且可以修改某些系统的配置信息。

9.2.4 实验内容

认识 proc 文件系统的文件和目录：

```
$ cd /proc
```

```
$ ls
```

通过 proc 文件系统查看系统当前进行状态：

```
$ cat /proc/self/status
```

查询文件句柄的当前使用情况：

```
$ cat /proc/sys/fs/file-nr
```

```
426      15252458
```

file-nr 文件显示了三个参数：分配的文件句柄总数、当前使用的文件句柄数以及可以分配的最大文件句柄数。如果需要增大/proc/sys/fs/file-max 中的值，请确保正确设置 ulimit 。对于 2.4.20，通常将其设置为 unlimited 。使用 ulimit 命令来验证 ulimit 设置：

```
$ ulimit
```

```
unlimited
```

通过 proc 文件系统修改内核中预定的一些变量。

修改整个系统中文件句柄的最大数量：

```
$ cat /proc/sys/fs/file-max
```

```
52458
```

```
$ echo 65536 >/proc/sys/fs/file-max
```

```
$ cat /proc/sys/fs/file-max
```

```
65536
```

修改网络 TTL：

```
$ cat /proc/sys/net/ipv4/ip_default_ttl
```

```
64
```

```
$ echo 128 >/proc/sys/net/ipv4/ip_default_ttl
```

```
$ cat /proc/sys/net/ipv4/ip_default_ttl
```

```
128
```

修改系统中最大进程数量：

```
$ cat /proc/sys/kernel/pid_max
```

```
32768
```

```
$ echo 65536 >/proc/sys/kernel/pid_max
```

```
$ cat /proc/sys/kernel/pid_max
```

```
65536
```

修改普通用户的最大 RTC 频率：



```
$ cat /proc/sys/dev/rtc/max-user-freq
64
$ echo 128 >/proc/sys/dev/rtc/max-user-freq
$ cat /proc/sys/dev/rtc/max-user-freq
128
$ cat /proc/cpuinfo      //查看 CPU 信息
$ cat /proc/interrupts   //查看中断信息
$ cat /proc/ioports      //设备 IO 端口
$ cat /proc/meminfo      //内存信息
$ cat /proc/partitions   //所有设备的所有分区
$ cat /proc/pci           //PCI 设备的信息
$ cat /proc/swaps         //所有 Swap 分区的信息
$ cat /proc/version       //Linux 的版本号
```

9.3 使用 fork、exit 和 exec 系统调用编写多进程程序

9.3.1 实验目的

本实验将通过编写 fork 等系统调用的程序，加深对系统进程及其控制的了解。

9.3.2 实验平台

华清远见开发环境

9.3.3 实验原理

fork 后父子进程会同步运行，但父子进程的返回顺序是不确定的。设两个变量 global 和 test 来检测父子进程共享资源的情况。

9.3.4 实验步骤

命令行终端，执行：

```
$ cd ~/workdir/linux/application/9-process
$ mkdir fork
$ cd fork
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\09. Linux 系统 fork 等系统调用实验\实验代码】目录下的“93.c”拷贝到该目录下。

```
$ gcc 93.c -o fork
$ ./fork    //执行观察结果
```



9.3.5 实验现象

可以看出父子进程打印出了各自的进程号和对应变量的值，显然 `global` 和 `test` 在父子进程间是独立的，

其各自的操作不会对对方的值有影响。

```
93.c
linux@ubuntu64-vm:~/workdir/linux/application/9-process/fork$ gcc 93.c -o fork
linux@ubuntu64-vm:~/workdir/linux/application/9-process/fork$ ./fork
the test content!
fork test!
global=24 test=2 Parent,my PID is 32845
global=23 test=1 Child,my PID is 32846
linux@ubuntu64-vm:~/workdir/linux/application/9-process/fork$
```

9.3.6 实验代码

93.c

```
1#include <stdio.h>
2#include <sys/types.h>
3#include <sys/wait.h>
4#include <unistd.h>
5#include <stdlib.h>
6#include <fcntl.h>
7
8int global = 22;
9char buf[] = "the test content!\n";
10
11int main(void)
12{
13    int test = 0, stat;
14    pid_t pid;
15    if(write(STDOUT_FILENO, buf, sizeof(buf)) != sizeof(buf))
16    {
17        perror("write error!");
18    }
19    printf(" fork test!\n");
20    /* fork */
21    pid = fork(); /*we should check the error*/
22    if (pid == -1)
23    {
24        perror("fork");
25        exit(0);
26    }
27    else if (pid == 0)
28    {
29        global++;
```



```
30     test++;
31     printf("global=%d test=%d Child,my PID is %d\n", global, test, getpid());
32     exit(0);
33 }
34 /*else be the parent*/
35 global += 2;
36 test += 2;
37 printf("global=%d test=%d Parent,my PID is %d\n", global, test, getpid());
38 exit(0);
39 //printf("global=%d test=%d Parent,my PID is %d",global,test,getpid());
//_exit(0);}
```

9.4 Linux 系统守护进程实验

9.4.1 实验目的

通过实验熟悉守护进程的编写过程。

9.4.2 实验平台

华清远见开发环境

9.4.3 实验原理

守护进程编写的主要步骤如下：

将程序进入后台执行。由于守护进程最终脱离控制终端，到后台去运行。方法是在进程中调用 `fork` 使父进程终止，让 `Daemon` 在子进程中后台执行。这就是常说的“脱壳”。子进程继续函数 `fork()` 的定义如下：

```
pid_t fork(void);
```

脱离控制终端、登录会话和进程组。开发人员如果要摆脱它们，不受它们的影响，一般使用 `setsid()` 设置新会话的领头进程，并与原来的登录会话和进程组脱离。

禁止进程重新打开控制终端。

重设文件权限掩码

关闭打开的文件描述符，并重定向标准输入、标准输出和标准错误输出的文件描述符。进程从创建它的父进程那里继承了打开的文件描述符。如果不关闭，将会浪费系统资源，引起无法预料的错误。关闭三者的代码如下：

```
for (fd = 0, fdtablesize = getdtablesize(); fd < fdtablesize; fd++)
```

```
close(fd);
```

改变工作目录到根目录或特定目录进程活动时，其工作目录所在的文件系统不能卸下。



9.4.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令

```
$ cd ~/workdir/linux/application/9-process
$ mkdir daemon
$ cd daemon
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\11. Linux 系统守护进程实验\实验代码】目录下的“init.c”和“test.c”拷贝到该目录下

编译执行：

```
$ gcc init.c test.c -o test
$ ./test
```

9.4.5 实验现象

```
$ ps -ef //查看进程
```

```
linux@ubuntu64-vm:~/workdir/linux/application/9-process/daemon$ gcc init.c test.c -o test
linux@ubuntu64-vm:~/workdir/linux/application/9-process/daemon$ ./test
linux@ubuntu64-vm:~/workdir/linux/application/9-process/daemon$

linux@ubuntu64-vm:~/workdir/linux/application/9-process/daemon$ ps -ef
```

USER	PPID	PID	STATUS	TIME	COMMAND
linux	2685	1	0	04:47 ?	/usr/bin/python /usr/lib/unity-scope-video-remote/unity-scope-vi
linux	2710	2685	0	04:47 ?	sh -c /usr/lib/x86_64-linux-gnu/libproxy/0.4.7/pxgsettings org.g
linux	2711	2710	0	04:47 ?	/usr/lib/x86_64-linux-gnu/libproxy/0.4.7/pxgsettings org.gnome.s
linux	2725	2231	0	04:48 ?	/usr/lib/gnome-disk-utility/gdu-notification-daemon
linux	2735	2231	0	04:48 ?	telepathy-indicator
linux	2743	1	0	04:48 ?	/usr/lib/telepathy/mission-control-5
linux	2748	1	0	04:48 ?	/usr/lib/gnome-online-accounts/goa-daemon
linux	2756	2231	0	04:48 ?	gnome-screensaver
linux	2846	2231	0	04:48 ?	update-notifier
root	2863	1	0	04:48 ?	/usr/bin/python /usr/lib/system-service/system-service-d
root	3011	1	0	04:49 ?	/usr/sbin/sshd -D
linux	8249	2231	0	04:49 ?	/usr/lib/deja-dup/deja-dup/deja-dup-monitor
root	9092	399	0	04:53 ?	/sbin/udev --daemon
root	9093	2	0	04:53 ?	[kdmflush]
root	9094	399	0	04:53 ?	/sbin/udev --daemon
root	9103	2	0	04:53 ?	[kworker/2:0]
root	9165	2	0	04:54 ?	[jbd2/dm-0:8]
root	9166	2	0	04:54 ?	[ext4-dio-unwrit]
linux	29607	2533	0	10:15 pts/3	bash
root	32406	2	0	11:13 ?	[kworker/0:1]
root	32432	2	0	11:16 ?	[flush-8:0]
root	32636	2	0	11:23 ?	[kworker/0:0]
linux	32928	1	0	11:35 ?	./test
linux	32935	2551	0	11:36 pts/1	ps -ef
root	61721	2	0	07:19 ?	[kworker/0:2]

打开/tmp/test.log 可以看到程序的输出内容，从输出可以发现 test 守护进程的各种特性满足上面的要



求。

```
root      61721      2  0 07:19 ?          00:00:02 [kworker/0:2]
linux@ubuntu64-vm:~/workdir/linux/application/9-process/deamon$ cat /tmp/test.log
I'm here at Sat Aug  2 11:35:42 2014
nI'm here at Sat Aug  2 11:35:44 2014
nI'm here at Sat Aug  2 11:35:46 2014
nI'm here at Sat Aug  2 11:35:48 2014
nI'm here at Sat Aug  2 11:35:50 2014
nI'm here at Sat Aug  2 11:35:52 2014
nI'm here at Sat Aug  2 11:35:54 2014
nI'm here at Sat Aug  2 11:35:56 2014
nI'm here at Sat Aug  2 11:35:58 2014
nI'm here at Sat Aug  2 11:36:00 2014
nI'm here at Sat Aug  2 11:36:02 2014
nI'm here at Sat Aug  2 11:36:04 2014
nI'm here at Sat Aug  2 11:36:06 2014
nI'm here at Sat Aug  2 11:36:08 2014
nI'm here at Sat Aug  2 11:36:10 2014
nI'm here at Sat Aug  2 11:36:12 2014
nI'm here at Sat Aug  2 11:36:14 2014
nI'm here at Sat Aug  2 11:36:16 2014
nI'm here at Sat Aug  2 11:36:18 2014
nI'm here at Sat Aug  2 11:36:20 2014
```

9.4.6 实验代码

init.c

```
1#include <unistd.h>
2#include <sys/types.h>
3#include <sys/stat.h>
4#include <stdlib.h>
5void init_daemon(void)
6{
7    int pid;
8    int i;
9
10   if (pid = fork())
11   {
12       exit(0);
13   } //是父进程，结束父进程
14
15   else if (pid < 0)
16   {
17       exit( -1 );
18   } //fork 失败，退出
19
20   setsid(); //第一子进程成为新的会话组长和进程组长 并与控制终端分离
21
22   if (pid = fork())
```



```
23 {
24     exit(0);
25 } //是第一子进程, 结束第一子进程
26 else if (pid < 0)
27 {
28     exit(1);
29 } //fork 失败, 退出
30
31 //第二子进程不再是会话组长
32 for(i = 0; i < getdtablesize(); ++i)
33     //关闭打开的文件描述符
34     close(i);
35 chdir("/tmp"); //改变工作目录到 /tmp
36 umask(0); //重设文件创建掩模
37 return;
}
```

test.c

```
1#include <stdio.h>
2#include <time.h>
3void init_daemon(void); //守护进程初始化函数
4
5int main()
6{
7    FILE *fp;
8    time_t t;
9    init_daemon(); //初始化为 Daemon
10
11    while(1) //每隔 2 秒钟向 test.log 报告运行状态
12    {
13        sleep(2); //睡眠 2 秒钟
14        if((fp = fopen("test.log", "a")) != NULL)
15        {
16            t = time(0);
17            fprintf(fp, "I'm here at %sn", asctime(localtime(&t)) );
18            fclose(fp);
19        }
20    }
21    return 0;
22}
```




第 10 章 Linux 系统进程间通信实验

10.1 实验原理

通过前面的学习，读者已经知道了进程是一个程序的一次执行，系统资源分配的最小单元。这里所说的进程一般是指运行在用户态的进程，而由于处于用户态的不同进程之间是彼此隔离的，就像处于不同城市的人们，它们必须通过某种方式来进行通信，例如人们现在广泛使用的手机等方式。本章就是讲述如何建立这些不同的通话方式，就像人们有多种通信方式一样。

Linux 下的进程通信手段基本上是从 Unix 平台上的进程通信手段继承而来的。而对 Unix 发展做出重大贡献的两大主力 AT&T 的贝尔实验室及 BSD（加州大学伯克利分校的伯克利软件发布中心）在进程间的通信方面的侧重点有所不同。前者是对 Unix 早期的进程间通信手段进行了系统的改进和扩充，形成了“system V IPC”，其通信进程主要局限在单个计算机内；后者则跳过了该限制，形成了基于套接口（socket）的进程间通信机制。而 Linux 则把两者的优势都继承了下来，如图 4.1 所示。

- Unix 进程间通信（IPC）方式包括管道、FIFO 以及信号。
 - System V 进程间通信（IPC）包括 System V 消息队列、System V 信号量以及 System V 共享内存区。
 - Posix 进程间通信（IPC）包括 Posix 消息队列、Posix 信号量以及 Posix 共享内存区。
- 现在在 Linux 中使用较多的进程间通信方式主要有以下几种。



图 6.1 进程间通信发展历程

（1）管道（Pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。

（2）信号（Signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一样的。

（3）消息队列（Message Queue）：消息队列是消息的链接表，包括 Posix 消息队列 SystemV 消息队列。它克服了前两种通信方式中信息量有限的缺点，具有写权限的进程可以按照一定的规则向消息队列中添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。

（4）共享内存（Shared memory）：可以说这是最有效的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种通信方式需要依靠某种同步机制，如互斥锁和信号量等。



(5) 信号量 (Semaphore): 主要作为进程之间以及同一进程的不同线程之间的同步和互斥手段。

(6) 套接字 (Socket): 这是一种更为一般的进程间通信机制, 它可用于网络中不同机器之间的进程间通信, 应用非常广泛。

10.1.1 管道通信

管道是 Linux 中进程间通信的一种方式, 它把一个程序的输出直接连接到另一个程序的输入。Linux 的管道主要包括两种: 无名管道和有名管道。

(1) 无名管道

无名管道是 Linux 中管道通信的一种原始方法, 如图 6.2 (左图) 所示, 它具有如下特点。

- 它只能用于具有亲缘关系的进程之间的通信 (也就是父子进程或者兄弟进程之间)。
- 它是一个半双工的通信模式, 具有固定的读端和写端。
- 管道也可以看成是一种特殊的文件, 对于它的读写也可以使用普通的 `read()`、`write()` 等函数。但是它不是普通的文件, 并不属于其他任何文件系统, 并且只存在于内存中。

(2) 有名管道 (FIFO)

有名管道是对无名管道的一种改进, 如图 6.2 (右图) 所示, 它具有如下特点:

- 它可以使互不相关的两个进程实现彼此通信。
- 该管道可以通过路径名来指出, 并且在文件系统中是可见的。在建立了管道之后, 两个进程就可以把它当作普通文件一样进行读写操作, 使用非常方便。
- FIFO 严格地遵循先进先出规则, 对管道及 FIFO 的读总是从开始处返回数据, 对它们的写则把数据添加到末尾, 它们不支持如 `lseek()` 等文件定位操作。

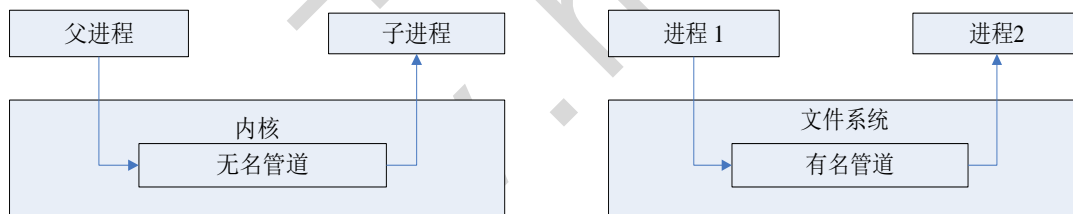


图 6.2 无名管道 (左) 和有名管道 (右)

10.1.2 无名管道系统调用

(1) 管道创建与关闭说明

管道是基于文件描述符的通信方式, 当一个管道建立时, 它会创建两个文件描述符 `fds[0]` 和 `fds[1]`, 其中 `fds[0]` 固定用于读管道, 而 `fd[1]` 固定用于写管道, 如图 6.3 所示, 这样就构成了一个半双工的通道。

管道关闭时只需将这两个文件描述符关闭即可, 可使用普通的 `close()` 函数逐个关闭各个文件描述符。

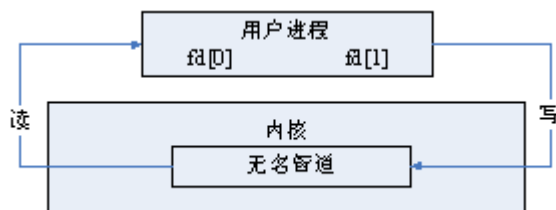


图 6.3 无名管道的读写机制

(2) 管道创建函数

创建管道可以通过调用 `pipe()` 来实现，下表 6.1 列出了 `pipe()` 函数的语法要点。

表 6.1 `pipe()` 函数语法要点

所需头文件	<code>#include <unistd.h></code>
函数原型	<code>int pipe(int fd[2])</code>
函数传入值	<code>fd[2]</code> : 管道的两个文件描述符，之后就可以直接操作这两个文件描述符
函数返回值	成功: 0
	出错: -1

(3) 管道读写说明

用 `pipe()` 函数创建的管道两端处于一个进程中，由于管道是主要用于在不同进程间通信的，因此这在实际应用中没有太大意义。实际上，通常先是一个管道，再调用 `fork()` 函数创建一子进程，该子进程会继承父进程所创建的管道，这时，父子进程管道的文件描述符对应关系如图 6.4 所示。

此时的关系看似非常复杂，实际上却已经给不同进程之间的读写创造了很好的条件。父子进程分别拥有自己的读写通道，为了实现父子进程之间的读写，只需把无关的读端或写端的文件描述符关闭即可。例如在图 6.5 中将父进程的写端 `fd[1]` 和子进程的读端 `fd[0]` 关闭。此时，父子进程之间就建立起了一条“子进程写入父进程读取”的通道。

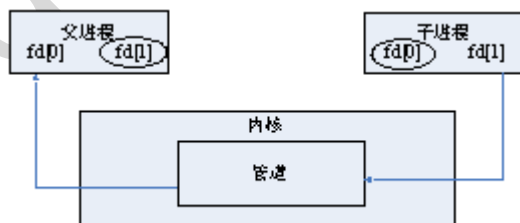


图 6.4 父子进程管道的文件描述符对应关系

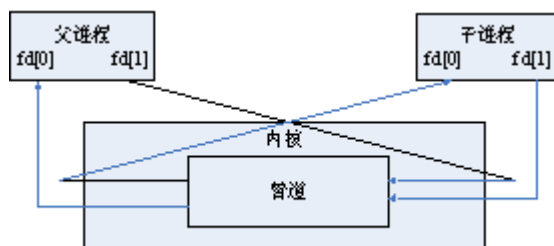


图 6.5 关闭父进程 fd[1]和子进程 fd[0]

同样，也可以关闭父进程的 fd[0]和子进程的 fd[1]，这样就可以建立一条“父进程写入子进程读取”的通道。另外，父进程还可以创建多个子进程，各个子进程都继承了相应的 fd[0]和 fd[1]，这时，只需要关闭相应端口就可以建立其各子进程之间的通道。

(4) 管道读写注意点

- 只有在管道的读端存在时，向管道写入数据才有意义。否则，向管道写入数据的进程将收到内核传来的 SIGPIPE 信号（通常为 Broken pipe 错误）。
- 向管道写入数据时，Linux 将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读取管道缓冲区中的数据，那么写操作将会一直阻塞。
- 父子进程在运行时，它们的先后次序并不能保证，因此，在为了保证父子进程已经关闭了相应的文件描述符，可在两个进程中调用 sleep()函数，当然这种调用不是很好的解决方法，在后面学到进程之间的同步与互斥机制之后，请读者自行修改本小节的实例程序。

10.1.3 标准流管道

(1) 标准流管道函数说明

与 Linux 的文件操作中有基于文件流的标准 I/O 操作一样，管道的操作也支持基于文件流的模式。这种基于文件流的管道主要是用来创建一个连接到另一个进程的管道，这里的“另一个进程”也就是一个可以进行一定操作的可执行文件，例如，用户执行“ls -l”或者自己编写的程序“./pipe”等。由于这一类操作很常用，因此标准流管道就将一系列的创建过程合并到一个函数 popen()中完成。它所完成的工作有以下几步。

- 创建一个管道。
- fork()一个子进程。
- 在父子进程中关闭不需要的文件描述符。
- 执行 exec 函数族调用。
- 执行函数中所指定的命令。

这个函数的使用可以大大减少代码的编写量，但同时也有一些不利之处，例如，它不如前面管道创建的函数那样灵活多样，并且用 popen()创建的管道必须使用标准 I/O 函数进行操作，但不能使用前面的 read()、write()一类不带缓冲的 I/O 函数。

与之相对应，关闭用 popen()创建的流管道必须使用函数 pclose()来关闭该管道流。该函数关闭标准



I/O 流，并等待命令执行结束。

(2) 函数格式

`popen()`和 `pclose()`函数格式如表 6.2 和表 6.3 所示。

表 6.2 `popen()`函数语法要点

所需头文件	#include <stdio.h>	
函数原型	FILE *popen(const char *command, const char *type)	
函数传入值	command: 指向的是一个以 null 结束符结尾的字符串，这个字符串包含一个 shell 命令，并被送到/bin/sh 以-c 参数执行，即由 shell 来执行	
	type:	“r”: 文件指针连接到 command 的标准输出，即该命令的结果产生输出 “w”: 文件指针连接到 command 的标准输入，即该命令的结果产生输入
函数返回值	成功: 文件流指针	
	出错: -1	

表 6.3 `pclose()`函数语法要点

所需头文件	#include <stdio.h>
函数原型	int pclose(FILE *stream)
函数传入值	stream: 要关闭的文件流
函数返回值	成功: 返回由 <code>popen()</code> 所执行的进程的退出码
	出错: -1

10.1.4 有名管道 (FIFO)

有名管道的创建可以使用函数 `mkfifo()`，该函数类似文件中的 `open()`操作，可以指定管道的路径和打开的模式。(用户还可以在命令行使用“`mknod 管道名 p`”来创建有名管道。)

在创建管道成功之后，就可以使用 `open()`、`read()`和 `write()`这些函数了。与普通文件的开发设置一样，对于为读而打开的管道可在 `open()`中设置 `O_RDONLY`，对于为写而打开的管道可在 `open()`中设置 `O_WRONLY`，在这里与普通文件不同的是阻塞问题。由于普通文件的读写时不会出现阻塞问题，而在管道的读写中却有阻塞的可能，这里的非阻塞标志可以在 `open()`函数中设定为 `O_NONBLOCK`。下面分别对阻塞打开和非阻塞打开的读写进行讨论。

对于读进程

- 若该管道是阻塞打开，且当前 FIFO 内没有数据，则对读进程而言将一直阻塞到有数据写入。
- 若该管道是非阻塞打开，则不论 FIFO 内是否有数据，读进程都会立即执行读操作。即如果 FIFO

内没有数据，则读函数将立刻返回 0。

对于写进程

- 若该管道是阻塞打开，则写操作将一直阻塞到数据可以被写入。



- 若该管道是非阻塞打开而不能写入全部数据，则读操作进行部分写入或者调用失败。

表 4.4 列出了 mkfifo()函数的语法要点。

表 6.4 mkfifo()函数语法要点

所需头文件	<pre>#include <sys/types.h> #include <sys/stat.h></pre>	
函数原型	<pre>int mkfifo(const char *filename, mode_t mode)</pre>	
函数传入值	filename: 要创建的管道	
函数传入值	mode:	O_RDONLY: 读管道
		O_WRONLY: 写管道
		O_RDWR: 读写管道
		O_NONBLOCK: 非阻塞
		O_CREAT: 如果该文件不存在，那么就创建一个新的文件，并用第三个参数为其设置权限
		O_EXCL: 如果使用 O_CREAT 时文件存在，那么可返回错误消息。这一参数可测试文件是否存在
函数返回值	成功: 0	
	出错: -1	

表 4.5 再对 FIFO 相关的出错信息做一归纳，以方便用户查错。

表 6.5 FIFO 相关的出错信息

EACCESS	参数 filename 所指定的目录路径无可执行的权限
EEXIST	参数 filename 所指定的文件已存在
ENAMETOOLONG	参数 filename 的路径名称太长
ENOENT	参数 filename 包含的目录不存在
ENOSPC	文件系统的剩余空间不足
ENOTDIR	参数 filename 路径中的目录存在但却非真正的目录
EROFS	参数 filename 指定的文件存在于只读文件系统内

10.1.5 信号通信

信号是在软件层次上对中断机制的一种模拟。在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。信号可以直接进行用户空间进程和内核进程之间的交互，内核进程也可以利用它来通知用户空间进程发生了哪些系统事件。它可以在任何时候发给某一进程，而无需知道该进程的状态。如果该进程当前并未处于执行态，则该信号就由内核保存起来，直到该进程恢复执行再传递给它为止；如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传



递给进程。

信号是进程间通信机制中惟一的异步通信机制，可以看作是异步通知，通知接收信号的进程有哪些事情发生了。信号机制经过 POSIX 实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息。

信号事件的发生有两个来源：硬件来源（比如我们按下了键盘或者其他硬件故障）；软件来源，最常用发送信号的函数有 kill()、raise()、alarm()、setitimer()和 sigqueue()等，软件来源还包括一些非法运算等操作。

进程可以通过 3 种方式来响应一个信号。

(1) 忽略信号

即对信号不做任何处理，其中，有两个信号不能忽略：SIGKILL 及 SIGSTOP。

(2) 捕捉信号

定义信号处理函数，当信号发生时，执行相应的处理函数。

(3) 执行缺省操作

Linux 对每种信号都规定了默认操作，如表 6.6 所示。

表 6.6 常见信号的含义及其默认操作

信 号 名	含 义	默 认 操 作
SIGHUP	该信号在用户终端连接（正常或非正常）结束时发出，通常是在终端的控制进程结束时，通知同一会话内的各个进程与控制终端不再关联	终止
SIGINT	该信号在用户键入 INTR 字符（通常是 Ctrl-C）时发出，终端驱动程序发送此信号并送到前台进程中的每一个进程	终止
SIGQUIT	该信号和 SIGINT 类似，但由 QUIT 字符（通常是 Ctrl-\）来控制	终止
SIGILL	该信号在一个进程企图执行一条非法指令时（可执行文件本身出现错误，或者试图执行数据段、堆栈溢出时）发出	终止
SIGFPE	该信号在发生致命的算术运算错误时发出。这里不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术的错误	终止
SIGKILL	该信号用来立即结束程序的运行，并且不能被阻塞、处理和忽略	终止
SIGALRM	该信号当一个定时器到时的时候发出	终止
SIGSTOP	该信号用于暂停一个进程，且不能被阻塞、处理或忽略	暂停进程



SIGTSTP	该信号用于交互停止进程，用户可键入 SUSP 字符时（通常是 Ctrl+Z）发出这个信号	停止进程
SIGCHLD	子进程改变状态时，父进程会收到这个信号	忽略

一个完整的信号生命周期可以分为 3 个重要阶段，这 3 个阶段由 4 个重要事件来刻画的：信号产生、信号在进程中注册、信号在进程中注销、执行信号处理函数。这里信号的产生、注册、注销等是指信号的内部实现机制，而不是信号的函数实现。因此，信号注册与否与本节后面讲到的发送信号函数（如 kill（）等）以及信号安装函数（如 signal（）等）无关，只与信号值有关。

相邻两个事件的时间间隔构成信号生命周期的一个阶段。要注意这里的信号处理有多种方式，一般是由内核完成的，当然也可以由用户进程来完成，故在此没有明确画出。

信号的处理包括信号的发送、捕获以及信号的处理，它们各自相对应的常见函数有

发送信号的函数：kill()、raise()。

捕获信号的函数：alarm()、pause()。

处理信号的函数：signal()、sigaction()。

(1) 信号发送：kill()和 raise()

kill()函数同读者熟知的 kill 系统命令一样，可以发送信号给进程或进程组（实际上，kill 系统命令只是 kill()函数的一个用户接口）。这里需要注意的是，它不仅中止进程（实际上发出 SIGKILL 信号），也可以向进程发送其他信号。

与 kill()函数所不同的是，raise()函数允许进程向自身发送信号。

表 6.7 列出了 kill()函数的语法要点。

表 6.7 kill()函数语法要点

所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid, int sig)	
函数传入值	pid	正数：要发送信号的进程号
		0：信号被发送到所有和当前进程在同一个进程组的进程
		-1：信号发给所有的进程表中的进程（除了进程号最大的进程外）
		<-1：信号发送给进程组号为-pid 的每一个进程
	sig: 信号	
函数返回值	成功：0	
	出错：-1	

表 6.8 列出了 raise()函数的语法要点。

表 6.8 raise()函数语法要点

所需头文件	#include <signal.h>
-------	---------------------



	#include <sys/types.h>
函数原型	int raise(int sig)
函数传入值	sig: 信号
函数返回值	成功: 0
	出错: -1

(2) 信号捕捉: alarm()、pause()

alarm()也称为闹钟函数,它可以在进程中设置一个定时器,当定时器指定的时间到时,它就向进程发送 SIGALRM 信号。要注意的是,一个进程只能有一个闹钟时间,如果在调用 alarm()之前已设置过闹钟时间,则任何以前的闹钟时间都被新值所代替。

pause()函数是用于将调用进程挂起直至捕捉到信号为止。这个函数很常用,通常可以用于判断信号是否已到。

表 6.9 列出了 alarm()函数的语法要点。

表 6.9 alarm()函数语法要点

所需头文件	#include <unistd.h>
函数原型	unsigned int alarm(unsigned int seconds)
函数传入值	seconds: 指定秒数,系统经过 seconds 秒之后向该进程发送 SIGALRM 信号
函数返回值	成功: 如果调用此 alarm()前,进程中已经设置了闹钟时间,则返回上一个闹钟时间的剩余时间,否则返回 0
	出错: -1

表 6.10 列出了 pause()函数的语法要点。

表 6.10 pause()函数语法要点

所需头文件	#include <unistd.h>
函数原型	int pause(void)
函数返回值	-1, 并且把 error 值设为 EINTR

(3) 信号的处理

信号处理的主要方法有两种,一种是使用简单的 signal()函数,另一种是使用信号集函数组。下面分别介绍这两种处理方式。

● 信号处理函数

使用 signal()函数处理时,只需要指出要处理的信号和处理函数即可。它主要是用于前 32 种非实时信号的处理,不支持信号传递信息,但是由于使用简单、易于理解,因此也受到很多程序员的欢迎。Linux 还支持一个更健壮更新的信号处理函数 sigaction(),推荐使用该函数。

signal()函数的语法要点如表 6.11 所示。



表 6.11 signal()函数语法要点

所需头文件	#include <signal.h>	
函数原型	typedef void (*sighandler_t)(int); sighandler_t signal(int signum, sighandler_t handler);	
函数传入值	signum: 指定信号代码	
	handler:	SIG_IGN: 忽略该信号
		SIG_DFL: 采用系统默认方式处理信号 自定义的信号处理函数指针
函数返回值	成功: 以前的信号处理配置	
	出错: -1	

这里需要对这个函数原型进行说明。这个函数原型有点复杂。首先该函数原型整体指向一个无返回值并且带一个整型参数的函数的指针，也就是信号的原始配置函数。接着该原型又带有两个参数，其中的第二个参数可以是用户自定义的信号处理函数的函数指针。

表 6.12 列举了 sigaction()的语法要点。

表 6.12 sigaction()函数语法要点

所需头文件	#include <signal.h>	
函数原型	int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)	
函数传入值	signum: 信号代码，可以为除 SIGKILL 及 SIGSTOP 外的任何一个特定有效的信号	
	act: 指向结构 sigaction 的一个实例的指针，指定对特定信号的处理	
	oldact: 保存原来对相应信号的处理	
函数返回值	成功: 0	
	出错: -1	

这里要说明的是 sigaction()函数中第 2 个和第 3 个参数用到的 sigaction 结构。这是一个看似非常复杂的结构，希望读者能够慢慢阅读此段内容。

首先给出了 sigaction 的定义，如下所示：

```
struct sigaction
{
    void (*sa_handler)(int signo);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restore)(void);
}
```



sa_handler 是一个函数指针，指定信号处理函数，这里除可以是用户自定义的处理函数外，还可以为 SIG_DFL（采用缺省的处理方式）或 SIG_IGN（忽略信号）。它的处理函数只有一个参数，即信号值。

sa_mask 是一个信号集，它可以指定在信号处理程序执行过程中哪些信号应当被屏蔽，在调用信号捕获函数之前，该信号集要加入到信号的信号屏蔽字中。

sa_flags 中包含了许多标志位，是对信号进行处理的各个选择项。它的常见可选值如下表 6.13 所示。

表 6.13 常见信号的含义及其默认操作

选 项	含 义
SA_NODEFER / SA_NOMASK	当捕捉到此信号时，在执行其信号捕捉函数时，系统不会自动屏蔽此信号
SA_NOCLDSTOP	进程忽略子进程产生的任何 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 信号
SA_RESTART	令重启的系统调用起作用
SA_ONESHOT / SA_RESETHAND	自定义信号只执行一次，在执行完毕后恢复信号的系统默认动作

- 信号集函数组

使用信号集函数组处理信号时涉及一系列的函数，这些函数按照调用的先后次序可分为以下几大功能模块：创建信号集合、注册信号处理函数以及检测信号。

其中，创建信号集合主要用于处理用户感兴趣的一些信号，其函数包括以下几个。

- sigemptyset(): 将信号集合初始化为空。
- sigfillset(): 将信号集合初始化为包含所有已定义的信号的集合。
- sigaddset(): 将指定信号加入到信号集合中去。
- sigdelset(): 将指定信号从信号集合中删去。
- sigismember(): 查询指定信号是否在信号集合之中。

注册信号处理函数主要用于决定进程如何处理信号。这里要注意的是，信号集里的信号并不是真正可以处理的信号，只有当信号的状态处于非阻塞状态时才会真正起作用。因此，首先使用 sigprocmask() 函数检测并更改信号屏蔽字（信号屏蔽字是用来指定当前被阻塞的一组信号，它们不会被进程接收），然后使用 sigaction() 函数来定义进程接收到特定信号之后的行为。检测信号是信号处理的后续步骤，因为被阻塞的信号不会传递给进程，所以这些信号就处于“未处理”状态（也就是进程不清楚它的存在）。sigpending() 函数允许进程检测“未处理”信号，并进一步决定对它们作何处理。

首先介绍创建信号集合的函数格式，表 6.14 列举了这一组函数的语法要点。

表 6.14 创建信号集合函数语法要点

所需头文件	#include <signal.h>
函数原型	int sigemptyset(sigset_t *set)
	int sigfillset(sigset_t *set)
	int sigaddset(sigset_t *set, int signum)



	int sigdelset(sigset_t *set, int signum)
	int sigismember(sigset_t *set, int signum)
函数传入值	set: 信号集
	signum: 指定信号代码
函数返回值	成功: 0 (sigismember 成功返回 1, 失败返回 0)
	出错: -1

表 6.15 列举了 sigprocmask 的语法要点。

表 6.15 sigprocmask 函数语法要点

所需头文件	#include <signal.h>	
函数原型	int sigprocmask(int how, const sigset_t *set, sigset_t *oset)	
函数传入值	how: 决定函数的 操作方式	SIG_BLOCK: 增加一个信号集合到当前进程的阻塞集合之中
		SIG_UNBLOCK: 从当前的阻塞集合之中删除一个信号集合
		SIG_SETMASK: 将当前的信号集合设置为信号阻塞集合
	set: 指定信号集	
	oset: 信号屏蔽字	
函数返回值	成功: 0	
	出错: -1	

此处, 若 set 是一个非空指针, 则参数 how 表示函数的操作方式; 若 how 为空, 则表示忽略此操作。

最后, 表 6.16 列举了 sigpending 函数的语法要点。

表 4.16 sigpending 函数语法要点

所需头文件	#include <signal.h>
函数原型	int sigpending(sigset_t *set)
函数传入值	set: 要检测的信号集
函数返回值	成功: 0
	出错: -1

总之, 在处理信号时, 一般遵循如图 6.6 所示的操作流程。

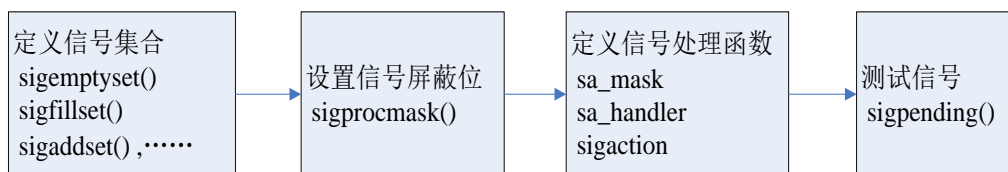


图 6.6 一般的信号操作处理流程

10.1.6 信号量

在多任务操作系统环境下，多个进程会同时运行，并且一些进程之间可能存在一定的关联。多个进程可能为了完成同一个任务会相互协作，这样形成进程之间的同步关系。而且在不同进程之间，为了争夺有限的系统资源（硬件或软件资源）会进入竞争状态，这就是进程之间的互斥关系。

进程之间的互斥与同步关系存在的根源在于临界资源。临界资源是在同一个时刻只允许有限个（通常只有一个）进程可以访问（读）或修改（写）的资源，通常包括硬件资源（处理器、内存、存储器以及其他外围设备等）和软件资源（共享代码段，共享结构和变量等）。访问临界资源的代码叫做临界区，临界区本身也会成为临界资源。

信号量是用来解决进程之间的同步与互斥问题的一种进程之间通信机制，包括一个称为信号量的变量和在该信号量下等待资源的进程等待队列，以及对信号量进行的两个原子操作（PV 操作）。其中信号量对应于某一种资源，取一个非负的整型值。信号量值指的是当前可用的该资源的数量，若它等于 0 则意味着目前没有可用的资源。

PV 原子操作的具体定义为：

P 操作：如果有可用的资源（信号量值>0），则占用一个资源（给信号量值减去一，进入临界区代码）；如果没有可用的资源（信号量值等于 0），则被阻塞到，直到系统将资源分配给该进程（进入等待队列，一直等到资源轮到该进程）。

V 操作：如果在该信号量的等待队列中有进程在等待资源，则唤醒一个阻塞进程。如果没有进程等待它，则释放一个资源（给信号量值加一）。

常见的使用信号量访问临界区的伪代码如下所示：

```

{
    /* 设 R 为某种资源，S 为资源 R 的信号量*/
    INIT_VAL(S);          /* 对信号量 S 进行初始化 */
    非临界区;
    P(S);                  /* 进行 P 操作 */
    临界区（使用资源 R）;  /* 只有有限个（通常只有一个）进程被允许进入该区*/
    V(S);                  /* 进行 V 操作 */
    非临界区;
}
  
```

最简单的信号量是只能取 0 和 1 两种值，这种信号量被叫做二维信号量。在本节中，主要讨论二维信号量。二维信号量的应用比较容易地扩展到使用多维信号量的情况。



1. 函数说明

在 Linux 系统中，使用信号量通常分为以下几个步骤。

第一步：创建信号量或获得在系统已存在的信号量，此时需要调用 `semget()` 函数。不同进程通过使用同一个信号量键值来获得同一个信号量。

第二步：初始化信号量，此时使用 `semctl()` 函数的 `SETVAL` 操作。当使用二维信号量时，通常将信号量初始化为 1。

第三步：进行信号量的 PV 操作，此时调用 `semop()` 函数。这一步是实现进程之间的同步和互斥的核心工作部分。

第四步：如果不需要信号量，则从系统中删除它，此时使用 `semctl()` 函数的 `IPC_RMID` 操作。此时需要注意，在程序中不应该出现对已经被删除的信号量的操作。

2. 函数格式

表 6.17 列举了 `semget()` 函数的语法要点。

表 6.17 `semget()` 函数语法要点

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h></pre>
函数原型	<code>int semget(key_t key, int nsems, int semflg)</code>
函数传入值	key: 信号量的键值，多个进程可以通过它访问同一个信号量，其中有个特殊值 <code>IPC_PRIVATE</code> 。它用于创建当前进程的私有信号量。
	nsems: 需要创建的信号量数目，通常取值为 1。
	semflg: 同 <code>open()</code> 函数的权限位，也可以用八进制表示法，其中使用 <code>IPC_CREAT</code> 标志创建新的信号量，即使该信号量已经存在（具有同一个键值的信号量已在系统中存在），也不会出错。如果同时使用 <code>IPC_EXCL</code> 标志可以创建一个新的唯一的信号量，此时如果该信号量已经存在，该函数会返回出错。
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值。
	出错：-1

表 6.18 列举了 `semctl()` 函数的语法要点。

表 6.18 `semctl()` 函数语法要点



所需 头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h></pre>
函数 原型	<pre>int semctl(int semid, int semnum, int cmd, union semun arg)</pre>
函数 传入值	<p>semid: semget()函数返回的信号量标识符。</p> <p>semnum: 信号量编号，当使用信号量集时才会被用到。通常取值为 0，就是使用单个信号量（也是第一个信号量）。</p> <p>cmd: 指定对信号量的各种操作，当使用单个信号量（而不是信号量集）时，常用的操作有以下几种：</p> <p>IPC_STAT: 获得该信号量（或者信号量集合）的 semid_ds 结构，并存放在由第四个参数 arg 结构变量的 buf 域指向的 semid_ds 结构中。semid_ds 是在系统中描述信号量的数据结构。</p> <p>IPC_SETVAL: 将信号量值设置为 arg 的 val 值。</p> <p>IPC_GETVAL: 返回信号量的当前值。</p> <p>IPC_RMID: 从系统中，删除信号量（或者信号量集）</p> <p>arg: 是 union semnn 结构，可能在某些系统中不给出该结构的定义，此时必须由程序员自己定义。</p> <pre>union semun { int val; struct semid_ds *buf; unsigned short *array; }</pre>
函数 返回值	<p>成功：根据 cmd 值的不同而返回不同的值。</p> <p>IPC_STAT、IPC_SETVAL、IPC_RMID: 返回 0</p> <p>IPC_GETVAL: 返回信号量的当前值</p> <p>出错：-1</p>

表 6.19 列举了 semop()函数的语法要点。

表 6.19 semop()函数语法要点

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h></pre>
函数原型	<pre>int semop(int semid, struct sembuf *sops, size_t nsops)</pre>



函数传入 值	semid: semget()函数返回的信号量标识符。
	sops: 指向信号量操作数组, 一个数组包括以下成员: struct sembuf { short sem_num; /* 信号量编号, 使用单个信号量时, 通常取 值为 0 */ short sem_op; /* 信号量操作: 取值为-1 则表示 P 操作, 取值为+1 则表示 V 操作*/ short sem_flg; /* 通常设置为 SEM_UNDO。这样在进程没释放信号量而退出 时, 系统自动 释放该进程中未释放的信号量 */ }
	nsops: 操作数组 sops 中的操作个数 (元素数目), 通常取值为 1 (一个操作)
函数返回 值	成功: 信号量标识符, 在信号量的其他函数中都会使用该值。
	出错: -1

因为信号量相关的函数调用接口比较复杂, 我们可以将它们封装成二维单个信号量的几个基本函数。它们分别为信号量初始化函数 (或者信号量赋值函数) `init_sem()`、P 操作函数 `sem_p()`、V 操作函数 `sem_v()` 以及删除信号量的函数 `del_sem()` 等, 具体实现如下所示:

```
/* sem_com.c */
#include "sem_com.h"
/* 信号量初始化 (赋值) 函数*/
int init_sem(int sem_id, int init_value)
{
    union semun sem_union;
    sem_union.val = init_value; /* init_value 为初始值 */
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1)
    {
        perror("Initialize semaphore");
        return -1;
    }
    return 0;
}
```




```
/* 从系统中删除信号量的函数 */
int del_sem(int sem_id)
{
    union semun sem_union;
    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
    {
        perror("Delete semaphore");
        return -1;
    }
}

/* P 操作函数 */
int sem_p(int sem_id)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0; /* 单个信号量的编号应该为 0 */
    sem_b.sem_op = -1; /* 表示 P 操作 */
    sem_b.sem_flg = SEM_UNDO; /* 系统自动释放将会在系统中残留的信号量 */
    if (semop(sem_id, &sem_b, 1) == -1)
    {
        perror("P operation");
        return -1;
    }
    return 0;
}

/* V 操作函数 */
int sem_v(int sem_id)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0; /* 单个信号量的编号应该为 0 */
    sem_b.sem_op = 1; /* 表示 V 操作 */
    sem_b.sem_flg = SEM_UNDO; /* 系统自动释放将会在系统中残留的信号量 */
    if (semop(sem_id, &sem_b, 1) == -1)
    {
        perror("V operation");
        return -1;
    }
}
```



```

    }
    return 0;
}

```

10.1.7 共享内存

可以说，共享内存是一种最为高效的进程间通信方式。因为进程可以直接读写内存，不需要任何数据的拷贝。为了在多个进程间交换信息，内核专门留出了一块内存区。这段内存区可以由需要访问的进程将其映射到自己的私有地址空间。因此，进程就可以直接读写这一内存区而不需要进行数据的拷贝，从而大大提高了效率。当然，由于多个进程共享一段内存，因此也需要依靠某种同步机制，如互斥锁和信号量等（请参考本章的共享内存实验）。其原理示意图如图 6.7 所示。

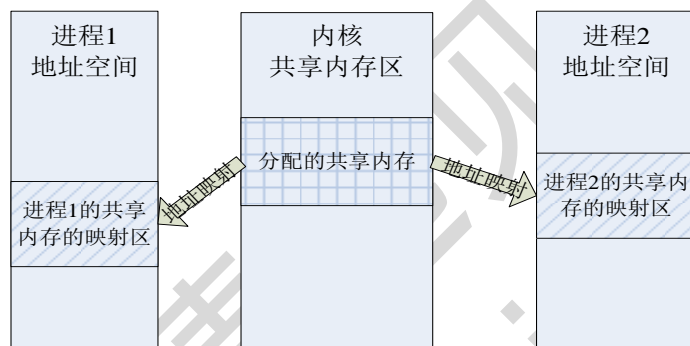


图 4.7 共享内存原理示意图

共享内存的实现分为两个步骤，第一步是创建共享内存，这里用到的函数是 `shmget()`，也就是从内存中获得一段共享内存区域。第二步映射共享内存，也就是把这段创建的共享内存映射到具体的进程空间中，这里使用的函数是 `shmat()`。到这里，就可以使用这段共享内存了，也就是可以使用不带缓冲的 I/O 读写命令对其进行操作。除此之外，当然还有撤销映射的操作，其函数为 `shmdt()`。这里就主要介绍这 3 个函数。

表 6.20 列举了 `shmget()` 函数的语法要点。

表 6.20 `shmget()` 函数语法要点

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h></pre>
函数原型	<code>int shmget(key_t key, int size, int shmflg)</code>
函数传入值	<p>key: 共享内存的键值，多个进程可以通过它访问同一个共享内存，其中有个特殊值 <code>IPC_PRIVATE</code>。它用于创建当前进程的私有共享内存。</p> <p>size: 共享内存区大小</p> <p>shmflg: 同 <code>open()</code> 函数的权限位，也可以用八进制表示法</p>
函数返回值	成功：共享内存段标识符



出错：-1

表 6.21 列举了 shmat()函数的语法要点。

表 6.21 shmat()函数语法要点

所需头文件	#include <sys/types.h>	
	#include <sys/ipc.h>	
	#include <sys/shm.h>	
函数原型	char *shmat(int shmid, const void *shmaddr, int shmflg)	
函数传入值	shmid: 要映射的共享内存区标识符	
	shmaddr: 将共享内存映射到指定地址（若为 0 则表示系统自动分配地址并把该段共享内存映射到调用进程的地址空间）	
	shmflg	SHM_RDONLY: 共享内存只读 默认 0: 共享内存可读写
函数返回值	成功: 被映射的段地址	
	出错: -1	

表 6.22 列举了 shmdt()函数的语法要点。

表 6.22 shmdt()函数语法要点

所需头文件	#include <sys/types.h>	
	#include <sys/ipc.h>	
	#include <sys/shm.h>	
函数原型	int shmdt(const void *shmaddr)	
函数传入值	shmaddr: 被映射的共享内存段地址	
函数返回值	成功: 0	
	出错: -1	

10.1.8 消息队列

顾名思义，消息队列就是一些消息的列表。用户可以在消息队列中添加消息和读取消息等。从这点上看，消息队列具有一定的 FIFO 特性，但是它可以实现消息的随机查询，比 FIFO 具有更大的优势。同时，这些消息又是存在于内核中的，由“队列 ID”来标识。

消息队列的实现包括创建或打开消息队列、添加消息、读取消息和控制消息队列这四种操作。其中创建或打开消息队列使用的函数是 msgget()，这里创建的消息队列的数量会受到系统消息队列数量的限制；添加消息使用的函数是 msgsnd()函数，它把消息添加到已打开的消息队列末尾；读取消息使用的函数是 msgrcv()，它把消息从消息队列中取走，与 FIFO 不同的是，这里可以取走指定的某一条消息；最后控制消息队列使用的函数是 msgctl()，它可以完成多项功能。

表 6.23 列举了 msgget()函数的语法要点。

表 6.23 msgget()函数语法要点



所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h></pre>
函数原型	<code>int msgget(key_t key, int msgflg)</code>
函数传入值	<p>key: 消息队列的键值，多个进程可以通过它访问同一个消息队列，其中有个特殊值 <code>IPC_PRIVATE</code>。它用于创建当前进程的私有消息队列。</p> <p>msgflg: 权限标志位</p>
函数返回值	<p>成功：消息队列 ID</p> <p>出错：-1</p>

表 6.24 列举了 `msgsnd()` 函数的语法要点。

表 6.24 msgsnd() 函数语法要点

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h></pre>
函数原型	<code>int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)</code>
函数传入值	<p>msqid: 消息队列的队列 ID</p> <p>msgp: 指向消息结构的指针。该消息结构 <code>msgbuf</code> 通常为：</p> <pre>struct msgbuf { long mtype; /* 消息类型，该结构必须从这个域开始 */ char mtext[1]; /* 消息正文 */ }</pre> <p>msgsz: 消息正文的字节数（不包括消息类型指针变量）</p> <p>msgflg: <ul style="list-style-type: none"> <code>IPC_NOWAIT</code> 若消息无法立即发送（比如：当前消息队列已满），函数会立即返回。 <code>0:</code> <code>msgsnd</code> 调用阻塞直到发送成功为止。 </p>
函数返回值	<p>成功：0</p> <p>出错：-1</p>

表 6.25 列举了 `msgrcv()` 函数的语法要点。

表 6.25 msgrcv() 函数语法要点

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h></pre>
-------	---



函数原型	int msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg)	
函数传入值	msgid:	消息队列的队列 ID
	msgp:	消息缓冲区, 同于 msgsnd()函数的 msgp
	msgsz:	消息正文的字节数 (不包括消息类型指针变量)
	msgtyp:	0: 接收消息队列中第一个消息
		大于 0: 接收消息队列中第一个类型为 msgtyp 的消息
		小于 0: 接收消息队列中第一个类型值不小于 msgtyp 绝对值且类型值又最小的消息
	msgflg:	MSG_NOERROR: 若返回的消息比 msgsz 字节多, 则消息就会截短到 msgsz 字节, 且不通知消息发送进程
		IPC_NOWAIT 若在消息队列中并没有相应类型的消息可以接收, 则函数立即返回
		0: msgsnd()调用阻塞直到接收一条相应类型的消息为止
函数返回值	成功: 0	
	出错: -1	

表 6.26 列举了 msgctl()函数的语法要点。

表 6.26 msgctl()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgctl (int msgqid, int cmd, struct msqid_ds *buf)	
函数传入值	msgid:	消息队列的队列 ID
	cmd: 命令 参数	IPC_STAT: 读取消息队列的数据结构 msqid_ds, 并将其存储在 buf 指定的地址中
		IPC_SET: 设置消息队列的数据结构 msqid_ds 中的 ipc_perm 域 (IPC 操作权限描述结构) 值。这个值取自 buf 参数
		IPC_RMID: 从系统内核中删除消息队列
	buf: 描述消息队列的 msqid_ds 结构类型变量	
	成功: 0	
函数返回值	出错: -1	



10.2 Linux 系统无名管道通信实验

10.2.1 实验目的

熟悉 UNIX/LINUX 支持的无名管道通信方式。

10.2.2 实验平台

华清远见开发环境

10.2.3 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/10-comm  
$ mkdir pipe  
$ cd pipe
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\12. Linux 系统无名管道通信实验\实验代码】目录下的“pipe.c”拷贝到该目录下。

执行命令：

```
$ gcc pipe.c -o pipe  
$ ./pipe
```

10.2.4 实验现象

运行结果，延迟 5 秒后显示：

child 1 process is sending message!

再显示：

child 2 process is sending message!

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ cd pipe/  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/pipe$ ls  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/pipe$ gcc pipe.c -o pipe  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/pipe$ ./pipe  
child 1 process is sending message!nchild 2 process is sending message!nlinux@ubuntu64-vm:~/workdir/linux/appl  
ication/10-comm/pipe$
```

10.2.5 实验代码

pipe.c

```
1#include <unistd.h>  
2#include <stdio.h>  
3#include <stdlib.h>  
4  
5int pid1, pid2;  
6  
7main( )
```



```
8{
9    int fd[2];
10   char outpipe[100], inpipe[100];
11   pipe(fd);                      /*创建一个管道*/
12   while ((pid1 = fork( )) == -1);
13   if (pid1 == 0)
14   {
15       lockf(fd[1], 1, 0);
16       sprintf(outpipe, "child 1 process is sending message!");
17       /*把串放入数组 outpipe 中*/
18       write(fd[1], outpipe, 50);    /*向管道写长为 50 字节的串*/
19       sleep(5);                    /*自我阻塞 5 秒*/
20       lockf(fd[1], 0, 0);
21       exit(0);
22   }
23   else
24   {
25       while((pid2 = fork( )) == -1);
26       if (pid2 == 0)
27       {
28           lockf(fd[1], 1, 0);        /*互斥*/
29           sprintf(outpipe, "child 2 process is sending message!");
30           write(fd[1], outpipe, 50);
31           sleep(5);
32           lockf(fd[1], 0, 0);
33           exit(0);
34       }
35       else
36       {
37           wait(NULL);                /*同步*/
38           read(fd[0], inpipe, 50);    /*从管道中读长为 50 字节的串*/
39           printf("%s/n", inpipe);
40           wait(NULL);
41           read(fd[0], inpipe, 50);
42           printf("%s/n", inpipe);
43           exit(0);
44       }
45   }
46 }
```



10.3 Linux 系统有名管道通信实验

10.3.1 实验目的

熟悉 UNIX/LINUX 支持的有名管道通信方式。

10.3.2 实验平台

华清远见开发环境

10.3.3 实验内容

创建有名管道；

本进程执行循环等待数据被写入到管道中并读有名管道；

打开有名管道并写数据到名管道；

10.3.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/10-comm  
$ mkdir fifo  
$ cd fifo
```

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ mkdir fifo  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ cd fifo/  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\13. Linux 系统有名管道通信实验\实验代码】目录下的“create_fifo.c”、“read_fifo.c”、“write_fifo.c”拷贝到该目录下。

```
a.out      create_fifo.c  read_fifo.c  write_fifo.c  
linux@ubuntu64-vm:~/workdir/linux/application$ cp /mnt/hgfs/share/2.\ Linux系统部分/13.\ Linux系统有名管道通信实  
验/实验代码/* 10-comm/fifo/  
linux@ubuntu64-vm:~/workdir/linux/application$
```

执行编译命令

```
$ gcc create_fifo.c -o create_fifo
```

运行程序

```
$ ./create_fifo test
```

在当前目录下，创建了管道文件“test”。

执行编译命令：

```
$ gcc read_fifo.c -o read_fifo  
$ gcc write_fifo.c -o write_fifo
```




```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ gcc create_fifo.c read_fifo.c write_fifo.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ gcc create_fifo.c -o create_fifo
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ ./create_fifo
Usage: ./create_fifo <filename>
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ ./create_fifo test
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ gcc read_fifo.c -o read_fifo
read_fifo.c: 在函数‘main’中:
read_fifo.c:56: 警告: 格式‘%d’需要类型‘int’, 但实参 3 的类型为‘ssize_t’
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ gcc write_fifo.c -o write_fifo
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$
```

\$./read_fifo test //执行读管道文件的程序

打开另一个终端，进入该目录

\$./write_fifo //执行写管道文件的程序

写入 hello

open fifo ./write_fifo for writting succeeded.

hello

```
linux@ubuntu64-vm:~/workdir/linux/application$ cd 10-comm/fifo/
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ ls
a.out create_fifo create_fifo.c read_fifo read_fifo.c test write_fifo write_fifo.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ ./write_fifo test
open fifo ./write_fifo for writting succeeded.
hello
```

10.3.5 实验现象

在另一个终端会显示:

open fifo ./read_fifo for reading succeeded.

read 6 bytes from fifo: hello

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/fifo$ ./read_fifo test
open fifo ./read_fifo for reading succeeded.
read 6 bytes from fifo: hello
```

10.3.6 实验代码

create_fifo.c

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <sys/stat.h>
4#include <string.h>
5#include <errno.h>
6
7int main(int argc, char **argv)
8{
9    if (argc < 2)
10    {
```



```
11     fprintf(stdout, "Usage: %s <filename>\n", argv[0]);
12     exit(1);
13 }
14
15 //int mkfifo(const char *path, mode_t mode);
16 if (mkfifo(argv[1], 0644) < 0)
17 {
18     fprintf(stderr, "mkfifo() failed: %s\n", strerror(errno));
19     exit(1);
20 }
21
22 return 0;
23 }
```

read_fifo.c

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <unistd.h>
4#include <sys/stat.h>
5#include <fcntl.h>
6#include <string.h>
7#include <errno.h>
8
9#define BUFFER_SIZE 1024
10
11int main(int argc, char **argv)
12{
13     int fd;
14
15     if (argc < 2)
16     {
17         fprintf(stdout, "Usage: %s <filename>\n", argv[0]);
18         exit(1);
19     }
20
21     //int open(const char *path, int oflag, ...);
22     if ((fd = open(argv[1], O_RDONLY)) < 0)
23     {
24         fprintf(stderr, "open fifo %s for reading failed: %s\n",
25                 argv[1], strerror(errno));
26         exit(1);
27     }
28
29     fprintf(stdout, "open fifo %s for reading succeeded.\n", argv[0]);
30     char buffer[BUFFER_SIZE];
```



```
31     ssize_t n;
32
33     while (1)
34     {
35 again:
36         //ssize_t read(int fd, void *buf, size_t count);
37         if ((n = read(fd, buffer, BUFFER_SIZE)) < 0)
38         {
39             if (errno == EINTR)
40             {
41                 goto again;
42             }
43             else
44             {
45                 fprintf(stderr, "read failed on %s: %s\n", argv[1], strerror(errno));
46                 exit(1);
47             }
48         }
49         else if (n == 0)
50         {
51             fprintf(stderr, "peer closed fifo.\n");
52             break;
53         }
54         else
55         {
56             buffer[n] = '\0';
57             fprintf(stdout, "read %d bytes from fifo: %s\n", n, buffer);
58         }
59     }
60     return 0;
61 }
```

write_fifo.c

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <unistd.h>
4#include <sys/stat.h>
5#include <fcntl.h>
6#include <signal.h>
7#include <string.h>
8#include <errno.h>
9
10#define BUFFER_SIZE 1024
11
12void signal_handler(int s);
```



```
13
14 int main(int argc, char **argv)
15 {
16     int fd;
17
18     if (argc < 2)
19     {
20         fprintf(stdout, "Usage: %s <filename>\n", argv[0]);
21         exit(1);
22     }
23
24     signal(SIGPIPE, signal_handler);
25
26     //int open(const char *path, int oflag, ...);
27     if ((fd = open(argv[1], O_WRONLY)) < 0)
28     {
29         fprintf(stderr, "open fifo %s for writting failed: %s\n", argv[1],
30                                     strerror(errno));
31         exit(1);
32     }
33
34     fprintf(stdout, "open fifo %s for writting succeeded.\n", argv[0]);
35
36     char buffer[BUFFER_SIZE];
37     ssize_t n;
38
39     //char *fgets(char *s, int size, FILE * stream);
40     while (fgets(buffer, BUFFER_SIZE, stdin))
41     {
42 again:
43         //ssize_t write(int fd, const void *buf, size_t count);
44         if ((n = write(fd, buffer, strlen(buffer))) < 0)
45         {
46             if (errno == EINTR)
47             {
48                 goto again;
49             }
50             else
51             {
52                 fprintf(stderr, "write() failed on fifo: %s\n", strerror(errno));
53                 break;
54             }
55         } // end if
56     } // end while
```



```
57
58     return 0;
59 }
60
61 void signal_handler(int signo)
62 {
    fprintf(stdout, "Caught signal %d\n", signo);
}
```

10.4 Linux 系统信号机制实验

10.4.1 实验目的

熟悉 LINUX 系统中进程之间信号通信的基本原理。

10.4.2 实验平台

华清远见开发环境

10.4.3 实验内容

编写一段程序，使用系统调用 `fork()` 创建两个子进程，再用系统调用 `signal()` 让父进程捕捉键盘上来的中断信号（即按 `ctrl+c` 键），当捕捉到中断信号后，父进程用系统调用 `kill()` 向两个子进程发出信号，子进程捕捉到父进程发来的信号后，分别输出下列信息后终止：

Child process 1 is killed by parent!

Child process 2 is killed by parent!

父进程等待两个子进程终止后，输出以下信息后终止：

Parent process exit!

10.4.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/10-comm
```

```
$ mkdir signal
```

```
$ cd signal
```

```
10-comm 11-pthread 12-network 13-shell 14-gcc 15-makefile 16-to 17-process
linux@ubuntu64-vm:~/workdir/linux/application$ cd 10-comm/
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ ls
fifo pipe
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ mkdir signal
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ cd signal/
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/signal$
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\14. Linux 系统信号机制实验\实验代码】目录下的“signal.c”、拷贝到该目录下。



执行编译命令：

```
$ gcc signal.c -o signal
```

执行程序：

```
$ ./signal
```

10.4.5 实现现象

按“Ctrl+c”键，可以看到终端输出结果：

```
^Cchild process 1 is killed by parent!
```

```
child process 2 is killed by parent!
```

```
parent process exit!
```

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/signal$ ls
signal.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/signal$ gcc signal.c -o signal
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/signal$ ./signal
^Cchild process 2 is killed by parent!
child process 1 is killed by parent!
parent process exit!
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/signal$
```

10.4.6 实验代码

signal.c

```
1#include<stdio.h>
2#include<signal.h>
3#include<unistd.h>
4#include<stdlib.h>
5
6int wait_mark;
7void waiting(), stop();
8
9int main(int argc, char *argv[])
10{
11    int p1, p2;
12    signal(SIGINT, stop);
13
14    while((p1 = fork()) == -1);
15
16    if (p1 > 0)                /*在父进程中*/
17    {
18        while((p2 = fork()) == -1);
19
20        if (p2 > 0)            /*在父进程中*/
21        {
```



```
22     wait_mark = 1;
23     waiting();
24     kill(p1, 10);
25     kill(p2, 12);
26     wait(NULL);
27     wait(NULL);
28     printf("parent process exit!\n");
29     exit(0);
30 }
31 else                                     /*在子进程 2 中*/
32 {
33     wait_mark = 1;
34     signal(12, stop);
35     waiting();
36     lockf(1, 1, 0);
37     printf("child process 2 is killed by parent!\n");
38     lockf(1, 0, 0);
39     exit(0);
40 }
41 }
42 else                                     /*在子进程 1 中*/
43 {
44     wait_mark = 1;
45     signal(10, stop);
46     waiting();
47     lockf(1, 1, 0);
48     printf("child process 1 is killed by parent!\n");
49     lockf(1, 0, 0);
50     exit(0);
51 }
52}
53
54void waiting()
55{
56    while(wait_mark != 0);
57}
58
59void stop()
60{
61    wait_mark = 0;
62}
```



10.5 Linux 系统共享内存通信实验

10.5.1 实验目的

熟悉 UNIX/LINUX 支持的共享内存通信方式。

10.5.2 实验平台

华清远见开发环境

10.5.3 实验内容

reader 和 writer 两个进程通过共享内存交换数据。writer 从标准输入读入字符串写入共享内存，reader 把共享内存里的字符串打印到标准输出。

reader 和 writer 通过信号实现同步。

reader 和 writer 通过信号通信必须获取对方的进程号，可利用共享内存保存双方的进程号。

reader 和 writer 运行的顺序不确定，可约定先运行的进程创建共享内存并初始化。

10.5.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/10-comm
```

```
$ mkdir shmem
```

```
$ cd shmem
```

```
linux@ubuntu64-vm:~/workdir/linux/application$ cd 10-comm/  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ ls  
fifo pipe signal  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ mkdir shmem  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ ls  
fifo pipe shmem signal  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ cd shmem/  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shmem$
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\16. Linux 系统共享内存通信实验\实验代码】目录下的“reader.c”、“writer.c”拷贝到该目录下。

执行编译命令：

```
$ gcc reader.c -o reader
```

```
$ gcc writer.c -o writer
```

```
$ ./reader //执行 read 程序
```




```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ cd shm/
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shm$ ls
reader.c  writer.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shm$ gcc reader.c -o reader
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shm$ gcc writer.c -o writer
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shm$ ./reader
```

开启另一个终端，进入目录

```
$ ./writer           //执行 write 程序
please input : hello //输入 “hello”
```

```
linux@ubuntu64-vm:~/workdir/linux/application$ cd 10-comm/shm/
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shm$ ls
reader reader.c writer writer.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shm$ ./writer
please input : hello
please input :
```

10.5.5 实验现象

在执行 read 程序的终端，会出现：

message from shm : hello

这样就成功的实现了对共享内存的读和写

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shm$ gcc writer.c -o writer
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/shm$ ./reader
message from shm : hello
```

10.5.6 实验代码

reader.c

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <sys/types.h>
4#include <sys/shm.h>
5#include <sys/ipc.h>
6#include <unistd.h>
7#include <signal.h>
8#include <errno.h>
9
10#define N 64
11
12typedef struct
13{
14     int pid;
15     char buf[N];
16} shmbuf;
```



```
17
18 void handler(int signo)
19 {
20     return;
21 }
22
23 int main(int argc, char *argv[])
24 {
25     int shmid;
26     key_t key;
27     pid_t pid;
28     shmbuf *shmaddr;
29
30     signal(SIGUSR1, handler);
31     if ((key = ftok(".", 'a')) < 0)
32     {
33         perror("fail to ftok");
34         exit(-1);
35     }
36     if ((shmid = shmget(key, sizeof(shmbuf), IPC_CREAT | IPC_EXCL | 0666)) < 0)
37     {
38         if (errno == EEXIST)
39         {
40             shmid = shmget(key, sizeof(shmbuf), 0666);
41             shmaddr = (shmbuf *)shmat(shmid, NULL, 0);
42             pid = shmaddr->pid;
43             shmaddr->pid = getpid();
44             kill(pid, SIGUSR1);
45         }
46         else
47         {
48             perror("fail to shmget");
49             exit(-1);
50         }
51     }
52     else
53     {
54         shmaddr = (shmbuf *)shmat(shmid, NULL, 0);
55         shmaddr->pid = getpid();
56         pause();
57         pid = shmaddr->pid;
58     }
59
60     while ( 1 )
```



```
61 {
62     pause();
63     if ( strcmp(shmaddr->buf, "quit", 4) == 0)
64     {
65         break;
66     };
67     printf("message from shm : %s", shmaddr->buf);
68     usleep(100000);
69     kill(pid, SIGUSR1);
70 }
71 shmdt(shmaddr);
72
73     return 0;
74 }
```

writer.c

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <sys/types.h>
4#include <sys/shm.h>
5#include <sys/ipc.h>
6#include <unistd.h>
7#include <signal.h>
8#include <errno.h>
9
10#define N 64
11
12typedef struct
13{
14     int pid;
15     char buf[N];
16} shmbuf;
17
18void handler(int signo)
19{
20     return;
21}
22
23int main(int argc, char *argv[])
24{
25     int shmid;
26     key_t key;
27     pid_t pid;
28     shmbuf *shmaddr;
29
```



```
30 signal(SIGUSR1, handler);
31 if ((key = ftok(".", 'a')) < 0)
32 {
33     perror("fail to ftok");
34     exit(-1);
35 }
36 if((shmid = shmget(key, sizeof(shmbuf), IPC_CREAT | IPC_EXCL | 0666)) < 0)
37 {
38     if (errno == EEXIST)
39     {
40         shmid = shmget(key, sizeof(shmbuf), 0666);
41         shmaddr = (shmbuf *)shmat(shmid, NULL, 0);
42         pid = shmaddr->pid;
43         shmaddr->pid = getpid();
44         kill(pid, SIGUSR1);
45     }
46     else
47     {
48         perror("fail to shmget");
49         exit(-1);
50     }
51 }
52 else
53 {
54     shmaddr = (shmbuf *)shmat(shmid, NULL, 0);
55     shmaddr->pid = getpid();
56     pause();
57     pid = shmaddr->pid;
58 }
59
60 while ( 1 )
61 {
62     printf("please input : ");
63     fgets(shmaddr->buf, N, stdin);
64     kill(pid, SIGUSR1);
65     if (strncmp(shmaddr->buf, "quit", 4) == 0)
66     {
67         break;
68     }
69     pause();
70 }
71 sleep(1);
72 shmdt(shmaddr);
73 shmctl(shmid, IPC_RMID, NULL);
```



```
74  
75     return 0;  
}
```

10.6 Linux 系统消息队列实验

10.6.1 实验目的

熟悉 UNIX/LINUX 支持的消息队列通信方式。

10.6.2 实验平台

华清远见开发环境

10.6.3 实验内容

本实验需要用消息队列设计一个简易的双人聊天程序（一个服务器，两个客户端）。消息队列重点在于消息类型的匹配，客户端和服务端的“通信协议”的设计。设计思想如下：

服务器端：接受客户端发来的任何信息，并根据其消息类型，转发给对应的客户端。同时，检测是否有退出标志，有则给所有的客户端发送退出标志，等待 1s 后，确定客户端都退出后，删除消息队列，释放空间，并退出。

客户端：A 和 B。A 给 B 发送信息，先发给服务器，由服务器根据自定义协议转发该消息给 B。同 B 可以也通过服务器给 A 发消息。

10.6.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/10-comm  
$ mkdir msg  
$ cd msg
```

```
linux@ubuntu64-vm:~/workdir/linux/application$ cd 10-comm/  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ ls  
fifo pipe shmem signal  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ mkdir msg  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ cd msg/  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ls  
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$
```

将【华清远见-CORTEXA9 资料： 程序源码\Linux 应用实验源码\20. Linux 系统消息队列实验\实验代码】目录下的“client1.c”、“client2.c”、“server.c”拷贝到该目录下。



执行编译命令：

```
$ gcc client1.c -o client1
$ gcc client2.c -o client2
$ gcc server.c -o server
```

运行测试。先编译运行服务器，确保消息队列已经创建完毕，并进入等待状态。结果如下：

```
$ ./server
```

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ls
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ls
client1.c client2.c server.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ gcc server.c -o server
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ gcc client1.c -o client1
client1.c: 在函数‘main’中:
client1.c:25: 警告：不建议使用‘gets’(声明于 /usr/include/stdio.h:638)
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ gcc client2.c -o client2
client2.c: 在函数‘main’中:
client2.c:33: 警告：不建议使用‘gets’(声明于 /usr/include/stdio.h:638)
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ./server
```

打开另一个终端，运行客户端 A，结果如下：

```
$ ./client1
```

```
linux@ubuntu64-vm:~/workdir/linux/application$ cd 10-comm/msg/
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ls
client1 client1.c client2 client2.c server server.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ./client1
input the msg to client2:
```

打开另一个终端，运行客户端 B，结果如下：

```
$ ./client2
```

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ ls
fifo msg pipe shm signal
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ cd msg/
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ls
client1 client1.c client2 client2.c server server.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ./client2
```

10.6.5 实验现象

在客户端 A 下输入 hello client1，如下图所示。



```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ls
client1 client1.c client2 client2.c server server.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ./client1
input the msg to client2:hello client1
```

客户端 B 如下图所示

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ls
client1 client1.c client2 client2.c server server.c
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ./client2
Receive from client1, message: hello client1
input the msg to client1:
```

服务器端如下图所示

```
client2.c:33: 警告：不建议使用‘gets’(声明于 /usr/include/stdio.h:638)
linux@ubuntu64-vm:~/workdir/linux/application/10-comm/msg$ ./server
Receive client1 message: hello client1
```

10.6.6 实验代码

server.c

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <sys/types.h>
4#include <sys/ipc.h>
5#include <sys/msg.h>
6#include <string.h>
7
8#define KEY_MSG 0x101 //使用共有的 IPC key
9#define MSGSIZE 64
10
11typedef struct //定义消息结构体：消息类型和消息数据
12{
13    long mtype;
14    char mtext[MSGSIZE];
15} msgbuf;
16
17#define LEN sizeof(msgbuf) - sizeof(long)
18
19int main()
20{
21    int msgid;
```



```

22 msgbuf buf1, buf2;
23 msgid = msgget( KEY_MSG, IPC_CREAT | 0666 );
24 while(1)
25 {
26     msgrcv(msgid, &buf1, LEN, 1L, 0);
27     //接受客户端 1 的消息
28     printf( "Receive client1 message: %s\n", buf1.mtext ); //打印收到的消息
29     if ( buf1.mtext[0] == 'x' || buf1.mtext[0] == 'X' )
30         //若是退出标志, 则给 2 个客户端都发退出信息
31     {
32         strcpy( buf1.mtext, "x" );
33         buf1.mtype = 3L;
34         msgsnd( msgid, &buf1, LEN, 0 );
35         buf1.mtype = 4L;
36         msgsnd( msgid, &buf1, LEN, 0 );
37         break;
38     }
39     buf1.mtype = 4L;
40     msgsnd( msgid, &buf1, LEN, 0 ); //将客户端 1 的消息转发给客户端 2
41     msgrcv( msgid, &buf2, LEN, 2L, 0 ); //接受客户端 2 的消息
42     printf( "Receive client2 message: %s\n", buf2.mtext ); //打印收到的消息
43     if ( buf2.mtext[0] == 'x' || buf2.mtext[0] == 'X' )
44         //若是退出标志, 则给 2 个客户端发退出信息
45     {
46         strcpy( buf2.mtext, "x" );
47         buf2.mtype = 3L;
48         msgsnd( msgid, &buf2, LEN, 0 );
49         buf2.mtype = 4L;
50         msgsnd( msgid, &buf2, LEN, 0 );
51         break;
52     }
53     buf2.mtype = 3L;
54     msgsnd( msgid, &buf2, LEN, 0 ); //将客户端 2 的消息转发给客户端 1
55 }
56 sleep(1); //若退出, 则先等待, 以确保客户端程序退出
57 msgctl( msgid, IPC_RMID, NULL ); //删除消息队列, 释放空间
58 exit(0);
59 }

```

client1.c

```

1#include <stdio.h>
2#include <sys/types.h>
3#include <sys/ipc.h>
4#include <sys/msg.h>
5#include <string.h>

```




```
6
7#define KEY_MSG 0x101
8#define MSGSIZE 64
9
10typedef struct
11{
12    long mtype;
13    char mtext[MSGSIZE];
14} msgbuf;
15
16#define LEN sizeof(msgbuf) - sizeof(long)
17
18int main()
19{
20    int msgid;
21    msgbuf buf1, buf2;
22    msgid = msgget( KEY_MSG, 0666 );
23    while( 1 )
24    {
25        printf( "input the msg to client2:" );
26        gets( buf1.mtext );
27        buf1.mtype = 1L;
28        msgsnd( msgid, &buf1, LEN, 0 ); //客户端 1 获取消息并发往服务器
29
30        msgrcv( msgid, &buf2, LEN, 3L, 0 ); //准备从客户端 2 获取消息
31        if ( buf2.mtext[0] == 'x' || buf2.mtext[0] == 'X' )
32        {
33            printf( "client1 will quit!\n" );
34            break;
35        }
36        printf( "Receive from client2, message: %s\n", buf2.mtext );
37    }
38    return 0;
39}
```

client2.c

```
1#include <stdio.h>
2#include <sys/types.h>
3#include <sys/ipc.h>
4#include <sys/msg.h>
5#include <string.h>
6
7#define KEY_MSG 0x101
8#define MSGSIZE 64
9
```



```
10 typedef struct
11 {
12     long mtype;
13     char mtext[MSGSIZE];
14 } msgbuf;
15
16 #define LEN  sizeof(msgbuf) - sizeof(long)
17
18 int main()
19 {
20     int msgid;
21     msgbuf buf1, buf2;
22     msgid = msgget( KEY_MSG, 0666 );
23     while( 1 )
24     {
25         msgrcv( msgid, &buf2, LEN, 4L, 0 ); //等待客户端 1 发消息
26         if( buf2.mtext[0] == 'x' || buf2.mtext[0] == 'X' )
27         {
28             printf( "client2 will quit!\n" );
29             break;
30         }
31         else printf( "Receive from client1, message: %s\n", buf2.mtext );
32
33         sleep(1); //等待一秒, 以确保客户端 1 已经收到了回执
34         printf( "input the msg to client1:" );
35         gets( buf1.mtext );
36         buf1.mtype = 2L;
37         msgsnd( msgid, &buf1, LEN, 0 ); //给客户端 1 发送回执消息
38     }
39 }
```



第 11 章 嵌入式 Linux 多线程编程实验

11.1 实验目的

通过实验学习 Linux 下多线程编程。

11.2 实验平台

华清远见开发环境。

11.3 实验原理

11.3.1 线程基本编程

这里要讲的线程相关操作都是用户空间中的线程的操作。在 Linux 中，一般 pthread 线程库是一套通用的线程库，是由 POSIX 提出的，因此具有很好的可移植性。

创建线程实际上就是确定调用该线程函数的入口点，这里通常使用的函数是 `pthread_create()`。在线程创建之后，就开始运行相关的线程函数，在该函数运行完之后，该线程也就退出了，这也是线程退出的一种方法。另一种退出线程的方法是使用函数 `pthread_exit()`，这是线程的主动行为。这里要注意的是，在使用线程函数时，不能随意使用 `exit()` 退出函数进行出错处理，由于 `exit()` 的作用是使调用进程终止，往往一个进程包含多个线程，因此，在使用 `exit()` 之后，该进程中的所有线程都终止了。因此，在线程中就可以使用 `pthread_exit()` 来代替进程中的 `exit()`。

由于一个进程中的多个线程是共享数据段的，因此通常在线程退出之后，退出线程所占用的资源并不会随着线程的终止而得到释放。正如进程之间可以用 `wait()` 系统调用来同步终止并释放资源一样，线程之间也有类似机制，那就是 `pthread_join()` 函数。`pthread_join()` 可以用于将当前线程挂起来等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源就被收回。

前面已提到线程调用 `pthread_exit()` 函数主动终止自身线程。但是在很多线程应用中，经常会遇到在别的线程中要终止另一个线程的问题。此时调用 `pthread_cancel()` 函数实现这种功能，但在被取消的线程的内部需要调用 `pthread_setcancel()` 函数和 `pthread_setcanceltype()` 函数设置自己的取消状态，例如被取消的线程接收到另一个线程的取消请求之后，是接受还是忽略这个请求；如果是接受，则再判断立刻采取终止操作还是等待某个函数的调用等。

表 7.1 列出了 `pthread_create()` 函数的语法要点。

表 7.1 `pthread_create()` 函数语法要点

所需 头文件	<code>#include <pthread.h></code>
函数 原型	<code>int pthread_create ((pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg))</code>
函数 传入值	<p><code>thread</code>: 线程标识符</p> <p><code>attr</code>: 线程属性设置（其具体设置参见 6.4.3 小节），通常取为 <code>NULL</code></p>



函数 返回值	start_routine: 线程函数的起始地址, 是一个以指向 void 的指针作为参数和返回值的函数指针
	arg: 传递给 start_routine 的参数
	成功: 0 出错: 返回错误码

表 7.2 列出了 pthread_exit()函数的语法要点。

表 7.2 pthread_exit()函数语法要点

所需头文件	#include <pthread.h>
函数原型	void pthread_exit(void *retval)
函数传入值	retval: 线程结束时的返回值, 可由其他函数如 pthread_join()来获取

表 7.3 列出了 pthread_join()函数的语法要点。

表 7.3 pthread_join()函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_join((pthread_t th, void **thread_return))
函数传入值	th: 等待线程的标识符
	thread_return: 用户定义的指针, 用来存储被等待线程结束时的返回值 (不为 NULL 时)
函数返回值	成功: 0
	出错: 返回错误码

表 7.4 列出了 pthread_cancel()函数的语法要点。

表 7.4 pthread_cancel()函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_cancel(pthread_t th)
函数传入值	th: 要取消的线程的标识符



函数返回值	成功: 0
出错: 返回错误码	

11.3.2 线程之间的同步与互斥

由于线程共享进程的资源和地址空间，因此在对这些资源进行操作时，必须考虑到线程间资源访问的同步与互斥问题。这里主要介绍 POSIX 中两种线程同步机制，分别为互斥锁和信号量。这两个同步机制可以互相通过调用对方来实现，但互斥锁更适合用于同时可用的资源是唯一的情况；信号量更适合用于同时可用的资源为多个的情况。

1. 互斥锁线程控制

互斥锁是用一种简单的加锁方法来控制对共享资源的原子操作。这个互斥锁只有两种状态，也就是上锁和解锁，可以把互斥锁看作某种意义上的全局变量。在同一时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经被上锁的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。可以说，这把互斥锁保证让每个线程对共享资源按顺序进行原子操作。

互斥锁机制主要包括下面的基本函数。

- 互斥锁初始化: `pthread_mutex_init()`
- 互斥锁上锁: `pthread_mutex_lock()`
- 互斥锁判断上锁: `pthread_mutex_trylock()`
- 互斥锁解锁: `pthread_mutex_unlock()`
- 消除互斥锁: `pthread_mutex_destroy()`

其中，互斥锁可以分为快速互斥锁、递归互斥锁和检错互斥锁。这三种锁的区别主要在于其他未占有互斥锁的线程在希望得到互斥锁时是否需要阻塞等待。快速锁是指调用线程会阻塞直至拥有互斥锁的线程解锁为止。递归互斥锁能够成功地返回，并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。默认属性为快速互斥锁。

表 7.5 列出了 `pthread_mutex_init()` 函数的语法要点。

表 7.5 pthread_mutex_init() 函数语法要点

所需头文件	#include <pthread.h>	
函数原型	int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)	
函数传入值	mutex: 互斥锁	
函数传入值	Mutexattr	PTHREAD_MUTEX_INITIALIZER: 创建快速互斥锁
		PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP: 创建递归互斥锁
		PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP: 创建检错互斥锁
函数返回值	成功: 0	



出错：返回错误码

表 7.6 列出了 pthread_mutex_lock()等函数的语法要点。

表 7.6 pthread_mutex_lock()等函数语法要点

所需头文件	#include <pthread.h>
函数原型	<pre>int pthread_mutex_lock(pthread_mutex_t *mutex,) int pthread_mutex_trylock(pthread_mutex_t *mutex,) int pthread_mutex_unlock(pthread_mutex_t *mutex,) int pthread_mutex_destroy(pthread_mutex_t *mutex,)</pre>
函数传入值	mutex：互斥锁
函数返回值	成功：0
	出错：-1

2. 信号量线程控制

在前面已经讲到，信号量也就是操作系统中所用到的 PV 原子操作，它广泛用于进程或线程间的同步与互斥。信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。这里先来简单复习一下 PV 原子操作的工作原理。

PV 原子操作是对整数计数器信号量 sem 的操作。一次 P 操作使 sem 减一，而一次 V 操作使 sem 加一。进程（或线程）根据信号量的值来判断是否对公共资源具有访问权限。当信号量 sem 的值大于等于零时，该进程（或线程）具有公共资源的访问权限；相反，当信号量 sem 的值小于零时，该进程（或线程）就将阻塞直到信号量 sem 的值大于等于 0 为止。

PV 原子操作主要用于进程或线程间的同步和互斥这两种典型情况。若用于互斥，几个进程（或线程）往往只设置一个信号量 sem，它们的操作流程如图 5.1 所示。

当信号量用于同步操作时，往往会设置多个信号量，并安排不同的初始值来实现它们之间的顺序执行，它们的操作流程如图 5.2 所示。

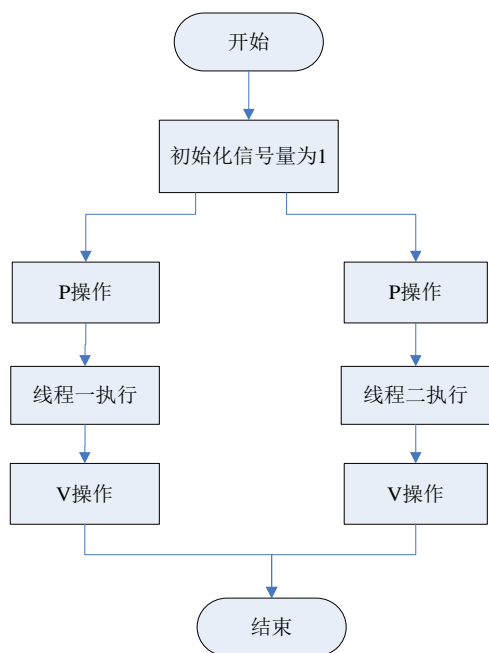


图 7.1 信号量互斥操作

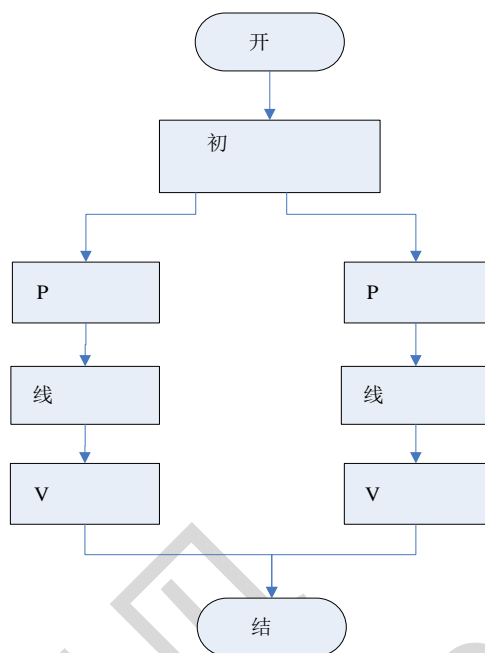


图 7.2 信号量同步操作

Linux 实现了 POSIX 的无名信号量，主要用于线程间的互斥与同步。这里主要介绍几个常见函数。

- `sem_init()`用于创建一个信号量，并初始化它的值。
- `sem_wait()`和 `sem_trywait()`都相当于 P 操作，在信号量大于零时它们都能将信号量的值减一，两者的区别在于若信号量小于零时，`sem_wait()`将会阻塞进程，而 `sem_trywait()`则会立即返回。
- `sem_post()`相当于 V 操作，它将信号量的值加一同时发出信号来唤醒等待的进程。
- `sem_getvalue()`用于得到信号量的值。
- `sem_destroy()`用于删除信号量。

表 7.7 列出了 `sem_init()`函数的语法要点。

表 7.7 `sem_init()`函数语法要点

所需头文件	#include <semaphore.h>
函数原型	int sem_init(sem_t *sem,int pshared,unsigned int value)
函数传入值	sem: 信号量指针
	pshared: 决定信号量能否在几个进程间共享。由于目前 Linux 还没有实现进程间共享信号量，所以这个值只能取 0，就表示这个信号量是当前进程的局部信号量
	value: 信号量初始化值
函数返回值	成功: 0
	出错: -1

表 7.8 列出了 `sem_wait()`等函数的语法要点。

表 7.8 `sem_wait()`等函数语法要点



所需头文件	#include <pthread.h>
函数原型	<pre>int sem_wait(sem_t *sem) int sem_trywait(sem_t *sem) int sem_post(sem_t *sem) int sem_getvalue(sem_t *sem) int sem_destroy(sem_t *sem)</pre>
函数传入值	sem: 信号量指针
函数返回值	成功: 0 出错: -1

11.3.3 线程属性

读者是否还记得 `pthread_create()` 函数的第二个参数 (`pthread_attr_t *attr`) 表示线程的属性。在上一个实例中, 将该值设为 `NULL`, 也就是采用默认属性, 线程的多项属性都是可以更改的。这些属性主要包括绑定属性、分离属性、堆栈地址、堆栈大小以及优先级。其中系统默认的属性为非绑定、非分离、缺省 1M 的堆栈以及与父进程同样级别的优先级。下面首先对绑定属性和分离属性的基本概念进行讲解。

- 绑定属性

前面已经提到, Linux 中采用“一对一”的线程机制, 也就是一个用户线程对应一个内核线程。绑定属性就是指一个用户线程固定地分配给一个内核线程, 因为 CPU 时间片的调度是面向内核线程 (也就是轻量级进程) 的, 因此具有绑定属性的线程可以保证在需要的时候总有一个内核线程与之对应。而与之对应的非绑定属性就是指用户线程和内核线程的关系不是始终固定的, 而是由系统来控制分配的。

- 分离属性

分离属性是用来决定一个线程以什么样的方式来终止自己。在非分离情况下, 当一个线程结束时, 它所占用的系统资源并没有被释放, 也就是没有真正的终止。只有当 `pthread_join()` 函数返回时, 创建的线程才能释放自己占用的系统资源。而在分离属性情况下, 一个线程结束时立即释放它所占有的系统资源。这里要注意的一点是, 如果设置一个线程的分离属性, 而这个线程运行又非常快, 那么它很可能在 `pthread_create()` 函数返回之前就终止了, 它终止以后就可能将线程号和系统资源移交给其他的线程使用, 这时调用 `pthread_create()` 的线程就得到了错误的线程号。

这些属性的设置都是通过特定的函数来完成的, 通常首先调用 `pthread_attr_init()` 函数进行初始化, 之后再调用相应的属性设置函数, 最后调用 `pthread_attr_destroy()` 函数对分配的属性结构指针进行清理和回收。设置绑定属性的函数为 `pthread_attr_setscope()`, 设置线程分离属性的函数为 `pthread_attr_setdetachstate()`, 设置线程优先级的相关函数为 `pthread_attr_getschedparam()` (获取线程优先级) 和 `pthread_attr_setschedparam()` (设置线程优先级)。在设置完这些属性后, 就可以调用 `pthread_create()` 函数来创建线程了。

表 7.9 列出了 `pthread_attr_init()` 函数的语法要点。

表 7.9 `pthread_attr_init()` 函数语法要点



所需头文件	#include <pthread.h>
函数原型	int pthread_attr_init(pthread_attr_t *attr)
函数传入值	attr: 线程属性结构指针
函数返回值	成功: 0
	出错: 返回错误码

表 7.10 列出了 pthread_attr_setscope()函数的语法要点。

表 7.10 pthread_attr_setscope()函数语法要点

所需头文件	#include <pthread.h>	
函数原型	int pthread_attr_setscope(pthread_attr_t *attr, int scope)	
函数传入值	attr: 线程属性结构指针	
	scope	PTHREAD_SCOPE_SYSTEM: 绑定
		PTHREAD_SCOPE_PROCESS: 非绑定
函数返回值	成功: 0	
	出错: -1	

表 7.11 列出了 pthread_attr_setdetachstate()函数的语法要点。

表 7.11 pthread_attr_setdetachstate()函数语法要点

所需头文件	#include <pthread.h>	
函数原型	int pthread_attr_setscope(pthread_attr_t *attr, int detachstate)	
函数传入值	attr: 线程属性	
	detachstate	PTHREAD_CREATE_DETACHED: 分离
		PTHREAD_CREATE_JOINABLE: 非分离
函数返回值	成功: 0	
	出错: 返回错误码	

表 5.12 列出了 pthread_attr_getschedparam()函数的语法要点。

表 5.12 pthread_attr_getschedparam()函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param)
函数传入值	attr: 线程属性结构指针
	param: 线程优先级
函数返回值	成功: 0



出错：返回错误码

表 7.13 列出了 pthread_attr_setschedparam()函数的语法要点。

表 7.13 pthread_attr_setschedparam()函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_setschedparam (pthread_attr_t *attr, struct sched_param *param)
函数传入值	attr: 线程属性结构指针
	param: 线程优先级
函数返回值	成功: 0
	出错: 返回错误码

11.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/11-pthread
```

```
$ mkdir pthread
```

```
$ cd pthread
```

```
linux@ubuntu64-vm:~/workdir/linux/application/10-comm$ cd ..
linux@ubuntu64-vm:~/workdir/linux/application$ ls
10-comm 5-shell 6-gcc 7-Makefile 8-io 9-process
linux@ubuntu64-vm:~/workdir/linux/application$ mkdir 11-pthread
linux@ubuntu64-vm:~/workdir/linux/application$ cd 1
bash: cd: 1: 没有那个文件或目录
linux@ubuntu64-vm:~/workdir/linux/application$ cd 11-pthread/
linux@ubuntu64-vm:~/workdir/linux/application/11-pthread$ ls
linux@ubuntu64-vm:~/workdir/linux/application/11-pthread$ mkdir pthread
linux@ubuntu64-vm:~/workdir/linux/application/11-pthread$ cd pthread/
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\20. Linux 系统消息队列实验\实验代码】目录下的“thread.c”拷贝到该目录下。

执行编译命令：

```
$ gcc thread.c -o thread -lpthread
```

运行程序：

```
$ ./thread
```



11.4.1 实验现象

```
更多信息请见: "vim -h"
linux@ubuntu64-vm:~/workdir/linux/application/11-pthread/pthread$ gcc thread.c -o thread -lpthread
linux@ubuntu64-vm:~/workdir/linux/application/11-pthread/pthread$ ./thread
dadf

welcome
***
result:welcome
```

11.5 实验代码

thread.c

```
1#include <stdio.h>
2#include <time.h>
3#include <stdlib.h>
4#include <errno.h>
5#include <unistd.h>
6#include <pthread.h>
7
8//char buf[] = "welcome";
9
10void *handler(void *p) //void *p = buf;
11{
12    getchar();
13    printf("%s\n", (char *)p);
14
15    //return p;
16    pthread_exit(p);
17}
18
19void f(char **p)//char **p = &s;
20{
21    char *q = "dadf";
22    *p = q;
23}
24
25int main(int argc, char *argv[])
26{
27    char *s = NULL;
28    f(&s);
29    printf("%s\n", s);
30
31#if 1
32    pthread_t thread;
```



```
33 char buf[] = "welcome";
34 void *result;
35
36 if (pthread_create(&thread, NULL, handler, (void *)buf))
37     exit(-1);
38
39 pthread_join(thread, &result); //exit(0); waitpid(pid, &status, 0);
40
41 printf("***\n");
42 printf("result:%s\n", (char *)result);
43
44 while (1);
45 #endif
46
47 exit(0);
}
```

华清远见
dev.hqyj.com



第 12 章 嵌入式 Linux 网络编程实验

12.1 实验原理

12.1.1 TCP/IP 的分层模型

读者一定都听说过著名的 OSI 协议参考模型，它是基于国际标准化组织（ISO）的建议发展起来的，它分为 7 个层次：应用层、表示层、会话层、传输层、网络层、数据链路层及物理层。这个 7 层的协议模型虽然规定得非常细致和完善，但在实际中却得不到广泛的应用，其重要的原因之一就在于它过于复杂。但它仍是此后很多协议模型的基础。与此相区别的 TCP/IP 协议模型将 OSI 的 7 层协议模型简化为 4 层，从而更有利于实现和使用。TCP/IP 的协议参考模型和 OSI 协议参考模型的对应关系如下图 8.1 所示。

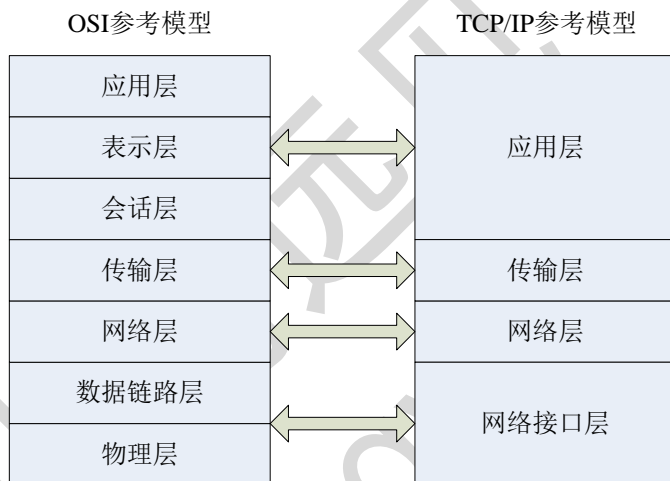


图 8.1 OSI 模型和 TCP/IP 参考模型对应关系

TCP/IP 协议是一个复制的协议，是由一组专业化协议组成的。这些协议包括 IP、TCP、UDP、ARP、ICMP 以及其他的一些被称为子协议的协议。TCP/IP 协议的前身是由美国国防部在 20 世纪 60 年代末期为其远景研究规划署网络（ARPANET）而开发的。由于低成本以及在多个不同平台通信的可靠性，TCP/IP 迅速发展并开始流行。它实际上是一个关于因特网的标准，迅速成为局域网的首选协议。下面具体讲解各层在 TCP/IP 整体架构中的作用。

1. 网络接口层（Network Interface Layer）

网络接口层是 TCP/IP 协议软件的最底层，负责将二进制流转换为数据帧，并进行数据帧的发送和接收。数据帧是网络传输的基本单元。

2. 网络层（Internet Layer）

网络层负责在主机之间的通信中选择数据报的传输路径，即路由。当网络层接收到传输层的请求后，传输某个具有目的地址信息的分组。该层把分组封装在 IP 数据报中，填入数据报的首部，使用路由



算法来确定是直接交付数据报，还是把它传递给路由器，然后把数据报交给适当的网络接口进行传输。

网络层还要负责处理传入的数据报，检验其有效性，使用路由算法来决定应该对数据报进行本地处理还是应该转发。

如果数据报的目的机处于本机所在的网络，该层软件就会除去数据报的首部，再选择适当的运输层协议来处理这个分组。最后，网络层还要根据需要发出和接收 ICMP（Internet 控制报文协议）差错和控制报文。

3. 传输层（Transport Layer）

传输层负责提供应用程序之间的通信服务。这种通信又称为端到端通信。传输层要系统地管理信息的流动，还要提供可靠的传输服务，以确保数据到达无差错、无乱序。为了达到这个目的，传输层协议软件要进行协商，让接收方回送确认信息及让发送方重发丢失的分组。传输层协议软件把要传输的数据流划分为分组，把每个分组连同目的地址交给网络层去发送。

4. 应用层（Application Layer）

应用层是分层模型的最高层，在这个最高层中，用户调用应用程序通过 TCP/IP 互联网来访问可行的服务。与各个传输层协议交互的应用程序负责接收和发送数据。每个应用程序选择适当的传输服务类型，把数据按照传输层的格式要求封装好向下层传输。

综上所述，TCP/IP 分层模型每一层负责不同的通信功能，整体联动合作，就可以完成互联网的大部分传输要求。

12.1.2 TCP/IP 分层模型特点

TCP/IP 是目前 Internet 上最成功、使用最频繁的互联协议。虽然现在已有很多协议都适用于互联网，但 TCP/IP 的使用最广泛。下面讲解一下 TCP/IP 的特点。

1. TCP/IP 模型边界特性

TCP/IP 分层模型中有两大边界特性：一个是地址边界特性，它将 IP 逻辑地址与底层网络的硬件地址分开；一个是操作系统边界特性，它将网络应用与协议软件分开，如图 8.2 所示。

TCP/IP 分层模型边界特性是指在模型中存在一个地址上的边界，它将底层网络的物理地址与网络层的 IP 地址分开。该边界出现在网络层与网络接口层之间。

应用层	操作系统外部
传输层	操作系统内部
网络层	IP地址
网络接口层	物理地址



图 6.2 TCP/IP 分层模型边界特性

网络层和其上的各层均使用 IP 地址，网络接口层则使用物理地址，即底层网络设备的硬件地址。TCP/IP 提供在两种地址之间进行映射的功能。划分地址边界的目的是为了屏蔽底层物理网络的地址细节，以便使互联网软件地址上易于实现和理解。

TCP/IP 软件在操作系统内具体的位置和 TCP/IP 的实现有关，但大部分实现都类似于图 6.2 所示情况。影响操作系统边界划分的最重要因素是协议的效率问题，在操作系统内部实现的协议软件，其数据传递的效率明显要高。

2. IP 层特性

IP 层作为通信子网的最高层，提供无连接的数据报传输机制，但 IP 协议并不能保证 IP 报文传递的可靠性，IP 的机制是点到点的。用 IP 进行通信的主机或路由器位于同一物理网络，对等机器之间拥有直接的物理连接。

TCP/IP 设计原则之一是为包容各种物理网络技术，包容性主要体现在 IP 层中。各种物理网络技术在帧或报文格式、地址格式等方面差别很大，TCP/IP 的重要思想之一就是通过对 IP 将各种底层网络技术统一起来，达到屏蔽底层细节，提供统一虚拟网的目的。

IP 向上层提供统一的 IP 报文，使得各种网络帧或报文格式的差异性对高层协议不复存在。IP 层是 TCP/IP 实现异构网互联最关键的一层。

3. TCP/IP 的可靠性特性

在 TCP/IP 网络中，IP 采用无连接的数据报机制，对数据进行“尽力而为”的传递机制，即只管将报文尽力传送到目的主机，无论传输正确与否，不做验证，不发确认，也不保证报文的顺序。TCP/IP 的可靠性体现在传输层协议之一的 TCP 协议。TCP 协议提供面向连接的服务，因为传输层是端到端的，所以 TCP/IP 的可靠性被称为端到端可靠性。

综上所述，TCP/IP 的特点就是将不同的底层物理网络、拓扑结构隐藏起来，向用户和应用程序提供通用、统一的网络服务。这样，从用户的角度看，整个 TCP/IP 互联网就是一个统一的整体，它独立于具体的各种物理网络技术，能够向用户提供一个通用的网络服务。

TCP/IP 网络完全撇开了底层物理网络的特性，是一个高度抽象的概念，正是由于这个原因，其为 TCP/IP 网络赋予了巨大的灵活性和通用性。

12.1.3 TCP/IP 核心协议

在 TCP/IP 协议族中，有很多种协议，如图 8.3 所示。

TCP/IP 协议群中的核心协议被设计运行在网络层和传输层，它们为网络中的各主机提供通信服务，也为模型的最高层——应用层中的协议提供服务。

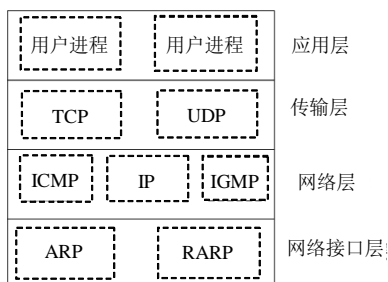


图 8.3 TCP/IP 协议族不同分层中的协议

在此主要介绍在网络编程中涉及的传输层 TCP 和 UDP 协议。

1. TCP

(1) 概述

TCP 的上一层是应用层，TCP 向应用层提供可靠的面向对象的数据流传输服务，TCP 数据传输实现了从一个应用程序到另一个应用程序的数据传递。它能提供高可靠性通信(即数据无误、数据无丢失、数据无失序、数据无重复到达的通信。)，应用程序通过向 TCP 层提交数据接发送/收端的地址和端口号而实现应用层的数据通信。

通过 IP 的源/目的可以惟一地区分网络中两个设备的连接，通过 socket 的源/目的可以惟一地区分网络中两个应用程序的连接。

(2) 3 次握手协议

TCP 是面向连接的，所谓面向连接，就是当计算机双方通信时必需先建立连接，然后进行数据通信，最后拆除连接三个过程。TCP 在建立连接时又分三步走：

第一步 (A->B)：主机 A 向主机 B 发送一个包含 SYN 即同步 (Synchronize) 标志的 TCP 报文，SYN 同步报文会指明客户端使用的端口以及 TCP 连接的初始序号；

第二步 (B->A)：主机 B 在收到客户端的 SYN 报文后，将返回一个 SYN+ACK 的报文，表示主机 B 的请求被接受，同时 TCP 序号被加一，ACK 即确认 (Acknowledgement)。

第三步 (A->B)：主机 A 也返回一个确认报文 ACK 给服务器端，同样 TCP 序列号被加一，到此一个 TCP 连接完成。图 8.4 就是这个流程的简单示意图。

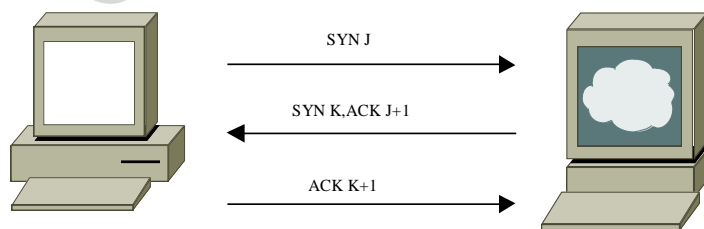


图 8.4 TCP3 次握手协议

TCP 实体所采用的基本协议是滑动窗口协议。当发送方传送一个数据报时，它将启动计时器。当该数据报到达目的地后，接收方的 TCP 实体向回发送一个数据报，其中包含有一个确认序号，它意思是希



望收到的下一个数据报的序号。如果发送方的定时器在确认信息到达之前超时，那么发送方会重发该数据报。

(3) TCP 数据报头

图 8.5 为 TCP 数据报头的格式。

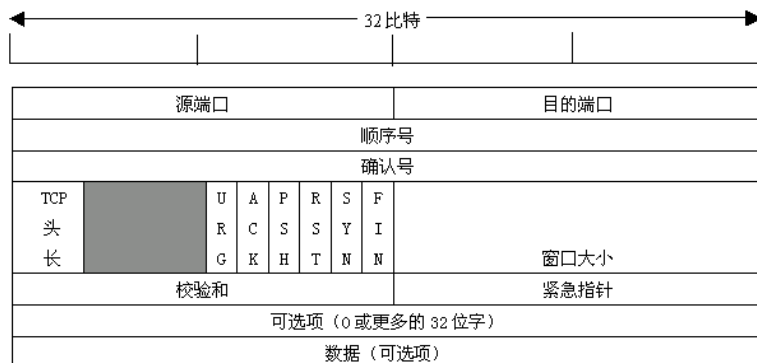


图 8.5 TCP 数据报头的格式

- 源端口、目的端口：16 位长，标识出远端和本地的端口号。
- 序号：32 位长，标识发送的数据报的顺序。
- 确认号：32 位长，希望收到的下一个数据报的序列号。
- TCP 头长：4 位长，表明 TCP 头中包含多少个 32 位字。
- 6 位未用。
- ACK：ACK 位置 1 表明确认号是合法的；如果 ACK 为 0，那么数据报不包含确认信息，确认字段被省略。
- PSH：表示是带有 PUSH 标志的数据。因此请求数据报一到接收方便可送往应用程序而不必等到缓冲区装满时才传送。
- RST：用于复位由于主机崩溃或其他原因而出现的错误的连接，还可以用于拒绝非法的数据报或拒绝连接请求。
- SYN：用于建立连接。
- FIN：用于释放连接。
- 窗口大小：16 位长，窗口大小字段表示在确认了字节之后还可以发送多少个字节。
- 校验和：16 位长，是为了确保高可靠性而设置的，它校验头部、数据和伪 TCP 头部之和。
- 可选项：0 个或多个 32 位字，包括最大 TCP 载荷、窗口比例、选择重发数据报等选项。

2. UDP

(1) 概述

UDP 即用户数据报协议，是一种面向无连接的不可靠传输协议，不需要通过 3 次握手来建立一个连接，同时，一个 UDP 应用可同时作为应用的客户或服务方。

由于 UDP 协议并不需要建立一个明确的连接，因此建立 UDP 应用要比建立 TCP 应用简单得多。



UDP 比 TCP 协议更为高效，也能更好地解决实时性的问题，如今，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都使用 UDP 协议。

(2) UDP 数据包头

UDP 数据包头如图 8.6 所示。

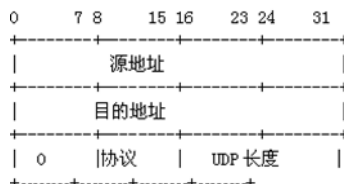


图 8.6 UDP 数据包头

- 源地址、目的地址：16 位长，标识出远端和本地的端口号。
- 数据报的长度是指包括报头和数据部分在内的总的字节数。因为报头的长度是固定的，所以该域主要用来计算可变长度的数据部分（又称为数据负载）。

3. 协议的选择

协议的选择应该考虑到数据可靠性、应用的实时性和网络的可靠性。

- 对数据可靠性要求高的应用需选择 TCP 协议，而对数据的可靠性要求不那么高的应用可选择 UDP 传送。
- TCP 协议中的 3 次握手、重传确认等手段可以保证数据传输的可靠性，但使用 TCP 协议会有较大的时延，因此不适合对实时性要求较高的应用；而 UDP 协议则有很好的实时性。

网络状况不是很好的情况下需选用 TCP 协议（如在广域网等情况），网络状况很好的情况下选择 UDP 协议可以减少网络负荷。

12.1.4 套接字（socket）概述

1. 套接字定义

在 Linux 中的网络编程是通过 socket 接口来进行的。套接字（socket）是一种特殊的 I/O 接口，它也是一种文件描述符。socket 是一种常用的进程之间通信机制，通过它不仅能实现本地机器上的进程之间的通信，而且通过网络能够在不同机器上的进程之间进行通信。

每一个 socket 都用一个半相关描述{协议、本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。socket 也有一个类似于打开文件的函数调用，该函数返回一个整型的 socket 描述符，随后的连接建立、数据传输等操作都是通过 socket 来实现的。



2. 套接字类型

常见的 socket 有 3 种类型如下。

(1) 流式套接字 (SOCK_STREAM)

流式套接字提供可靠的、面向连接的通信流；它使用 TCP 协议，从而保证了数据传输的可靠性和顺序性。

(2) 数据报套接字 (SOCK_DGRAM)

数据报套接字定义了一种无可靠、面向无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证是可靠、无差错的。它使用数据报协议 UDP。

(3) 原始套接字 (SOCK_RAW)

原始套接字允许对底层协议如 IP 或 ICMP 进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

12.1.5 地址及顺序处理

1. 地址结构相关处理

(1) 数据结构介绍

下面首先介绍两个重要的数据类型：sockaddr 和 sockaddr_in，这两个结构类型都是用来保存 socket 信息的，如下所示：

```
struct sockaddr
{
    unsigned short sa_family; /*地址族*/
    char sa_data[14]; /*14 字节的协议地址，包含该 socket 的 IP 地址和端口号。*/
};

struct sockaddr_in
{
    short int sa_family; /*地址族*/
    unsigned short int sin_port; /*端口号*/
    struct in_addr sin_addr; /*IP 地址*/
    unsigned char sin_zero[8]; /*填充 0 以保持与 struct sockaddr 同样大小*/
};
```

这两个数据类型是等效的，可以相互转化，通常 sockaddr_in 数据类型使用更为方便。在建立 socketadd 或 sockaddr_in 后，就可以对该 socket 进行适当的操作了。

(2) 结构字段



表 8.1 列出了该结构 `sa_family` 字段可选的常见值。

表 8.1 `sa_family` 字段值

结构定义头文件	<code>#include <netinet/in.h></code>
<code>sa_family</code>	<code>AF_INET</code> : IPv4 协议
	<code>AF_INET6</code> : IPv6 协议
	<code>AF_LOCAL</code> : UNIX 域协议
	<code>AF_LINK</code> : 链路地址协议
	<code>AF_KEY</code> : 密钥套接字

`sockaddr_in` 其他字段的含义非常清楚，具体的设置涉及到其他函数，在后面会有详细的讲解。

2. 数据存储优先顺序

(1) 函数说明

计算机数据存储有两种字节优先顺序：高位字节优先（称为大端模式）和低位字节优先（称为小端模式，PC 机通常采用小端模式）。Internet 上数据以高位字节优先顺序在网络上传输，因此在有些情况下，需要对这两个字节存储优先顺序进行相互转化。这里用到了四个函数：`htons()`、`ntohs()`、`htonl()`和`ntohl()`。这四个地址分别实现网络字节序和主机字节序的转化，这里的 `h` 代表 `host`，`n` 代表 `network`，`s` 代表 `short`，`l` 代表 `long`。通常 16 位的 IP 端口号用 `s` 代表，而 IP 地址用 `l` 来代表。调用这些函数只是使其得到相应的字节序，用户不需清楚该系统的主机字节序和网络字节序是否真正相等。如果是相同不需要转换的话，该系统的这些函数会定义成空宏。

(2) 函数格式

表 8.2 列出了这 4 个函数的语法格式。

表 8.2 `htons` 等函数语法要点

所需头文件	<code>#include <netinet/in.h></code>
函数原型	<code>uint16_t htons(uint16_t host16bit)</code> <code>uint32_t htonl(uint32_t host32bit)</code> <code>uint16_t ntohs(uint16_t net16bit)</code> <code>uint32_t ntohs(uint32_t net32bit)</code>
函数传入值	<code>host16bit</code> : 主机字节序的 16bit 数据
	<code>host32bit</code> : 主机字节序的 32bit 数据
	<code>net16bit</code> : 网络字节序的 16bit 数据
	<code>net32bit</code> : 网络字节序的 32bit 数据
函数返回值	成功: 返回要转换的字节序
	出错: -1



3. 地址格式转化

(1) 函数说明

用户在表达地址时通常采用点分十进制表示的数值字符串（或者是以冒号分开的十进制 IPv6 地址），而在通常使用的 socket 编程中所使用的则是二进制值（例如，用 `in_addr` 结构和 `in6_addr` 结构分别表示 IPv4 和 IPv6 中的网络地址），这就需要将这两个数值进行转换。

这里在 IPv4 中用到的函数有 `inet_aton()`、`inet_addr()` 和 `inet_ntoa()`，而 IPv4 和 IPv6 兼容的函数有 `inet_pton()` 和 `inet_ntop()`。由于 IPv6 是下一代互联网的标准协议，因此，本书讲解的函数都能够同时兼容 IPv4 和 IPv6，但在具体举例时仍以 IPv4 为例。`inet_pton()` 函数是将点分十进制地址字符串转换为二进制地址（例如：将 IPv4 的地址字符串“192.168.1.123”转换为 4 个字节的数据（从低字节起依次为 192、168、1、123）），而 `inet_ntop()` 是 `inet_pton()` 的反操作，将二进制地址转换为点分十进制地址字符串。

(2) 函数格式

表 8.3 列出了 `inet_pton()` 函数的语法要点。

表 8.3 `inet_pton()` 函数语法要点

所需头文件	#include <arpa/inet.h>	
函数原型	int inet_pton(int family, const char *strptr, void *addrptr)	
函数传入值	family	AF_INET: IPv4 协议 AF_INET6: IPv6 协议
	strptr:	要转化的值（十进制地址字符串）
	addrptr:	转化后的地址
函数返回值	成功: 0	
	出错: -1	

表 8.4 列出了 `inet_ntop()` 函数的语法要点。

表 8.4 `inet_ntop()` 函数语法要点

所需头文件	#include <arpa/inet.h>	
函数原型	int inet_ntop(int family, void *addrptr, char *strptr, size_t len)	
函数传入值	family	AF_INET: IPv4 协议 AF_INET6: IPv6 协议
	addrptr:	要转化的地址
	strptr:	转化后的十进制地址字符串
	len:	转化后值的大小
函数返回值	成功: 0	
	出错: -1	

4. 名字地址转化



(1) 函数说明

在 Linux 中有一些函数可以实现主机名和地址的转化，如 `gethostbyname()`、`gethostbyaddr()` 和 `getaddrinfo()` 等，它们都可以实现 IPv4 和 IPv6 的地址和主机名之间的转化。

其中 `gethostbyname()` 是将主机名转化为 IP 地址，`gethostbyaddr()` 则是逆操作，是将 IP 地址转化为主机名，另外 `getaddrinfo()` 还能实现自动识别 IPv4 地址和 IPv6 地址。

`gethostbyname()` 和 `gethostbyaddr()` 都涉及到一个 `hostent` 的结构体，如下所示：

```
struct hostent
{
    char *h_name;           /*正式主机名*/
    char **h_aliases;       /*主机别名*/
    int h_addrtype;         /*地址类型*/
    int h_length;           /*地址字节长度*/
    char **h_addr_list;     /*指向 IPv4 或 IPv6 的地址指针数组*/
}
```

调用 `gethostbyname()` 函数或 `gethostbyaddr()` 函数后就能返回 `hostent` 结构体的相关信息。`getaddrinfo()` 函数涉及到一个 `addrinfo` 的结构体，如下所示：

```
struct addrinfo
{
    int ai_flags;           /*AI_PASSIVE, AI_CANONNAME;*/
    int ai_family;          /*地址族*/
    int ai_socktype;        /*socket 类型*/
    int ai_protocol;        /*协议类型*/
    size_t ai_addrlen;      /*地址字节长度*/
    char *ai_canonname;     /*主机名*/
    struct sockaddr *ai_addr; /*socket 结构体*/
    struct addrinfo *ai_next; /*下一个指针链表*/
}
```

`hostent` 结构体而言，`addrinfo` 结构体包含更多的信息。

(2) 函数格式

表 8.5 列出了 `gethostbyname()` 函数的语法要点。

表 8.5 `gethostbyname` 函数语法要点



所需头文件	#include <netdb.h>
函数原型	struct hostent *gethostbyname(const char *hostname)
函数传入值	hostname: 主机名
函数返回值	成功: hostent 类型指针
	出错: -1

调用该函数时可以首先对 hostent 结构体中的 h_addrtype 和 h_length 进行设置, 若为 IPv4 可设置为 AF_INET 和 4; 若为 IPv6 可设置为 AF_INET6 和 16; 若不设置则默认为 IPv4 地址类型。

表 8.6 列出了 getaddrinfo() 函数的语法要点。

表 8.6 getaddrinfo() 函数语法要点

所需头文件	#include <netdb.h>
函数原型	int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **result)
函数传入值	node: 网络地址或者网络主机名
	service: 服务名或十进制的端口号字符串
	hints: 服务线索
	result: 返回结果
函数返回值	成功: 0
	出错: -1

在调用之前, 首先要对 hints 服务线索进行设置。它是一个 addrinfo 结构体, 表 8.7 列举了该结构体常见的选项值。

表 8.7 addrinfo 结构体常见选项值

结构体头文件	#include <netdb.h>
ai_flags	AI_PASSIVE: 该套接口是用作被动地打开
	AI_CANONNAME: 通知 getaddrinfo 函数返回主机的名字
ai_family	AF_INET: IPv4 协议
	AF_INET6: IPv6 协议
	AF_UNSPEC: IPv4 或 IPv6 均可
ai_socktype	SOCK_STREAM: 字节流套接字 socket (TCP)
	SOCK_DGRAM: 数据报套接字 socket (UDP)
ai_protocol	IPPROTO_IP: IP 协议
	IPPROTO_IPV4: IPv4 协议
	IPPROTO_IPV6: IPv6 协议



	IPPROTO_UDP: UDP
	IPPROTO_TCP: TCP

注意:

- 通常服务器端在调用 `getaddrinfo()` 之前, `ai_flags` 设置 `AI_PASSIVE`, 用于 `bind()` 函数 (用于端口和地址的绑定, 后面会讲到), 主机名 `nodename` 通常会设置为 `NULL`。
- 客户端调用 `getaddrinfo()` 时, `ai_flags` 一般不设置 `AI_PASSIVE`, 但是主机名 `nodename` 和服务名 `servname` (端口) 则应该不为空。

即使不设置 `ai_flags` 为 `AI_PASSIVE`, 取出的地址也可以被绑定, 很多程序中 `ai_flags` 直接设置为 0, 即 3 个标志位都不设置, 这种情况下只要 `hostname` 和 `servname` 设置的没有问题就可以正确绑定。

12.1.6 套接字编程

(1) 函数说明

socket 编程的基本函数有 `socket()`、`bind()`、`listen()`、`accept()`、`send()`、`sendto()`、`recv()` 以及 `recvfrom()` 等, 其中根据客户端还是服务端, 或者根据使用 TCP 协议还是 UDP 协议, 这些函数的调用流程都有所区别, 这里先对每个函数进行说明, 再给出各种情况下使用的流程图。

- `socket()`: 该函数用于建立一个套接字, 一条通信路线的端点。在建立了 `socket` 之后, 可对 `sockaddr` 或 `sockaddr_in` 结构进行初始化, 以保存所建立的 `socket` 地址信息。
 - `bind()`: 该函数是用于将 `sockaddr` 结构的地址信息与套接字进行绑定, 它主要用于 TCP 的连接, 而在 UDP 的连接中则无必要 (但可以使用)。
 - `listen()`: 在服务端程序成功建立套接字和与地址进行绑定之后, 还需要准备在该套接字上接收新的连接请求。此时调用 `listen()` 函数来创建一个等待队列, 在其中存放未处理的客户端连接请求。
 - `accept()`: 服务端程序调用 `listen()` 函数创建等待队列之后, 调用 `accept()` 函数等待并接收客户端的连接请求。它通常从由 `listen()` 所创建的等待队列中取出第一个未处理的连接请求。
 - `connect()`: 客户端通过一个未命名套接字 (未使用 `bind()` 函数) 和服务器监听套接字之间建立连接的方法来连接到服务器。这个工作客户端通过使用 `connect()` 函数来实现。
 - `send()` 和 `recv()`: 这两个函数分别用于发送和接收数据, 可以用在 TCP 中, 也可以用在 UDP 中。当用在 UDP 时, 可以在 `connect()` 函数建立连接之后再使用。
 - `sendto()` 和 `recvfrom()`: 这两个函数的作用与 `send()` 和 `recv()` 函数类似, 也可以用在 TCP 和 UDP 中。当用在 TCP 时, 后面的几个与地址有关参数不起作用, 函数作用等同于 `send()` 和 `recv()`; 当用在 UDP 时, 可以用在之前没有使用 `connect()` 的情况下, 这两个函数可以自动寻找指定地址并进行连接。
- 服务器端和客户端使用 TCP 协议的流程如图 8.7 所示。

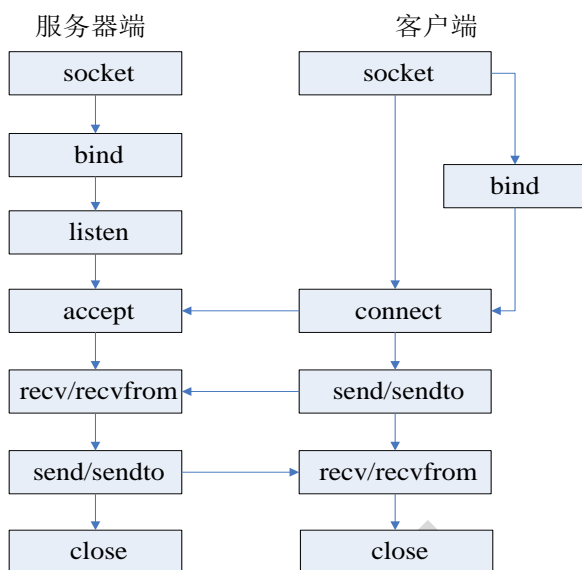


图 8.7 使用 TCP 协议 socket 编程流程图

服务器端和客户端使用 UDP 协议的流程图如图 8.8 所示。

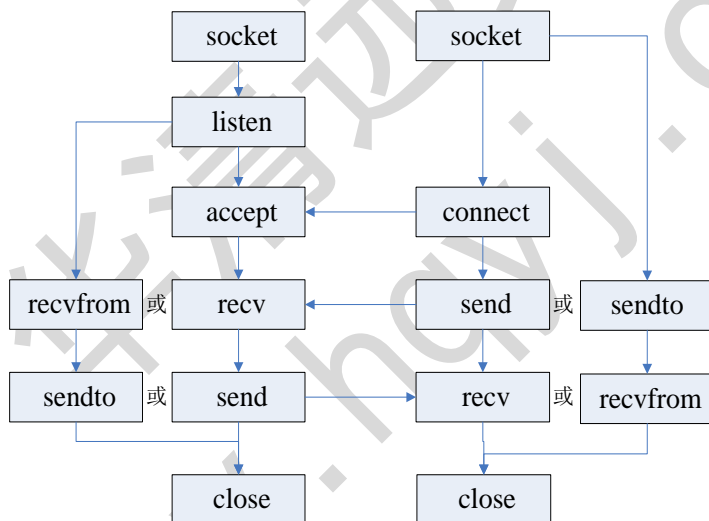


图 8.8 使用 UDP 协议 socket 编程流程图

(2) 函数格式

表 8.8 列出了 socket() 函数的语法要点。

表 8.8 socket() 函数语法要点

所需头文件	#include <sys/socket.h>	
函数原型	int socket(int family, int type, int protocol)	
函数传入值	:	family
		AF_INET: IPv4 协议
		AF_INET6: IPv6 协议
		AF_LOCAL: UNIX 域协议
		AF_ROUTE: 路由套接字 (socket)
		AF_KEY: 密钥套接字 (socket)



	type:	SOCK_STREAM: 字节流套接字 socket
	套接字	SOCK_DGRAM: 数据报套接字 socket
	类型	SOCK_RAW: 原始套接字 socket
	protocol:	0 (原始套接字除外)
函数返回值	成功:	非负套接字描述符
	出错:	-1

表 8.9 列出了 bind()函数的语法要点。

表 8.9 bind()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
函数传入值	sockfd: 套接字描述符
	my_addr: 本地地址
	addrlen: 地址长度
函数返回值	成功: 0
	出错: -1

端口号和地址在 my_addr 中给出了, 若不指定地址, 则内核随意分配一个临时端口给该应用程序。IP 地址可以直接指定 (比如: inet_addr("192.168.1.112")), 或者使用宏 INADDR_ANY, 允许将套接字与服务器的任一网络接口 (比如: eth0、eth0:1、eth1 等) 进行绑定。

表 8.10 列出了 listen()函数的语法要点。

表 8.10 listen()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int listen(int sockfd, int backlog)
函数传入值	sockfd: 套接字描述符
	backlog: 请求队列中允许的最大请求数, 大多数系统缺省值为 5
函数返回值	成功: 0
	出错: -1

表 8.11 列出了 accept()函数的语法要点。

表 8.11 accept()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
函数传入值	sockfd: 套接字描述符
	addr: 客户端地址
	addrlen: 地址长度



函数返回值	成功：接收到的非负的套接字
	出错：-1

表 8.12 列出了 connect()函数的语法要点。

表 8.12 connect()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
函数传入值	sockfd: 套接字描述符
	serv_addr: 服务器端地址
	addrlen: 地址长度
函数返回值	成功：0
	出错：-1

表 8.13 列出了 send()函数的语法要点。

表 8.13 send()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int send(int sockfd, const void *msg, int len, int flags)
函数传入值	sockfd: 套接字描述符
	msg: 指向要发送数据的指针
	len: 数据长度
	flags: 一般为 0
函数返回值	成功：实际发送的字节数
	出错：-1

表 8.14 列出了 recv()函数的语法要点。

表 8.14 recv()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int recv(int sockfd, void *buf, int len, unsigned int flags)
函数传入值	sockfd: 套接字描述符
	buf: 存放接收数据的缓冲区
	len: 数据长度
	flags: 一般为 0
函数返回值	成功：实际接收到的字节数
	出错：-1



表 8.15 列出了 sendto()函数的语法要点。

表 8.15 sendto()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int sendto(int sockfd, const void *msg,int len, unsigned int flags, const struct sockaddr *to, int tolen)
函数传入值	sockfd: 套接字描述符
	msg: 指向要发送数据的指针
	len: 数据长度
	flags: 一般为 0
	to: 目的地机的 IP 地址和端口号信息
	tolen: 地址长度
函数返回值	成功: 实际发送的字节数
	出错: -1

表 8.16 列出了 recvfrom()函数的语法要点。

表 8.16 recvfrom()函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int recvfrom(int sockfd,void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen)
函数传入值	sockfd: 套接字描述符
	buf: 存放接收数据的缓冲区
	len: 数据长度
	flags: 一般为 0
	from: 源主机的 IP 地址和端口号信息
	tolen: 地址长度
函数返回值	成功: 实际接收到的字节数
	出错: -1

!

在实际情况中，人们往往遇到多个客户端连接服务器端的情况。在之前介绍的例子中我们使用阻塞函数，因此如果资源没有准备好，则调用该函数的进程将进入睡眠状态，这样就无法处理其他请求的情况了。本节给出了三种解决 I/O 多路复用的解决方法，分别为非阻塞和异步式处理（使用 fcntl()函数）以及多路复用处理（使用 select()或 poll()函数）。此外有多进程和多线程编程是在网络编程中常用的事务处理方法，下一节的例子中会用到多进程网络编程。读者可以尝试使用多线程机制修改书上的所有例子，会发现多线程编程在网络编程中非常有效的方法之一。



1. 非阻塞和异步 I/O

在 socket 编程中可以使用函数 `fcntl(int fd, int cmd, int arg)` 的如下的编程特性。

- 获得文件状态标志：将 `cmd` 设置为 `F_GETFL`，会返回由 `fd` 指向的文件的状态标志。
- 非阻塞 I/O：将 `cmd` 设置为 `F_SETFL`，将 `arg` 设置为 `O_NONBLOCK`。
- 异步 I/O：将 `cmd` 设置为 `F_SETFL`，将 `arg` 设置为 `O_ASYNC`，后面有说明。

尽管在很多情况下，使用阻塞式和非阻塞式以及多路复用等机制可以有效地进行网络通信，但效率最高的方法是使用异步通知机制。这种方法在设备 I/O 编程中常见的。

内核通过使用异步 I/O，在某一个进程需要处理的事件发生（例如，接收到新的连接请求）时，向该进程发送一个 `SIGIO` 信号。这样，应用程序不需要不停地等待着某些事件的发生，而可以往下运行，以完成其它的工作。只有收到从内核发来的 `SIGIO` 信号时，去处理它（例如，读取数据）就可以。

我们使用 `fcntl()` 函数就可以实现高效率的异步 I/O 的方法。首先必须使用 `fcntl` 的 `F_SETOWN` 命令，使套接字归属于当前进程，以内核能够判断应该向哪个进程发送信号。接下来，使用 `fcntl` 的 `F_SETFL` 命令将套接字的状态标志位设置为异步通知方式（使用 `O_ASYNC` 参数）。

2. 使用多路复用

使用 `fcntl()` 函数虽然可以实现非阻塞，但在实际使用时往往会对资源是否准备完毕进行循环测试，这样就大大增加了不必要的 CPU 资源的占用。在这里可以使用 `select()` 函数（或者使用 `poll()` 函数）来解决这个问题，同时，使用 `select()` 函数还可以设置等待的时间，可以说功能更加强大。

12.2 Linux 系统 tcp 网络协议编程实验

12.2.1 实验目的

熟悉 socket 网络编程的基本方法

12.2.2 实验平台

华清远见开发环境

12.2.3 实验内容

本实验通过一个简单的 tcp 服务器端，接收客户端的连接请求，并接受客户端发来的信息。

12.2.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/12-network
$ mkdir tcp
```



```
$ cd tcp
```

```
linux@ubuntu64-vm:~/workdir/linux/application$ mkdir 12-network
linux@ubuntu64-vm:~/workdir/linux/application$ cd 12-network/
linux@ubuntu64-vm:~/workdir/linux/application/12-network$ ls
linux@ubuntu64-vm:~/workdir/linux/application/12-network$ mkdir tcp
linux@ubuntu64-vm:~/workdir/linux/application/12-network$ cd tcp/
linux@ubuntu64-vm:~/workdir/linux/application/12-network/tcp$ ls
linux@ubuntu64-vm:~/workdir/linux/application/12-network/tcp$
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\17. Linux 系统 tcp 网络协议编程实验\实验代码】目录下的“client.c”、“server.c”拷贝到该目录下。

执行编译命令：

```
$ gcc client.c -o client
$ gcc server.c -o server
```

运行程序：

首先运行服务器程序：

```
$ ./server 192.168.100.192 8888 //其中 ip 地址是 ubuntu 系统实际的 ip 地址
```

另外打开一个终端，进入目录，运行客户端程序

```
$ ./client 192.168.100.192 8888
```

```
>hello //向服务器端发送消息
```

```
Hello
```

```
>^C
linux@ubuntu64-vm:~/workdir/linux/application/12-network/tcp$ ./client 192.168.100.192 8888
>hello
hello
>
```

12.2.5 实验现象

服务器所在终端会显示：

```
from 192.168.100.192:xxxxx
n=6 hello
```

```
>^C
linux@ubuntu64-vm:~/workdir/linux/application/12-network/tcp$ ./server 192.168.100.192 8888
from 192.168.100.192:60776
n=6 hello
```

12.2.6 实验代码

client.c



```
1#include <stdio.h>
2#include <stdlib.h>
3#include <unistd.h>
4#include <sys/types.h>
5#include <sys/socket.h>
6#include <errno.h>
7#include <string.h>
8#include <arpa/inet.h>
9#include <netinet/in.h>
10
11#define N 64
12
13int main(int argc, char *argv[])//./server ip port
14{
15    int sockfd;
16    struct sockaddr_in servaddr, myaddr;
17    char buf[N] = {0};
18
19    if (argc < 3)
20    {
21        printf("usage:%s ip port\n", argv[0]);
22        return 0;
23    }
24
25    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
26    {
27        perror("socket");
28        exit(-1);
29    }
30
31    #if 0
32    memset(&myaddr, 0, sizeof(myaddr));
33    myaddr.sin_family = AF_INET;
34    myaddr.sin_port = htons(8000);//"6000"--6000 htons(6000);
35    myaddr.sin_addr.s_addr = inet_addr(argv[1]);
36
37    if (bind(sockfd, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1)
38    {
39        perror("bind");
40        exit(-1);
41    }
42    #endif
43
44    memset(&servaddr, 0, sizeof(servaddr));
```



```

45 servaddr.sin_family = AF_INET;
46 servaddr.sin_port = htons(atoi(argv[2])); //"6000"--6000 htons(6000);
47 servaddr.sin_addr.s_addr = inet_addr(argv[1]);
48
49 if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1)
50 {
51     perror("connect");
52     exit(-1);
53 }
54
55 printf(">");
56 while (fgets(buf, N, stdin) != NULL)
57 {
58     //abc\n    --- a b c \n \0----
59     send(sockfd, buf, strlen(buf), 0);
60
61     memset(buf, 0, sizeof(buf));
62     recv(sockfd, buf, N, 0);
63     printf("%s\n", buf);
64
65     printf(">");
66 }
67 close(sockfd);
68
69 return 0;
70 }

```

server.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <errno.h>
7 #include <string.h>
8 #include <arpa/inet.h>
9 #include <netinet/in.h>
10
11 #define N 64
12
13 int main(int argc, char *argv[]) // ./server ip port
14 {
15     int listenfd, connfd;
16     struct sockaddr_in myaddr, peeraddr;

```




```
17  socklen_t len;
18  char buf[N] = {0};
19  ssize_t n;
20
21  if (argc < 3)
22  {
23      printf("usage:%s ip port\n", argv[0]);
24      return 0;
25  }
26
27  if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
28  {
29      perror("socket");
30      exit(-1);
31  }
32
33  memset(&myaddr, 0, sizeof(myaddr));
34  myaddr.sin_family = AF_INET;
35  myaddr.sin_port = htons(atoi(argv[2])); // "6000" -- 6000 htons(6000);
36  myaddr.sin_addr.s_addr = inet_addr(argv[1]);
37
38  if (bind(listenfd, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1)
39  {
40      perror("bind");
41      exit(-1);
42  }
43
44  if (-1 == listen(listenfd, 5))
45  {
46      perror("listen");
47      exit(-1);
48  }
49
50  memset(&peeraddr, 0, sizeof(peeraddr));
51  len = sizeof(peeraddr);
52
53  while (1)
54  {
55      if ((connfd = accept(listenfd, (struct sockaddr *)&peeraddr, &len)) == -1)
56      {
57          perror("accept");
58          exit(-1);
59      }
60      printf("from %s:%d\n", inet_ntoa(peeraddr.sin_addr), ntohs(peeraddr.sin_port));
```



```
61
62     while (1)
63     {
64         memset(buf, 0, sizeof(buf));
65         n = recv(connfd, buf, N, 0); // a b c \n n=4
66         if (n == 0)
67             break;
68         buf[n] = '\0';
69         printf("n=%d %s", n, buf);
70
71         send(connfd, buf, n, 0);
72     }
73
74     close(connfd);
75 }
76
77 return 0;
78 }
```

12.3 Linux 系统 udp 网络协议编程实验

12.3.1 实验目的

熟悉 socket 网络编程的基本方法。

12.3.2 实验平台

华清远见开发环境

12.3.3 实验内容

本实验通过一个简单的 udp 服务器端，接收客户端的连接请求，并接受客户端发来的信息。

12.3.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/12-network
```

```
$ mkdir udp
```

```
$ cd udp
```

```
linux@ubuntu64-vm:~/workdir/linux/application/12-network$ mkdir udp
linux@ubuntu64-vm:~/workdir/linux/application/12-network$ cd udp/
linux@ubuntu64-vm:~/workdir/linux/application/12-network/udp$ ls
linux@ubuntu64-vm:~/workdir/linux/application/12-network/udp$
```



将【华清远见-CORTEXA9 资料： \程序源码\Linux 应用实验源码\18. Linux 系统 udp 网络协议编程实验\实验代码】目录下的“client.c”、“server.c”拷贝到该目录下。

执行编译命令：

```
$ gcc client.c -o client
```

```
$ gcc server.c -o server
```

```
client.c server.c
linux@ubuntu64-vm:~/workdir/linux/application/12-network/udp$ gcc client.c -o client
linux@ubuntu64-vm:~/workdir/linux/application/12-network/udp$ gcc server.c -o server
linux@ubuntu64-vm:~/workdir/linux/application/12-network/udp$
```

运行程序：

首先运行服务器程序：

```
$ ./server 192.168.100.192 8888 //其中 ip 地址是 ubuntu 系统实际的 ip 地址
```

另外打开一个终端，进入目录，运行客户端程序

```
$ ./client 192.168.100.192 8888
```

```
>hello //向服务器端发送消息
```

```
hello
```

```
client client.c server server.c
linux@ubuntu64-vm:~/workdir/linux/application/12-network/udp$ ./client 192.168.100.192 8888
>hello
hello
```

12.3.5 实验现象

服务器所在终端会显示：

```
from 192.168.100.192:xxxxx hello
```

```
linux@ubuntu64-vm:~/workdir/linux/application/12-network/udp$ gcc server.c -o server
linux@ubuntu64-vm:~/workdir/linux/application/12-network/udp$ ./server 192.168.100.192 8888
from 192.168.100.192:49113 hello
```

12.3.6 实验代码

client.c

```
1#include <stdio.h>
2#include <stdlib.h>
3#include <unistd.h>
4#include <sys/types.h>
5#include <sys/socket.h>
6#include <errno.h>
7#include <string.h>
8#include <arpa/inet.h>
9#include <netinet/in.h>
1
```



```
0#define N 64
1
1int main(int argc, char *argv[])//./server ip port
1{
2    int sockfd;
1    struct sockaddr_in servaddr;
3    char buf[N] = {0};
1
4    if (argc < 3)
1    {
5        printf("usage:%s ip port\n", argv[0]);
1        return 0;
6    }
1
7    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
1    {
8        perror("socket");
1        exit(-1);
9    }
2
0
2    memset(&servaddr, 0, sizeof(servaddr));
1    servaddr.sin_family = AF_INET;
2    servaddr.sin_port = htons(atoi(argv[2]));//"6000"--6000 htons(6000);
2    servaddr.sin_addr.s_addr = inet_addr(argv[1]);
2
3    while (1)
2    {
4        printf(">");
2        fgets(buf, N, stdin);
5        sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *) &servaddr, sizeof(servaddr)
2);
6
2        memset(buf, 0, sizeof(buf));
7        recvfrom(sockfd, buf, N, 0, NULL, NULL);
2        printf("%s\n", buf);
8    }
2    close(sockfd);
9
    return 0;
}
```

server.c

```
1#include <stdio.h>
2#include <stdlib.h>
```



```
3#include <unistd.h>
4#include <sys/types.h>
5#include <sys/socket.h>
6#include <errno.h>
7#include <string.h>
8#include <arpa/inet.h>
9#include <netinet/in.h>
10
11#define N 64
12
13int main(int argc, char *argv[])//./server ip port
14{
15    int sockfd;
16    struct sockaddr_in myaddr, peeraddr;
17    char buf[N] = {0};
18    socklen_t len;
19    ssize_t n;
20
21    if (argc < 3)
22    {
23        printf("usage:%s ip port\n", argv[0]);
24        return 0;
25    }
26
27    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
28    {
29        perror("socket");
30        exit(-1);
31    }
32
33
34    memset(&myaddr, 0, sizeof(myaddr));
35    myaddr.sin_family = AF_INET;
36    myaddr.sin_port = htons(atoi(argv[2]));//"6000"--6000 htons(6000);
37    myaddr.sin_addr.s_addr = inet_addr(argv[1]);
38
39    if (bind(sockfd, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1)
40    {
41        perror("bind");
42        exit(-1);
43    }
44
45    memset(&peeraddr, 0, sizeof(peeraddr));
46    len = sizeof(peeraddr);
```



```
47
48     while (1)
49     {
50         memset(buf, 0, sizeof(buf));
51         n = recvfrom(sockfd, buf, N, 0, (struct sockaddr *)&peeraddr, &len);
52         printf("from %s:%d %s\n", inet_ntoa(peeraddr.sin_addr),
53             ntohs(peeraddr.sin_port), buf);
54         sendto(sockfd, buf, n, 0, (struct sockaddr *)&peeraddr, sizeof(peeraddr));
55     }
56     close(sockfd);
57
58     return 0;
}
```

12.4 Linux 系统 select IO 复用实验

12.4.1 实验目的

通过 select 来实现网络 I/O 的复用。

12.4.2 实验平台

华清远见开发环境

12.4.3 实验内容

tcp 服务器端在处理客户端连接请求的同时，对已连接客户实现 echo 服务

12.4.4 实验步骤

运行 Ubuntu 系统，打开命令行终端。

执行命令：

```
$ cd ~/workdir/linux/application/12-network
$ mkdir select
$ cd select
```

```
tcp udp
linux@ubuntu64-vm:~/workdir/linux/application/12-network$ mkdir select
linux@ubuntu64-vm:~/workdir/linux/application/12-network$ cd select/
linux@ubuntu64-vm:~/workdir/linux/application/12-network/select$ ls
linux@ubuntu64-vm:~/workdir/linux/application/12-network/select$
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 应用实验源码\19. Linux 系统 select IO 复用实验\实验代码\selectecho】目录下的“client.c”、“select_server.c”拷贝到该目录下。

执行编译命令：



```
$ gcc client.c -o client
```

```
$ gcc select_server.c -o select_server
```

```
linux@ubuntu64-vm:~/workdir/linux/application/12-network/select$ gcc client.c -o client
client.c: 在函数'main'中:
client.c:53: 警告: 格式'%d'需要类型'int', 但实参 2 的类型为'ssize_t'
linux@ubuntu64-vm:~/workdir/linux/application/12-network/select$ gcc select_server.c -o select_server
select_server.c: 在函数'main'中:
select_server.c:95: 警告: 格式'%d'需要类型'int', 但实参 2 的类型为'ssize_t'
linux@ubuntu64-vm:~/workdir/linux/application/12-network/select$
```

运行程序:

首先运行服务器程序:

```
$ ./select_server 192.168.100.192 8888 //其中 ip 地址是 ubuntu 系统实际的 ip 地址
```

另外打开一个终端, 进入目录, 运行客户端程序

```
$ ./client 192.168.100.192 8888
```

```
>hello
```

```
n=64 buf=hello //
```

```
select_server.c:95: 警告: 格式和 需要类型 int, 但实参 2 的类型为 ssize_t
linux@ubuntu64-vm:~/workdir/linux/application/12-network/select$ ./client 192.168.100.192 8888
>hello
n=64 buf=hello
>
```

12.4.5 实验现象

服务器所在终端会显示:

```
welcome 192.168.100.192 xxxxx
```

```
n=64 hello
```

```
client client.c select select_server select_server.c
linux@ubuntu64-vm:~/workdir/linux/application/12-network/select$ ./select_server 192.168.100.192 8888
welcome 192.168.100.192 60778
n=64 hello
```

12.4.6 实验代码

client.c

```
1#include <stdio.h>
2#include <unistd.h>
3#include <string.h>
4#include <stdlib.h>
5#include <sys/types.h>
6#include <sys/socket.h>
7#include <errno.h>
8#include <strings.h>
9#include <netinet/in.h>
10#include <arpa/inet.h>
11
12typedef struct sockaddr SA;
```



```
13 #define N 64
14
15 int main(int argc, char *argv[])
16 {
17     int sockfd;
18     ssize_t n;
19     struct sockaddr_in servaddr;
20     char buf[N] = {0};
21
22     if (argc < 3)
23     {
24         fprintf(stdout, "usage:%s ip port\n", argv[0]);
25         exit(0);
26     }
27
28     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
29     {
30         perror("socket");
31         exit(-1);
32     }
33
34     bzero(&servaddr, sizeof(servaddr));
35     servaddr.sin_family = PF_INET;
36     servaddr.sin_port = htons(atoi(argv[2])); // "9000"---9000
37     servaddr.sin_addr.s_addr = inet_addr(argv[1]);
38
39     if (connect(sockfd, (SA *)&servaddr, sizeof(servaddr)) == -1)
40     {
41         perror("connect");
42         exit(-1);
43     }
44
45     printf(">");
46     while (fgets(buf, N, stdin) != NULL) // abc\n
47     {
48         buf[strlen(buf) - 1] = 0; // abc\0
49         send(sockfd, buf, N, 0);
50
51         bzero(buf, sizeof(buf));
52         n = recv(sockfd, buf, N, 0);
53         printf("n=%d buf=%s\n", n, buf);
54         printf(">");
55     }
56
```




```
57     close(sockfd);
58
59     exit(0);
60 }
```

select_server.c

```
1#include <stdio.h>
2#include <string.h>
3#include <stdlib.h>
4#include <sys/types.h>          /* See NOTES */
5#include <sys/time.h>
6#include <unistd.h>
7#include <sys/select.h>
8#include <sys/socket.h>
9#include <errno.h>
10#include <strings.h>
11#include <netinet/in.h>
12#include <arpa/inet.h>
13
14#define N 64
15typedef struct sockaddr SA;
16
17int main(int argc, char *argv[])
18{
19     int listenfd, connfd, maxfd, i;
20     struct sockaddr_in servaddr, peeraddr;
21     socklen_t len;
22     char buf[N] = {0};
23     fd_set rdfs, bakrdfs;
24     ssize_t n;
25
26     if (argc < 3)
27     {
28         fprintf(stdout, "usage:%s <ip> <port>\n", argv[0]);
29         exit(0);
30     }
31
32     bzero(&servaddr, sizeof(servaddr));
33
34     if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
35     {
36         perror("socket");
37         exit(-1);
38     }
39
```



```
40 servaddr.sin_family = PF_INET;
41 servaddr.sin_port = htons(atoi(argv[2]));
42 servaddr.sin_addr.s_addr = inet_addr(argv[1]);
43
44 if (bind(listenfd, (SA *)&servaddr, sizeof(servaddr)) == -1)
45 {
46     perror("bind");
47     exit(-1);
48 }
49
50 listen(listenfd, 5);
51 maxfd = listenfd;
52
53 FD_ZERO(&bakrdfs);
54 FD_SET(listenfd, &bakrdfs);
55
56 len = sizeof(peeraddr);
57 while (1)
58 {
59     rdfs = bakrdfs;
60
61     if (select(maxfd + 1, &rdfs, NULL, NULL, NULL) == -1)
62     {
63         perror("select");
64         exit(-1);
65     }
66
67     for (i = 0; i <= maxfd; i++)
68     {
69         if (FD_ISSET(i, &rdfs))
70         {
71             if (i == listenfd)
72             {
73                 if ((connfd = accept(i, (SA *)&peeraddr, &len)) == -1)
74                 {
75                     perror("accept");
76                     exit(-1);
77                 }
78                 fprintf(stdout, "welcome %s %d\n",
79                     inet_ntoa(peeraddr.sin_addr),
80                     ntohs(peeraddr.sin_port));
81
82                 FD_SET(connfd, &bakrdfs);
83                 maxfd = (maxfd > connfd) ? maxfd : connfd;
```



```
84         }
85         else
86         {
87             bzero(buf, sizeof(buf));
88             if ((n = recv(i, buf, N, 0)) == 0)
89             {
90                 close(i);
91                 FD_CLR(i, &bakrdfs);
92             }
93             else
94             {
95                 printf("n=%d %s\n", n, buf);
96                 send(i, buf, N, 0);
97             }
98         }
99     }
100 }
101 }
102
103 exit(0);
}
```

华清远见
dev.hay



第 13 章 BootLoader (Uboot) 移植实验

13.1 实验原理

13.1.1 概念

简单地说, Bootloader 就是在操作系统内核运行之前运行的一段程序, 它类似于 PC 机中的 BIOS 程序。通过这段程序, 可以完成硬件设备的初始化, 并建立内存空间的映射图的功能, 从而将系统的软硬件环境带到一个合适的状态, 为最终调用系统内核做好准备。

通常, Bootloader 是严重地依赖于硬件实现的, 特别是在嵌入式中。因此, 在嵌入式世界里建立一个通用的 Bootloader 几乎是不可能的。尽管如此, 仍然可以对 Bootloader 归纳出一些通用的概念来指导用户特定的 Bootloader 设计与实现。

(1) Bootloader 所支持的 CPU 和嵌入式开发板

每种不同的 CPU 体系结构都有不同的 Bootloader。有些 Bootloader 也支持多种体系结构的 CPU, 如后面要介绍的 U-Boot 就同时支持 ARM 体系结构和 MIPS 体系结构。除了依赖于 CPU 的体系结构外, Bootloader 实际上也依赖于具体的嵌入式板级设备的配置。

(2) Bootloader 的安装媒介

系统加电或复位后, 所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备 (比如 ROM、EEPROM 或 FLASH 等) 被映射到这个预先安排的地址上。因此在系统加电后, CPU 将首先执行 Bootloader 程序。

(3) Bootloader 的启动过程分为单阶段和多阶段两种。通常多阶段的 Bootloader 能提供更为复杂的功能, 以及更好的可移植性。

(4) Bootloader 的操作模式。大多数 Bootloader 都包含两种不同的操作模式: “启动加载” 模式和 “下载” 模式, 这种区别仅对于开发人员才有意义。

- 启动加载模式: 这种模式也称为 “自主” 模式。也就是 Bootloader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行, 整个过程并没有用户的介入。这种模式是嵌入式产品发布时的通用模式。

- 下载模式: 在这种模式下, 目标机上的 Bootloader 将通过串口连接或网络连接等通信手段从主机 (Host) 下载文件, 比如: 下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Bootloader 保存到目标机的 RAM 中, 然后再被 Bootloader 写到目标机上的 FLASH 类固态存储设备中。Bootloader 的这种模式是在更新时使用。工作于这种模式下的 Bootloader 通常都会向它的终端用户提供一个简单的命令行接口。

(5) Bootloader 与主机之间进行文件传输所用的通信设备及协议, 最常见的情况就是, 目标机上的 Bootloader 通过串口与主机之间进行文件传输, 传输协议通常是 xmodem/ ymodem/zmodem 协议中的一种。但是, 串口传输的速度是有限的, 因此通过以太网连接并借助 TFTP 协议来下载文件是个更好的选择。

13.1.2 Bootloader 启动流程

Bootloader 的启动流程一般分为两个阶段: stage1 和 stage2, 下面分别对这两个阶段进行讲解:



(1) Bootloader 的 stage1

在 stage1 中 Bootloader 主要完成以下工作。

- 基本的硬件初始化，包括屏蔽所有的中断、设置 CPU 的速度和时钟频率、RAM 初始化、初始化 LED、关闭 CPU 内部指令和数据 cache 灯。
- 为加载 stage2 准备 RAM 空间，通常为了获得更快的执行速度，通常把 stage2 加载到 RAM 空间中执行，因此必须为加载 Bootloader 的 stage2 准备好一段可用的 RAM 空间范围。
- 拷贝 stage2 到 RAM 中，在这里要确定两点：①stage2 的可执行映像 在固态存储设备的存放起始地址和终止地址；②RAM 空间的起始地址。
- 设置堆栈指针 sp，这是为执行 stage2 的 C 语言代码做好准备。

(2) Bootloader 的 stage2

在 stage2 中 Bootloader 主要完成以下工作。

- 用汇编语言跳转到 main 入口函数

由于 stage2 的代码通常用 C 语言来实现，目的是实现更复杂的功能和取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是，在编译和链接 Bootloader 这样的程序时，不能使用 glibc 库中的任何支持函数。

- 初始化本阶段要使用到的硬件设备，包括初始化串口、初始化计时器等。在初始化这些设备之前、可以输出一些打印信息。
- 检测系统的内存映射，所谓内存映射就是指在整个 4GB 物理地址空间中有指出哪些地址范围被分配用来寻址系统的 RAM 单元。
- 加载内核映像和根文件系统映像，这里包括规划内存占用的布局 和从 Flash 上拷贝数据。
- 设置内核的启动参数。

13.1.3 Bootloader 的种类

嵌入式系统世界已经有各种各样的 Bootloader，种类划分也有多种方式。除了按照处理器体系结构不同划分以外，还有功能复杂程度的不同。

首先区分一下“Bootloader”和“Monitor”的概念。严格来说，“Bootloader”只是引导设备并且执行主程序的固件；而“Monitor”还提供了更多的命令行接口，可以进行调试、读写内存、烧写 Flash、配置环境变量等。“Monitor”在嵌入式系统开发过程中可以提供很好的调试功能，开发完成以后，就完全设置成了一个“Bootloader”。所以，习惯上大家把它们统称为 Bootloader。

表 1.4 列出了 Linux 的开放源码引导程序及其支持的体系结构。表中给出了 X86、ARM、PowerPC 体系结构的常用引导程序，并且注明了每一种引导程序是不是“Monitor”。

表 1.4 开放源码的 Linux 引导程序

Bootloader	Monitor	描 述	X86	ARM	PowerPC
LILO	否	Linux 磁盘引导程序	是	否	否
GRUB	否	GNU 的 LILO 替代程序	是	否	否



Loadlin	否	从 DOS 引导 Linux	是	否	否
ROLO	否	从 ROM 引导 Linux 而不需要 BIOS	是	否	否
Etherboot	否	通过以太网卡启动 Linux 系统的固件	是	否	否
LinuxBIOS	否	完全替代 BIOS 的 Linux 引导程序	是	否	否
BLOB	否	LART 等硬件平台的引导程序	否	是	否
U-Boot	是	通用引导程序	是	是	是
RedBoot	是	基于 eCos 的引导程序	是	是	是

对于每种体系结构，都有一系列开放源码 Bootloader 可以选用。

(1) X86

X86 的工作站和服务器的上一般使用 LILO 和 GRUB。LILO 是 Linux 发行版主流的 Bootloader。不过 Redhat Linux 发行版已经使用了 GRUB，GRUB 比 LILO 有更好的显示接口，使用配置也更加灵活方便。

在某些 X86 嵌入式单板机或者特殊设备上，会采用其他的 Bootloader，如 ROLO。这些 Bootloader 可以取代 BIOS 的功能，能够从 Flash 中直接引导 Linux 启动。现在 ROLO 支持的开发板已经并入 U-Boot，所以 U-Boot 也可以支持 X86 平台。

(2) ARM

ARM 处理器的芯片商很多，所以每种芯片的开发板都有自己的 Bootloader。结果 ARM Bootloader 也变得多种多样。最早有为 ARM720 处理器的开发板的固件，又有了 armboot，StrongARM 平台的 BLOB，还有 S3C2410 处理器开发板上的 vivi 等。现在 armboot 已经并入了 U-Boot，所以 U-Boot 也支持 ARM/XSCALE 平台。U-Boot 已经成为 ARM 平台事实上的标准 Bootloader。

(3) PowerPC

PowerPC 平台的处理器有标准的 Bootloader，就是 PPCBOOT。PPCBOOT 在合并 armboot 等之后，创建了 U-Boot，成为各种体系结构开发板的通用引导程序。U-Boot 仍然是 PowerPC 平台的主要 Bootloader。

(4) MIPS

MIPS 公司开发的 YAMON 是标准的 Bootloader，也有许多 MIPS 芯片商为自己的开发板写了 Bootloader。现在，U-Boot 也已经支持 MIPS 平台。

(5) SH

SH 平台的标准 Bootloader 是 sh-boot。RedBoot 在这种平台上也很好用。

(6) M68K

M68K 平台没有标准的 Bootloader。RedBoot 能够支持 M68K 系列的系统。

值得说明的是 RedBoot，它几乎能够支持所有的体系结构，包括 MIPS、SH、M68K 等。RedBoot 是以



eCos 为基础, 采用 GPL 许可的开源软件工程。现在由 core eCos 的开发人员维护, 源码下载网站是 <http://www.ecoscentric.com/snapshots>。RedBoot 的文档也相当完善, 有详细的使用手册《RedBoot User's Guide》。

13.1.4 U-Boot 概述

U-Boot (UniversalBootloader), 是遵循 GPL 条款的开放源码项目。它是从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源码目录、编译形式与 Linux 内核很相似, 事实上, 不少 U-Boot 源码就是相应的 Linux 内核源程序的简化, 尤其是一些设备的驱动程序, 这从 U-Boot 源码的注释中能体现这一点。但是 U-Boot 不仅仅支持嵌入式 Linux 系统的引导, 而且还支持 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 嵌入式操作系统。其目前要支持的目标操作系统是 OpenBSD、NetBSD、FreeBSD、4.4BSD、Linux、SVR4、Esix、Solaris、Irix、SCO、Dell、NCR、VxWorks、LynxOS、pSOS、QNX、RTEMS、ARTOS。这是 U-Boot 中 Universal 的一层含义, 另外一层含义则是 U-Boot 除了支持 PowerPC 系列的处理器外, 还能支持 MIPS、x86、ARM、NIOS、XScale 等诸多常用系列的处理器。这两个特点正是 U-Boot 项目的开发目标, 即支持尽可能多的嵌入式处理器和嵌入式操作系统。就目前为止, U-Boot 对 PowerPC 系列处理器支持最为丰富, 对 Linux 的支持最完善。

U-Boot 的特点如下。

- 开放源码;
- 支持多种嵌入式操作系统内核, 如 Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、

LynxOS;

- 支持多个处理器系列, 如 PowerPC、ARM、x86、MIPS、XScale;
- 较高的可靠性和稳定性;
- 高度灵活的功能设置, 适合 U-Boot 调试, 操作系统不同引导要求, 产品发布等;
- 丰富的设备驱动源码, 如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等;
- 较为丰富的开发调试文档与强大的网络技术支持。

U-Boot 可支持的主要功能列表。

- 系统引导: 支持 NFS 挂载、RAMDISK (压缩或非压缩) 形式的根文件系统。支持 NFS 挂载, 并从 FLASH 中引导压缩或非压缩系统内核。

- 基本辅助功能: 强大的操作系统接口功能; 可灵活设置、传递多个关键参数给操作系统, 适合系统在不同开发阶段的调试要求与产品发布, 尤其对 Linux 支持最为强劲; 支持目标板环境参数多种存储方式, 如 FLASH、NVRAM、EEPROM; CRC32 校验, 可校验 FLASH 中内核、RAMDISK 镜像文件是否完好。

- 设备驱动: 串口、SDRAM、FLASH、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI、RTC 等驱动支持。

- 上电自检功能: SDRAM、FLASH 大小自动检测; SDRAM 故障检测; CPU 型号。
- 特殊功能: XIP 内核引导。



13.1.5 U-Boot 的常用命令

U-Boot 上电启动后，按任意键可以退出自动启动状态，进入命令行。

```
U-Boot 2013.01 (Aug 24 2014 - 12:01:19) for FS4412
```

```
CPU:   Exynos4412@1000MHz
```

```
Board: FS4412
```

```
DRAM:  1 GiB
```

```
WARNING: Caches not enabled
```

```
MMC:   MMC0: 3728 MB
```

```
In:    serial
```

```
Out:   serial
```

```
Err:   serial
```

```
MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK
```

```
eMMC CLOSE Success.!!
```

```
Checking Boot Mode ... EMMC4.41
```

```
Net:   dm9000
```

```
Hit any key to stop autoboot:  0
```

```
FS4412 #
```

在命令行提示符下，可以输入 U-Boot 的命令并执行。U-Boot 可以支持几十个常用命令，通过这些命令，可以对开发板进行调试，可以引导 Linux 内核，还可以擦写 Flash 完成系统部署等功能。掌握这些命令的使用，才能够顺利地进行嵌入式系统的开发。

输入 help 命令，可以得到当前 U-Boot 的所有命令列表。每一条命令后面是简单的命令说明。

```
FS4412 # help
```

```
?          - alias for 'help'
```

```
base       - print or set address offset
```

```
bdinfo     - print Board Info structure
```

```
boot       - boot default, i.e., run 'bootcmd'
```

```
bootd      - boot default, i.e., run 'bootcmd'
```

```
bootelf    - Boot from an ELF image in memory
```

```
bootm      - boot application image from memory
```

```
bootp      - boot image via network using BOOTP/TFTP protocol
```

```
bootvx     - Boot vxWorks from an ELF image
```




cmp - memory compare
coninfo - print console devices and information
cp - memory copy
crc32 - checksum calculation
dhcp - boot image via network using DHCP/TFTP protocol
echo - echo args to console
editenv - edit environment variable
emmc - Open/Close eMMC boot Partition
env - environment handling commands
erase - erase FLASH memory
exit - exit script
false - do nothing, unsuccessfully
fatinfo - print information about filesystem
fatload - load binary file from a dos filesystem
fatls - list files in a directory (default /)
fdisk - fdisk - fdisk for sd/mmc.

fdt - flattened device tree utility commands
flinfo - print FLASH memory information
go - start application at address 'addr'
help - print command description/usage
iminfo - print header information for application image
imxtract- extract a part of a multi-image
itest - return true/false on integer compare
loadb - load binary file over serial line (kermit mode)
loads - load S-Record file over serial line
loady - load binary file over serial line (ymodem mode)
loop - infinite loop on address range
md - memory display
mm - memory modify (auto-incrementing address)
mmc - MMC sub system
mmcinfo - display MMC info
movi - movi - sd/mmc r/w sub system for SMDK board

mtest - simple RAM read/write test



```
mw      - memory write (fill)
nm      - memory modify (constant address)
ping    - send ICMP ECHO_REQUEST to network host
printenv- print environment variables
protect - enable or disable FLASH write protection
reset   - Perform RESET of the CPU
run     - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv  - set environment variables
showvar - print local hushshell variables
sleep   - delay execution for some time
source  - run script from memory
test    - minimal test like /bin/sh
tftpboot- boot image via network using TFTP protocol
true    - do nothing, successfully
version - print monitor, compiler and linker version
```

U-Boot 还提供了更加详细的命令帮助，通过 `help` 命令还可以查看每个命令的参数说明。由于开发过程的需要，有必要先把 U-Boot 命令的用法弄清楚。接下来，根据每一条命令的帮助信息，解释一下这些命令的功能和参数。

13.2 实验目的

熟悉交叉工具链的使用、u-boot 常用命令、u-boot 的代码结构和移植方法。

13.3 实验平台

华清远见开发环境，FS4412 平台

13.4 实验步骤

13.4.1 建立自己的平台

拷贝源码（下载源码）：

（我们可以在下面这个网站上下载最新的和以前任一版本的 u-boot <ftp://ftp.denx.de/pub/u-boot/>）

```
$ mkdir ~/workdir/uboot -p
$ cd ~/workdir/uboot
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 移植实验源码\U-Boot 移植目录下的“u-boot-2013.01.tar.bz2”】拷贝到共享目录下。



名称	修改日期	类型	大小
CodeSign4SecureBoot	2016/9/24 17:57	文件夹	
u-boot-2013.01.tar.bz2	2014/11/26 13:27	WinRAR 压缩文件	9,489 KB

```
linux@ubuntu64-vm: ~/workdir/u-boot
linux@ubuntu64-vm:~$ mkdir ~/workdir/u-boot -p
linux@ubuntu64-vm:~$ cd workdir/u-boot/
linux@ubuntu64-vm:~/workdir/u-boot$ cp /mnt/hgfs/share/u-boot-2013.01.tar.bz2 ./
linux@ubuntu64-vm:~/workdir/u-boot$ tar xvf u-boot-2013.01.tar.bz2
```

```
tar xvf u-boot-2013.01.tar.bz2
cd u-boot-2013.01
```

我们关心的板级相关文件或目录

```
u-boot-2013.01/Makefile
u-boot- 2013.01/include/configs/origen.h
u-boot- 2013.01 /arch/arm/cpu/armv7/start.S
u-boot- 2013.01 /board/samsung/origen
u-boot- 2013.01 /arch/arm/lib
```

origen 是使用 exynos4412 芯片的参考板，我们在其基础之上移植 fs4412

```
$ cp -rf board/samsung/origen/ board/samsung/fs4412
```

```
$ mv board/samsung/fs4412/origen.c board/samsung/fs4412/fs4412.c
```

```
$ vim board/samsung/fs4412/Makefile
```

修改 origen.o 为 fs4412.o

```
$ cp include/configs/origen.h include/configs/fs4412.h
```

```
$ vim include/configs/fs4412.h
```

修改

```
#define CONFIG_SYS_PROMPT "ORIGEN #"
```

为

```
#define CONFIG_SYS_PROMPT "fs4412 #"
```

修改

```
#define CONFIG_IDENT_STRING for ORIGEN
```

为



```
#define CONFIG_IDENT_STRING for fs4412
```

```
$ vim boards.cfg
```

参考

```
origen arm armv7 origen samsung exynos
```

并在后面新增

```
fs4412 arm armv7 fs4412 samsung exynos
```

修改 u-boot 顶层目录下的 Makefile，指定交叉工具链

```
$ cd /home/linux/u-boot/u-boot-2013.01/
```

```
$ vim Makefile
```

在

```
ifeq ($(HOSTARCH), $(ARCH))
```

```
CROSS_COMPILE ?=
```

```
endif
```

下添加：

```
ifeq (arm, $(ARCH))
```

```
CROSS_COMPILE ?= arm-none-linux-gnueabi-
```

```
endif
```

编译 u-boot-2013.01：

```
$ make distclean
```

```
$ make fs4412_config
```

```
$ make
```

编译完成后生成的 u-boot.bin 就是可执行的镜像文件。但是该文件只能在 origen 平台上运行，我们需要对 u-boot 源代码进行相应的修改。

13.4.2 u-boot 移植

1、点灯确定代码执行

- 修改【arch/arm/cpu/armv7/start.S】在 134 行后添加

```
#if 1
```

```
ldr r0, =0x11000c40 @GPK2_7 led2
```

```
ldr r1, [r0]
```

```
bic r1, r1, #0xf0000000
```

```
orr r1, r1, #0x10000000
```



```
str r1, [r0]

ldr r0, =0x11000c44
mov r1, #0xff
str r1, [r0]

#endif
```

这部分代码是一段点灯程序，通过板上 LED2 的状态确定 uboot 第一行代码是否实行。

- 将【华清远见-CORTEXA9 资料：程序源码\Linux 移植实验源码\U-Boot 移植】目录下的 CodeSign4SecureBoot、build.sh 拷贝到 u-boot-2013.01 目录下

CodeSign4SecureBoot 是板级初始化代码，来源是三星的 BSP，用来做安全启动；build.sh 是编译脚本。

为编译脚本添加执行权限

```
$ chmod 777 u-boot-2013.01/build.sh
```

- 修改 Makefile

在

```
$(obj)u-boot.bin: $(obj)u-boot
    $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
    $(BOARD_SIZE_CHECK)
```

下添加

```
@#./mkuboot

@split -b 14336 u-boot.bin bl2

@+make -C sdfuse_q/

@#cp u-boot.bin u-boot-4212.bin

@#cp u-boot.bin u-boot-4412.bin

@#./sdfuse_q/add_sign

@./sdfuse_q/chksum

@./sdfuse_q/add_padding

@rm bl2a*

@echo
```

注意是 tab 键缩进的，否则 Makefile 编译报错

注意如果执行了 make dist clean 需重新拷贝 CodeSign4SecureBoot

- 编译

```
./build.sh
```



编译生成所需文件 u-boot_fs4412.bin

按前文烧写新的 u-boot_fs4412.bin

复位，发现灯有点亮，说明 我们的 u-boot 有运行到，不过这样原有 uboot 被覆盖，如需继续按前文使用 SD 卡重新烧写 uboot 到 EMMC 中。

2、实现串口输出

修改 lowlevel_init.S 文件

```
$vim board/samsung/fs4412/lowlevel_init.S
```

- 添加临时栈

在

```
lowlevel_init:
```

后添加

```
ldr sp,=0x02060000 @use iRom stack in bl2
```

- 添加关闭看门狗代码

在

```
beq wakeup_reset
```

后添加

```
#if 1 /*for close watchdog */
    /* PS-Hold high */
    ldr r0, =0x1002330c
    ldr r1, [r0]
    orr r1, r1, #0x300
    str r1, [r0]
    ldr r0, =0x11000c08
    ldr r1, =0x0
    str r1, [r0]
    /* Clear MASK_WDT_RESET_REQUEST */
    ldr r0, =0x1002040c
    ldr r1, =0x00
    str r1, [r0]
#endif
```

- 添加串口初始化代码

在 uart_asm_init: 的

```
str r1, [r0, #EXYNOS4_GPIO_A1_CON_OFFSET]
```



后添加

```
ldr r0,=0x10030000
ldr r1,=0x666666
ldr r2,=CLK_SRC_PERIL0_OFFSET
str r1,[r0,r2]
ldr r1,=0x777777
ldr r2,=CLK_DIV_PERIL0_OFFSET
str r1,[r0,r2]
```

注释掉 trustzone 初始化

注释掉

```
bl uart_asm_init
```

下的

```
bl tzpc_init
```

重新编译 u-boot

```
$ ./build.sh
```

烧写新的 u-boot_fs4412.bin

复位会看到串口信息

```
U-Boot 2013.01 (Aug 12 2014 - 10:31:53) for FS4412

CPU:      Exynos4412@1000MHz

Board: ORIGIN
DRAM:  1 GiB
WARNING: Caches not enabled
MMC:  SAMSUNG SDHCI: 0
Using default environment

In:      serial
Out:     serial
Err:     serial
Hit any key to stop autoboot:  0
FS4412 # █
```

3、网卡移植

- 添加网络初始化代码

```
$ vim board/samsung/fs4412/fs4412.c
```

在 struct exynos4_gpio_part2 *gpio2; 后添加

```
#ifdef CONFIG_DRIVER_DM9000
```

```
#define EXYNOS4412_SROMC_BASE 0X12570000
```



```
#define DM9000_Tacs      (0x1)
#define DM9000_Tcos      (0x1)
#define DM9000_Tacc      (0x5)
#define DM9000_Tcoh      (0x1)
#define DM9000_Tah      (0xC)
#define DM9000_Tacp      (0x9)
#define DM9000_PMC      (0x1)

struct exynos_sromc {
    unsigned int bw;
    unsigned int bc[6];
};

/*
 * s5p_config_sromc() - select the proper SROMC Bank and configure the
 * band width control and bank control registers
 * srom_bank      - SROM
 * srom_bw_conf   - SMC Band width reg configuration value
 * srom_bc_conf   - SMC Bank Control reg configuration value
 */
void exynos_config_sromc(u32 srom_bank, u32 srom_bw_conf, u32 srom_bc_conf)
{
    unsigned int tmp;
    struct exynos_sromc *srom = (struct exynos_sromc *) (EXYNOS4412_SROMC_BASE);

    /* Configure SMC_BW register to handle proper SROMC bank */
    tmp = srom->bw;
    tmp &= ~(0xF << (srom_bank * 4));
    tmp |= srom_bw_conf;
    srom->bw = tmp;

    /* Configure SMC_BC register */
    srom->bc[srom_bank] = srom_bc_conf;
}
```




```
static void dm9000aep_pre_init(void)
{
    unsigned int tmp;
    unsigned char smc_bank_num = 1;
    unsigned int    smc_bw_conf=0;
    unsigned int    smc_bc_conf=0;

    /* gpio configuration */
    writel(0x00220020, 0x11000000 + 0x120);
    writel(0x00002222, 0x11000000 + 0x140);
    /* 16 Bit bus width */
    writel(0x22222222, 0x11000000 + 0x180);
    writel(0x0000FFFF, 0x11000000 + 0x188);
    writel(0x22222222, 0x11000000 + 0x1C0);
    writel(0x0000FFFF, 0x11000000 + 0x1C8);
    writel(0x22222222, 0x11000000 + 0x1E0);
    writel(0x0000FFFF, 0x11000000 + 0x1E8);
    smc_bw_conf &= ~(0xf<<4);
    smc_bw_conf |= (1<<7) | (1<<6) | (1<<5) | (1<<4);
    smc_bc_conf = ((DM9000_Tacs << 28)
                   | (DM9000_Tcos << 24)
                   | (DM9000_Tacc << 16)
                   | (DM9000_Tcoh << 12)
                   | (DM9000_Tah << 8)
                   | (DM9000_Tacp << 4)
                   | (DM9000_PMC));
    exynos_config_sromc(smc_bank_num,smc_bw_conf,smc_bc_conf);
}
#endif
```

在 `gd->bd->bi_boot_params = (PHYS_SDRAM_1 + 0x100UL);` 后添加

```
#ifdef CONFIG_DRIVER_DM9000
    dm9000aep_pre_init();
#endif
```

在文件末尾添加



```
#ifdef CONFIG_CMD_NET
int board_eth_init(bd_t *bis)
{
    int rc = 0;
#ifdef CONFIG_DRIVER_DM9000
    rc = dm9000_initialize(bis);
#endif
    return rc;
}
#endif
```

- 修改配置文件添加网络相关配置

```
$ vim include/configs/fs4412.h
```

修改

```
#undef CONFIG_CMD_PING
```

为

```
#define CONFIG_CMD_PING
```

修改

```
#undef CONFIG_CMD_NET
```

为

```
#define CONFIG_CMD_NET
```

在文件末尾

```
#endif /* __CONFIG_H */
```

前面添加

```
#ifdef CONFIG_CMD_NET
#define CONFIG_NET_MULTI
#define CONFIG_DRIVER_DM9000 1
#define CONFIG_DM9000_BASE 0x05000000
#define DM9000_IO CONFIG_DM9000_BASE
#define DM9000_DATA (CONFIG_DM9000_BASE + 4)
#define CONFIG_DM9000_USE_16BIT
#define CONFIG_DM9000_NO_SROM 1
#define CONFIG_ETHADDR 11:22:33:44:55:66
#define CONFIG_IPADDR 192.168.9.200
#define CONFIG_SERVERIP 192.168.9.120
```



```
#define CONFIG_GATEWAYIP      192.168.9.1
#define CONFIG_NETMASK        255.255.255.0
#endif
```

- 重新编译 u-boot

```
$ ./build.sh
```

烧写新的 u-boot_fs4412.bin
复位后

```
# ping 192.168.9.120
```

```
FS4412 # ping 192.168.9.120
dm9000 i/o: 0x5000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
operating at 100M full duplex mode
Using dm9000 device
host 192.168.9.120 is alive
```

4、FLASH 移植 (EMMC)

- 初始化 EMMC

```
$cp movi.c arch/arm/cpu/armv7/exynos/
$vim arch/arm/cpu/armv7/exynos/Makefile
在 pinmux.o 后添加 movi.o
```

修改板级文件

```
$vim board/samsung/fs4412/fs4412.c
```

在

```
#include <asm/arch/mmc.h>
```

后面添加

```
#include <asm/arch/clk.h>
```

```
#include "origen_setup.h"
```

在

```
#ifdef CONFIG_GENERIC_MMC
```

后面添加

```
u32 sclk_mmc4; /*clock source for emmc controller*/
```

```
#define __REGMY(x) (*((volatile u32 *) (x)))
```

```
#define CLK_SRC_FSYS __REGMY(EXYNOS4_CLOCK_BASE + CLK_SRC_FSYS_OFFSET)
```



```
#define CLK_DIV_FSYS3 __REGMY(EXYNOS4_CLOCK_BASE + CLK_DIV_FSYS3_OFFSET)
```

```
int emmc_init()
```

```
{
```

```
    u32 tmp;
```

```
    u32 clock;
```

```
    u32 i;
```

```
    /* setup_hsmmc_clock */
```

```
    /* MMC4 clock src = SCLKMPLL */
```

```
    tmp = CLK_SRC_FSYS & ~(0x000f0000);
```

```
    CLK_SRC_FSYS = tmp | 0x00060000;
```

```
    /* MMC4 clock div */
```

```
    tmp = CLK_DIV_FSYS3 & ~(0x0000ff0f);
```

```
    clock = get_pll_clk(MPLL)/1000000;
```

```
    for(i=0 ; i<=0xf; i++) {
```

```
        sclk_mmc4=(clock/(i+1));
```

```
        if(sclk_mmc4 <= 160) //200
```

```
        {
```

```
            CLK_DIV_FSYS3 = tmp | (i<<0);
```

```
            break;
```

```
        }
```

```
    }
```

```
    emmcdbg("[mjdbg] sclk_mmc4:%d MHZ; mmc_ratio: %d\n",sclk_mmc4,i);
```

```
    sclk_mmc4 *= 1000000;
```

```
    /*
```

```
    * MMC4 EMMC GPIO CONFIG
```

```
    *
```

```
    * GPK0[0]    SD_4_CLK
```

```
    * GPK0[1]    SD_4_CMD
```

```
    * GPK0[2]    SD_4_CDn
```

```
    * GPK0[3:6]  SD_4_DATA[0:3]
```

```
    */
```



```
writel(readl(0x11000048)&~(0xf),0x11000048); //SD_4_CLK/SD_4_CMD pull-down enable
writel(readl(0x11000040)&~(0xff),0x11000040); //cdn set to be output

writel(readl(0x11000048)&~(3<<4),0x11000048); //cdn pull-down disable
writel(readl(0x11000044)&~(1<<2),0x11000044); //cdn output 0 to shutdown the emmc power
writel(readl(0x11000040)&~(0xf<<8)|(1<<8),0x11000040); //cdn set to be output
udelay(100*1000);
writel(readl(0x11000044)|(1<<2),0x11000044); //cdn output 1

writel(0x0333133, 0x11000040);

writel(0x00003FF0, 0x11000048);
writel(0x00002AAA, 0x1100004C);

#ifdef CONFIG_EMMC_8Bit
    writel(0x04444000, 0x11000060);
    writel(0x00003FC0, 0x11000068);
    writel(0x00002AAA, 0x1100006C);
#endif

#ifdef USE_MMC4
    smdk_s5p_mshc_init();
#endif
}
```

将 int board_mmc_init(bd_t *bis)函数内容改写为

```
int board_mmc_init(bd_t *bis)
{
    int i, err;

#ifdef CONFIG_EMMC
    err = emmc_init();
#endif

    return err;
}
```



在末尾添加

```
#ifndef CONFIG_BOARD_LATE_INIT
#include <movi.h>
int  chk_bootdev(void)//mj for boot device check
{
    char run_cmd[100];
    struct mmc *mmc;
    int boot_dev = 0;
    int cmp_off = 0x10;
    ulong  start_blk, blkcnt;

    mmc = find_mmc_device(0);

    if (mmc == NULL)
    {
        printf("There is no eMMC card, Booting device is SD card\n");
        boot_dev = 1;
        return boot_dev;
    }

    start_blk = (24*1024/MOVI_BLKSIZE);
    blkcnt = 0x10;

    sprintf(run_cmd,"emmc open 0");
    run_command(run_cmd, 0);

    sprintf(run_cmd,"mmc read 0 %lx %lx %lx",CFG_PHY_KERNEL_BASE,start_blk,blkcnt);
    run_command(run_cmd, 0);

    /* switch mmc to normal paritition */
    sprintf(run_cmd,"emmc close 0");
    run_command(run_cmd, 0);

    return 0;
}

int board_late_init (void)
```



```
{  
  
    int boot_dev=0;  
    char boot_cmd[100];  
    boot_dev = chk_bootdev();  
    if(!boot_dev)  
    {  
        printf("\n\nChecking Boot Mode ... EMMC4.41\n");  
    }  
    return 0;  
}  
#endif
```

- 添加相关命令

```
$ cp    cmd_movi.c  common/  
$ cp    cmd_mmc.c  common/  
$ cp    cmd_mmc_fdisk.c  common/
```

修改 Makefile

```
$ vim    common/Makefile
```

在

```
COBJS-$(CONFIG_CMD_MMC) += cmd_mmc.o
```

后添加

```
COBJS-$(CONFIG_CMD_MMC) += cmd_mmc_fdisk.o  
COBJS-$(CONFIG_CMD_MOVINAND) += cmd_movi.o
```

添加驱动

```
$ cp    mmc.c  drivers/mmc/  
$ cp    s5p_mshc.c  drivers/mmc/  
$ cp    mmc.h  include/  
$ cp    movi.h  include/  
$ cp    s5p_mshc.h  include/
```

修改 Makefile

```
$vim  drivers/mmc/Makefile
```

添加

```
COBJS-$(CONFIG_S5P_MSHC) += s5p_mshc.o
```



- 添加 EMMC 相关配置

```
$vim include/configs/fs4412.h
```

添加

```
#define CONFIG_EVT1      1      /* EVT1 */

#ifndef CONFIG_EVT1

#define CONFIG_EMMC44_CH4 //eMMC44_CH4 (OMPIN[5:1] = 4)

#ifndef CONFIG_SDMMC_CH2
#define CONFIG_S3C_HSMMC
#undef DEBUG_S3C_HSMMC
#define USE_MMC2
#endif

#ifndef CONFIG_EMMC44_CH4
#define CONFIG_S5P_MSHC
#define CONFIG_EMMC      1
#define USE_MMC4
/* #define CONFIG_EMMC_8Bit */
#define CONFIG_EMMC_EMERGENCY
/* #define emmcdbg(fmt,args...) printf(fmt,##args) */
#define emmcdbg(fmt,args...)
#endif

#endif /*end CONFIG_EVT1*/

#define CONFIG_CMD_MOVINAND
#define CONFIG_CLK_1000_400_200
#define CFG_PHY_UBOOT_BASE      CONFIG_SYS_SDRAM_BASE + 0x3e00000
#define CFG_PHY_KERNEL_BASE    CONFIG_SYS_SDRAM_BASE + 0x8000

#define BOOT_MMCSDB      0x3
#define BOOT_EMMC43      0x6
#define BOOT_EMMC441     0x7
#define CONFIG_BOARD_LATE_INIT
```

- 重新编译 u-boot

```
$ ./build.sh
```




烧写新的 u-boot_fs4412.bin

复位后

mmcinfo

```
CPU:      Exynos4412@1000MHz

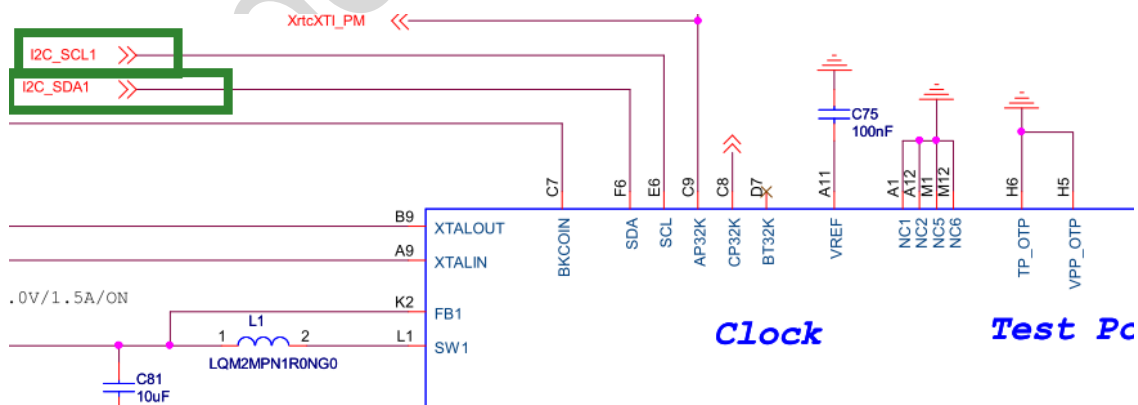
Board: FS4412
DRAM:    1 GiB
WARNING: Caches not enabled
MMC:     MMC0: 3728 MB
In:       serial
Out:      serial
Err:      serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK
eMMC CLOSE Success.!!

Checking Boot Mode ... EMMC4.41
Net:      dm9000
Hit any key to stop autoboot:  0
FS4412 # mmcinfo
Device: S5P_MSHC4
Manufacturer ID: 15
OEM: 100
Name: 4YMD3
Tran Speed: 0
Rd Block Len: 512
MMC version 4.0
High Capacity: Yes
Capacity: 7.3 MiB
Bus Width: 2-bit
FS4412 #
```

5、移植电源管理

核心板采用的电源管理芯片为 Samsung 公司的 S5M8767A 芯片，电源管理在 uboot 中一般称之为 PMIC 或者 PMU；该芯片通过 I2C1 和处理器连接，系统通过读写 I2C 总线初始化电源管理芯片，配置芯片部分输出电压。下图是 PMIC 部分电路图。



S5M8767A 有 9 路 BUCK 和 28 路 LDO，暂且可以当成有 37 路供电电路，这 37 路供电电路最低可以使用 6.25mv 的步进电压，多达 60 多个档位可以做到对输出电压的精确控制。另外该芯片还



有一个硬件 RTC，可以在有外部电池供电的情况下保存时钟信息。

该芯片的 I2C 通信从机地址（Power Manager 和 RTC 的地址不同）：

4.1 Slave Address

Slave address is used to select the S5M8767A on the I2C bus. This address consists of 8-bit data. The LSB 1-bit determines the read or write mode. If the LSB 1-bit is low, the write mode is selected.

Section	Address selection	Access Mode
PM section	1100 1100 (0xCC)	PM Write
	1100 1101 (0xCD)	PM Read
RTC section	0000 1100 (0x0C)	RTC Write
	0000 1101 (0x0D)	RTC Read

用户确保自己当前位置在 u-boot 源码的顶层目录下，执行如下命令。

```
$ touch drivers/power/pmic/pmic_s5m8767.c
```

在该文件中添加如下内容：

```
1  #include <errno.h>
2
3  #define I2C_WRITE 0
4  #define I2C_READ 1
5
6  #define I2C_OK      0
7  #define I2C_NOK     1
8  #define I2C_NACK    2
9  #define I2C_NOK_LA  3      /* Lost arbitration */
10 #define I2C_NOK_TOUT 4      /* time out */
11
12 #define I2CSTAT_BSY 0x20    /* Busy bit */
13 #define I2CSTAT_NACK 0x01   /* Nack bit */
14 #define I2CCON_IRPND 0x10   /* interrupt pending bit */
15 #define I2C_MODE_MT 0xC0    /* Master Transmit Mode */
16 #define I2C_MODE_MR 0x80    /* Master Receive Mode */
17 #define I2C_START_STOP 0x20 /* START / STOP */
18 #define I2C_TXRX_ENA 0x10    /* I2C Tx/Rx enable */
19 #define I2C_TIMEOUT 1 // 1 second
20
21 #define S5M8767_I2C_ADDR (0xCC)
22 //I2C 1 add by jinhc
23 #define rIICCON *((volatile unsigned long *) 0x13870000)
24 #define rIICSTAT *((volatile unsigned long *) 0x13870004)
25 #define rIICADD *((volatile unsigned long *) 0x13870008)
26 #define rIICDS *((volatile unsigned long *) 0x1387000c)
27 #define rIICIC *((volatile unsigned long *) 0x13870010)
28
29 #define BUCK9_CTRL_ADDR 0x5A
30 #define LD018_CTRL_ADDR 0x70
```



```

31
32 #define CONFIG_PM_VDD_ARM          1.2
33 #define CONFIG_PM_VDD_INT          1.0
34 #define CONFIG_PM_VDD_G3D          1.1
35 #define CONFIG_PM_VDD_MIF          1.1
36 #define CONFIG_PM_VDD_LD014        1.8
37
38 #define CALC_S5M8767_VOLT1(x)      ((x < 600) ? 0 : ((x - 600) / 6.25))
39 #define CALC_S5M8767_VOLT2(x)      ((x < 650) ? 0 : ((x - 650) / 6.25))
40
41 typedef enum {
42     PMIC_BUCK1 = 0,
43     PMIC_BUCK2,
44     PMIC_BUCK3,
45     PMIC_BUCK4,
46     PMIC_LD014,
47     PMIC_LD010,
48     PMIC_BUCK8,
49 }PMIC_RegNum;
50 static
51 int WaitForXfer (void)
52 {
53     int i;
54     unsigned long status;
55
56     i = I2C_TIMEOUT * 10000;
57     status = rIICCON;
58     while ((i > 0) && !(status & I2CCON_IRPND)) {
59         udelay (100);
60         status = rIICCON;
61         i--;
62     }
63
64     return (status & I2CCON_IRPND) ? I2C_OK : I2C_NOK_TOUT;
65 }
66
67 static
68 int IsACK (void)
69 {
70     return (!(rIICSTAT & I2CSTAT_NACK));
71 }
72
73 static
74 void ReadWriteByte (void)

```



```
75 {
76     rIICCON &= ~I2CCON_IRPND;
77 }
78
79 /*
80  * cmd_type is 0 for write, 1 for read.
81  *
82  * addr_len can take any value from 0-255, it is only limited
83  * by the char, we could make it larger if needed. If it is
84  * 0 we skip the address write cycle.
85  */
86 static
87 int i2c_transfer (unsigned char cmd_type,
88                  unsigned char chip,
89                  unsigned char addr[],
90                  unsigned char addr_len,
91                  unsigned char data[], unsigned short data_len)
92 {
93     int i, status, result;
94
95     if (data == 0 || data_len == 0) {
96
97         return I2C_NOK;
98     }
99
100     i = I2C_TIMEOUT * 1000;
101     status = rIICSTAT;
102     while ((i > 0) && (status & I2CSTAT_BSY)) {
103         udelay (1000);
104         status = rIICSTAT;
105         i--;
106     }
107
108     if (status & I2CSTAT_BSY)
109         return I2C_NOK_TOUT;
110
111     rIICCON |= 0x80;
112     result = I2C_OK;
113
114     switch (cmd_type) {
115     case I2C_WRITE:
116         if (addr && addr_len) {
117             rIICDS = chip;
```



```

119         rIICSTAT = I2C_MODE_MT | I2C_TXRX_ENA | I2C_START_STOP;
120         i = 0;
121         while ((i < addr_len) && (result == I2C_OK)) {
122             result = WaitForXfer ();
123             rIICDS = addr[i];
124             ReadWriteByte ();
125             i++;
126         }
127         i = 0;
128         while ((i < data_len) && (result == I2C_OK)) {
129             result = WaitForXfer ();
130             rIICDS = data[i];
131             ReadWriteByte ();
132             i++;
133         }
134     } else {
135         rIICDS = chip;
136         rIICSTAT = I2C_MODE_MT | I2C_TXRX_ENA | I2C_START_STOP;
137         i = 0;
138         while ((i < data_len) && (result == I2C_OK)) {
139             result = WaitForXfer ();
140             rIICDS = data[i];
141             ReadWriteByte ();
142             i++;
143         }
144     }

145
146     if (result == I2C_OK)
147         result = WaitForXfer ();
148
149     rIICSTAT = I2C_MODE_MR | I2C_TXRX_ENA;
150     ReadWriteByte ();
151     break;
152 case I2C_READ:
153     rIICSTAT = I2C_MODE_MT | I2C_TXRX_ENA ;
154     rIICDS = chip;
155     rIICSTAT |= I2C_START_STOP;
156     i = 0;
157     result = WaitForXfer ();
158     if (IsACK ()) {
159
160         i = 0;
161         while ((i < addr_len) && (result == I2C_OK)) {
162             rIICDS = addr[i];

```



```

163         ReadWriteByte ();
164         result = WaitForXfer ();
165         i++;
166     }
167
168     rIICDS = chip;
169     rIICSTAT = I2C_MODE_MR | I2C_TXRX_ENA |
170         I2C_START_STOP;
171
172     ReadWriteByte ();
173
174     result = WaitForXfer ();
175     i = 0;
176
177     while ((i < data_len) && (result == I2C_OK)) {
178         if (i == data_len - 1)
179             rIICCON &= ~0x80;
180         ReadWriteByte ();
181         result = WaitForXfer ();
182         data[i] = rIICDS;
183         i++;
184     }
185
186     } else {
187         result = I2C_NACK;
188     }
189
190     rIICSTAT = I2C_MODE_MR | I2C_TXRX_ENA;
191     ReadWriteByte ();
192     break;
193 default:
194     result = I2C_NOK;
195     break;
196 }
197
198     return (result);
199 }
200 static int i2c_read(unsigned char chip, unsigned int addr,
201     int alen, unsigned char *buffer, int len)
202 {
203     unsigned char xaddr[4];
204     int ret;
205     if (alen > 4)
206         return 1;

```



```

207
208     if (alen > 0) {
209         xaddr[0] = (addr >> 24) & 0xFF;
210         xaddr[1] = (addr >> 16) & 0xFF;
211         xaddr[2] = (addr >> 8) & 0xFF;
212         xaddr[3] = addr & 0xFF;
213     }
214
215     if ((ret = i2c_transfer(I2C_READ, chip, &xaddr[4 - alen], alen,
216         buffer, len)) != 0) {
217         return 1;
218     }
219     return 0;
220 }
221 static int i2c_write(unsigned char chip, unsigned int addr,
222     int alen, unsigned char *buffer, int len)
223 {
224     unsigned char xaddr[4];
225     int ret;
226
227     if (alen > 4)
228         return 1;
229     if (alen > 0) {
230         xaddr[0] = (addr >> 24) & 0xFF;
231         xaddr[1] = (addr >> 16) & 0xFF;
232         xaddr[2] = (addr >> 8) & 0xFF;
233         xaddr[3] = addr & 0xFF;
234 #if 0
235         printf("xaddr[3] = %02x\n", xaddr[3]);
236 #endif
237     }
238
239     return(i2c_transfer(I2C_WRITE, chip, &xaddr[4 - alen], alen,
240         buffer, len) != 0);
241
242 }
243 static void i2c1_init(void) {
244     int i, status;
245     volatile unsigned int* gpd1_con_reg = (unsigned int*)0x114000C0; //GPD1CON
246     unsigned int value;
247
248     value = *gpd1_con_reg;
249     value &= ~(0xFF<<8); //clear bits gpio_reg1 [15:7]
250     value |= 0x2200; //set GPD1CON2 & GPD1CON3 as I2C_1_SDA I2C_1_SCL

```



```

251     *gpd1_con_reg = value;
252
253
254     /*wait for some time to give previous transfer a chance to finish */
255     i = I2C_TIMEOUT * 1000;
256     status = rIICCON;
257     while ((i > 0) && !(status & I2CSTAT_BSY)) {
258         udelay(1000);
259         status = rIICSTAT;
260         i--;
261     }
262
263     rIICCON = (1 << 6) | (1 << 5) | (7&0xF);
264     rIICSTAT = 0;
265     rIICADD = S5M8767_I2C_ADDR >> 1;  //7-bit slave address Slave address[7:1]
266
267     rIICSTAT = I2C_MODE_MT | I2C_TXRX_ENA;
268 }
269
270 void lowlevel_init_s5m8767(unsigned char Address, unsigned char *Val, int flag)
271 {
272     unsigned char addr;
273     unsigned char data[4] = {0, 0, 0, 0};
274     unsigned char write_data = 0;
275     int ret;
276
277     addr = Address;
278     data[0] = *Val;
279     write_data = *Val;
280
281     //flag == 0 is read reg, flag==1 is write reg;
282     if (flag == 0){
283         ret = i2c_read(S5M8767_I2C_ADDR, addr, 1, data, 1);
284         *Val = data[0];
285         //printf("s5m8767: addr 0x%x, value 0x%x\n", addr, data[0]);
286     }else {
287         ret = i2c_write(S5M8767_I2C_ADDR, addr, 1, data, 1);
288         data[0] = 0;
289     #if 0
290         ret = i2c_read(S5M8767_I2C_ADDR, addr, 1, data, 1);
291         *Val = data[0];
292         printf("s5m8767: addr 0x%x, write value 0x%x, read value0x%x\n", addr, write_data,
293             data[0]);
294     #endif

```




```

295     }
296 }
297
298 void I2C_S5M8767_VolSetting(PMIC_RegNum eRegNum, unsigned char ucVolLevel, unsigned char u
299 nEnable)
300 {
301     unsigned char reg_addr, vol_level;
302
303     switch(eRegNum) {
304         case PMIC_BUCK1: reg_addr = 0x33; break;
305         case PMIC_BUCK2: reg_addr = 0x35; break;
306         case PMIC_BUCK3: reg_addr = 0x3E; break;
307         case PMIC_BUCK4: reg_addr = 0x47; break;
308         case PMIC_BUCK8: reg_addr = 0x59; break;
309         default:
310             while(1);
311     }
312     vol_level = ucVolLevel&0xFF;
313
314     lowlevel_init_s5m8767(reg_addr, &vol_level, 1);
315 }
316
317 void s5m8767_voltage_set(void)
318 {
319     float vdd_arm, vdd_int, vdd_g3d;
320     float vdd_mif;
321     unsigned char read_data;
322
323     vdd_arm = CONFIG_PM_VDD_ARM;
324     vdd_int = CONFIG_PM_VDD_INT;
325     vdd_g3d = CONFIG_PM_VDD_G3D;
326     vdd_mif = CONFIG_PM_VDD_MIF;
327
328     I2C_S5M8767_VolSetting(PMIC_BUCK1, CALC_S5M8767_VOLT2(vdd_mif * 1000), 1);
329     I2C_S5M8767_VolSetting(PMIC_BUCK2, CALC_S5M8767_VOLT1(vdd_arm * 1000), 1);
330     I2C_S5M8767_VolSetting(PMIC_BUCK3, CALC_S5M8767_VOLT1(vdd_int * 1000), 1);
331     I2C_S5M8767_VolSetting(PMIC_BUCK4, CALC_S5M8767_VOLT1(vdd_g3d * 1000), 1);
332     //set BUCK8 to 1.55v, because LD02's out decide by BUCK8
333     I2C_S5M8767_VolSetting(PMIC_BUCK8, CALC_S5M8767_VOLT1(1.55 * 1000), 1);
334
335 }
336 int Is_TC4_Dvt = 0;
337 void pmic_s5m8767_init(void)
338 {

```



```

339     unsigned char id;
340     unsigned int val;
341
342     /* init i2c_1 */
343     i2c1_init();
344     /* read s5m8767's chip id */
345     lowlevel_init_s5m8767(0, &id, 0);
346 #if 0
347     printf("s5m8767 chip id = %02x\n", id);
348 #endif
349     if (id == 0x15) {
350         printf("PMIC: S5M8767(VER5.0)\n");
351         s5m8767_voltage_set();
352         Is_TC4_Dvt = 2;
353     }
354     if (Is_TC4_Dvt == 2) {
355         val = 0x58;
356         lowlevel_init_s5m8767(BUCK9_CTRL_ADDR, &val, 1);
357     }
358     val = 0x32;
359     lowlevel_init_s5m8767(LDO18_CTRL_ADDR, &val, 1);
    }

```

添加完成后保存退出。

修改[u-boot]/drivers/power/pmic/Makefile:

```
$ vi drivers/power/pmic/Makefile
```

添加

```
COBJS-$(CONFIG_POWER_S5M8767) += pmic_s5m8767.o
```

```

25
26 LIB := $(obj)libpmic.o
27
28 COBJS-$(CONFIG_POWER_S5M8767) += pmic_s5m8767.o
29 COBJS-$(CONFIG_POWER_MAX8998) += pmic_max8998.o
30 COBJS-$(CONFIG_POWER_MAX8997) += pmic_max8997.o
31 COBJS-$(CONFIG_POWER_MUIC_MAX8997) += muic_max8997.o
32 COBJS-$(CONFIG_POWER_MAX77686) += pmic_max77686.o
33
34 COBJS := $(COBJS-y)
35 SRCS := $(COBJS:.o=.c)
drivers/power/pmic/Makefile

```

修改板级配置文件

```
$ vim include/configs/fs4412.h
```

添加

```
#define CONFIG_POWER_S5M8767
```



```
80 #define CONFIG_ENV_OVERWRITE
81
82 #define CONFIG_POWER_S5M8767
83 /* Command definition */
84 #include <config_cmd_default.h>
85
86 #define CONFIG_CMD_PING
87 #define CONFIG_CMD_ELF
#include/configs/fs4412.h
```

修改板级文件

```
$ vim board/samsung/fs4412/fs4412.c
```

在 board_init()函数中添加

```
#ifdef CONFIG_POWER_S5M8767
    pmic_s5m8767_init();
#endif
```

```
110 int board_init(void)
111 {
112     gpio1 = (struct exynos4_gpio_part1 *) EXYNOS4_GPIO_PART1_BASE;
113     gpio2 = (struct exynos4_gpio_part2 *) EXYNOS4_GPIO_PART2_BASE;
114
115     gd->bd->bi_boot_params = (PHYS_SDRAM_1 + 0x100UL);
116
117 #ifdef CONFIG_DRIVER_DM9000
118     dm9000aep_pre_init();
119 #endif
120
121 #ifdef CONFIG_POWER_S5M8767
122     pmic_s5m8767_init();
123 #endif
124     return 0;
125 }
```

重新编译 uboot

```
$ ./build.sh
```

编译成功后，按照前文烧写 u-boot-fs4412.bin

重新启动开发板后，现象如下图：

```
U-Boot 2013.01 (Nov 02 2016 - 09:20:06) for FS4412

CPU:   Exynos4412@1000MHz

Board: FS4412
DRAM:  1 GiB
WARNING: Caches not enabled
PMIC: S5M8767(VER5.0)
MMC:   MMC0: 14910 MB
In:    serial
Out:   serial
Err:   serial
```

有该现象说明已经移植成功（必要时可以打开部分调试语句，调试完成后关闭调试语句）。



6、2G 内存适配

确定板子内存大小，如果是 2G 的，那么需要修改如下代码：

```
$ vim include/configs/fs4412.h
```

修改

```
#define SDRAM_BANK_SIZE (256UL << 20UL)
```

为

```
#define SDRAM_BANK_SIZE (512UL << 20UL)
```

重新编译 uboot 即可。

注意在 fs4412.h 中添加注释，必须重/* 注释*/ 注释，而不能用双斜杠。

13.5 实验现象

```
COM1 - PuTTY
NAND: 1024 MiB
*** Warning - bad CRC or NAND, using default environment

In: serial
Out: serial
Err: serial
Net: dm9000
FS210 #
FS210 #
FS210 #
FS210 # print
bootdelay=3
baudrate=115200
ethaddr=11:22:33:44:55:66
ipaddr=192.168.0.200
serverip=192.168.0.110
gatewayip=192.168.0.1
stdin=serial
stdout=serial
stderr=serial
ethact=dm9000

Environment size: 175/131068 bytes
FS210 #
```



第 14 章 Linux 系统移植实验

14.1 实验原理

Linux 内核是 Linux 操作系统的核心，也是整个 Linux 功能体现。它是用 C 语言编写，符合 POSIX 标准。Linux 最早是由芬兰黑客 Linus Torvalds 为尝试在英特尔 X86 架构上提供自由免费的类 Unix 操作系统而开发的。该计划开始于 1991 年，这里有一份 Linus Torvalds 当时在 Usenet 新闻组 comp.os.minix 所登载的帖子，这份著名的帖子标志着 Linux 计划的正式开始。在计划的早期有一些 Minix 黑客提供了协助，而今天全球无数程序员正在为该计划无偿提供帮助。

今天 Linux 是一个一体化内核（Monolithic Kernel）系统。设备驱动程序可以完全访问硬件。Linux 内的设备驱动程序可以方便地以模块化（Modularize）的形式设置，并在系统运行期间可直接装载或卸载。

Linux 内核主要功能包括：进程管理、内存管理、文件管理、设备管理、网络管理等。

- 进程管理：进程是在计算机系统中资源分配的最小单元。内核负责创建和销毁进程，而且由调度程序采取合适的调度策略，实现进程之间的合理且实时的处理器资源的共享。从而内核的进程管理活动实现了多个进程在一个或多个处理器之上的抽象。内核还负责实现不同进程之间、进程和其他部件之间的通信。

- 内存管理：内存是计算机系统最主要的资源。内核使得多个进程安全而合理地共享内存资源，为每个进程在有限的物理资源上建立一个虚拟地址空间。内存管理部分代码可以分为硬件无关部分和硬件有关部分：硬件无关部分实现进程和内存之间的地址映射等功能；硬件有关部分实现不同体系结构上的内存管理相关功能并为内存管理提供硬件无关的虚拟接口。

- 文件管理：在 Linux 系统中的任何一个概念几乎都可以看作一个文件。内核在非结构化的硬件之上建立了一个结构化的虚拟文件系统，隐藏了各种硬件的具体细节。从而在整个系统的几乎所有机制中使用文件的抽象。Linux 在不同物理介质或虚拟结构上支持数十种文件系统。例如，Linux 支持磁盘的标准文件系统 ext3 和虚拟的特殊文件系统。

- 设备管理：Linux 系统中几乎每个系统操作最终都映射到一个或多个物理设备上。除了处理器，内存等少数的硬件资源之外，任何一种设备控制操作都由设备特定的驱动代码来进行。内核中必须提供系统中可能要操作的每一种外设的驱动。

网络管理：内核支持各种网络标准协议和网络设备。网络管理部分可分为网络协议栈和网络设备驱动程序。网络协议栈负责实现每种可能的网络传输协议（TCP/IP 协议等）；网络设备驱动程序负责与各种网络硬件设备或虚拟设备进行通讯。

Linux 内核源代码非常庞大，随着版本的发展不断增加。它使用目录树结构，并且使用 Makefile 组织配置编译。

初次接触 Linux 内核，最好仔细阅读顶层目录的 readme 文件，它是 Linux 内核的概述和编译命令说明。readme 的说明更加针对 X86 等通用的平台，对于某些特殊的体系结构，可能有些特殊的地方。

顶层目录的 Makefile 是整个内核配置编译的核心文件，负责组织目录树中子目录的编译管理，还可以设置体系结构和版本号等。



内核源码的顶层有许多子目录，分别组织存放各种内核子系统或者文件。具体的目录说明见表 1.7。

表 1.7 Linux 内核源码顶层目录说明

Arch/	体系结构相关的代码，如 arch/i386、arch/arm、arch/ppc
crypto	常用加密和散列算法（如 AES、SHA 等），以及一些压缩和 CRC 校验算法
drivers/	各种设备驱动程序，例如，drivers/char、drivers/block、...
documentation/	内核文档
fs/	文件系统，例如，fs/ext3、fs/jffs2、...
include/	内核头文件：include/asm 是体系结构相关的头文件，它是 include/asm-arm、include/asm-i386 等目录的链接。include/linux 是 Linux 内核基本的头文件
init/	Linux 初始化，如 main.c
ipc/	进程间通信的代码
kernel/	Linux 内核核心代码（这部分比较小）
lib/	各种库子程序，如 zlib、crc32
mm/	内存管理代码
net/	网络支持代码，主要是网络协议
sound	声音驱动的支持
scripts/	内部或者外部使用的脚本
usr/	用户的代码

14.2 内核的配置和编译

14.2.1 解压内核

```
$ cd ~
$ mkdir kernel
$ cd kernel
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 移植实验源码\内核移植】目录下的“linux-3.14.tar.xz”拷贝到该目录下

```
$ tar xvf linux-3.14.tar.xz
$ cd linux-3.14
```

14.2.2 修改内核顶层目录下的 Makefile

```
$ vi Makefile
```

修改：

```
ARCH      ?= $(SUBARCH)
```



```
CROSS_COMPILE      ?= $(CONFIG_CROSS_COMPILE:"%"=%)
```

为:

```
ARCH      ?= arm
```

```
CROSS_COMPILE      ?= arm-none-linux-gnueabi-
```

14.2.3 拷贝标准板配置文件

```
$ cp arch/arm/configs/exynos_defconfig .config
```

14.2.4 配置内核

```
$ make menuconfig
```

```
System Type  --->
```

```
(2) S3C UART to use for low-level messages
```

该命令执行时会弹出一个菜单，我们可以对内核进行详细的配置。这里我们先查看一下，内核都提供了哪些功能！

14.2.5 编译内核

```
$ make uImage
```

通过上述操作我们能够在 arch/arm/boot 目录下生成一个 uImage 文件，这就是经过压缩的内核镜像。

如果编译过程中提示缺少 mkimage 工具，需将上一章 uboot 源码中的 tools/mkimage 拷贝到 ubuntu 的 /usr/bin 目录下

```
$ cp u-boot-2013.01/tools/mkimage /usr/bin
```

14.2.6 生成设备树

生成设备树文件，以参考板 origen 的设备树文件为参考。

```
$ cp arch/arm/boot/dts/exynos4412-origen.dts arch/arm/boot/dts/exynos4412-fs4412.dts
```

添加新文件需修改 Makefile 才能编译

```
$ vim arch/arm/boot/dts/Makefile
```

在

```
exynos4412-origen.dtb \
```

下添加如下内容

```
exynos4412-fs4412.dtb \
```

编译设备树文件

```
$ make dtbs
```

拷贝内核和设备树文件到/tftpboot 目录下

```
$ cp arch/arm/boot/uImage /tftpboot
```

```
$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/
```



修改 uboot 启动参数

重启板子在系统倒计时时按任意键结束启动，输入如下内容修改 uboot 环境变量：

```
# setenv serverip 192.168.9.120
# setenv ipaddr 192.168.9.233
# setenv bootcmd tftp 41000000 uImage;tftp 42000000 exynos4412-fs4412.dtb;bootm 41000000 -
42000000
#setenv bootargs root=/dev/nfs nfsroot=192.168.9.120:/source/rootfs rw console=ttySAC2,115200
init=/linuxrc ip=192.168.9.233
# saveenv
```

注意：192.168.9.120 对应 Ubuntu 的 ip

192.168.9.233 对应板子的 ip

这两个 ip 应该根据自己的实际情况适当修改

重启开发板查看现象，注意由于此时还没有移植网卡驱动，所以无法正常挂载网络文件系统。

14.3 以太网卡驱动移植

14.3.1 实验目的

网卡是嵌入式产品最常用的设备，这里我们需要完成网卡驱动的移植。FS4412 使用的是 DM9000 网卡，我们通过这个实验能够了解如何在内核中添加网卡驱动及网络功能的基本配置。

说明：在本系统移植课程实验中命令行提示符 “\$” 表示是在主机上执行，“#” 表示在目标板执行

14.3.2 实验平台

华清远见开发环境，FS4412 平台；

14.3.3 实验步骤

运行 Ubuntu 12.04 系统，打开命令行终端，使用实验 14.2 的内核

```
$ cd ~/kernel/ linux-3.14
```

设备树文件修改

```
$ vim arch/arm/boot/dts/exynos4412-fs4412.dts
```

添加如下内容，添加到文件末尾 “}” 里面

```
srom-cs1@5000000 {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    reg = <0x50000000 0x1000000>;
```




```
ranges;

ethernet@5000000 {
    compatible = "davicom,dm9000";
    reg = <0x5000000 0x2 0x5000004 0x2>;
    interrupt-parent = <&gpx0>;
    interrupts = <6 4>;
    davicom,no-eeeprom;
    mac-address = [00 0a 2d a6 55 a2];
};

};
```

修改文件 driver/clock/clock.c

修改

```
static bool clk_ignore_unused;
```

为

```
static bool clk_ignore_unused = true;
```

配置内核:

make menuconfig

[*] Networking support --->

Networking options --->

<*> Packet socket

<*> Unix domain sockets

[*] TCP/IP networking

[*] IP: kernel level autoconfiguration

Device Drivers --->

[*] Network device support --->

[*] Ethernet driver support (NEW) --->

<*> DM9000 support

File systems --->

[*] Network File Systems (NEW) --->

<*> NFS client support

[*] NFS client support for NFS version 2

[*] NFS client support for NFS version 3

[*] NFS client support for the NFSv3 ACL protocol extension



[*] Root file system on NFS

编译内核和设备树

```
$ make uImage
```

```
$ make dtbs
```

测试:

拷贝内核和设备树文件到/tftpboot 目录下

```
$ cp arch/arm/boot/uImage /tftpboot
```

```
$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/
```

启动开发板，修改内核启动参数，通过 NFS 方式挂载根文件系统，文件系统暂用【华清远见-CORTEXA9 资料：\程序源码\Linux 移植实验源码\移植后的源码】下的 rootfs.tar.xz。

14.3.4 实验现象

文件系统挂载成功，说明网卡驱动移植成功:

```
COM6 - PuTTY
[ 1.645000] dm9000 5000000.ethernet eth0: link down
[ 1.690000] mmc0: new high speed SDHC card at address 619b
[ 1.695000] mmcblk1: mmc0:619b SE08G 7.28 GiB
[ 1.700000] mmcblk1: p1
[ 1.905000] dm9000 5000000.ethernet eth0: link up, 100Mbps, full-duplex, lpa 0xCDE1
[ 1.910000] IP-Config: Guessing netmask 255.255.255.0
[ 1.910000] IP-Config: Complete:
[ 1.910000] device=eth0, hwaddr=00:0a:2d:a6:55:a2, ipaddr=192.168.3.8, mask=255.255.255.0, gw=255.255.255.255
[ 1.910000] host=192.168.3.8, domain=, nis-domain=(none)
[ 1.910000] bootserver=255.255.255.255, rootserver=192.168.3.3, rootpath=
[ 1.910000] clk: Not disabling unused clocks
[ 1.930000] usb 1-3: new high-speed USB device number 2 using exynos-ehci
[ 1.960000] VFS: Mounted root (nfs filesystem) on device 0:10.
[ 1.970000] devtmpfs: mounted
[ 1.970000] Freeing unused kernel memory: 228K (c0530000 - c0569000)
[ 2.100000] hub 1-3:1.0: USB hub found
[ 2.105000] hub 1-3:1.0: 3 ports detected
[root@farsight]# ls
bin      etc      linuxrc  proc     sbin     tmp      var
dev      lib      mnt     root     sys      usr
```

14.4 SD 卡驱动移植

14.4.1 实验目的

熟悉基于 Linux 操作系统下的 SD 卡驱动移植过程。



14.4.2 实验平台

华清远见开发环境，FS4412 平台；

14.4.3 实验步骤

运行 Ubuntu 12.04 系统，打开命令行终端。

```
$ cd ~/kernel/ linux-3.14
```

修改设备树文件

```
$ vim arch/arm/boot/dts/exynos4412-fs4412.dts
```

修改

```
sdhci@12530000 {  
    bus-width = <4>;  
    pinctrl-0 = <&sd2_clk &sd2_cmd &sd2_bus4 &sd2_cd>;  
    pinctrl-names = "default";  
    vmmc-supply = <&mmc_reg>;  
    status = "okay";  
};
```

为:

```
sdhci@12530000 {  
    bus-width = <4>;  
    pinctrl-0 = <&sd2_clk &sd2_cmd &sd2_bus4>;  
    cd-gpios = <&gpx0 7 0>;  
    cd-inverted = <0>;  
    pinctrl-names = "default";  
    /*vmmc-supply = <&mmc_reg>;*/  
    status = "okay";  
};
```

配置内核

```
$ make menuconfig
```

Device Drivers --->

<*> MMC/SD/SDIO card support --->

<*> Secure Digital Host Controller Interface support

<*> SDHCI support on Samsung S3C SoC

File systems --->

DOS/FAT/NT Filesystems --->

<*> MSDOS fs support

<*> VFAT (Windows-95) fs support



```
(437) Default codepage for FAT
(iso8859-1) Default iocharset for FAT

-*- Native language support --->

<*>  Codepage 437 (United States, Canada)
<*>  Simplified Chinese charset (CP936, GB2312)
<*>  ASCII (United States)
<*>  NLS ISO 8859-1  (Latin 1; Western European Languages)
<*>  NLS UTF-8
```

编译内核和设备树

```
$ make uImage
$ make dtbs
$ cp arch/arm/boot/uImage /tftpboot
$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/
```

14.4.4 实验现象

启动开发板，通过 NFS 方式挂载根文件系统。

把 SD 卡插到 FS4412 的 SD 卡槽里，在串口终端可以看到如下图所示：

```
[ 1.620000] mmc0: new high speed SDHC card at address cd6d
[ 1.625000] mmcblk1: mmc0:cd6d SE08G 7.28 GiB
[ 1.630000] mmcblk1: p1(mmcblk1 为设备名 p1 为分区名)
```

挂载，注意不要挂在 eMMC 的分区

```
# mount -t vfat /dev/mmcblk1p1 /mnt
```

查看/mnt/目录即可看到 sd 卡中内容

14.5 USB 驱动移植

14.5.1 实验目的

熟悉 Linux 操作系统上 USB 驱动的移植过程。

14.5.2 实验平台

华清远见开发环境，FS4412 平台；

14.5.3 实验步骤

运行 Ubuntu 12.04 系统，打开命令行终端。

```
$ cd ~/kernel/ linux-3.14
```

修改设备树文件



```
$ vim arch/arm/boot/dts/exynos4412-fs4412.dts
```

添加如下内容:

```
usbphy: usbphy@125B0000 {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "samsung,exynos4x12-usb2phy";
    reg = <0x125B0000 0x100>;
    ranges;

    clocks = <&clock 2>, <&clock 305>;
    clock-names = "xusbxti", "otg";

    usbphy-sys {
        reg = <0x10020704 0x8 0x1001021c 0x4>;
    };
};

ehci@12580000 {
    status = "okay";
    usbphy = <&usbphy>;
};

usb3503@08 {
    compatible = "smc,usb3503";
    reg = <0x08 0x4>;
    connect-gpios = <&gpm3 3 1>;
    intn-gpios = <&gpx2 3 1>;
    reset-gpios = <&gpm2 4 1>;
    initial-mode = <1>;
};
```

配置内核

```
make menuconfig
```

```
Device Drivers --->
```

```
[*] USB support --->
```

```
<*>    EHCI HCD (USB 2.0) support
```

```
<*>    EHCI support for Samsung S5P/EXYNOS SoC Series
```



```
<*> USB Mass Storage support
<*> USB3503 HSIC to USB20 Driver
USB Physical Layer drivers --->
<*> Samsung USB 2.0 PHY controller Driver
SCSI device support --->
<*> SCSI device support
<*> SCSI disk support
<*> SCSI generic support
```

编译内核和设备树

```
$ make uImage
```

```
$ make dtbs
```

拷贝内核和设备树文件到/tftpboot 目录下

```
$ cp arch/arm/boot/uImage /tftpboot
```

```
$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/
```

实验现象

启动开发板，通过 NFS 方式挂载根文件系统。

插入 U 盘显示如下

```
[ 72.695000] usb 1-3.2: USB disconnect, device number 3
[ 74.435000] usb 1-3.2: new high-speed USB device number 4 using exynos-ehci
[ 74.555000] usb-storage 1-3.2:1.0: USB Mass Storage device detected
[ 74.560000] scsi1 : usb-storage 1-3.2:1.0
[ 75.645000] scsi 1:0:0:0: Direct-Access Kingston DataTraveler 160 PMAP PQ: 0 ANSI: 4
[ 75.660000] sd 1:0:0:0: Attached scsi generic sg0 type 0
[ 76.695000] sd 1:0:0:0: [sda] 15556608 512-byte logical blocks: (7.96 GB/7.41 GiB)
[ 76.700000] sd 1:0:0:0: [sda] Write Protect is off
[ 76.705000] sd 1:0:0:0: [sda] No Caching mode page found
[ 76.710000] sd 1:0:0:0: [sda] Assuming drive cache: write through
[ 76.725000] sd 1:0:0:0: [sda] No Caching mode page found
[ 76.730000] sd 1:0:0:0: [sda] Assuming drive cache: write through
[ 76.760000] sda: sda1 (sda是设备名 sda1是分区名)
[ 76.770000] sd 1:0:0:0: [sda] No Caching mode page found
[ 76.770000] sd 1:0:0:0: [sda] Assuming drive cache: write through
[ 76.780000] sd 1:0:0:0: [sda] Attached SCSI removable disk
```



在终端上执行挂载的设备与上边显示相关

```
# mount -t vfat /dev/sda1 /mnt  
# ls
```

可以查看到 U 盘内容，即完成实验。由于内核中 USB 控制器驱动并不是很完善，所以会出现挑盘现象，这里并不涉及驱动优化，建议使用实验箱配备的 sd 卡读卡器及 sd 卡配套测试。

华清远见
dev.hqyj.com



14.6 LCD 驱动移植

14.6.1 实验目的

熟悉 Linux 操作系统上 LCD 驱动的移植过程。

14.6.2 实验平台

华清远见开发环境，FS4412 平台；

14.6.3 实验步骤

Exynos4412 内部集成了一个显示控制器 FIMD (Fully Interactive Mobile Display, 完全交互式移动显示设备)。该控制器支持三种接口, 分别是 RGB 接口、indirect-i80 接口和 YUV 接口。在 FS4412 平台上使用的是 RGB 接口连接外部的 LCD 屏。相关的电路原理图如图 9.8 和图 9.9 所示, 图中省略了中间的缓冲器。

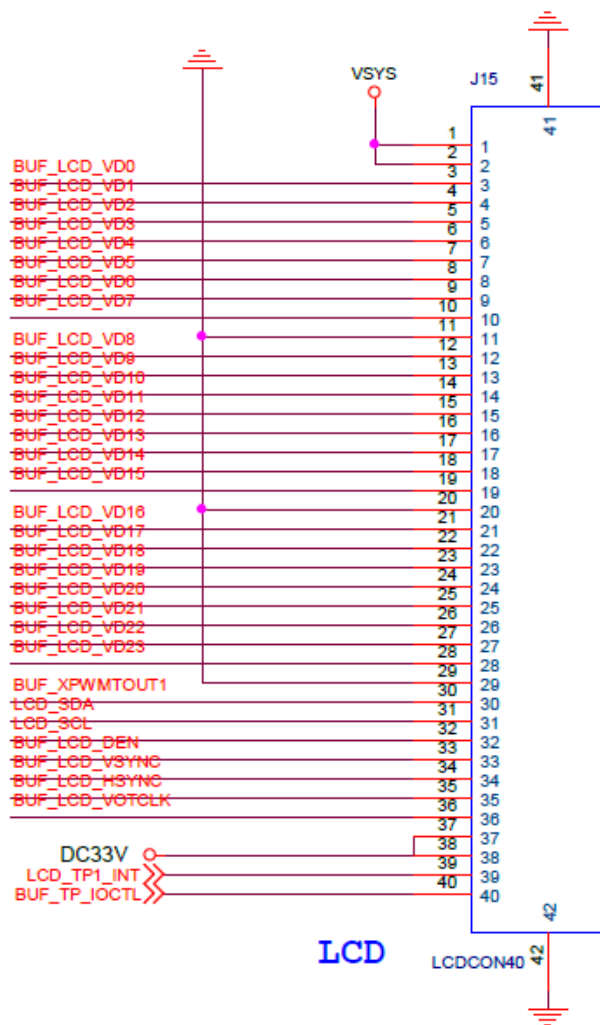


图 FS4412 LCD 接口电路图 (LCD 侧)

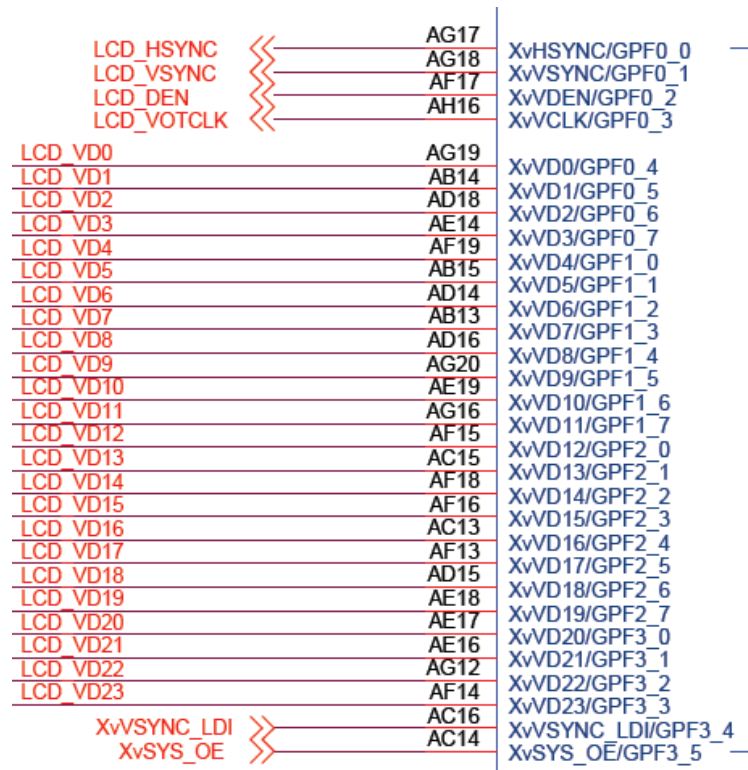


图 FS4412 LCD 接口电路图（CPU 侧）

由前面两幅图可知，LCD 是 24 位 RGB 模式，由 Exynos4412 的 PWM1 的输出来控制 LCD 屏的背光。

在了解了 LCD 部分的原理图后，接下来就是要选配相应的驱动。使用 menuconfig 打开配置界面，选择如下的选项。

```
Device Drivers --->
Graphics support --->
  <*> Direct Rendering Manager (XFree86 4.1.0 and higher DRI support) ----
  <*> DRM Support for Samsung SoC EXYNOS Series
    [*] Exynos DRM FIMD
    [*] Bootup logo -->
      --- Bootup logo
        [*] Standard black and white Linux logo
        [*] Standard 16-color Linux logo
        [*] Standard 224-color Linux logo
```

然后根据 Documentation/devicetree/bindings/video/samsung-fimd.txt 和 Documentation/devicetree/binding s/video/display-timing.txt 文档来修改 arch/arm/boot/dts/exynos4412-fs4412.dts 设备树源文件中关于 FIMD 设备节点和显示时序的描述，具体是将下面的内容：

```
fimd@11c00000 {
    pinctrl-0 = <&lcd_clk &lcd_data24 &pwm1_out>;
    pinctrl-names = "default";
    status = "okay";
```



```
};

display-timings {
    native-mode = <&timing0>;
    timing0: timing {
        clock-frequency = <47500000>;
        hactive = <1024>;
        vactive = <600>;
        hfront-porch = <64>;
        hback-porch = <16>;
        hsync-len = <48>;
        vback-porch = <64>;
        vfront-porch = <16>;
        vsync-len = <3>;
    };
};
```

改为:

```
fimd: fimd@11c00000 {
    pinctrl-0 = <&lcd_clk &lcd_data24 &pwm1_out>;
    pinctrl-names = "default";
    status = "okay";

    display-timings {
        native-mode = <&timing0>;
        timing0: timing {
            hsync-active = <0>;
            vsync-active = <0>;
            de-active = <0>;
            pixelclk-active = <1>;

            clock-frequency = <51206400>;
            hactive = <1024>;
            vactive = <600>;
            hfront-porch = <150>;
            hback-porch = <160>;
            hsync-len = <10>;
            vback-porch = <22>;
            vfront-porch = <12>;
            vsync-len = <1>;
        };
    };
};
```

在上面的修改中，首先是将 `display-timings` 节点移到了 `fimd` 节点中，如果不这样做的话，内核在启



动过程中将会打印未找到时序节点的错误信息。第二个改动就是添加了行、场同步信号，数据使能信号和像素时钟信号的极性属性。其属性值根据 Documentation/devicetree/bindings/video/display-timing.txt 文档中的描述可知，0 表示低电平有效，1 表示高电平有效。这些属性值需要对照具体使用的 LCD 屏的时序图来设定。最后修改了时序参数，这些参数完全根据 LCD 的数据手册来设定，FS4412 使用的 LCD 屏的数据手册上关于时序方面的内容如表 9.1 所示。

表 9.1 LCD 屏时序

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Clock Frequency	fclk	40.8	51.2	67.2	MHz	Frame rate=60Hz
Horizontal display area	thd	1024			DCLK	
HS period time	th	1114	1344	1400	DCLK	
HS Blanking	thb	90	320	376	DCLK	
Vertical display area	tvd	600			H	
VS period time	tv	610	635	800	H	
VS Blanking	tvb	10	35	200	H	

上表表示该屏的水平分辨率为 1024，垂直分辨率为 600，典型的 hfront-porch、hback-porch 和 hsync-len 的和为 320，典型的 vback-porch、vfront-porch 和 vsync-len 的值为 35。上述属性的具体含义请参考 Documentation/devicetree/bindings/video/display-timing.txt 文档。现在的问题是手册只给出了和值，并未给出各个参数具体的值，那么只有不停修改这些属性值，直到观察到 LCD 屏的边缘没有黑边，没有图片显示不全的情况位置。所以这需要在 LCD 驱动移植成功够，编写测试代码显示一张测试图片来确定。需要注意的是，在修改这些属性值时，其和值保持不变。对于 clock-frequency 这个属性，指的是像素时钟的频率，对于该值的计算可以使用下面的公式：

$$(\text{hactive} + \text{hfront-porch} + \text{hback-porch} + \text{hsync-len}) \times (\text{vactive} + \text{vback-porch} + \text{vfront-porch} + \text{vsync-len})$$

将内核和设备树重新编译后重新启动内核，并没有看到 LCD 屏上显示的企鹅图标。但是在/dev 目录下有了 fb0 这个设备节点。通过测试程序操作这个设备发现一切正常，只是没有显示。根据以上现象基本可以断定 LCD 的驱动移植是成功的，可能是有些配置还没有完全正确。对于这类现象首先应该怀疑的就是背光是否没有，导致屏幕没有显示。但是仔细观察 LCD 屏会发现，LCD 屏的四周有白色的背光，说明背光是有的。接下来就应该怀疑控制器的输出是否使能，这需要使用一些调试手段来查看 LCD 的寄存器（相关内容请见下一章）。通过调试发现，LCD 模块寄存器的使能位都是打开的，但是在 Exynos4412 的系统控制器中有个关于 FIMD 选择的寄存器 LCDBLK_CFG 的比特 1 设置不对。修改驱动代码，设置该位后，发现屏幕终于有了输出，但是明显感觉到刷新频率很慢。对于这种现象很容易想到是时钟配置方



面的问题，所以再查看关于 LCD 时钟配置相关的寄存器，发现确实有问题，所以针对这个结论编写了如下的两个文件（exynos_drm_fbclk.h 和 exynos_drm_fbclk.c）的代码。

```
/* drivers/gpu/drm/exynos/exynos_drm_fbclk.h */
```

```
#ifndef _EXYNOS_DRM_FBCLK_H_
```

```
#define _EXYNOS_DRM_FBCLK_H_
```

```
void exynos_drm_fbclk_preinit(void);
```

```
#endif
```

```
/* drivers/gpu/drm/exynos/exynos_drm_fbclk.c */
```

```
#include <linux/io.h>
```

```
#include <linux/ioport.h>
```

```
#include <mach/map.h>
```

```
#define CLK_SRC_LCD0 (S5P_VA_CMU + 0xC234)
```

```
#define CLK_SRC_MASK_LCD (S5P_VA_CMU + 0xC334)
```

```
#define CLK_DIV_LCD (S5P_VA_CMU + 0xC534)
```

```
#define CLK_DIV_STAT_LCD (S5P_VA_CMU + 0xC634)
```

```
#define CLK_GATE_IP_LCD (S5P_VA_CMU + 0xC934)
```

```
#define CLK_GATE_BLOCK (S5P_VA_CMU + 0xC970)
```

```
#define LCDBLK_CFG (S3C_VA_SYS + 0x0210)
```

```
void exynos_drm_fbclk_preinit(void)
```

```
{
```

```
/* FIMD0_SEL = SCLKVPLL, 350000000 */
```

```
__raw_writel((__raw_readl(CLK_SRC_LCD0) & ~(0x0F)) | (0x08), CLK_SRC_LCD0);
```

```
/* FIMD0_RATIO = 3, SCLK_FIMD0 = MOUTFIMD0/(FIMD0_RATIO + 1) */
```

```
__raw_writel((__raw_readl(CLK_DIV_LCD) & ~(0x0F)) | (0x04), CLK_DIV_LCD);
```

```
/* unmask output clock of MUXFIMD0 */
```

```
__raw_writel(__raw_readl(CLK_SRC_MASK_LCD) | 0x1, CLK_SRC_MASK_LCD);
```

```
__raw_writel(__raw_readl(CLK_GATE_IP_LCD) | 0x1, CLK_GATE_IP_LCD);
```

```
__raw_writel(__raw_readl(CLK_GATE_BLOCK) | (0x1 << 4), CLK_GATE_BLOCK);
```

```
/* select fimd */
```

```
__raw_writel(__raw_readl(LCDBLK_CFG) | (0x1 << 1), LCDBLK_CFG);
```

```
}
```

在上面的代码中首先设置了 LCD 的时钟源为 SCLKVPLL，然后设置的分频值和时钟使能位，最后选



择了 FIMD。将上面两个文件拷贝到 `drivers/gpu/drm/exynos/` 目录下并修改该目录下的 `Makefile` 文件，添加 `exynos_drm_fbclk.o`。

```
exynosdrm-$(CONFIG_DRM_EXYNOS_FIMD)+= exynos_drm_fimd.o exynos_drm_fbclk.o
```

最后修改 `drivers/gpu/drm/exynos/exynos_drm_drv.c`，在 `exynos_drm_fbdev_init(dev)` 函数调用前添加 `exynos_drm_fbclk_preinit()` 函数调用（见下面的斜体字部分）。

```
/*
 * create and configure fb helper and also exynos specific
 * fbdev object.
 */
#include <exynos_drm_fbclk.h>
exynos_drm_fbclk_preinit();
ret = exynos_drm_fbdev_init(dev);
if (ret) {
    DRM_ERROR("failed to initialize drm fbdev\n");
    goto err_drm_device;
}
```

14.6.4 实验现象

经过修改之后，再次编译内核，发现在系统启动后正常显示了企鹅图标，运行测试程序，图片显示不闪烁，说明 LCD 驱动移植成功。



第 15 章 Linux 文件系统构建实验

15.1 实验原理

文件是计算机系统的软件资源，操作系统本身和大量的用户程序、数据都是以文件形式组织和存放的，对这些资源的有效管理和充分利用是操作系统的重要任务之一。本章介绍文件和文件系统的概念，并对操作系统的文件管理功能给予简要说明。

15.1.1 磁盘的物理组织

文件系统主要完成对磁盘上的数据和程序进行管理，所以首先需要了解磁盘（硬盘）的结构。

磁盘又叫硬盘，由一组盘片组成。每个盘片的上下两面都涂有磁粉，磁化后可以存储信息数据。每个盘片的上下两面都安装有磁头。磁头被安装在梳状的可以做直线运动的小车上以便寻道。每个盘面被格式化成为若干条同心圆的磁道，并规定最外面的磁道是 0 磁道，次外层是 1 磁道。每个磁道又被分成若干个扇区，并顺序排为 1 号扇区、2 号扇区，等等。通常，一个扇区可以储存 512 字节的二进制信息位。这也是 CPU 进行磁盘 I/O 操作时能够读出和写入的最小单位。每个盘面上的同号磁道组成一个柱面，也就是说每个盘面的 0 号磁道组成 0 号柱面，所有的 1 号磁道组成 1 号柱面，等等。硬盘的结构如图 8.1 所示。

磁盘的结构

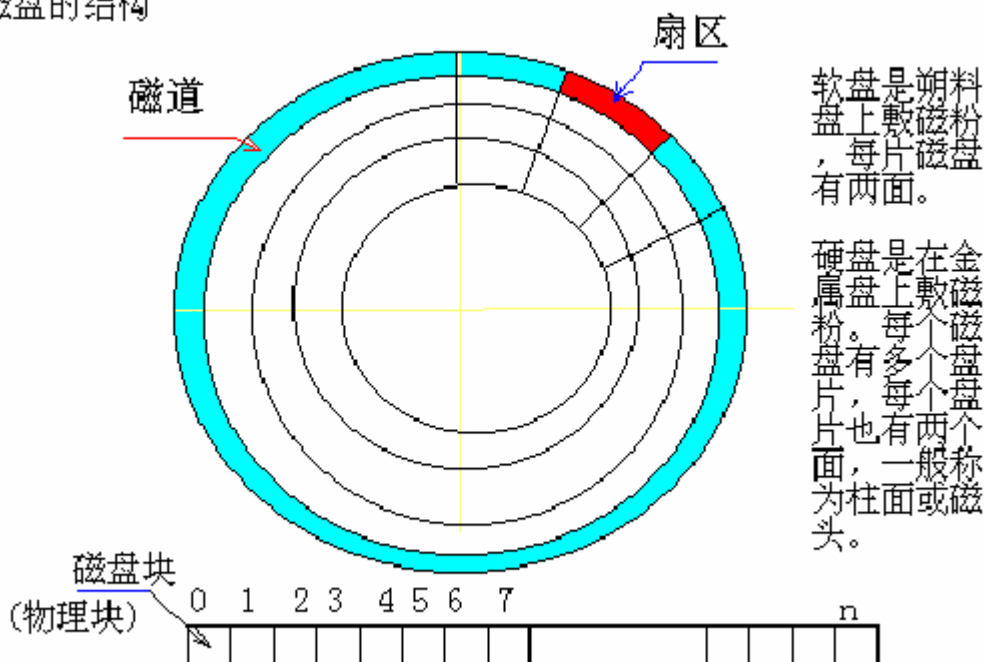


图 8.1 硬盘结构示意图

在 Linux 操作系统中，内核采用的方法是把物理磁盘抽象为逻辑磁盘来管理文件系统。所谓逻辑磁盘是把物理磁盘按照磁头号、磁道号、扇区号以及盘面号划分成磁盘块的线性数组，也叫线性序列。例如：把 1 号盘面 0 号磁道的 0 号扇区定义为 0 号磁盘块；把 1 号盘面 0 号磁道的 1 号扇区定义为 1 号磁盘块等等；当然一个磁盘块可以包含多个扇区，一般扇区数是 2 的整次幂。显然，当把实际的磁盘看成



是磁盘块的线性数组时，就把物理磁盘存储数据的实际地址（即磁道号、扇区号以及盘面号）隐藏起来。因此呈现在系统高层面前的已经不是物理磁盘，而是一个经过加工以后的逻辑磁盘。图 8.2 给出了逻辑磁盘的结构示意图，它比物理硬盘的结构要简单的多。当系统执行磁盘 I/O 操作时，系统给出试图访问的逻辑磁盘块号。由设备驱动程序根据该块号计算出物理磁盘的磁道号、磁头号以及扇区号，然后启动硬盘把磁头向前或向后移动到相应的柱面，这就是所谓的寻道。寻道是磁盘 I/O 操作中最为耗时的一个操作。一旦磁头找到磁道，并且相应的扇区转到磁头下面，数据传输就开始。

15.1.2 文件和目录

文件是一个具有符号的一组相关联元素的有序序列。文件可以包含范围非常广泛的内容。系统和用户都可以将具有一定独立功能的程序模块、一组数据或一组文字命名为一个文件。在电脑里看见的东西都叫文件。文件是以单个名称在计算机上存储的信息集合。文件可以是文本文档、图片、程序等等。文件通常具有三个字母的文件扩展名，用于指示文件类型。

文件有很多种，运行的方式也各有不同。一般来说我们可以通过文件名来识别这个文件是哪种类型，特定的文件都会有特定的图标，也只有安装了相应的软件，才能正确显示这个文件的图标。

15.1.3 文件的分类

在计算机系统中存放着大量的文件，它们有着不同的内容、用途和形式。

为了便于对文件进行在理合加工，通常把众多的文件从不同的角度进行分类。下面介绍几种经常使用的文件分类方法。

按文件的创建角度，文件分为：

- **系统文件**：即由操作系统创建的文件。这些文件包含着操作系统执行的程序和处理的数据。系统文件仅供操作系统使用，不对用户开放。
- **用户文件**：即由用户创建的文件。这些文件包含的是用户的信息，如用户的程序、数据和其他各种形式的信息。
- **库文件**：即由系统创建、供系统和用户使用的文件。它们是一些由标准函数或子程序及常用的应用程序组成的文件。库文件允许用户调用，但是不允许用户修改。在某些系统中，允许用户通过系统向库文件中添加信息。

按文件的读取权限，文件可以分为：

- **只读文件**：只允许对文件进行读操作，而不允许写操作的文件。
- **读写文件**：指既可以进行读操作，又可以写操作的文件。
- **可执行文件**：指只可以调入到内存中执行，而不能对它们进行读写操作的文件。
- **不保护文件**：这种文件不作任何保护，所有用户都可以使用。

按文件在系统中的信息流向，文件分为：

- (1) **输入文件**：这种文件只能从输入设备中读入到内存，如读卡器上的文件。
- (2) **输出文件**：这种文件只能从系统写入到输出设备中，如打印机上的文件。
- (3) **输入/输出文件**：这种文件既可以从输入设备中读取，又可以向输出设备写入，如



磁盘上的文件。

按文件信息的逻辑结构，文件分为：

- **流文件**：以字符为基本单位，按照一定顺序组成的文件。文件内的信息就是一连串的字符，不再划分结构。它使用结束符来标志文件的结束。
- **记录文件**：把文件内的信息划分成多个记录，记录是文件组织和操作的基本单位。记录文件可以分为下面两种：
 - a) **文本文件**：即由字符代码组成的文件。文本文件可以直接显示在屏幕上，或者在打印机上打印，也可以使用编辑器进行编辑。
 - b) **二进制文件**：即由二进制数据组成的文件，如可执行程序、图像文件、声音文件等。二进制文件不能直接显示或打印。

15.1.4 目录

目录是一类特殊的文件，利用它可以构成文件系统的分层树型结构。如同普通文件那样，目录文件也包含数据；但目录文件与普通文件的差别是，核心对这些数据加以结构化，它是由成对的“i 节点号/文件名”构成的列表。

- i 节点号是检索 I 节点表的下标，i 节点中存放有文件的状态信息。
- 文件名是给一个文件分配的文本形式的字符串，用来标识该文件。在一个指定的目录中，任何两项都不能有同样的名字。

每个目录的第一项都表示目录本身，并以“.”作为它的文件名。每个目录的第二项的名字是“..”，表示该目录的父目录。当把文件添加到一个目录中的时候，该目录的大小会增长，以便容纳新文件名。当删除文件时，目录的尺寸并不减少，而是核心对该目录项做上特殊标记，以便下次添加一个文件时重新使用它。

Linux 文件系统采用带链接的树形目录结构，即只有一个根目录（通常用“/”表示），其中含有下级子目录或文件的信息；子目录中又可含有更下级的子目录或者文件的信息。这样一层一层地延伸下去，构成一棵倒置的树，如图 8.2 所示。

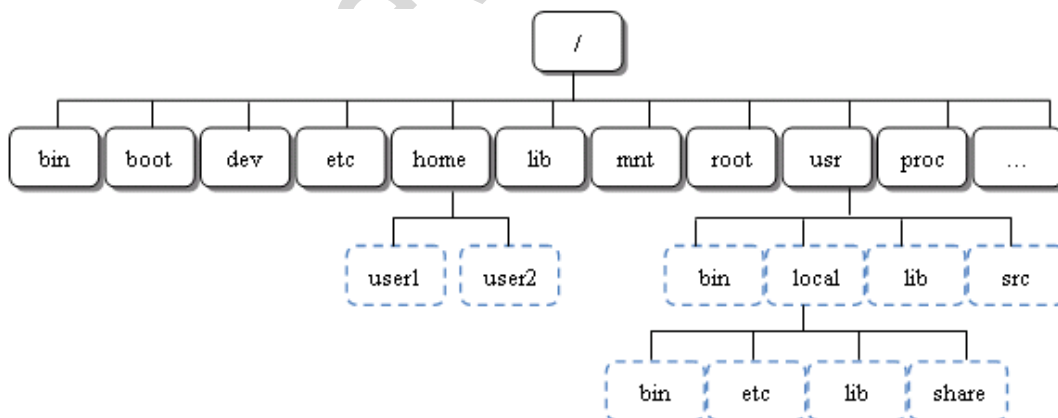


图 8.2 LINUX 系统目录

在目录树中，根节点和中间节点都必须是目录，而普通文件和特别文件只能作为“叶子”出现。



当然，目录也可以作为叶子。

15.1.5 文件系统

文件系统指文件存在的物理空间。在 Linux 系统中，每个分区都是一个文件系统，都有自己的目录层次结构。Linux 的最重要特征之一就是支持多种文件系统，这样它更加灵活，并可以和许多其它种操作系统共存。由于系统已将 Linux 文件系统的所有细节进行了转换，所以 Linux 核心的其它部分及系统中运行的程序将看到统一的文件系统。

大部分 UNIX 文件系统种类具有类似的通用结构，即使细节有些变化。其中心概念是超级块 superblock, i 节点 inode, 数据块 data block, 目录块 directory block, 和间接块 indirection block。超级块包括文件系统的总体信息，比如大小(其准确信息依赖文件系统)。i 节点包括除了名字外的一个文件的所有信息，名字与 i 节点数目一起存在目录中，目录条目包括文件名和文件的 i 节点数目。i 节点包括几个数据块的数目，用于存储文件的数据。i 节点中只有少量数据块数的空间，如果需要更多，会动态分配指向数据块的指针空间。这些动态分配的块是间接块；为了找到数据块，这名字指出它必须先找到间接块的号码。

Linux 支持多种操作系统，其实现机制我们在后面会讲到。

15.1.6 虚拟文件系统

Linux 系统允许众多不同种类的文件系统共存，如 ext3、vfat 等。通过使用同一套文件 I/O 系统调用，即可对 Linux 中的任意文件进行操作，而无需考虑其所在的文件系统的具体格式。此外，Linux 还支持跨文件系统的文件操作，即对文件的操作可以跨文件系统地进行。如图 8.3 所示。实现这种操作的机制正是虚拟文件系统。

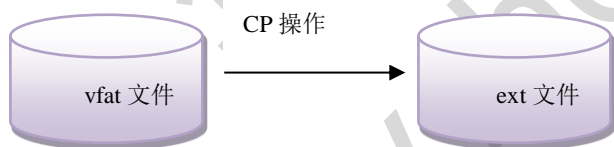


图 8.3 跨文件系统操作

15.1.7 虚拟文件系统概述

Linux 文件系统分为三个部分，第一部分是 Virtual File System Switch (VFS)，它是 Linux 文件系统对外的接口，任何要使用文件系统的程序都必须经由这层接口来使用它。另外两部分是属于文件系统的内部实现，分别是 cache 和真正的文件系统（例如 Ext3，VFAT 等）。

虚拟文件系统是 Linux 内核中的一个软件抽象层，它一方面用于给用户空间的程序提供文件系统接口，另一方面还提供了内核中的一个抽象功能，它通过一些数据结构及其方法向实际的文件系统提供接口，实现不同文件系统在 Linux 中共存。系统中所有文件系统不但依赖于 VFS 共存，同时也要依靠 VFS 协同工作。

为了能支持各种文件系统，VFS 定义了所有文件系统都必须支持基本的、概念上的接口和数据结构，例如超级块、节点、文件操作函数入口等。换句话说，一个实际的文件系统要想被 Linux 支持，就必须提供一个符合 VFS 标准的接口，这才能与 VFS 协同工作。VFS 不是实际的操作系统，它只是一种转



换机制，仅存在于内存中，不存在于任何外存空间。图 8.4 显示了 VFS 在内核中与实际的文件系统的协同关系。

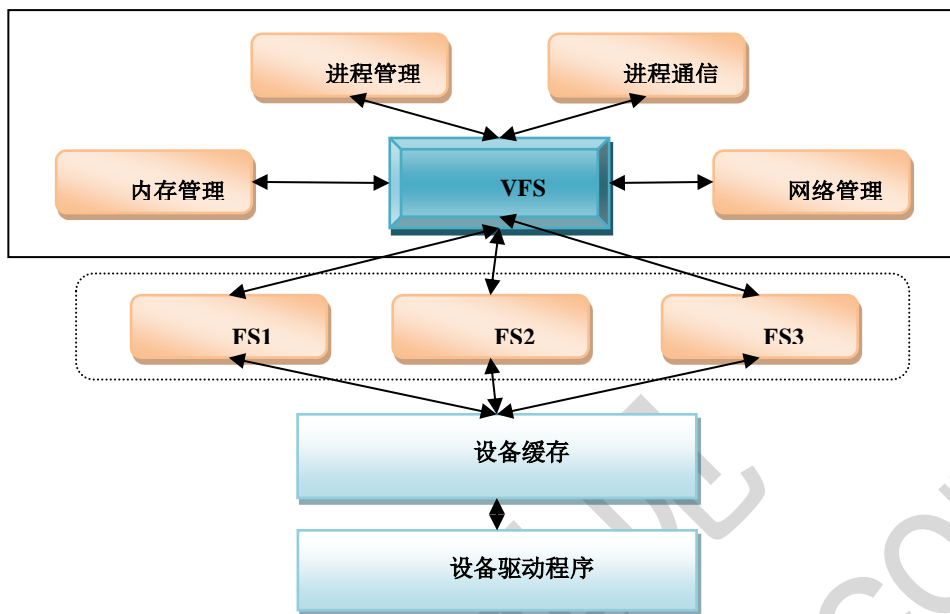


图 8.4 VFS 在内核中与实际文件系统的关系图

概述的说，VFS 主要提供以下一些功能：

- 记录可用的文件系统类型；
- 将设备与对应的文件系统相联系；
- 处理一些面向文件的通用操作；
- 涉及针对文件系统的操作时，VFS 把它们映射到与控制文件、目录以及 inode 相关的物理文件系统。

15.2 根文件系统开发实验

15.2.1 实验目的

熟悉 Linux 文件系统目录结构，创建自己的文件系统，通过 NFS 方式测试。

15.2.2 实验平台

华清远见开发环境，FS4412 平台；

15.2.3 实验步骤

1. 根文件系统制作

可以从 <http://busybox.net/downloads/> 网站下载 busybox 源码用于制作 Linux 文件系统，为了方便，已将源码放进了光盘。

运行 Ubuntu 12.04 系统，打开命令行终端：



```
$ cd ~
```

```
$ mkdir busybox
```

```
$ cd busybox
```

将【华清远见-CORTEXA9 资料：\程序源码\Linux 移植实验源码\文件系统移植” 目录下的“busybox-1.17.3.tar.bz2】拷贝至该目录。

```
$ tar xvf busybox-1.22.1.tar.bz2 //解压源码
```

```
$ cd busybox-1.22.1
```

配置 busybox 源码：

```
$ make menuconfig
```

Busybox Settings --->

Build Options --->

- ☒ [*] Build BusyBox as a static binary (no shared libs)
- ☐ [] Force NOMMU build
- ☐ [] Build with Large File Support (for accessing files > 2 GB)
- (arm-none-linux-gnueabi-) Cross Compiler prefix
- ☐ () Additional CFLAGS

编译源码：

```
$ make
```

安装：

busybox 默认安装路径为源码目录下的_install

```
$ make install
```

进入安装目录：

```
$ cd _install
```

```
$ ls
```

bin linuxrc sbin usr

创建其他需要的目录：

```
$ mkdir dev etc mnt proc var tmp sys root
```

添加库：

将工具链中的库拷贝到_install 目录下：

```
$ cp /usr/local/toolchain/ toolchain-4.6.4/arm-arm1176jzfssf-linux-gnueabi/lib / . -a
```

删除静态库和共享库文件中的符号表：

```
$ sudo rm lib/*.a
```

```
# arm-none-linux-gnueabi-strip lib/*.so （超级用户下执行命令）
```

删除不需要的库，确保所有库大小不超过 4M：

```
$ du -mh lib/
```



添加系统启动文件:

在 etc 下添加文件 inittab, 文件内容如下:

```
#this is run first except when booting in single-user mode.

::sysinit:/etc/init.d/rcS

# /bin/sh invocations on selected ttys

# start an "askfirst" shell on the console (whatever that may be)

::askfirst:/bin/sh

# stuff to do when restarting the init process

::restart:/sbin/init

# stuff to do before rebooting

::ctrlaltdel:/sbin/reboot
```

在 etc 下添加文件 fstab , 文件内容如下:

#device	mount-point	type	options	dump	fsck order
proc	/proc	proc	defaults	0	0
tmpfs	/tmp	tmpfs	defaults	0	0
sysfs	/sys	sysfs	defaults	0	0
tmpfs	/dev	tmpfs	defaults	0	0

这里我们挂载的文件系统有三个 proc、sysfs 和 tmpfs 。在内核中 proc 和 sysfs 默认都支持, 而 tmpfs 是没有支持的, 我们需要添加 tmpfs 的支持

修改 Linux 内核配置:

```
$ cd ~/kernel/linux-3.14
$ make menuconfig
File systems --->
    Pseudo filesystems --->
        [*] Tmpfs Virtual memory file system support (former shm fs)
        [*] Tmpfs POSIX Access Control Lists
```

重新编译内核:

```
$ make uImage
$ cp arch/arm/boot/uImage /tftpboot
```

回到创建的文件系统处, 在 etc 下创建 init.d 目录, 并在 init.d 下创建 rcS 文件, rcS 文件内容为:

```
#!/bin/sh

# This is the first script called by init process

/bin/mount -a
```



```
echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev -s
```

为 rcS 添加可执行权限:

```
$ chmod +x init.d/rcS
```

在 etc 下添加 profile 文件, 文件内容为:

```
#!/bin/sh
export HOSTNAME=farsight
export USER=root
export HOME=root
export PS1="[$USER@$HOSTNAME \W]\# "
PATH=/bin:/sbin:/usr/bin:/usr/sbin
LD_LIBRARY_PATH=/lib:/usr/lib:$LD_LIBRARY_PATH
export PATH LD_LIBRARY_PATH
```

重要: 新制作的文件系统尺寸若超出 8M, 删除不需要的库文件

2. NFS 测试

删除原先的/source/rootfs:

```
$ sudo rm -rf /source/rootfs
```

将我们新建的根文件系统拷贝到/source/rootfs 目录下

```
$sudo mkdir /source/rootfs
```

```
$ sudo cp _install/* /source/rootfs -a
```

15.2.4 实验现象

按照 3.4.10 节设置 uboot 环境变量:

重新启动开发板, 查看是否能够正常挂载, 功能是否正常



```
COM6 - PuTTY
[ 1.670000] dm9000 5000000.ethernet eth0: link down
[ 1.710000] mmc0: new high speed SDHC card at address 619b
[ 1.715000] mmcblk1: mmc0:619b SE08G 7.28 GiB
[ 1.720000]   mmcblk1: p1
[ 1.920000] dm9000 5000000.ethernet eth0: link up, 100Mbps, full-duplex, lpa
0xCDE1
[ 1.925000] IP-Config: Guessing netmask 255.255.255.0
[ 1.925000] IP-Config: Complete:
[ 1.925000]   device=eth0, hwaddr=00:0a:2d:a6:55:a2, ipaddr=192.168.3.8, m
ask=255.255.255.0, gw=255.255.255.255
[ 1.925000]   host=192.168.3.8, domain=, nis-domain=(none)
[ 1.925000]   bootserver=255.255.255.255, rootserver=192.168.3.3, rootpath
=
[ 1.925000] clk: Not disabling unused clocks
[ 1.950000] usb 1-3: new high-speed USB device number 2 using exynos-ehci
[ 1.980000] VFS: Mounted root (nfs filesystem) on device 0:10.
[ 1.985000] devtmpfs: mounted
[ 1.990000] Freeing unused kernel memory: 228K (c0530000 - c0569000)
[ 2.120000] hub 1-3:1.0: USB hub found
[ 2.125000] hub 1-3:1.0: 3 ports detected
[root@farsight]# ls
bin      etc      linuxrc  proc     sbin     tmp      var
dev      lib      mnt      root     sys      usr
[root@farsight]#
```

15.3 Ramdisk 文件系统制作实验

15.3.1 实验目的

将通过 nfs 测试好的文件系统内容打包成 ramdisk 文件镜像，并烧写到目标平台测试。通过实验掌握 ramdisk 文件系统的特性及镜像制作方法。

15.3.2 实验平台

华清远见开发环境，FS4412 平台；

15.3.3 实验步骤

- 1、制作一个大小为 8M 的镜像文件

```
$ cd ~
```

```
$ dd if=/dev/zero of=ramdisk bs=1k count=8192 (ramdisk 为 8M)
```

- 2、格式化这个镜像文件为 ext2

```
$ mkfs.ext2 -F ramdisk
```

- 3、在 mount 下面创建 initrd 目录作为挂载点

```
$ sudo mkdir /mnt/initrd
```

- 4、将这个磁盘镜像文件挂载到/mnt/initrd 下

注意这里的 ramdisk 不能存放在 rootfs 目录中

```
$ sudo mount -t ext2 -o loop ramdisk /mnt/initrd
```



5、将我们的文件系统复制到 initrd.img 中

将测试好的文件系统里的内容全部拷贝到 /mnt/initrd 目录下面

```
$ sudo cp /source/rootfs/* /mnt/initrd -a
```

6、卸载 initrd

```
$ sudo umount /mnt/initrd
```

7、压缩 initrd.img 为 initrd.img.gz 并拷贝到/tftpboot 下

```
$ gzip --best -c ramdisk > ramdisk.gz
```

8、格式化为 uboot 识别的格式

```
$ mkimage -n "ramdisk" -A arm -O linux -T ramdisk -C gzip -d ramdisk.gz  
ramdisk.img
```

```
$ cp ramdisk.img /tftpboot
```

9、配置内核支持 RAMDISK

制作完 ramdisk.img 后，需要配置内核支持 RAMDISK 作为启动文件系统，修改内核配置

```
$ cd ~/kernel/linux-3.14
```

```
make menuconfig
```

```
File systems --->
```

```
<*> Second extended fs support
```

```
Device Drivers
```

```
SCSI device support --->
```

```
<*> SCSI disk support
```

```
Block devices --->
```

```
<*>RAM block device support
```

```
(16)Default number of RAM disks
```

```
(8192) Default RAM disk size (kbytes) (修改为 8M)
```

```
General setup --->
```

```
[*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
```

重新编译内核，复制到/tftpboot

10、在 U-BOOT 命令行重新设置启动参数：

```
# setenv bootcmd tftp 41000000 uImage;tftp 42000000 exynos4412-fs4412.dtb;tftp 43000000  
ramdisk.img;bootm 41000000 43000000 42000000  
  
# saveenv
```

重新启动开发板查看能否正常启动。



第 16 章 LED 驱动开发实验

16.1 实验原理

如图所示，LED2~LED5 分别与 GPX2_7、GPX1_0、GPF3_4、GPF3_5 相连，通过 GPX2_7、GPX1_0、GPF3_4、GPF3_5 引脚的高低电平来控制三极管的导通性，从而控制 LED 的亮灭。

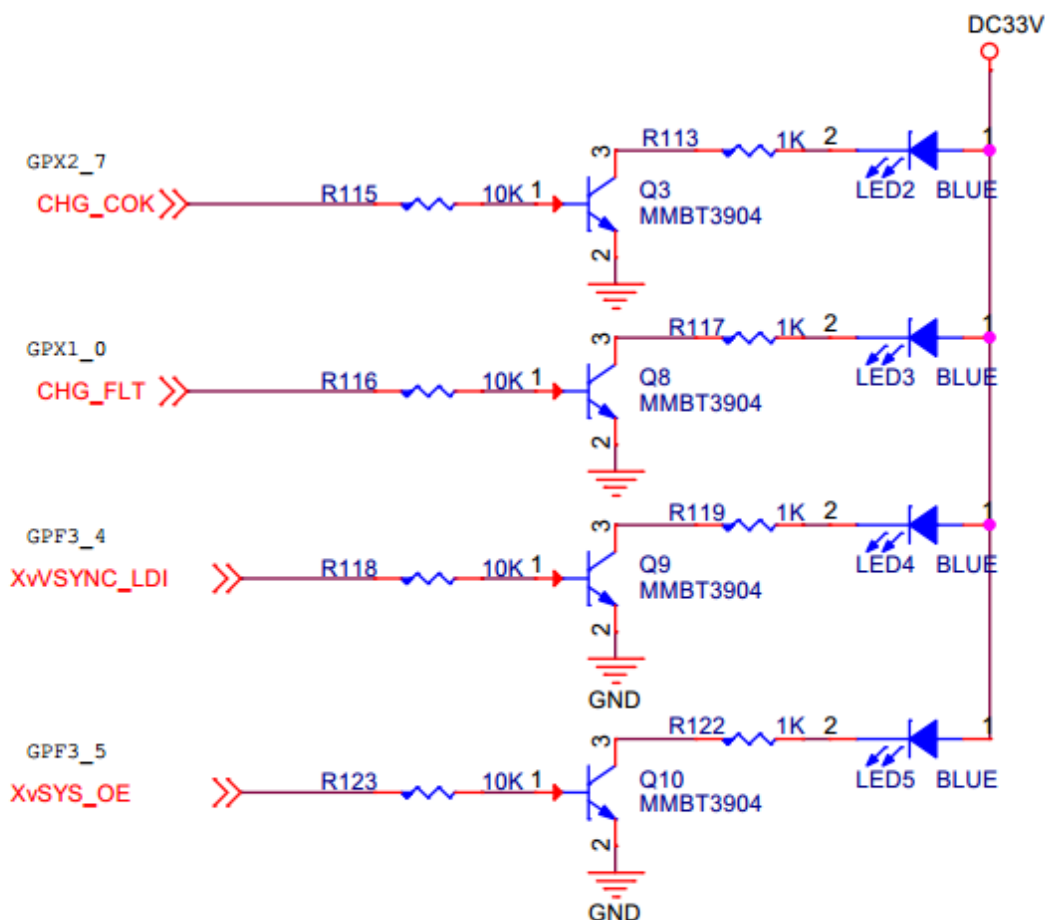


图 LED 接线原理图

因此，当这几个引脚输出高电平时发光二极管点亮；反之，发光二极管熄灭。

16.2 实验目的

示例将利用 Exynos4412 的 GPX2_7、GPX1_0、GPF3_4、GPF3_5 这 4 个 I/O 引脚控制 4 个 LED 发光二极管，使其闪烁。

16.3 实验平台

华清远见开发环境，FS4412 平台



16.4 实验步骤

16.4.1 环境准备

按照 3.4 节的步骤，搭建 TFTP 和 NFS 环境（3.4.10 节的现象为准）。

```
COM6 - PuTTY
[ 1.565000] mmcblk0: p1 p2 p3 p4
[ 1.565000] mmcblk0boot1: unknown partition table
[ 1.570000] mmcblk0boot0: unknown partition table
[ 1.645000] dm9000 5000000.ethernet eth0: link down
[ 1.700000] mmc0: new high speed SDHC card at address 619b
[ 1.705000] mmcblk1: mmc0:619b SE08G 7.28 GiB
[ 1.710000] mmcblk1: p1
[ 1.870000] usb 1-3: new high-speed USB device number 2 using exynos-ehci
[ 1.890000] IP-Config: Guessing netmask 255.255.255.0
[ 1.890000] IP-Config: Complete:
[ 1.895000] dm9000 5000000.ethernet eth0: link up, 100Mbps, full-duplex, lpa
0xCDE1
[ 1.900000] device=eth0, hwaddr=00:0a:2d:a6:55:a2, ipaddr=192.168.3.8, m
ask=255.255.255.0, gw=255.255.255.255
[ 1.910000] host=192.168.3.8, domain=, nis-domain=(none)
[ 1.920000] bootserver=255.255.255.255, rootserver=192.168.3.3, rootpath
=
[ 1.925000] clk: Not disabling unused clocks
[ 1.940000] VFS: Mounted root (nfs filesystem) on device 0:10.
[ 1.945000] devtmpfs: mounted
[ 1.950000] Freeing unused kernel memory: 228K (c0530000 - c0569000)
[ 2.025000] hub 1-3:1.0: USB hub found
[ 2.030000] hub 1-3:1.0: 3 ports detected
[root@farsight ]#
```

16.4.2 代码准备

将【华清远见-CORTEXA9 资料:\程序源码\Linux 移植实验源码\移植后的源码\linux-3.14-fs4412.tar.xz】目录拷贝到共享目录下（此内核为移植好的内核，如果用户做完 Linux 内核移植实验，可以使用自己的内核，按照实际情况修改路径）。

名称	修改日期	类型	大小
linux-3.14-fs4412.tar.xz	2016/1/20 14:45	WinRAR 压缩文件	79,854 KB
rootfs.tar.xz	2014/11/26 13:29	WinRAR 压缩文件	1,545 KB
u-boot-2013.01-fs4412.tar.xz	2014/11/26 13:26	WinRAR 压缩文件	7,624 KB
如用户使用busybox制作的文件系统无法...	2016/9/24 15:46	TXT 文件	0 KB
为方便用户使用，我们已将需要修改的源...	2016/9/24 15:46	TXT 文件	0 KB

将【华清远见-CORTEXA9 资料: \程序源码\Linux 驱动实验源码\设备驱动例程】目录拷贝到共享目录下。

建立工作目录，拷贝源码

```
$ mkdir ~/workdir/driver -p
$ cd ~/workdir/driver/
$ cp /mnt/hgfs/share/linux-3.14-fs4412.tar.xz ./ 说明：也可以用 15 章自己做的内核
$ cp /mnt/hgfs/share/Linux3.14Drivers/ ./ -a
```



解压内核源码

```
$ tar xvf linux-3.14-fs4412.tar.xz
```

编译内核源码

```
$ cd ~/workdir/driver/linux-3.14-fs4412
$ make uImage
```

```
Kernel: arch/arm/boot/Image is ready
Kernel: arch/arm/boot/zImage is ready
UIMAGE arch/arm/boot/uImage
Image Name:   Linux-3.14.0
Created:      Sat Aug 30 12:13:19 2014
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    3027976 Bytes = 2957.01 kB = 2.89 MB
Load Address: 40008000
Entry Point:  40008000
Image arch/arm/boot/uImage is ready
```

16.4.3 编译代码

```
$ cd ~/workdir/driver/Linux3.14Drivers/fs4412_led
```

修改 Makefile 文件第 3、4 行。

```
linux@ubuntu64-vm: ~/workdir/driver/Linux3.14Drivers/fs4412_led
1 ifeq ($(KERNELRELEASE),)
2
3 KERNELDIR ?= /home/linux/linux-3.14-fs4412/
4 #KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5 PWD := $(shell pwd)
6
7 modules:
8     $(MAKE) -C $(KERNELDIR) M=$(PWD)
9
10 modules_install:
11     $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
12
13 clean:
14     rm -rf *.o *~ core .depend *.cmd *.ko *.mod.c .tmp_versions Module* mod
15
16 .PHONY: modules modules_install clean
17
18 else
19     obj-m := fs4412_led.o
20 endif
~
~
Makefile 12,0-1 全部
"Makefile" 20L, 418C
```

修改为用户内核源码的路径和交叉工具链。



```
linux@ubuntu64-vm: ~/workdir/driver/Linux3.14Drivers/fs4412_led
1 ifeq ($(KERNELRELEASE),)
2
3 KERNELDIR ?= /home/linux/workdir/driver/linux-3.14-fs4412/
4 #KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5 PWD := $(shell pwd)
6
7 modules:
8     $(MAKE) -C $(KERNELDIR) M=$(PWD)
9
10 modules_install:
11     $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
12
13 clean:
14     rm -rf *.o *~ core .depend *.cmd *.ko *.mod.c .tmp_versions Module* mod
15
16 .PHONY: modules modules_install clean
17
18 else
19     obj-m := fs4412_led.o
20 endif
~
~
Makefile [+]3,58 全部
```

保存退出。执行 make 命令编译源码。

```
$ make
```

```
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_led$ make
make -C /home/linux/workdir/driver/linux-3.14-fs4412/ M=/home/linux/workdir/driver/Linux3.14Drivers/fs4412_led
make[1]: 正在进入目录 `/home/linux/workdir/driver/linux-3.14-fs4412'
LD      /home/linux/workdir/driver/Linux3.14Drivers/fs4412_led/built-in.o
CC [M]  /home/linux/workdir/driver/Linux3.14Drivers/fs4412_led/fs4412_led.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/linux/workdir/driver/Linux3.14Drivers/fs4412_led/fs4412_led.mod.o
LD [M]  /home/linux/workdir/driver/Linux3.14Drivers/fs4412_led/fs4412_led.ko
make[1]:正在离开目录 `/home/linux/workdir/driver/linux-3.14-fs4412'
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_led$
```

查看编译生成的 ko 文件，并拷贝到 nfs 文件系统目录中。

```
$ ls
```

```
$ cp fs4412_led.ko /source/rootfs/
```

```
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_led$ ls
built-in.o  fs4412_led.ko  fs4412_led.o  Module.symvers
fs4412_led.c  fs4412_led.mod.c  Makefile      test.c
fs4412_led.h  fs4412_led.mod.o  modules.order
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_led$
```

执行命令编译测试文件。

```
$ arm-none-linux-gnueabi-gcc test.c -o test
```

```
$ cp test /source/rootfs
```



```
412_led/fs4412_led.o
2_led$ arm-none-linux-gnueabi-gcc test.c -o test
2_led$ cp test /source/rootfs
2_led$
```

16.4.4 执行代码

启动开发板，查看文件系统内容

```
# ls
```

```
COM6 - PuTTY
dev    lib    mnt    root    sys    tmp    var
[root@farsight]# ls
[  49.760000] nfs: server 192.168.100.192 not responding, still trying
[  82.840000] nfs: server 192.168.100.192 OK
bin    fs4412_led.ko  mnt    sbin    tmp
dev    lib           proc    sys     usr
etc    linuxrc       root    test    var
[root@farsight]# ls
bin    fs4412_led.ko  mnt    sbin    tmp
dev    lib           proc    sys     usr
etc    linuxrc       root    test    var
[root@farsight]#
```

加载驱动。

```
# insmod fs4412_led.ko
# mknod /dev/led c 500 0
# ./test
```

```
bin    led_test    proc    tmp
dev    lib         root    usr
etc    linuxrc     sbin    var
[root@farsight /]# insmod fs210_led.ko
Disabling lock debugging due to kernel taint
led: driver installed, with major 250!
[root@farsight /]#
```

```
COM6 - PuTTY
[  1.995000] device=eth0, hwaddr=00:0a:2d:a6:55:a2, ipaddr=192.168.
100.191, mask=255.255.255.0, gw=255.255.255.255
[  1.995000] host=192.168.100.191, domain=, nis-domain=(none)
[  1.995000] bootserver=255.255.255.255, rootserver=192.168.100.192
, rootpath=
[  1.995000] clk: Not disabling unused clocks
[  2.045000] VFS: Mounted root (nfs filesystem) on device 0:10.
[  2.050000] devtmpfs: mounted
[  2.055000] Freeing unused kernel memory: 228K (c0530000 - c0569000)
[  2.065000] hub 1-3:1.0: USB hub found
[  2.065000] hub 1-3:1.0: 3 ports detected
[root@farsight]# insmod fs4412_led.ko
[ 10.165000] Led init
[root@farsight]# ./test
open: No such file or directory
[root@farsight]# mknod /dev/led c 500 0
[root@farsight]# ./test
```



卸载驱动

```
# rmmod fs4412_led
```

卸载驱动时可能出现如下错误:

```
rmmod: can't change directory to '/lib/modules': No such file or directory
rmmod: can't change directory to '3.x.x': No such file or directory
```

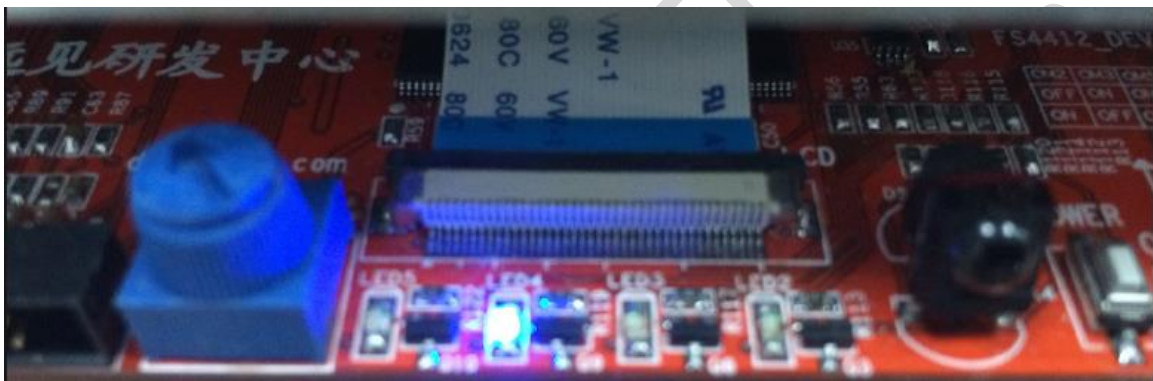
用户需要执行如下命令: (``符号一般情况下位于键盘中 ESC 键下方)

```
# mkdir -p /lib/modules/`uname -r`
```

执行完成后, 即能成功卸载驱动。

16.4.5 实验现象

可以看到 4 个 LED 间隔闪烁。





第 17 章 PWM 驱动开发实验

17.1 实验原理

如图所示，定时器 1 的输出引脚 TOUT1 和蜂鸣器的三极管相连，此电路的三极管是 PNP 性，当 TOUT1 是高电平时，此三极管处于饱和状态，电路导通，电流流过蜂鸣器，此时蜂鸣器发声；反之，当 TOUT1 是高电平时，此三极管处于截止状态，电路关断，时蜂鸣器停止发声。蜂鸣器发声的长短和频率，完全有 TOUT1 控制导通时间，一般都是设定一段延时就可以了，长短可以自己实验。

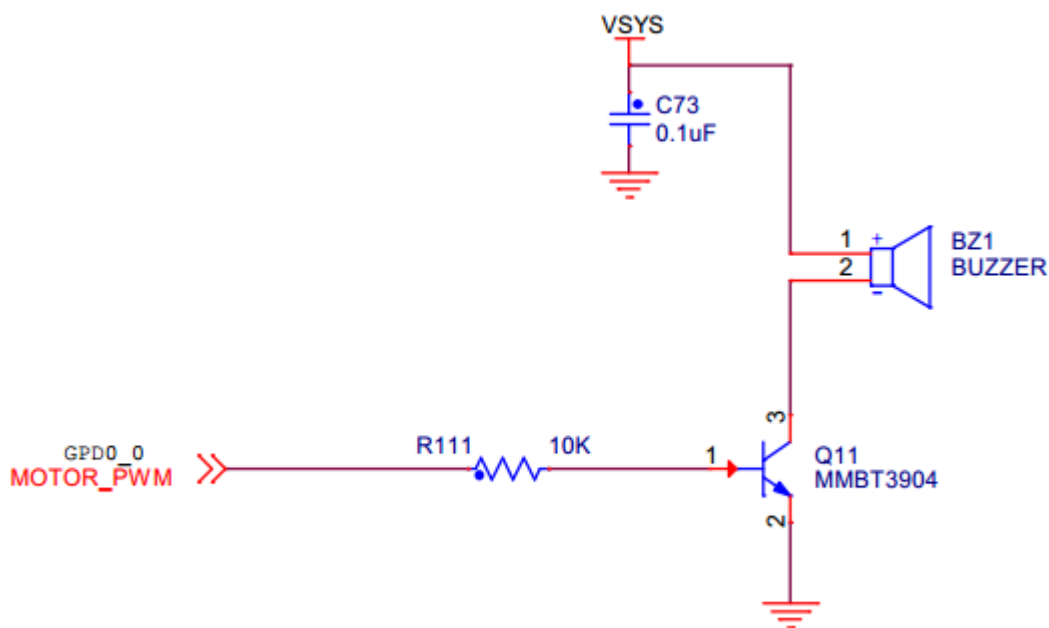


图 蜂鸣器控制电路

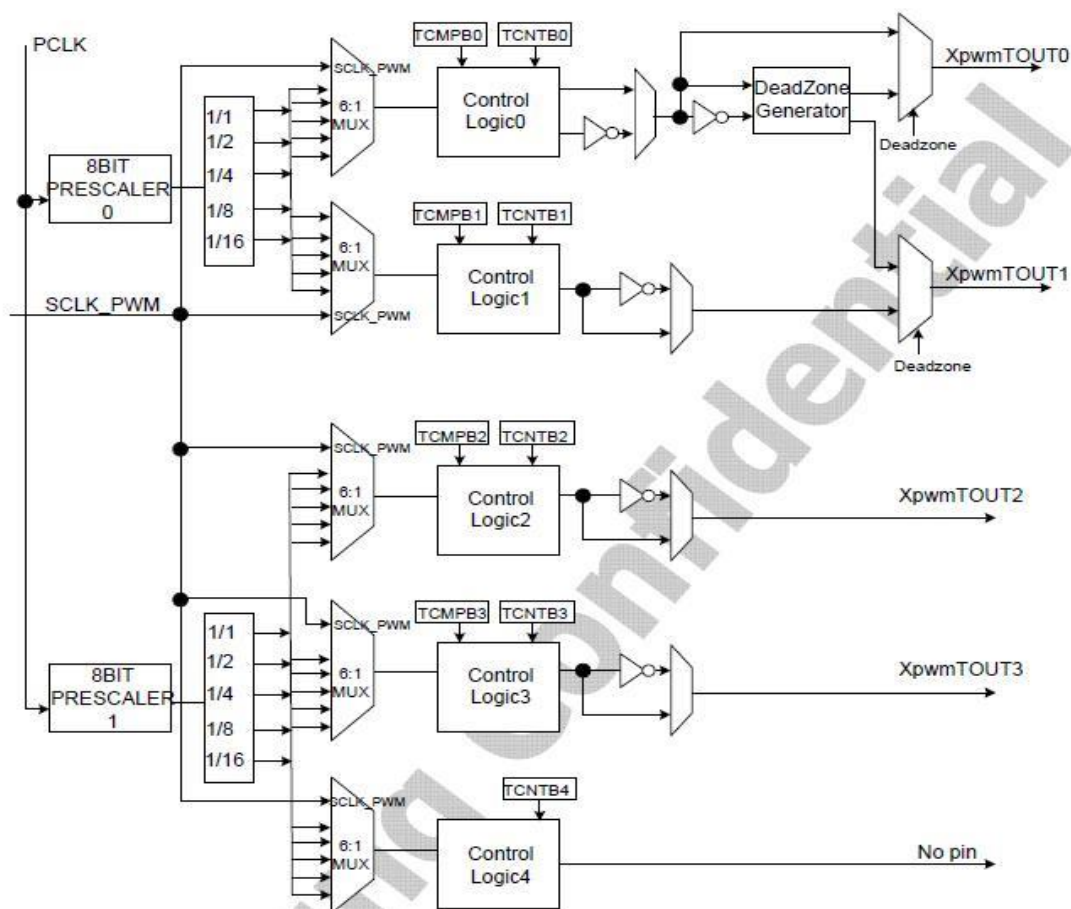


图 Exynos4412 PWM timer 结构框图

17.2 实验目的

利用 PWM 定时器实现蜂鸣器控制。

17.3 实验平台

华清远见开发环境，FS4412 平台

17.4 实验步骤

17.4.1 环境准备

参见【LED 驱动开发实验】章节，如已经完成上述实验则跳过此步骤。

17.4.2 代码准备

参见【LED 驱动开发实验】章节，如已经完成上述实验则跳过此步骤。

17.4.3 编译代码

```
$ cd ~/workdir/driver/Linux3.14Drivers/fs4412_pwm
```

修改 Makefile 文件第 3、4 行。



```
1 ifeq ($(KERNELRELEASE),)
2
3 KERNELDIR ?= /home/linux/linux-3.14-fs4412/
4 #KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5 PWD := $(shell pwd)
```

修改为用户内核源码的路径和交叉工具链。

```
linux@ubuntu64-vm: ~/workdir/driver/Linux3.14Drivers/fs4412_led
1 ifeq ($(KERNELRELEASE),)
2
3 KERNELDIR ?= /home/linux/workdir/driver/linux-3.14-fs4412/
4 #KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5 PWD := $(shell pwd)
6
```

保存退出。执行 make 命令编译源码。

```
$ make

linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_pwm$ make
make -C /home/linux/workdir/driver/linux-3.14-fs4412/ M=/home/linux/workdir/driver/Linux3.14Drivers/fs4412_pwm modules
make[1]: 正在进入目录 `/home/linux/workdir/driver/linux-3.14-fs4412/'
CC [M] /home/linux/workdir/driver/Linux3.14Drivers/fs4412_pwm/fs4412_pwm.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/linux/workdir/driver/Linux3.14Drivers/fs4412_pwm/fs4412_pwm.mod.o
LD [M] /home/linux/workdir/driver/Linux3.14Drivers/fs4412_pwm/fs4412_pwm.ko
make[1]:正在离开目录 `/home/linux/workdir/driver/linux-3.14-fs4412/'
cp *.ko /source/rootfs
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_pwm$
```

查看编译生成的 ko 文件，并拷贝到 nfs 文件系统目录中。

```
$ ls

$ cp fs4412_pwm.ko /source/rootfs/
```

```
LD [M] /home/linux/workdir/driver/Linux3.14Drivers/fs4412_pwm/fs4412_pwm.ko
make[1]:正在离开目录 `/home/linux/workdir/driver/linux-3.14-fs4412/'
cp *.ko /source/rootfs
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_pwm$ ls
fs4412_pwm.c fs4412_pwm.ko fs4412_pwm.mod.o Makefile Module.symvers pwm_music.h
fs4412_pwm.h fs4412_pwm.mod.c fs4412_pwm.o modules.order pwm_music.c
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_pwm$
```

执行 make pwm_test 命令编译测试文件。

```
$ arm-none-linux-gnueabi-gcc pwm_music.c -o pwm_music

$ cp pwm_music /source/rootfs
```

```
pwm$ arm-none-linux-gnueabi-gcc pwm_music.c -o pwm_music
pwm$ cp pwm_music /source/rootfs
```

17.4.4 执行代码

启动开发板，查看文件系统内容

```
# ls
```




```
COM6 - PuTTY
[ 2.050000] devtmpfs: mounted
[ 2.055000] Freeing unused kernel memory: 228K (c0530000 - c0569000)
[ 2.065000] hub 1-3:1.0: USB hub found
[ 2.065000] hub 1-3:1.0: 3 ports detected
[root@farsight]# insmod fs4412_led.ko
[ 10.165000] Led init
[root@farsight]# ./test
open: No such file or directory
[root@farsight]# mknod /dev/led c 500 0
[root@farsight]# ./test
[ 190.195000] random: nonblocking pool is initialized
^C
[root@farsight]# ls
bin          fs4412_pwm.ko  proc          sys          var
dev          lib            pwm_music    test
etc          linuxrc       root          tmp
fs4412_led.ko mnt           sbin         usr
[root@farsight]#
```

加载驱动。

```
# insmod fs4412_pwm.ko
# mknod /dev/pwm c 501 0
# ./pwm_music
```

卸载驱动

```
# rmmod fs4412_led
```

卸载驱动时可能出现如下错误：

```
rmmod: can't change directory to '/lib/modules': No such file or directory
rmmod: can't change directory to '3.x.x': No such file or directory
```

用户需要执行如下命令：（`符号一般情况下位于键盘中 ESC 键下方）

```
# mkdir -p /lib/modules/`uname -r`
```

执行完成后，即能成功卸载驱动。

17.4.5 实验现象

可以听到蜂鸣器会播放音乐。



第 18 章 按键驱动开发实验

18.1 实验原理

如图所示，三个按键 K2、K3、K4 分别对应三个 GPIO，且 GPIO 与中断复用，如图所示原理当按键没有按下 GPIO 处于高电平，当按键按下 GPIO 处于低电平，所以本例将 GPIO 设置为中断功能，当按键按下时下降沿出发中断，CPU 通过中断确定按键状态。

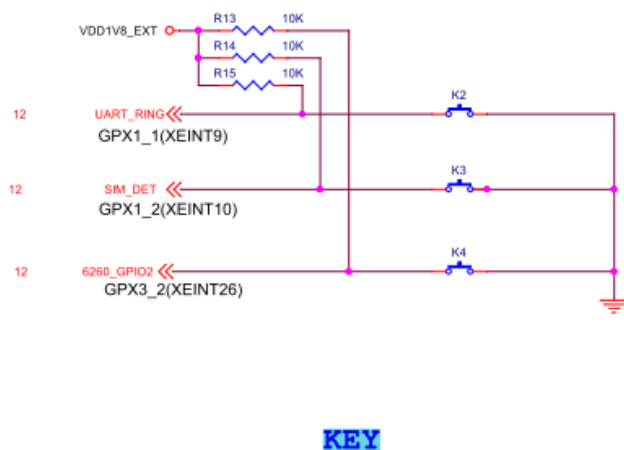


图 按键控制电路

18.2 实验目的

利用按键驱动的编写掌握 Linux 下中断的实现。

18.3 实验平台

华清远见开发环境，FS4412 平台

18.4 实验步骤

18.4.1 环境准备

参见【LED 驱动开发实验】章节，如已经完成上述实验则跳过此步骤。

18.4.2 代码准备

参见【LED 驱动开发实验】章节，如已经完成上述实验则跳过此步骤。

18.4.3 编译代码

```
$ cd ~/workdir/driver/Linux3.14Drivers/fs4412_key
```

修改 Makefile 文件第 3、4 行。



```

1  #KERNELDIR ?= /home/linux/workdir/4412/kernel/linux-3.14-fs4412/
2
3  KERNELDIR ?= /home/linux/workdir/4412/kernel/linux-3.14-fs4412/
4  #KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5  PWD := $(shell pwd)
6
7  modules:
8      $(MAKE) -C $(KERNELDIR) M=$(PWD)
9      cp *.ko /source/rootfs
10

```

修改为用户内核源码的路径和交叉工具链。

保存退出。执行 make 命令编译源码。

```
$ make
```

```

linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_key$ make
make -C /home/linux/workdir/driver/linux-3.14-fs4412/ M=/home/linux/workdir/driver/Linux3.14Drivers/fs4412_key
make[1]: 正在进入目录 `/home/linux/workdir/driver/linux-3.14-fs4412/'
LD      /home/linux/workdir/driver/Linux3.14Drivers/fs4412_key/built-in.o
CC [M]  /home/linux/workdir/driver/Linux3.14Drivers/fs4412_key/fs4412_key.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/linux/workdir/driver/Linux3.14Drivers/fs4412_key/fs4412_key.mod.o
LD [M]  /home/linux/workdir/driver/Linux3.14Drivers/fs4412_key/fs4412_key.ko
make[1]: 正在离开目录 `/home/linux/workdir/driver/linux-3.14-fs4412/'
cp *.ko /source/rootfs
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_key$

```

查看编译生成的 ko 文件，并拷贝到 nfs 文件系统目录中。

```
$ ls
```

```
$ cp fs4412_key.ko /source/rootfs/
```

18.4.4 修改设备树文件

进入 Linux 内核

```
$ cd /home/linux/workdir/driver/linux-3.14-fs4412
```

打开设备树文件

```
$ vim arch/arm/boot/dts/exynos4412-fs4412.dts
```

添加如下内容

C++ Code

```

1  fs4412-key {
2      compatible = "fs4412,key";
3      interrupt-parent = <&gpx1>;
4      interrupts = <1 2>, <2 2>;
5  };

```

重新编译设备树文件，并拷贝到/tftpboot 目录下

```
$ make dtbs
```

```
$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot
```

重新启动开发板即可



18.4.5 执行代码

启动开发板，查看文件系统内容

```
# ls
```

```
COM22 - PuTTY

[root@farsight ]# ls
bin          fs4412_key.ko  mnt          root         tmp
dev          lib            opt          sbin         usr
etc          linuxrc        proc         sys          var
[root@farsight ]#
```

加载驱动。

```
# insmod fs4412_key.ko
```

加载驱动后屏幕会显示 match OK 的字样，否则说明设备树修改错误，或设备树文件没有被正确加载

```
[root@farsight ]# insmod fs4412_key.ko
[ 107.725000] match OK
```

卸载驱动

```
# rmmod fs4412_led
```

卸载驱动时可能出现如下错误：

```
rmmod: can't change directory to '/lib/modules': No such file or directory
rmmod: can't change directory to '3.x.x': No such file or directory
```

用户需要执行如下命令：（`符号一般情况下位于键盘中 ESC 键下方）

```
# mkdir -p /lib/modules/`uname -r`
```

执行完成后，即能成功卸载驱动。

18.4.6 实验现象

连续按下 k2 和 k3，屏幕会打印出按键对应的中断号，如图：

```
COM22 - PuTTY

[root@farsight ]# ls
bin          fs4412_key.ko  mnt          root         tmp
dev          lib            opt          sbin         usr
etc          linuxrc        proc         sys          var
[root@farsight ]# insmod fs4412_key.ko
[ 107.725000] match OK
[root@farsight ]# [ 112.745000] irqno = 168
[ 114.005000] irqno = 169
[ 114.570000] irqno = 168
[ 115.180000] irqno = 169
[ 115.695000] irqno = 168
[ 115.835000] irqno = 168
[ 116.215000] irqno = 169
[ 116.640000] irqno = 168
[ 117.095000] irqno = 169
```



第 19 章 ADC 驱动开发实验

19.1 实验原理

如图所示，XadcAIN3 为 Exynos4412 adc 输入通道，Exynos4412 通过这个通道采集电压，然后再将其转换为数字量，Exynos4412 精度为 10bit 或 12bit，输入电压为 0~1.8V，通过旋转电位计可以调节电压，最终通过 ADC 转换，计算出当前电压。

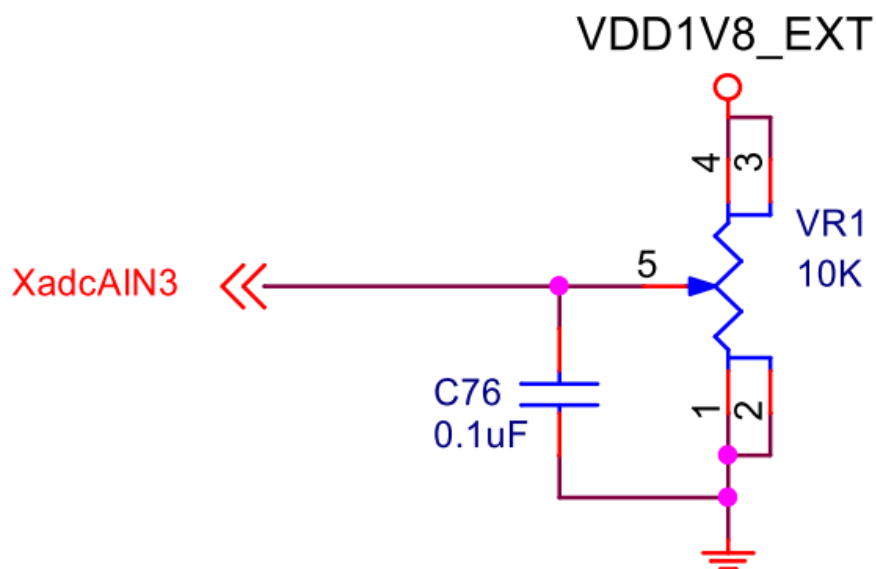


图 ADC 控制电路

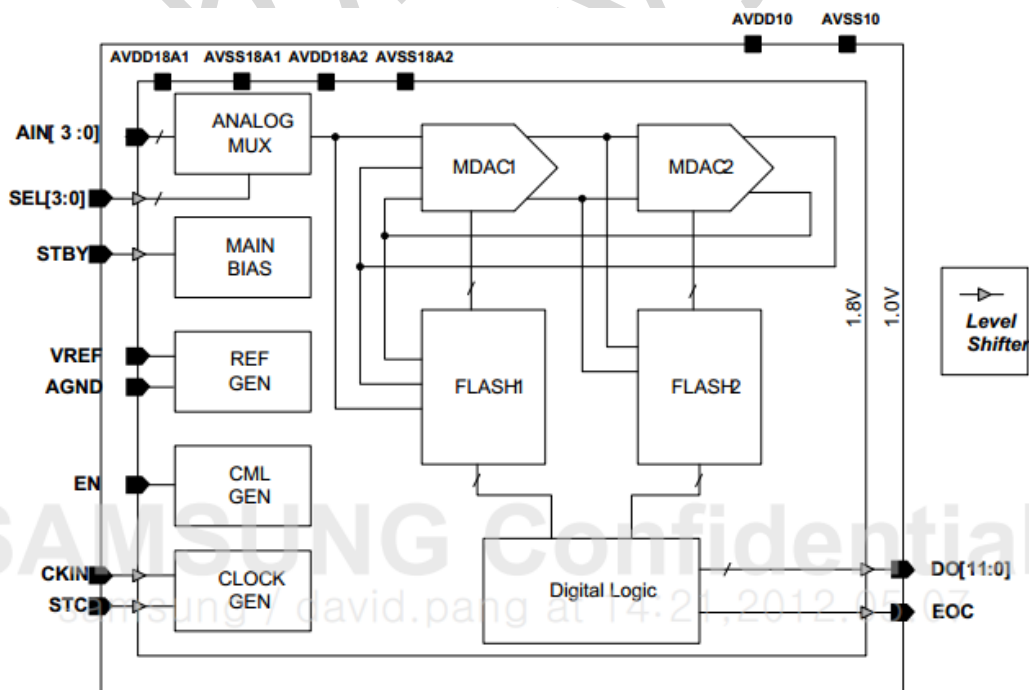


图 ADC 功能框图



19.2 实验目的

利用 ADC 驱动的编写掌握 Linux 下中断及 ADC 驱动的实现。

19.3 实验平台

华清远见开发环境，FS4412 平台

19.4 实验步骤

19.4.1 环境准备

参见【LED 驱动开发实验】章节，如已经完成上述实验则跳过此步骤。

19.4.2 代码准备

参见【LED 驱动开发实验】章节，如已经完成上述实验则跳过此步骤。

19.4.3 编译代码

```
$ cd ~/workdir/driver/Linux3.14Drivers/fs4412_adc
```

修改 Makefile 文件第 3、4 行。

```
1 ifeq ($(KERNELRELEASE),)
2
3 KERNELDIR ?= /home/linux/workdir/4412/kernel/linux-3.14-fs4412/
4 #KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5 PWD := $(shell pwd)
6
7 modules:
8     $(MAKE) -C $(KERNELDIR) M=$(PWD)
9     cp *.ko /source/rootfs
```

修改为我们内核源码的路径和交叉工具链。

保存退出。执行 make 命令编译源码。

```
$ make
```

```
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_adc$ make
make -C /home/linux/workdir/driver/linux-3.14-fs4412/ M=/home/linux/workdir/driver/Linux3.14Drivers/fs4412_adc
make[1]: 正在进入目录 `/home/linux/workdir/driver/linux-3.14-fs4412'
LD      /home/linux/workdir/driver/Linux3.14Drivers/fs4412_adc/built-in.o
CC [M]  /home/linux/workdir/driver/Linux3.14Drivers/fs4412_adc/fs4412_adc.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/linux/workdir/driver/Linux3.14Drivers/fs4412_adc/fs4412_adc.mod.o
LD [M]  /home/linux/workdir/driver/Linux3.14Drivers/fs4412_adc/fs4412_adc.ko
make[1]:正在离开目录 `/home/linux/workdir/driver/linux-3.14-fs4412'
cp *.ko /source/rootfs
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/fs4412_adc$
```

查看编译生成的 ko 文件，并拷贝到 nfs 文件系统目录中。

```
$ ls
```

```
$ cp fs4412_adc.ko /source/rootfs/
```



19.4.4 编译应用程序

编译应用程序并拷贝到根文件系统下

```
$ arm-none-linux-gnueabi-gcc test.c -o test
$ cp test /source/rootfs
```

19.4.5 修改设备树文件

进入 Linux 内核

```
$ cd /home/linux/workdir/driver/linux-3.14-fs4412
```

打开设备树文件

```
$ vim arch/arm/boot/dts/exynos4412-fs4412.dts
```

添加如下内容

C++ Code

```
1 fs4412-adc@126c0000{
2     compatible = "fs4412,adc";
3     reg = <0x126c0000 0x20>;
4     interrupt-parent = <&combiner>;
5     interrupts = <10 3>;
6 };
```

重新编译设备树文件，并拷贝到/tftpboot 目录下

```
$ make dtbs
$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot
```

重新启动开发板即可

19.4.6 执行代码

启动开发板，查看文件系统内容

```
# ls
```

```
[root@farsight]# ls
bin          fs4412_key.ko  opt          sys
dev          lib            proc         tmp
etc          linuxrc       root         usr
fs4412_adc.ko mnt           sbin         var
[root@farsight]#
```

加载驱动。

```
# insmod fs4412_adc.ko
```

加载驱动后屏幕会显示 match OK 的字样，否则说明设备树修改错误，或设备树文件没有被正确加载

```
[root@farsight]# insmod fs4412_adc.ko
[ 102.060000] match OK
[ 102.060000] mem = 126c0000: irq = 170
[ 102.065000] major = 500, minor = 0, devno = 1f400000
```

创建设备结点

```
# mknod /dev/adc c 500 0
```



执行应用程序

```
# ./test
```

19.4.7 实验现象

旋转电位计，会显示当前电压，如图：

```
[root@farsight ]# ./test  
Vol: 0.68V  
Vol: 0.68V  
Vol: 0.68V  
Vol: 0.69V  
Vol: 0.81V  
Vol: 0.90V  
Vol: 0.98V  
Vol: 1.02V  
Vol: 1.05V  
Vol: 1.07V  
Vol: 1.10V
```




第 20 章 I2C 设备驱动开发实验

20.1 实验原理

如图所示 I2C 总线上接入的是 MPU6050，MPU6050 包括陀螺仪和加速度传感器功能，陀螺仪和加速度传感器分别用三个 16bit 的数据表示，量程和精度都是可编程的，可按需求设置，具体 MPU6050 参数参照芯片手册。Exynos4412 共有八组 I2C 控制器，MPU6050 接在 I2C5 上。

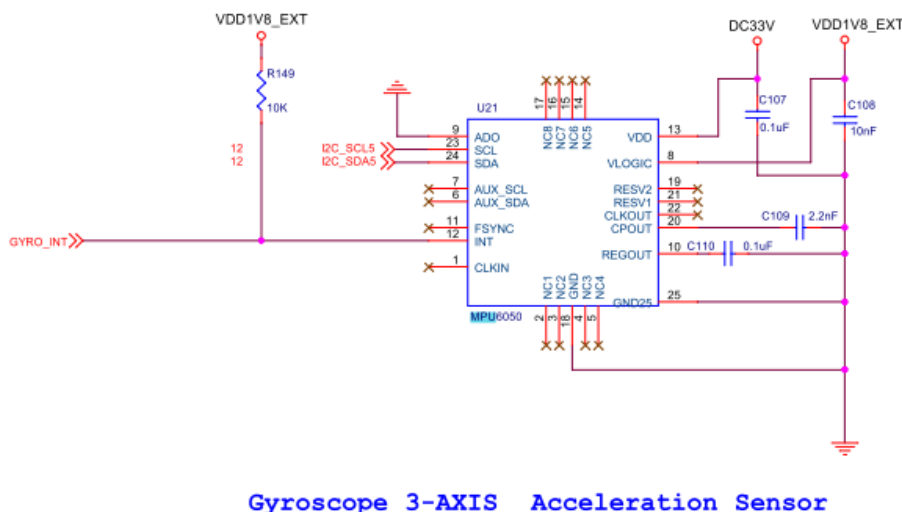


图 主板陀螺仪接口

20.2 实验目的

通过 MPU6050 加速度陀螺仪驱动的编写，掌握 Linux 下 I2C 设备驱动的编写。

20.3 实验平台

华清远见开发环境，FS4412 平台

20.4 实验步骤

20.4.1 环境准备

参见【LED 驱动开发实验】章节，如已经完成上述实验则跳过此步骤。

20.4.2 代码准备

参见【LED 驱动开发实验】章节，如已经完成上述实验则跳过此步骤。

20.4.3 编译代码

```
$ cd ~/workdir/driver/Linux3.14Drivers/mpu6050
```

修改 Makefile 文件第 3、4 行。



```

1 #KERNELDIR ?= /home/linux/workdir/4412/kernel/linux-3.14-fs4412/
2
3 KERNELDIR ?= /home/linux/workdir/4412/kernel/linux-3.14-fs4412/
4 #KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5 PWD := $(shell pwd)
6
7 modules:
8     $(MAKE) -C $(KERNELDIR) M=$(PWD)
9     cp *.ko /source/rootfs
10

```

修改为用户内核源码的路径和交叉工具链。

保存退出。执行 make 命令编译源码。

```
$ make
```

```

linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/mpu6050$ make
make -C /home/linux/workdir/driver/linux-3.14-fs4412/ M=/home/linux/workdir/driver/Linux3.14Drivers/mpu
make[1]: 正在进入目录 `/home/linux/workdir/driver/linux-3.14-fs4412'
LD      /home/linux/workdir/driver/Linux3.14Drivers/mpu6050/built-in.o
CC [M]  /home/linux/workdir/driver/Linux3.14Drivers/mpu6050/mpu6050.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/linux/workdir/driver/Linux3.14Drivers/mpu6050/mpu6050.mod.o
LD [M]  /home/linux/workdir/driver/Linux3.14Drivers/mpu6050/mpu6050.ko
make[1]:正在离开目录 `/home/linux/workdir/driver/linux-3.14-fs4412'
cp *.ko /source/rootfs
linux@ubuntu64-vm:~/workdir/driver/Linux3.14Drivers/mpu6050$

```

查看编译生成的 ko 文件，并拷贝到 nfs 文件系统目录中。

```
$ ls
```

```
$ cp mpu6050.ko /source/rootfs/
```

20.4.4 编译应用程序

编译应用程序并拷贝到根文件系统下

```
$ arm-none-linux-gnueabi-gcc test.c -o test
```

```
$ cp test /source/rootfs
```

20.4.5 修改设备树文件

进入 Linux 内核

```
$ cd /home/linux/workdir/driver/linux-3.14-fs4412
```

打开设备树文件

```
$ vim arch/arm/boot/dts/exynos4412-fs4412.dts
```

添加如下内容

C++ Code

```

1 i2c@138B0000 {
2     samsung,i2c-sda-delay = <100>;
3     samsung,i2c-max-bus-freq = <20000>;
4     pinctrl-0 = <&i2c5_bus>;
5     pinctrl-names = "default";
6     status = "okay";

```



```

6
7     mpu6050-3-axis@68 {
8         compatible = "invensense,mpu6050";
9         reg = <0x68>;
10        interrupt-parent = <&gpx3>;
11        interrupts = <3 2>;
12    };
13 };

```

重新编译设备树文件，并拷贝到/tftpboot 目录下

```
$ make dtbs
```

```
$ cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot
```

重新启动开发板即可

20.4.6 执行代码

启动开发板，查看文件系统内容

```
# ls
```

```

[root@farsight ]# ls
bin          fs4412_key.ko  mpu6050.ko    sbin         usr
dev          lib            opt           sys          var
etc          linuxrc       proc          test
fs4412_adc.ko mnt           root          tmp
[root@farsight ]#

```

加载驱动。

```
# insmod mpu6050.ko
```

加载驱动后屏幕会显示 match OK 的字样，否则说明设备树修改错误，或设备树文件没有被正确加载

```

[root@farsight ]# insmod mpu6050.ko
[  38.935000] match OK!
[root@farsight ]#

```

创建设备结点

```
# mknod /dev/mpu6050 c 500 0
```

执行应用程序

```
# ./test
```

卸载驱动

```
# rmmod fs4412_led
```

卸载驱动时可能出现如下错误：

```

rmmod: can't change directory to '/lib/modules': No such file or directory
rmmod: can't change directory to '3.x.x': No such file or directory

```

用户需要执行如下命令：（`符号一般情况下位于键盘中 ESC 键下方）

```
# mkdir -p /lib/modules/`uname -r`
```

执行完成后，即能成功卸载驱动。



20.4.7 实验现象

晃动开发板会发现加速度传感器和陀螺仪的数值会发生变化，如图：

```
[root@farsight ]# ./test
acceleration data: x = 01be, y = ff21, z = d23f
gyroscope data: x = 031b, y = fcba, z = 0454
acceleration data: x = 01bc, y = ff23, z = d2b0
gyroscope data: x = 0319, y = fcba, z = 0453
acceleration data: x = 01be, y = ff24, z = d267
gyroscope data: x = 031a, y = fcba, z = 0454
acceleration data: x = 02d2, y = 025b, z = d245
gyroscope data: x = 0684, y = fa31, z = 051f
acceleration data: x = 01a2, y = ffe0, z = d28d
gyroscope data: x = fec3, y = fcce, z = 0348
```

华清远见
dev.hqyj.com



华清远见
dev.hqyj.com