普通高等教育“十一五”国家级规划教材

Linux 高级程序设计

Advanced Programming in Linux

罗怡桂

高等教育出版社

普通高等教育“十一五”国家级规划教材

Linux 高级程序设计

Linux Gaoji Chengxu Sheji

罗怡桂

高等教育出版社·北京

内容简介

本书内容选择精要，由浅入深、循序渐进地阐述了 Linux 环境下的高级编程技术，体现了 Linux 高级编程人员必备的技术要求。全书共 10 章，主要包括 Linux 编程基础、文件与目录的操作、标准输入输出及系统信息、进程及进程的控制、信号、高级 I/O、进程之间的通信、服务进程、多进程的综合控制与多线程编程。书后的综合案例习题提供了一些小型案例项目，供读者自行实践。

本书可作为本科生或研究生的 Linux 编程技术课程的教材，也可以作为 Linux 编程技术人员的参考书。

图书在版编目(CIP)数据

Linux 高级程序设计/罗怡桂编著.—北京:高等教育出版社,2014.9

ISBN 978-7-04-040958-1

Ⅰ.①L… Ⅱ.①罗… Ⅲ.①Linux 操作系统-程序设计 Ⅳ.①TP316.89

中国版本图书馆 CIP 数据核字(2014)第 192923 号

策划编辑	倪文慧	责任编辑	倪文慧	封面设计	张 志	版式设计	杜微言
插图绘制	邓 超	责任校对	胡美萍	责任印制	毛斯璐		

出版发行	高等教育出版社	咨询电话	400-810-0598
社 址	北京市西城区德外大街 4 号	网 址	http://www.hep.edu.cn
邮政编码	100120		http://www.hep.com.cn
印 刷	北京北苑印刷有限责任公司	网上订购	http://www.landraco.com
开 本	787mm×1092mm 1/16		http://www.landraco.com.cn
印 张	17	版 次	2014 年 9 月第 1 版
字 数	380 千字	印 次	2014 年 9 月第 1 次印刷
购书热线	010-58581118	定 价	27.00 元

物 料 号 40958-00

前　言

Linux 操作系统从 1990 年诞生至今，已在全世界范围得到了广泛应用。Linux 行业的从业人员在不断增加，这种趋势在我国更为明显，因而基于 Linux 的编程技术已成为高校计算机类专业一门重要的课程。

应掌握什么样的编程技术才能成为高级 Linux 编程人员呢？要回答这一问题，首先要清楚什么样的人才算是高级 Linux 编程人员。假如你有参与应用软件系统开发的经验，如开发数据库软件、文本编辑软件、操作系统桌面管理软件等，那么可以说你已经是一个高级编程人员了。开发这些软件所必需的知识和技术就是高级编程技术，其中最为核心的就是基于 Linux 操作系统的程序优化。如何优化，前提条件是深刻地了解操作系统以及操作系统的系统调用。也许有人会说，这太简单了，找个手册看一下就知道了。是的，的确需要查找一些手册，但了解与之相关的操作系统的结构、Linux 操作系统环境的特点、编程模式等更为重要。当然，最重要的是如何将这些体现在 Linux 编程中，从而充分利用 Linux 环境写出高性能的程序。本书编者主要围绕此目的进行内容的组织。

本书共 10 章，分别为 Linux 编程基础，文件与目录的操作，标准输入输出及系统信息，进程及进程的控制，信号，高级 I/O，进程之间的通信，服务进程，多进程的综合控制和多线程编程，以由浅入深、循序渐进的方式进行阐述。书后的综合案例习题提供了一些小型案例项目，供读者自行实践。

书中所有程序以及配套的电子教案可以从中国高校计算机课程网下载，网址为 http://computer.cncourse.com。本书具有很强的实践性，建议读者边做边学。

本书由罗怡桂编写，研究生闫亚飞与顾文迪编写了第 4 章、第 9 章的例子并进行了调试。同济大学为本书提供了教材出版基金。本书的出版得到了同济大学软件学院领导、同事以及高等教育出版社的大力支持。在此致以诚挚的谢意。

由于作者水平有限，书中难免有不妥之处，敬请读者批评指正。

罗怡桂

2014 年 6 月

目　录

第1章 Linux编程基础

1.1 Linux的演化

1.1.1 UNIX时代

1965年，AT&T公司、MIT与GE公司合作，试图开发一个多用户交互式操作系统Multics，但这个项目以失败告终。1969年，AT&T公司的Thompson在PDP-7机上用汇编语言开发了第一个UNIX操作系统。1972年发布了UNIX V2，在这一时期C语言同时应运而生。1974年，著名的计算机技术杂志*Communications of the ACM*刊登了*The UNIX Time Sharing System*一文，这是UNIX与外界的首次接触。1975年、1979年UNIX V6与V7发表。在这期间许多大学开始加入UNIX操作系统开发的行列，如Berkeley对其内核做了大量的工作，包括内存的页管理、虚拟内存管理等，现在见到的BSD 4.0（1980）、BSD 4.1（1981）、BSD 4.2（1983）、BSD 4.3（1986）、BSD 4.4（1993）都是在这一基础上发展而来的。

1977年，AT&T公司开始提供UNIX OEM给计算机生产厂家进行应用。1982年，Sun公司发布了SunOS，这是一个基于BSD 4.2的UNIX操作系统。随后，Microsoft公司与SCO公司发布了XENIX，其中SCO公司将SVR3移植到了386机器上，因而后来称之为SCO UNIX。到了20世纪八九十年代，更多的商业版UNIX问市了，包括IBM公司的AIX、HP公司的HP-UX等。

随着各种版本UNIX欣欣向荣的发展，带来的一个问题是如何保证程序在运行着不同版本UNIX的计算机上兼容。这不仅是一个技术问题，也是各个公司之间合作与竞争的利益焦点。于是就出现了一系列UNIX标准，其中包括AT&T公司定义的SVID（System V Interface Definition），以及由IEEE组织支持的POSIX等。这些标准都定义了操作系统提供给应用程序的接口，即函数的原型及语义。

1.1.2 从UNIX到Linux

1990年，芬兰大学生Linus Torvalds在学习操作系统课程时在MINIX（a mini Unix OS，由荷兰Vrije大学的Tanenbaum教授完成）环境下开发了一些自己的内核程序。1991年，他完成了第一个Linux内核，并于1992年将其所有的代码公开给自由软件联盟（Free Software Foundation，FSF）的GNU计划。此后，世界范围内所有爱好自由软件的学者、学生和研究人员都

可以在此基础上不断地对 Linux 进行升级。

目前 Linux 正处于一个蓬勃发展的时代。全世界有众多计算机运行着 Linux，诸多公司都参与 Linux 的开发，包括 Microsoft、Intel、IBM、HP、DELL、联想、中兴、华为等。同时，Linux 正在越来越多的领域中发挥着巨大的作用，无论是在服务器系统，还是无处不在的嵌入式系统。

1.2 Linux 编程环境

通常来说，可以将编程分为程序的编辑、编译和调试等几个阶段。

1.2.1 程序编辑器

vi 是 Linux 下最基本的程序编辑工具。它支持两种工作模式：命令模式与编辑模式。其常用的命令如下。

（1）i，由命令模式进入到编辑模式，在光标处插入字符。

（2）a，用于编辑模式，在光标后面添加字符。

（3）x，删除字符。

（4）dd，删除一行。

（5）:行号，进入指定行。

（6）:/字符串，搜索这个字符串。

（7）:q，退出 vi。

（8）:wq，将文件存盘并退出 vi。

（9）ndd，删除 *n* 行。

（10）yy，复制当前行到剪贴板。

（11）p，将剪贴板上的内容粘贴到光标处。

（12）ny，从当前行开始复制 *n* 行到剪贴板。

1.2.2 程序编译器 gcc

gcc 是一个 GNU C 语言编译器，在 Linux 中常用。其功能比较丰富，能将 C 程序编译成汇编代码、目标代码或者可执行文件，也可仅进行预编译处理，生成预编译后的 C 文件。常用的命令示例如 D1-1[①] 所示。

D1-1 gcc 常用命令示例

```
gcc    test.c                  ->   a.out   /* 编译 test.c 生成默认的 a.out 可执行文件 */
```

① 注：为便于阐述，本书各章将相关命令和代码段进行编号，序号分别为 D1-1、D1-2 等。

```
gcc -s  test. c                  ->  test. s  /* 编译生成汇编文件 */
gcc -c  test. c                  ->  test. o  /* 编译生成目标文件 */
gcc -o  test  test. c            ->  test     /* 编译 test. c 生成为 test 的可执行文件 */
gcc -o  test test1. o  test2. o  ->  test     /* 将 test1. o,test2. o 两个目标文件链接成可执行文件 test */
```

1.3 常用调试工具

1.3.1 gdb

gdb 是一种 GNU 调试器，其基本使用方法如下。

- 调试可执行文件 test：gdb test
- 列出代码行：l
- 设置断点：b
- 运行程序：r
- 继续运行：c
- 单步运行：n
- 退出 gdb：q
- 删除断点：delete，d
- 运行 shell 命令：shell command
- 在使用 gdb 进行调试时，可执行程序应当在用 gcc 编译时加上 -g 选项，这样在编译时调试信息就会添加到可执行文件中。

1.3.2 mtrace

在 Linux 应用程序的开发过程中，内存溢出是很常见的错误，采用 mtrace 可以调试这一类错误。

启用 mtrace 后，每个 malloc、realloc、free 都会留下相应的日志信息。具体操作步骤如下。

（1）使用 export 命令，或者调用 putenv 函数注册一个环境变量 MALLOC_TRACE。

（2）在程序中调用 mtrace( )与 muntrace( )，从 mtrace 到 muntrace 之间的内存分配（malloc）与释放（free）将会记入日志文件。

（3）打开日志文件即可看到相关的内存使用情况。

例 1 首先注册一个环境变量，如 D1-2 所示。

D1-2 注册一个环境变量

```
$ export MALLOC_TRACE = . /mymemtrace. log
```

然后调用 mtrace，如 D1-3 所示。其中第 4 行引用头文件 mcheck. h。第 9 行开启 mtrace 功能，第 10 到 13 行进行内存的分配与释放，第 14 行关闭 mtrace 功能。

D1-3 应用 mtrace 示例

```
1  #include "stdlib. h"
2  #include "stdio. h"
3  #include <string. h>
4  #include <mcheck. h>
5  int main( )
6  {
7      char * tmp;
8      printf( "hello\n" );
9      mtrace( );
10     tmp = ( char * ) malloc( 16 );
11     memset( tmp,0,16 );
12     * tmp = 1234;
13     free( tmp );
14     muntrace( );
15     return 1;
16 }
```

运行该程序后，直接打开日志 mymemtrace. log，记录如 D1-4 所示。

D1-4 查看 mtrace 的日志

```
= Start
@ ./testmtrace. o: [0x8048442] +0x873b378 0x10
@ ./testmtrace. o: [0x8048474] - 0x873b378
= End
```

其中，日志中记录的是从虚拟地址 0x8048442 执行的代码处申请了起始地址为 0x873b378、大小为 0x10（即 16 个字节）的地址空间；而 0x8048474 处将 0x873b378 的地址空间释放了。其中 + 表示申请，- 表示释放。

进一步用 addr2line 命令可以看到具体申请地址与释放地址的代码，如 D1-5 所示。

D1-5 用 addr2line 查看具体的代码

```
[root@ localhost embedded_linux_example]# addr2line -e ./testmtrace. o 0x8048442
/mnt/my/embedded_linux_example/testmtrace. c:10
[root@ localhost embedded_linux_example]# addr2line -e ./testmtrace. o 0x8048474
```

/mnt/my/embedded_linux_example/testmtrace.c:14

因此，不难写一脚本文件用于对 mtrace 日志文件的解析，进而自动诊断内存的泄漏。

1.3.3 hook

GNU C 还提供了一种钩子（hook）函数用以对内存进行跟踪、调试。这一组函数如 D1-6 所示。

D1-6 hook 函数原型

```
void * ( * _malloc_hook) ( size_t size, const void * caller) ;
void * ( * _realloc_hook) ( void * ptr, size_t size, const void * caller) ;
void * ( * _memalign_hook) ( size_t alignment, size_t size, const void * caller) ;
void( * _free_hook) ( void * ptr, const void * caller) ;
void( * _malloc_initialize_hook) ( void) ;
void( * _after_morecore_hook) ( void) ;
```

分别对应于 malloc、realloc、memalign、free。free，after_morecore_hook 指向的函数在每次调用 sbrk( )时就会调用。_malloc_initialize_hook 指向的函数在每次这组钩子函数实现时调用。

例 2 对于例 1 中的程序，去掉 mtrace 与 muntrace，改用 hook 进行跟踪。

首先定义钩子函数如 D1-7 所示。第 3、4 行声明自定义钩子函数，第 6 行给 C 库内置函数指针赋值，指向自定义钩子函数。当调用 malloc 库函数时就会去检查这个_malloc_initialize_hook 指针，如果指向的值非空就会启用钩子函数。在第 7 行到第 10 行中，对 C 库内置变量_malloc_hook 赋以自定义的钩子函数指针。在第 11 行到第 21 行实现这个自定义钩子函数。

D1-7 hooks 应用示例

```
1  #include < stdio. h >
2  #include < malloc. h >
3  static void my_init_hook( void) ;
4  static void * my_malloc_hook( size_t, const void * ) ;
5  static void * ( * old_malloc_hook) ( size_t, const void * ) ;
6  void( * _malloc_initialize_hook) ( void) = my_init_hook;
7  static void my_init_hook( void) {
8   old_malloc_hook = _malloc_hook;
9   _malloc_hook = my_malloc_hook;
10  }
11 static void *
12     my_malloc_hook( size_t size, const void * caller) {
```

```
13          void * result;
14          _malloc_hook = old_malloc_hook;
15          result = malloc(size);
16          old_malloc_hook = _malloc_hook;
17          printf("malloc(%u) called from %p returns %p0,
18              (unsigned int) size,caller,result);
19          _malloc_hook = my_malloc_hook;
20          return result;
21      }
```

与 mtrace 相比，钩子函数无需去改变原来的应用程序主体，如不用在原来的应用程序代码中插入 mtrace 与 muntrace，只要将 D1-7 中的文件保存成 C 语言的头文件形式，然后在源代码中引用就可以了。将 D1-7 中的内容保存成头文件 myhook. h，然后将 D1-3 中的第 9 行与第 14 行去掉，并加上 myhook. h，就成了 D1-8 中的代码。

D1-8 hook 使用示例

```
1  #include "stdlib. h"
2  #include "stdio. h"
3  #include < string. h >
4  #include < myhook. h >
5  int main( )
6  {
7      char * tmp;
8      printf("hello\n");
9      //mtrace( );
10     tmp = (char * ) malloc(16);
11     memset(tmp,0,16);
12     * tmp = 1234;
13     free(tmp);
14     //muntrace( );
15     return 1;
16 }
```

将 D1-8 所示的例子编译运行，其结果如 D1-9 所示。从第二句可以看到，当在程序中调用 malloc 时就会自动调用 my_malloc_hook。执行结果的第二行正是 D1-7 第 17 行的执行结果。malloc 申请了 16 个字节，其起始地址是 0x80cb0080。类似地，hook 同样适应于 free、realloc、memalign 函数。

D1-9 D1-8 所示例子的运行结果

```
[root@localhost hooks]#./a.out
hello,0x8048485
malloc(16)called from 0x4c0fd0fe returns 0x80cb0080
```

其他内存调试工具如 purify、boundercheck、valgrind 等也是不错的选择。

1.3.4 Binutil 工具集

Binutil 工具集主要分成两部分：一部分是对二进制文件进行查看与格式转换，另一部分用来对文件进行编译与归档。主要包含以下工具。

（1）addr2line：可以用来根据目标文件或可执行文件的地址找到对应函数名的第几行，但是有一个前提条件是其符号表没有去掉。

（2）ar：建立、修改与提取归档文件。归档文件是包含多个文件内容的一个大文件。

（3）as：主要用来编译 gcc 编译器输出的汇编文件，产生的目标文件由连接器 ld 连接。

（4）ld：连接器，把一些目标和归档文件结合在一起，重定位数据并连接符号引用。通常建立一个新编译程序的最后一步就是调用 ld。

（5）nm：列出目标文件中的符号。

（6）objcopy：将目标文件的内容复制到另一目标文件中，当另一目标文件的格式设置之后可进行格式的转换。

（7）objdump：显示一个或者更多目标文件的信息。

（8）ranlib：产生归档文件索引，并将其保存到该归档文件中。

（9）readelf：用来显示 elf 可执行文件各个段的信息。

（10）size：列出目标文件每一个段及总的大小。

1.4 GNU 编程风格

Linux 是开源的，其支持的许多应用程序也是开源的。开源提供了一种公开交流的平台，但是源代码的作者来自不同的企业、不同的领域，其代码写作风格各不相同，尤其是很多黑客写的代码非常紧凑，可读性比较差。因此各代码之间保持一种明朗的风格很重要。

GNU 对源代码的要求主要有以下几个方面。

（1）不要复制带有版权的源代码。

（2）可接收 GNU 代码。

（3）书写 changelog，以记录对源代码的修改。

关于代码风格的具体描述参见文献［7］。

本章小结

本章主要介绍了 Linux 的产生与发展，Linux 的编程环境与编程风格。应该说这时读者已能编写简单的 Linux 程序了。Linux 程序的学习具有很强的实践性，只有实践才能掌握其中的技巧与奥妙。

习　题

1.1　简述 Linux 的发展历史。

1.2　写一个 hello.c 程序，并用本章学习的知识进行编译调试。

1.3　如何写程序的 changelog？试为习题 1.2 写一个 changelog。

第 2 章　文件与目录的操作

2.1　Linux 文件系统概述

2.1.1　文件及文件系统

一般来说，在计算机系统中除了内存以外，还有如硬盘、U 盘、光盘、闪存等外存。为了管理的方便，通常需要将设备上的数据抽象成大小相等的块。根据这些数据块的组织方式不同，文件系统可以分为 FAT、EXT2、NTFS 等。为了让计算机能够识别不同类型的文件系统，在 Linux 中建立了虚拟文件系统（Virtual File System，VFS）的概念。VFS 概括了所有文件系统的基本特征，包括设备文件。

虚拟文件系统与其他部分之间的组织关系如图 2-1 所示。所有的设备通过驱动程序抽象为设备文件。通过虚拟文件系统管理所有的文件。各种具体类型的文件系统，如 FAT、EXT2

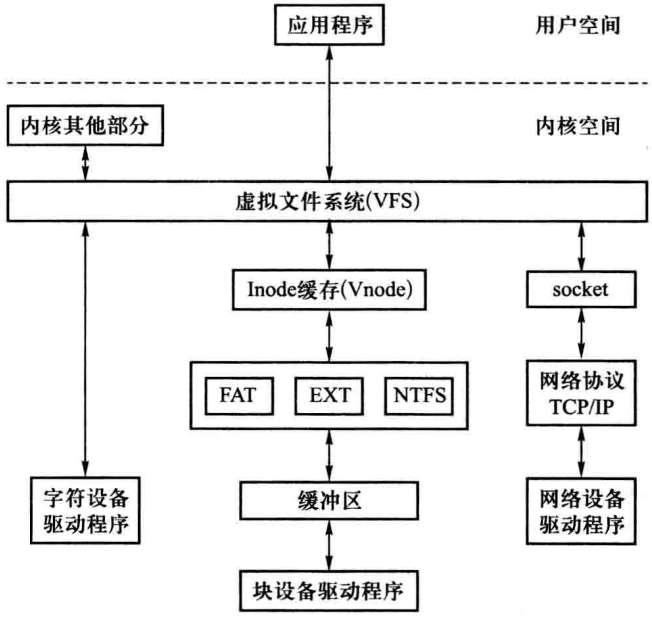

图 2-1　虚拟文件系统与其他部分之间的组织关系

等，需要提供与虚拟文件系统相同的函数接口才可被 Linux 访问。对于块设备文件的访问，操作系统根据不同文件系统类型、相应的操作方法，以及块设备上的 inode 信息生成一个统一结构的虚拟索引结点 vnode，并依据这些信息完成块设备文件的访问。

在虚拟文件系统中提供了文件操作函数接口、管理虚拟文件系统用的数据结构及缓存机制。操作函数接口在结构 file_operations 中提供，包括文件的读、写、定位等，如 D2-1 所示。数据结构主要有 struct super_block（超级块），其中包括了标识不同真实文件系统的 magic number、文件系统的标识（如只读等）、文件系统所在的块设备号、该文件系统本身的 inode 以及所在目录的 inode。根据超级块就可以找到该文件系统的根目录，进而可查找根目录下的子目录与文件。虚拟文件系统的超级块只是定义了一个结构，具体内容根据磁盘上真实文件系统中的超级块的内容进行填写，这项工作是在真实文件系统加载时完成的。

D2-1 file_operations 的结构定义

```
struct file_operations {
        struct module * owner;
        loff_t(* llseek) (struct file * ,loff_t,int);
        ssize_t(* read) (struct file * ,char _user * ,size_t,loff_t *);
        ssize_t(* aio_read) (struct kiocb * ,char _user * ,size_t,loff_t);
        ssize_t(* write) (struct file * ,const char _user * ,size_t,loff_t *);
        ssize_t(* aio_write) (struct kiocb * ,const char _user * ,size_t,loff_t);
        int(* readdir) (struct file * ,void * ,filldir_t);
        unsigned int(* poll) (struct file * ,struct poll_table_struct *);
        int(* ioctl) (struct inode * ,struct file * ,unsigned int,unsigned long);
        long(* unlocked_ioctl) (struct file * ,unsigned int,unsigned long);
        long(* compat_ioctl) (struct file * ,unsigned int,unsigned long);
        int(* mmap) (struct file * ,struct vm_area_struct *);
        int(* open) (struct inode * ,struct file *);
        int(* flush) (struct file *);
        int(* release) (struct inode * ,struct file *);
        int(* fsync) (struct file * ,struct dentry * ,int datasync);
        int(* aio_fsync) (struct kiocb * ,int datasync);
        int(* fasync) (int,struct file * ,int);
        int(* lock) (struct file * ,int,struct file_lock *);
        ssize_t(* readv) (struct file * ,const struct iovec * ,unsigned long,loff_t *);
        ssize_t(* writev) (struct file * ,const struct iovec * ,unsigned long,loff_t *);
        ssize_t(* sendfile) (struct file * ,loff_t * ,size_t,read_actor_t,void *);
        ssize_t(* sendpage) (struct file * ,struct page * ,int,size_t,loff_t * ,int);
        unsigned long(* get_unmapped_area) (struct file * ,unsigned long,unsigned long,unsigned long,unsigned
```