基于 Linux 的 C++

第十二讲　Linux系统编程基础

提　纲

程序执行环境

输入输出

文件系统

设　备

库

makefile文件

程序执行环境

- 参数列表
- 环境变量
- 程序退出码
- 系统调用错误处理
- 资源管理
- 系统日志
- 用户信息

参数列表

Linux命令行规范

- 短参数：以单横开头，后跟单一字符，例：ls -h
- 长参数：以双横开头，后跟字符串，例：ls --help

程序访问参数列表的方法

- 主函数的参数argc和argv
- 程序接受命令行的输入参数，并解释之

参数列表

编写程序，输出命令行参数

```
#include <iostream>
using namespace std;
int main( int argc, char* argv[] )
{
  cout << "The program name is " << argv[0] << "." << endl;
  if( argc > 1 )
  {
    cout << "With " << argc - 1 << " args as follows:" << endl;
    for( int i = 1; i < argc; ++i )
      cout << argv[i] << endl;
  }
  else
    cout << "With " << argc - 1 << " arguments." << endl;
  return 0;
}
```

参数列表

选项数组的定义

- 结构体类型option：系统已定义，直接使用即可

```
// 头文件：“getopt.h”
struct option
{
  // 选项长名称
  const char * name;
  // 该选项是否具有附加参数；0：无；1：有；2：可选
  int has_arg;
  // 指向整数，用于保存val值，设为0
  int * flag;
  // 选项短名称
  int val;
};
```

参数列表

函数getopt_long()

- 函数原型：int getopt_long(int argc, char * const * argv, const char * short_options, const struct option * long_options, int * long_index);
- 函数返回值为参数短名称，不存在时返回-1
- 如果为长选项，第五个参数输出该选项在长选项数组中的索引

参数处理方法

- 使用循环处理所有参数

参数列表

参数处理方法（续）

- 如果遇到错误选项，输出错误消息并终止程序执行
- 处理附加参数时，用全局变量optarg传递其地址
- 完成所有处理后，全局变量optind为首个非可选参数的索引

编写程序，接受如下三个选项并执行正确操作

- -h / --help：显示程序用法并退出
- -o filename / --output filename：指定文件名
- -v / --verbose：输出复杂信息

参数列表

```
#include <iostream>
#include <cstdlib>
#include <getopt.h>

using namespace std;

const char * program_name;

// 输出程序用法
void OutputInfo( ostream & os, int exit_code )
{
  os << "Usage: " << program_name << " options [filename]" << endl;
  os << " -h --help: Display this usage information." << endl;
  os << " -o --output filename: Write output to file." << endl;
  os << " -v --verbose: Print verbose messages." << endl;
  exit( exit_code );
}
```

参数列表

```
int main( int argc, char* argv[] )
{
  //  全部短选项的合并字符串，  “:”  表示带有附加参数
  const char * const short_opts = "ho:v";
  const struct option long_opts[] =
  {
    { "help", 0, NULL, 'h' },
    { "output", 1, NULL, 'o' },
    { "verbose", 0, NULL, 'v' },
    { NULL, 0, NULL, 0 }
  };
  //  参数指定的输出文件名
  const char * output_filename = NULL;
  //  是否显示复杂信息
  int verbose = 0;
  //  保存程序名
  program_name = argv[0];
  //  如果为长选项，第五个参数输出该选项在长选项数组中的索引
  int opt = getopt_long( argc, argv, short_opts, long_opts, NULL );
```

参数列表

```
while( opt != -1 ) {
  switch( opt ) {
  case 'h':           //   “-h” 或 “--help”
    OutputInfo( cout, 0 );
  case 'o':           //   “-o” 或 “--output” ，附加参数由optarg提供
    output_filename = optarg;     break;
  case 'v':           //   “-v” 或 “--verbose”
    verbose = 1;     break;
  case '?':           // 用户输入了无效参数
    OutputInfo( cerr, 1 );
  case -1:            // 处理完毕
    break;
  default:            // 未知错误
    abort();
  }
  opt = getopt_long( argc, argv, short_opts, long_opts, NULL );
}
return 0;
}
```

环境变量

典型Linux环境变量

- USER：你的用户名
- HOME：你的主目录
- PATH：分号分隔的Linux查找命令的目录列表

shell处理

- shell编程时查看环境变量：echo $USER
- 设置新的环境变量：EDITOR=emacs; export EDITOR或export EDITOR=emacs

环境变量

环境变量内部定义格式：VARIABLE=value

使用getenv()函数返回环境变量的值

使用全局变量environ处理环境变量

```
#include <iostream>
using namespace std;
extern char ** environ;
int main()
{
  char ** var;
  for( var = environ; *var != NULL; ++var )
    cout << *var << endl;
  return 0;
}
```

环境变量

编写客户端程序，在用户未指定服务器名时使用缺省服务器名称

```
#include <iostream>
#include <cstdlib>

int main ()
{
  char * server_name = getenv( "SERVER_NAME" );
  if( !server_name )
    //  SERVER_NAME环境变量未设置，使用缺省值
    server_name = "server.yours.com";
  cout << "accessing server" << server_name << endl;
  //  ......
  return 0;
}
```

程序退出码

程序：结束时传递给操作系统的整型数据

- 实际上是main()函数的返回值
- 其他函数也可以调用exit()函数返回特定退出码
- 退出码的变量名称经常为exit_code
- 应仔细设计程序退出码，确保它们能够区分不同错误

操作系统：响应程序退出码，如果必要，执行后续处理

- shell编程时查看上一次退出码的命令：echo $?

系统调用错误处理

实现逻辑

- C程序使用断言，C++程序使用断言或异常处理机制

两个主要问题

- 系统调用：访问系统资源的手段
- 系统调用失败原因：资源不足；因权限不足而被阻塞；调用参数无效，如无效内存地址或文件描述符；被外部事件中断；不可预计的外部原因
- 资源管理：已分配资源必须在任何情况下都能正确释放

系统调用错误处理

Linux使用整数表示系统调用错误

- 标准错误码为以“E”开头的全大写宏
- 宏errno（使用方法类似全局变量）：表示错误码，位于头文件“errno.h”中
- 每次错误都重写该值，处理错误时必须保留其副本
- 函数strerror()：返回宏errno对应的错误说明字符串，位于头文件“string.h”中

系统调用错误处理

```
// 将指定文件的拥有者改为指定的用户或组；第一个参数为文件名，
// 第二和第三个参数分别为用户和组id，-1表示不改变
rval = chown( path, user_id, -1 );
if( rval )
{
  // 必须存储errno，因为下一次系统调用会修改该值
  int error_code = errno;
  // 操作不成功，chown将返回-1
  assert( rval == -1 );
  // 检查errno，进行对应处理
  switch( error_code )
  {
  case EPERM:                 // 操作被否决
  case EROFS:                 // PATH位于只读文件系统中
  case ENAMETOOLONG:          // 文件名太长
  case ENOENT:                // 文件不存在
  case ENOTDIR:               // path的某个成分不是目录
```

系统调用错误处理

```
  case EACCES:                    //  path的某个成分不可访问
    cerr << "error when trying to change the ownership of " << path;
    cerr << ": “ << strerror( error_code ) << endl;
    break;
  case EFAULT:                    //  path包含无效内存地址，有可能为bug
    abort ();
  case ENOMEM:                    //  核心内存不足
    cerr << strerror( error_code ) << endl;
    exit( 1 );
  default:                        //  不可预见错误，最可能为程序bug
    abort ();
  };
}
```

资源管理

必须予以明确管理的资源类型

- 内存、文件描述符、文件指针、临时文件、同步对象等

资源管理流程

- 步骤1：分配资源
- 步骤2：正常处理流程
- 步骤3：如果流程失败，释放资源并退出，否则执行正常处理流程
- 步骤4：释放资源
- 步骤5：函数返回

资源管理

```
char * ReadFromFile( const char * filename, size_t length )
{
  char * buffer = new char[length];
  if( !buffer )
    return NULL;
  int fd = open( filename, O_RDONLY );    //  以只读模式打开文件
  if( fd == -1 )  {
    delete[] buffer,  buffer = NULL;
    return NULL;
  }
  size_t bytes_read = read( fd, buffer, length );
  if( bytes_read != length )  {
    delete[] buffer,  buffer = NULL;
    close( fd );
    return NULL;
  }
  close( fd );
  return buffer;
}
```

系统日志

日志：系统或程序运行的记录

系统日志进程：syslogd/rsyslogd

- 两者均为守护（daemon）进程，即在后台运行的进程，没有控制终端，也不会接收用户输入，父进程通常为init进程
- 日志文件一般为“/dev/log”，日志信息一般保存在“/var/log/”目录下
- rsyslogd既能接收用户进程输出的日志，也能接收内核日志；在接收到日志信息后，会输出到特定的日志文件中；日志信息的分发可配置

系统日志

日志生成函数：syslog()

- 头文件：“syslog.h”
- 原型：void syslog(int priority, const char * msg, ...);
- 可变参数列表，用于结构化输出
- priority：日志优先级，设施值（一般默认为LOG_USER）与日志级别的位或
- 日志级别：LOG_EMERG（0，系统不可用）、LOG_ALERT（1，报警，需立即采取行动）、LOG_CRIT（2，严重情况）、LOG_ERR（3，错误）、LOG_WARNING（4，警告）、LOG_NOTICE（5，通知）、LOG_INFO（6，信息）、LOG_DEBUG（7，调试）

系统日志

日志打开函数：openlog()

- 原型：void openlog(const char * ident, int logopt, int facility);
- 改变syslog()函数的默认输出方式，以进一步结构化日志内容
- ident：标志项，指定添加到日志消息的日期和时间后的字符串
- logopt：日志选项，用于配置syslog()函数的行为，取值为LOG_PID（在日志消息中包含程序PID）、LOG_CONS（如果日志不能记录至日志文件，则打印到终端）、LOG_ODELAY（延迟打开日志功能，直到第一次调用syslog()函数）、LOG_NDELAY（不延迟打开日志功能）的位或
- facility：用于修改syslog()函数的默认设施值，一般维持LOG_USER不变

系统日志

日志过滤函数：setlogmask()

- 原型：int setlogmask(int maskpri);
- 设置日志掩码，大于maskpri的日志级别信息被过滤
- 返回值：设置日志掩码前的日志掩码旧值

日志关闭函数：closelog()

- 原型：void closelog();

用户信息

UID、EUID、GID和EGID

- 每个进程拥有两个用户ID：UID（真实用户ID）和EUID（有效用户ID）
- EUID的目的：方便资源访问，运行程序的用户拥有该程序有效用户的权限
- 组与用户类似

用户信息处理函数

- 获取真实用户ID：uid_t getuid();
- 获取有效用户ID：uid_t geteuid();
- 获取真实组ID：gid_t getgid();
- 获取有效组ID：gid_t getegid();

用户信息

用户信息处理函数（续）

- 设置真实用户ID：int setuid(uid_t uid);
- 设置有效用户ID：int seteuid(uid_t uid);
- 设置真实组ID：int setgid(gid_t gid);
- 设置有效组ID：int setegid(gid_t gid);

程序示例

```
#include <unistd.h>
#include <stdio.h>
int main()
{
  uid_t uid = getuid(),  euid = geteuid();
  printf("uid: %d; euid: %d\n", uid, euid );
  return 0;
}
```

用户信息

```
qiaolin@Kylin: ~/C++/uid
qiaolin@Kylin:~$ cd C++
qiaolin@Kylin:~/C++$ cd uid
qiaolin@Kylin:~/C++/uid$ ls
main.c
qiaolin@Kylin:~/C++/uid$ gcc main.c
qiaolin@Kylin:~/C++/uid$ ls
a.out  main.c
qiaolin@Kylin:~/C++/uid$ ls -l
总用量 16
-rwxrwxr-x 1 qiaolin qiaolin 8657 11月 17 12:51 a.out
-rw-rw-r-- 1 qiaolin qiaolin  162 11月 17 12:51 main.c
qiaolin@Kylin:~/C++/uid$ ./a.out
uid: 1000; euid: 1000
qiaolin@Kylin:~/C++/uid$ sudo chown root:root ./a.out
[sudo] password for qiaolin:
qiaolin@Kylin:~/C++/uid$ ls -l
总用量 16
-rwxrwxr-x 1 root    root    8657 11月 17 12:51 a.out
-rw-rw-r-- 1 qiaolin qiaolin  162 11月 17 12:51 main.c
qiaolin@Kylin:~/C++/uid$ sudo chmod +s ./a.out
qiaolin@Kylin:~/C++/uid$ ls -l
总用量 16
-rwsrwsr-x 1 root    root    8657 11月 17 12:51 a.out
-rw-rw-r-- 1 qiaolin qiaolin  162 11月 17 12:51 main.c
qiaolin@Kylin:~/C++/uid$ ./a.out
uid: 1000; euid: 0
qiaolin@Kylin:~/C++/uid$
```

输入输出

- 标准输入输出流
- 文件描述符
- I/O函数
- 临时文件

标准输入输出流

标准输入流：stdin/cin

标准输出流：stdout/cout

- 数据有缓冲，在缓冲区满、程序正常退出、流被关闭或强制刷新（fflush()函数）时输出
- 等到缓冲区满后同时打印多个句号：while(1) { printf("."); sleep(1); }

标准错误流：stderr/cerr

- 数据无缓冲，直接输出
- 每秒打印一个句号：while(1) { fprintf(stderr, "."); sleep(1); }

文件描述符

文件描述符的意义与目的：在程序中代表文件

- 内核为每个进程维护一个文件打开记录表，文件描述符为该文件在表中的索引值

文件描述符为非负整数，范围从0至OPEN_MAX

- 不同操作系统可能具有不同范围，可以同时打开的文件数目不同

文件描述符的缺点

- 非UNIX/Linux操作系统可能没有文件描述符概念，跨平台编程时建议使用C/C++标准库函数和文件流类

文件描述符

预定义的标准输入输出流的文件描述符

- 标准输入流stdin：STDIN_FILENO（0）
- 标准输出流stdout：STDOUT_FILENO（1）
- 标准错误流stderr：STDERR_FILENO（2）

文件描述符的创建

- Linux中凡物皆文件，操作系统使用统一方式管理和维护
- 很多函数都可通过打开文件或设备的方式创建文件描述符

I/O函数

基本与高级I/O函数

- 打开关闭函数open()和close()：前者头文件“fcntl.h”，后者头文件“unistd.h”
- 读写函数read()和write()：头文件“unistd.h”
- 读写函数readv()和writev()：头文件“sys/uio.h”
- 文件发送函数sendfile()：头文件“sys/sendfile.h”
- 数据移动函数splice()：头文件“fcntl.h”
- 数据移动函数tee()：头文件“fcntl.h”
- 文件控制函数fcntl()：头文件“fcntl.h”

I/O函数

打开文件函数open()

- 原型：int open(const char * filename, int oflag, ...);
- 目的：打开filename指定的文件，返回其文件描述符，oflag为文件打开标志
- 若文件支持定位，读取时从当前文件偏移量处开始
- 文件打开标志：O_RDONLY（只读）、 O_WRONLY（只写）、 O_RDWR（读写）等

关闭文件函数close()

- 原型：int close(int fd);
- 目的：关闭文件描述符fd所代表的文件

I/O函数

读函数read()

- 原型：ssize_t read(int fd, void * buf, size_t count);
- 目的：将count个字节的数据从文件描述符fd所代表的文件中读入buf所指向的缓冲区
- 若文件支持定位，读取时从当前文件偏移量处开始
- 返回值：读取的字节数，0表示文件结尾，失败时返回-1并设置errno

写函数write()

- 原型：ssize_t write(int fd, const void * buf, size_t count);
- 目的：将count个字节的数据从buf所指向的缓冲区写入文件描述符fd所代表的文件中
- 参数与返回值的意义与read()相同或类似

I/O函数

分散读函数readv()

- 原型：ssize_t readv(int fd, const struct iovec * iov, int iovcnt);
- 目的：将数据从文件描述符所代表的文件中读到分散的内存块中
- 参数：fd为文件描述符；iov为写入的内存块结构体数组，每个数组元素只有内存基地址iov_base和内存块长度iov_len两个字段；iovcnt为读取的元素个数
- 返回值：读取的内存块数，失败时返回-1并设置errno

集中写函数writev()

- 原型：ssize_t writev(int fd, const struct iovec * iov, int iovcnt);
- 目的：将数据从分散的内存块中写入文件描述符所代表的文件中
- 参数与返回值的意义与readv()相同或类似

I/O函数

文件发送函数sendfile()

- 原型：ssize_t sendfile(int out_fd, int in_fd, off_t * offset, int count);
- 目的：在两个文件描述符所代表的文件间直接传递数据，以避免内核缓冲区和用户缓冲区之间的数据拷贝，提升程序效率；为网络文件传输而专门设计的函数
- 参数：out_fd为目的文件描述符；in_fd为源文件描述符；offset指定读取时的偏移量，为NULL表示从默认位置开始读取；count为传输的字节数
- 返回值：传输的字节数，失败时返回-1并设置errno

注意事项

- in_fd必须为支持类似mmap()函数的文件描述符，即必须代表真实的文件，不能为套接字和管道；out_fd必须为套接字

I/O函数

数据移动函数splice()

- 原型：ssize_t splice(int fd_in, loff_t * off_in, int fd_out, loff_t * off_out, ssize_t len, unsigned int flags);
- 目的：在两个文件描述符所代表的文件间移动数据
- 参数：fd_in为源文件描述符；off_in为输入数据偏移量，若fd_in为管道，则off_in必须设置为NULL；fd_out与off_out的意义类似；len为传输的字节数；flags控制数据如何移动，其取值为SPLICE_F_MOVE（新内核无效果）、SPLICE_F_NONBLOCK（非阻塞）、SPLICE_F_MORE（还有后续数据）和SPLICE_F_GIFT（无效果）的位或
- 返回值：传输的字节数，0表示无数据移动，失败时返回-1并设置errno

注意事项

- fd_in和fd_out必须至少有一个为管道文件描述符

I/O函数

数据移动函数tee()

- 原型：ssize_t tee(int fd_in, int fd_out, ssize_t len, unsigned int flags);
- 目的：在两个文件描述符所代表的管道间移动数据
- 参数：含义与splice()相同
- 返回值：传输的字节数，0表示无数据移动，失败时返回-1并设置errno

注意事项

- fd_in和fd_out必须为管道文件描述符

I/O函数

文件控制函数fcntl()

- 原型：int fcntl(int fd, int cmd, ...);
- 目的：对文件描述符所代表的文件或设备进行控制操作
- 参数： fd为文件描述符；cmd为控制命令
- 返回值：失败时返回-1并设置errno

常用操作

- 复制文件描述符：F_DUPFD/F_DUPFD_CLOEXEC，第三个参数型式long，成功时返回新创建的文件描述符
- 获取或设置文件描述符的标志：F_GETFD/F_SETFD，第三个参数前者无，后者型式long，成功时前者返回fd的标志，后者0

I/O函数

常用操作（续）

- 获取或设置文件描述符状态标志：F_GETFL/F_SETFL，第三个参数前者无，后者型式long，成功时前者返回fd的状态标志，后者0
- 获取或设置SIGIO和SIGURG信号的宿主进程PID或进程组的GID：F_GETOWN/F_SETOWN，第三个参数前者无，后者型式long，成功时前者返回信号的宿主进程的PID或进程组的GID，后者0
- 获取或设置信号：F_GETSIG/F_SETSIG，第三个参数前者无，后者型式long，成功时前者返回信号值，0表示SIGIO，后者0
- 获取或设置管道容量：F_GETPIPE_SZ/F_SETPIPE_SZ，第三个参数前者无，后者型式long，成功时前者返回管道容量，后者0

临时文件

使用临时文件时的注意事项

- 程序多个进程可能同时运行，它们可能应该使用不同的临时文件
- 必须小心设置文件属性，未授权用户不应具有临时文件访问权限
- 临时文件的生成应该外部不可预测，否则系统容易受到攻击

Linux临时文件函数mkstemp()

- 创建名称唯一的临时文件，使用“XXXXXX”作为模板，返回文件描述符
- 如果不希望外界看到临时文件，创建临时文件后应调用unlink()函数将其从目录项中删除，但文件本身仍存在
- 文件采用引用计数方式访问；本程序未结束，可用文件描述符访问该文件；文件引用计数降为0，系统自动删除临时文件

临时文件

```
#include <cstdlib>
#include <cstring>
#include <unistd.h>

// 向临时文件中写入数据
int  WriteToTempFile( char * buffer, size_t length )
{
  // 创建临时文件，“XXXXXX”将在生成时被替换，以保证文件名唯一性
  char temp_filename[] = "/tmp/temp_file.XXXXXX";
  int fd = mkstemp( temp_filename );
  // 取消文件链接，不显示该临时文件；关闭文件描述符后，临时文件被删除
  unlink( temp_filename );
  // 向临时文件中写入数据
  write( fd, &length, sizeof(length) );
  write( fd, buffer, length );
  // 返回临时文件的文件描述符
  return fd;
}
```

临时文件

```
// 从临时文件中读取数据
char * ReadFromTempFile( int fd, size_t * length )
{
  // 定位到文件开头
  lseek( fd, 0, SEEK_SET );
  // 读取数据
  read( fd, length, sizeof(*length) );
  char * buffer = new char[*length];
  read( fd, buffer, *length );
  // 关闭文件描述符，临时文件将被删除
  close( fd );
  return buffer;
}
```

文件系统

实际文件系统

- **ext、ext2、ext3、ext4**

虚拟文件系统VFS

特殊文件系统/proc

- **从/proc文件系统中抽取信息**

文件系统

实际文件系统：组成与功能描述

- 引导块、超级块、索引结点区、数据区
- 引导块：在文件系统开头，通常为一个扇区，存放引导程序，用于读入并启动操作系统
- 超级块：用于记录文件系统的管理信息，不同的文件系统拥有不同的超级块
- 索引结点区：一个文件或目录占据一个索引结点，首索引结点为该文件系统的根结点，可以利用根结点将一个文件系统挂在另一个文件系统的非叶结点上
- 数据区：用于存放文件数据或者管理数据

文件系统

虚拟文件系统VFS

- VFS的特点：只存于内存中，充当实际文件系统与操作系统之间的接口，提供实际文件系统的挂载，并管理实际文件系统
- VFS的构造：系统初始化时构造VFS目录树，建立其数据结构；每个实际文件系统使用struct file_system_type结构存储为结点，并形成链表
- VFS的意义与目的： 支持多种不同的文件系统，内核以一致的方式处理这些文件系统，从而对用户透明

文件系统

特殊文件系统/proc

- Linux内核的窗口，只存于内存中，并不占用磁盘空间

典型信息

- 进程信息：进程项、进程参数列表、进程环境、进程可执行文件、进程文件描述符、进程内存统计信息等
- 硬件信息：CPU信息、设备信息、PCI总线信息、串口信息等
- 内核信息：版本信息、主机名与域名信息、内存使用等
- 设备、挂载点与文件系统

设　备

- 设备类型
- 设备号
- 设备项
- 设备目录
- 硬件设备
- 特殊设备
- 设备控制与访问

设备类型

设备文件的性质

- 设备文件不是普通的磁盘文件
- 读写设备的数据需要与相应的设备驱动器通信

设备文件的类型

- 字符设备：读写串行数据字节流，如串口、终端等
- 块设备：随机读写固定尺寸数据块，如磁盘设备

说　明

- 磁盘挂载到文件系统后，使用文件和目录模式操作
- 程序一般不用块设备，内核实现文件系统时使用块设备操作文件

设备号

大设备号（major device number）

- 指定设备对应哪个设备驱动器
- 对应关系由内核确定

小设备号（ minor device number ）

- 区分由设备驱动器控制的单个设备或设备的某个组件

示　例

- 3号主设备为IDE控制器，IDE控制器可以连接多个设备（磁盘、磁带、CD-DVD驱动器等）
- 主设备的小设备号为0，而从设备的小设备号为64
- 主设备单独分区的小设备号从0至63，从设备单独分区的小设备号从64开始

设备项

设备项：与文件类似

- 可以使用mv、rm命令移动或删除
- 如果设备支持读写，cp命令可以从（向）设备读取（写入）数据

mknod系统调用：创建设备项（文件系统结点）

- 原型：int mknod(const char * pathname, mode_t mode, dev_t dev);
- 参数：pathname为设备项包含路径的名称；mode为设备的使用权限与结点类型；当文件类型为S_IFCHR或S_IFBLK时，dev表示设备的大小设备号，否则忽略
- 设备项仅仅是与设备通信的门户，在文件系统中创建设备项并不意味着设备可用
- 只有超级用户才可以创建设备项

设备目录

操作系统已知的设备目录：/dev

示　例

- 硬盘hda为块设备
- 硬盘有一个分区hda1

```
% ls  –l  /dev/hda  /dev/hda1
brw-rw----  1 root   disk  3, 0  Jul 20 2011  /dev/hda
brw-rw----  1 root   disk  3, 1  Jul 20 2011  /dev/hda1
```

硬件设备

设备描述	设备名称	大设备号	小设备号
第一软驱	/dev/fd0	2	0
第二软驱	/dev/fd1	2	1
主IDE控制器，主设备	/dev/hda	3	0
主IDE控制器，主设备，第一分区	/dev/hda1	3	1
主IDE控制器，从设备	/dev/hdb	3	64
主IDE控制器，从设备，第一分区	/dev/hdb1	3	65
次IDE控制器，主设备	/dev/hdc	22	0
次IDE控制器，从设备	/edv/hdd	22	64
第一SCSI设备	/dev/sda	8	0
第一SCSI设备，第一分区	/dev/sda1	8	1
第一SCSI CD - ROM驱动器	/dev/scd0	11	0
第二SCSI CD - ROM驱动器	/dev/scd1	11	1

硬件设备

设备描述	设备名称	大设备号	小设备号
并口0	/dev/lp0或/dev/par0	6	0
并口1	/dev/lp1或/dev/par1	6	1
第一串口	/dev/ttyS0	4	64
第二串口	/dev/ttyS1	4	65
IDE磁带设备	/dev/ht0	37	0
第一SCSI磁带设备	/dev/st0	9	0
第二SCSI磁带设备	/dev/st1	9	1
系统控制台	/dev/console	5	1
第一虚拟终端设备	/edv/tty1	4	1
第二虚拟终端设备	/dev/tty2	4	2
进程当前终端设备	/dev/tty	5	0
声卡	/dev/audio	14	5

特殊设备

/dev/null：哑设备

- 任何写入哑设备的数据被抛弃
- 从哑设备读取不到任何数据，例如cp /dev/null empty-file 命令将创建一个长度为0的空文件

/dev/zero：零设备

- 行为类似文件，长度无限，但内容全部为0

/dev/full：满设备

- 行为类似文件，没有空闲空间存储任何数据
- 对满设备的写入总是失败，并将errno设为ENOSPC

随机数设备

/dev/random和/dev/urandom：随机数设备

- C语言的rand()函数生成伪随机数

随机数设备原理

- 人的行为无法预测，因而是随机的
- Linux内核测量用户的输入活动（如键盘输入和鼠标操作）的时间延迟作为随机数

两者区别

- /dev/random：在用户没有输入操作时，阻塞随机数读取进程（没有数据可读取）
- /dev/urandom：永不阻塞；在用户没有输入操作时，生成伪随机数代替

设备访问与控制

设备访问

- 像文件一样操作设备
- 示例：向并口设备发送数据

```
int fd = open( "/dev/lp0", O_WRONLY );
write( fd, buffer, buffer_length );
close( fd );
```

设备访问与控制

控制硬件设备的函数： ioctl()

- 第一个参数为文件描述符，指定想要控制的设备；第二个参数为控制命令码，指定想要实施的操作

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  int fd = open( argv[1], O_RDONLY );          // 打开参数所表示的设备
  ioctl( fd, CDROMEJECT );                     // 弹出CD-ROM
  close( fd );
  return 0;
}
```

库

静态库（Archives）

- 后缀一般为“*.a”
- 使用两个目标文件创建单一静态库的编译与链接命令：ar cr libtest.a test1.o test2.o
- 链接器搜索静态库时，链接所有已引用而未处理的符号
- 将静态库的链接放置在命令行尾部，确保其引用被正确解析

动态库（Shared Object）

- 共享目标库（类似Windows的DLL），后缀一般为“*.so”
- 编译命令：g++ -shared -fPIC -o libtest.so test1.o test2.o
- PIC：位置无关代码（Position-Independent Code）
- 编译器首先链接动态库，其次才是静态库
- 如果要强制链接静态库，编译使用-static选项

标准库与库相关性

C标准库：libc

- 数学库单独：libm；需要调用数学函数时，显式链接数学库：g++ -o compute compute.c –lm

C++标准库：libstdc++

- 编译C++11程序，使用g++-4.8 -std=c++11；对于Code::Blocks等集成开发环境，在编译器设置对话框中选中相应的C++11选项

库的相关性

- 链接时需要注意交叉引用被正确解析，例如：libtiff库需要libjpeg库（jpeg图像处理）和libz库（压缩处理）
- 独立库链接： g++ -static -o tifftest tifftest.c -ltiff -ljpeg –lz
- 相关库链接： g++ -o app app.o -la -lb -la

动态库的装载与卸载

动态库装载函数dlopen()：头文件“dlfcn.h”

- 原型：void * dlopen(const char * filename, int flag);
- 参数：filename为动态库名称；flag为装载模式，必须为RTLD_LAZY或RTLD_NOW两者之一，并可与其他装载标识（如RTLD_GLOBAL、RTLD_LOCAL）组合
- 返回值：类型为void *，用以表示动态库句柄；调用失败返回NULL
- 示例：dlopen("libtest.so", RTLD_LAZY);

函数查找与装载函数dlsym()

- 原型：void * dlsym(void * handle, const char * symbol);
- 参数：handle为动态库句柄；symbol为函数名称字符串
- 返回值：目标函数装载在内存中的基地址

动态库的装载与卸载

动态库卸载函数dlclose()

- 原型：int dlclose(void * handle);
- 参数：handle为动态库句柄
- 返回值：成功时为0，其他为错误

动态库错误处理函数dlerror()

- 原型：char * dlerror();
- 返回值：其他三个函数调用时最后一次产生的错误描述字符串

动态库的装载与卸载

调用动态库中的函数，设函数名为g

- 混合C/C++编码时，C函数应封装于extern "C" { … } 块中，确保名解析正确工作（C不支持函数重载）
- 链接选项：“-ldl”

```
void * handle = dlopen( "libtest.so", RTLD_LAZY );
//  声明函数指针指向动态库中的函数，按被调函数的名称查找
void ( *test )() = dlsym( handle, "g" );
( *test )();    //  使用函数指针调用动态库中的函数
dlclose( handle );
```

makefile

make命令：负责C/C++程序编译与链接

- make根据指定命令进行建构
- 建构规则文件：GNUmakefile、makefile、Makefile

makefile文件格式

makefile语法

- 基本语法、变量、条件判断、循环、函数

makefile文件格式

makefile文件基本格式

```
target ... : prerequisites ...
[Tab键]        commands
```

makefile文件规则

- makefile文件由一系列规则构成
- 规则的目的：建构目标的先决条件是什么以及如何建构目标
- 如果未指定目标，缺省执行第一个目标
- 若prerequisites中有一个以上的文件比target文件要新，执行commands所定义的命令

makefile文件格式

target：目标

- 通常为编译期的文件名，以指定要建构的对象，也可以是执行文件，还可以是标签（操作名称，伪目标）
- 可以为单一目标，也可以为空格分隔的多个目标
- 每个目标都定义了一组处理规则，和其相关规则构成规则链

prerequisites：先决条件

- 为生成该目标所需的先决文件或目标（前置条件）
- 一般为空格分隔的文件名，指定目标是否重建的判断标准，即只要有一个先决文件不存在或有过更新，就重建目标

makefile文件格式

prerequisites：先决条件（续）

- 若目标先决条件本身需要重建，则匹配该先决条件的目标，执行其对应的命令

commands：命令

- 由一行或多行shell命令组成，命令前有Tab键
- 指示如何建构目标，一般为生成目标文件
- 每行命令都在单独的进程中执行，彼此没有继承关系，不能简单传递数据；解决办法：用分号将多条命令书写在单行（此时可用“\”折行），或者为该条规则添加指示“.ONESHELL :”

makefile文件格式

伪目标：操作名称，而不是文件名

- **删除编译后的二进制目标文件，例如：**

```
clean :
        rm -f *.o
```

- **执行命令时须指定伪目标：$ make clean**
- **若当前目录下有clean文件，则此规则不会被执行；此时可用“.PHONY : clean”明确指示clean为伪目标；make将跳过文件检查，执行其对应的命令**
- **执行清除任务的伪目标一般放置在脚本的最后**

makefile文件格式

伪目标惯例

- all：所有目标的目标，一般为编译所有的目标，对同时编译多个程序极为有用
- clean：删除由make创建的文件
- install：安装已编译好的程序，主要任务是完成目标执行文件的拷贝
- print：列出改变过的源文件
- tar：打包备份源程序，形成tar文件
- dist：创建压缩文件，一般将tar文件压缩成Z文件或gz文件
- TAGS：更新所有的目标，以备完整地重编译使用
- check和test：一般用来测试makefile的流程

makefile文件格式

示例：假设程序主文件“main.c”，使用library库

```
# 注释行
prog : main.o library.o
    cc -o prog main.o library.o
main.o : main.c library.h
    cc -c main.c
library.o : library.c library.h
    cc -c library.c
.PHONY : clean
clean :
    rm main.o library.o
```

makefile文件语法

行解析：命令按行解析

- 命令行的行首字符为Tab键，其他行的行首字符不得为Tab键，但可以使用多个空格缩进

换行：命令太长时，行尾用“\”换行

注释：行首字符为“#”的文本行

关闭回显：在行首字符后和命令前添加“@”

- 未关闭回显时，make会首先回显（打印）命令，然后执行该命令
- 通常仅在注释和纯显示的echo命令前使用此功能

include filename：包含其他文件

- 处理模式与C/C++类似
- 行首加“-”：忽略文件包含错误

makefile文件语法

通配符

- “*”（任意数目的任意字符），例如“*.c”表示所有C源文件
- “？”（任意一个字符），例如“?.c”表示所有单字符文件名的C源文件
- “[abc]”（存在括号内的某个字符），例如“lib[abc].c”表示第四个字符为“a”、“b”或“c”
- “[0-9]”（存在该集合中的某个字符），例如“lib[0-9].c”表示第四个字符为0～9之间的数字（含数字0和9）
- “[^abc]”（存在非括号内的某个字符），例如“lib[^abc].c”表示第四个字符不是“a”、“b”或“c”

makefile文件语法

变　量

- 基本变量定义： var_name = value
- $(变量名称)：引用变量（中间无多余空格）；shell变量用“$$”，例如“@echo $$HOME”
- 变量在使用时展开，形式上类似宏替换
- 变量的使用场合：目标、先决条件、命令、新的变量

内置变量

- $(CC)：当前使用的编译器；$(MAKE)：当前使用的make工具

自动变量

- $@：当前目标；$<：当前目标的首个先决条件；$?：比目标更新的所有先决条件；$^：所有先决条件；$(@D)和$(@F)：$@的目录名和文件名；$(<D)和$(<F)：$<的目录名和文件名

makefile文件语法

```
#  makefile样本
objs = main.o library.o

prog : $(objs)
    $(CC) -o prog $(objs)
    @echo "Constructed..."
main.o : main.c library.h
    $(CC) -c main.c
library.o : library.c library.h
    $(CC) -c library.c

.PHONY : clean
clean :
    rm -f prog $(objs) *~
```

makefile文件语法

变量定义格式

- var_name = value：在执行时扩展，允许递归，可以使用后续代码中出现的值
- var_name := value：在定义时扩展，不允许递归，使用右侧的现值，不能使用后续代码中出现的值
- var_name ?= value：只有在该变量为空时才设置值，否则维持原值
- var_name += value：将值追加到变量的尾部；若变量未定义，则“+=”自动解释为“=”；若变量已定义，则“+=”继承上次的操作符，并追加新值

makefile文件语法

多行变量

```
define var_name
        @echo "One"
        @echo "Two"
endef
```

- define和endef行首字符不能为Tab键，对齐时可使用空格
- 引用：$(var_name)
- 多行变量主要用于定义命令包，使用多行变量要小心，展开时有可能导致脚本错误

目标变量：类似局部变量，仅对本目标规则链有效

- target ... : var_name = value：定义目标变量

makefile文件语法

静态模式：以“%”通配

```
target ... : target-pattern : prerequisites ...
[Tab键]commands
```

- 目的：用于处理模式相同的多目标，简化脚本代码
- 示例：每个目标的文件以“.o”结尾，先决文件为对应的“.c”

```
objs = main.o library.o
$(objs) : %.o : %.c
        $(CC) -c $(CFLAGS) $< -o $@

main.o : main.c
        $(CC) -c $(CFLAGS) main.c -o main.o
library.o : library.c
        $(CC) -c $(CFLAGS) library.c -o library.o
```

makefile文件语法

条件判断基本格式

```
conditional-directive
        text-if-true
endif
```

```
conditional-directive
        text-if-true
else
        text-if-false
endif
```

可用的条件判断

- 判断两个参数是否相等：ifeq (arg1,arg2)、 ifeq 'arg1' 'arg2'、 ifeq "arg1" "arg2"
- 判断两个参数是否不等：ifneq（具体格式与ifeq相同）
- 判断某个变量是否已定义：ifdef variable_name
- 判断某个变量是否未定义：ifndef variable_name

makefile文件语法

循环：可以在makefile中使用shell循环

```
rulefor :
        for filename in `echo $(objs)`; \
        do \
                rm -f $$filename; \
        done
```

注意事项

- 循环为shell循环，为保证多行命令在同一个进程下执行，必须合并成单条命令并在行尾添加分行标识
- 可以使用反引号执行命令，所获得的结果集合可以作为循环的处理集合
- filename本身是shell变量，需使用“$$”引用

makefile文件语法

函数：像变量一样使用“$()”标识

- $(function arg1,arg2,...)：函数调用，函数名为function，后跟逗号分隔的参数列表，函数参数前后不能有多余的空格
- $(subst from,to,text) ：make的字符串替换函数，将text中的from字符串替换为to，返回替换后的字符串

```
comma :=,
#  定义空值
empty :=
#  定义空格
space := $(empty) $(empty)
foo := a b c
#  将“a b c”替换为“a,b,c”
bar := $(subst $(space),$(comma),$(foo))
```

编程实践

12.1 编写程序，通过随机数设备读取随机数，重新完成习题6.3及其后续习题7.2。

12.2 编写程序，测试临时文件的读写访问。

12.3 编写程序，查看CPU信息和进程信息。信息越全越好，本题将作为课程大作业“Web服务器”的服务。

12.4 为习题12.3编写makefile文件。