Exam2-要背的

1. 课内部分

1. Shell是用户与操作系统的接口，作为核外程序存在。具有双重角色：
    1. Shell是一种命令解释程序：Linux的开发启动过程(进程树)、Shell的工作步骤：打印提示符；得到命令行；解析命令；查找文件；准备参数；执行命令。
    2. Shell还是一种独立的程序设计语言解释器。
2. 静态库和动态库
    1. 二者的不同点在于代码被载入的时刻不同。
        1. 静态库的代码在编译过程中已经被载入可执行程序，因此体积比较大。
        2. 动态库(共享库)的代码在可执行程序运行时才载入内存，在编译过程中仅简单的引用，因此代码体积比较小。
        3. 不同的应用程序如果调用相同的库，那么在内存中只需要有一份该动态库(共享库)的实例。
        4. 静态库和动态库的最大区别：静态情况下，把库直接加载到程序中，而动态库链接的时候，它只是保留接口，将动态库与程序代码独立，这样就可以提高代码的可复用度，和降低程序的耦合度。
    2. 静态库：编译链接时，把库文件的代码全部加入可执行文件中，因此生成的文件比较大，但是运行时也就不需要库文件了，后缀名一般为`.a`
        1. 为什么需要静态库：通过静态库的方式降低复杂度，在升级更新时尽量做到增量更新，但是静态库会导致复用性降低，磁盘占用高。
    3. 动态库：在编译链接时并没有把库文件的代码加入可执行文件中，而是在程序执行时由运行时链接文件加载库，这样可以节省系统的开销，后缀名一般为`.so`。
        1. 动态库的作用
            1. 库文件不在可执行文件中，放置在外侧
            2. 升级更新会方便快捷
            3. 动态库会存在冲突(版本问题)
    4. gcc/g++在编译时默认使用动态库。无论静态库还是动态库，都是由.o文件构成的
3. makefile描述模块间的依赖关系，会记录所有文件的信息，在make时决定链接时需要重新编译哪些
4. 文件：是数据的集合，可以写入、读取或两者兼有的对象(文件具有某些属性，包括访问权限和类型)。逻辑上是字节，文件必然是整数字节。
5. 文件结构：字节流(Linux)、记录序列、记录树
6. 文件系统：操作系统中负责访问和管理文件的部分，是文件及其某种属性的集合，为引用文件的文件序列号提供了名称空间。
    1. 一种特定的文件格式
    2. 指按指定格式进行格式化的一块存储介质
    3. 指操作系统中(通常内核中)用来管理文件系统以及对文件进行操作的机制及其实现
7. 文件类型：七种
    1. `-` 普通文件：文件或代码数据，没有特别的内部结构
    2. `d` 目录
    3. `l` 符号链接
    4. `s` 套接字文件(网络接口):启动一个程序来监听客户端的要求，而客户端可以通过这个socket来进行数据的沟通

5. `b` 块设备文件：发大量数据，比如移动硬盘，字符设备按块读取
6. `c` 字符设备文件：发少量数据，按字符读取
7. `p` 命名管道文件：结果多个程序同时存取一个文件的错误问题

8. VFS：Virtual File System Switch:采用标准的Unix系统调用读写位于不同物理介质上的不同文件系统，使得open()等系统调用不用关心底层的存储介质和文件类型
9. VFS在系统内建立的四种结构体和含义
    1. 超级块(super block)：某一个磁盘的某一个分区的文件系统的信息，记录了文件系统类型和参数。
    2. i-node 对象(i-node object 索引节点)：
        1. 记录真正的文件，文件存储在磁盘上时是按照索引号访问文件的，软链接是不同的文件，但是硬链接是相同的inode号同一个文件。
        2. 文件的管理信息(名字和一些属性)，包括文件的创建/修改日期和它的访问权限，被保存在文件的inode中，它是文件系统中的一个特殊的数据块，它同时还包含文件的长度和文件在磁盘上的存放位置。系统使用的是文件的inode号，目录结构为文件命名仅仅是为了便于使用。
    3. 文件对象(file object)：记录了文件描述符、索引号，不对应真正的文件，文件打开会创建出文件对象，文件关闭才会释放内核中的文件对象。记录了文件的读写状态。
    4. 目录对象(dentry object)：维护了目录中的逻辑关系，若要通过目录来查找文件，都需要这个对象。在路径上，无论是目录还是文件，都是一个dentry对象对应到目录包含的i-node上，目录项包括索引节点编号，目录项名称长度以及名称。
10. 硬链接和软链接 掌握
    1. 硬链接
        1. 不同的文件名对应同一个inode
        2. 不能跨越文件系统
        3. 对应系统调用link
        4. ln -s [原文件名] [连接文件名]
    2. 软链接
        1. 存储被链接文件的文件名(而不是inode)实现链接
        2. 可以跨越文件系统
        3. 对应系统调用symlink
        4. ln [原文件名] [连接文件名]
11. 系统调用和C库区别
    1. 都以C库函数的形式出现
    2. 系统调用：需要切换到内核态来进行相应的操作，编译运行速度和效率高，是Linux内核对外的位移接口(用户程序和内核之间的唯一接口)，提供出最小接口。
    3. 库函数：库函数的可移植性好，依赖于系统调用，提供较为复杂的功能，例如标准I/O库
12. 系统调用头文件
    1. `<unistd.h>`
    2. `<sys/types.h>`
    3. `<fcntl.h>`
    4. `<sys/stat.h>`
    5. `<dirent.h>`
13. 系统调用
    1. `int open(const char *pathname, int flags, mode_t mode)`，可以没有mode，但是如果flags中有O_CREAT则必须要mode
        1. flags
            1. O_RDONLY：只读

    2. O_WRONLY：只写
    3. O_RDWR：读写
    4. O_APPEND：追加模式打开
    5. O_TRUNC：覆盖模式
    6. O_CREAT：文件不存在则创建
    7. O_EXCL：和O_CREAT同时使用，存在时出错
    8. O_NONBLOCK：非阻塞
  2. mode
    1. S_IRUSR
    2. S_IWUSR
    3. S_IXUSR
    4. S_IRWXU
    5. S_IRGRP
    6. S_IWGRP
    7. S_IXGRP
    8. S_IRWXG
    9. S_IROTH
    10. S_IWOTH
    11. S_IXOTH
    12. S_IRWXO
2. `off_t lseek(int fildes, off_t offset, int whence)` 如果成功返回偏移地址，否则为-1
    1. SEEK_SET：从头偏移offset
    2. SEEK_CUR：从当前偏移offset
    3. SEEK_END：从尾偏移offset
3. `int fcntl(int fd, int cmd, struct flock *lock)` 返回值；若成功则依赖于cmd，若出错则为-1
    1. F_DUPFD：复制文件描述符
    2. F_GETFD/F_SETFD：获取/设置文件描述符标志，为解决fork子进程执行其他任务(exec等)导致父进程的文件描述符被复制到子进程中，使得对应文件描述符无法被之后需要的进程获取。设置了这个标记后可以使得子进程在执行exce等命令后释放对应的文件描述符资源。
    3. F_GETFL/F_SETFL:获得/设置文件状态标志(open/creat中的flags 参数)，目前只能更改0_APPEND , 0_ASYNC, 0_DIRECT, 0_NOATIME,O_NONBLOCK
    4. F_GETOWN/F_SETOWN: 管理1/0可用相关的信号。获得或设置当前文件描述符会接受SIGI0和SIGURG信号的进程或进程组编号F_GETLK/F_SETLK/F_SETLKW: 获得/设置文件锁,设置为F_GETLK时需要传入flock* 指针用于存放最后的锁信息。S_SETLK 需要传入flock *指针表示需要改变的锁的内容,如果不能被设置,则立即返回EAGAIN。
    5. **F_GETLK**：获得文件的封锁信息
    6. **F_SETLK**：对文件的某个区域封锁或解封
    7. **F_SETLKW**：功能和F_SETLK，如果锁被占用则等待
4. `size_t read(int fd, void *buf, size_t count)` 返回值为读到的字节数，如果已经到文件尾为0，若出错为-1
5. `size_t write(int fd, const void *buf, size_t count)` 返回值为已经成功写的字节数，若出错为-1
6. `int close(int fd)` 关闭文件描述符，释放资源
7. `int stat(const char *filename, struct stat *buf);`
    1. 把对应文件名的相关信息存到对应地址里
    2. filename 文件名

3. 当文件是一个符号链接时，stat返回的是指向的文件的信息

8. `int fstat(int filedes, struct stat *buf);`
    1. 把对应文件描述符的相关信息存储到对应地址
    2. fstat系统调用的接受的是一个文件描述符，而其他接受的是文件全路径。
9. `int lstat(const char *file_ name, struct stat *buf);Return: 0 if success; -1 if failure)`：当文件是一个符号链接时，lstat返回的是符号链接本身的信息

14. 缓冲I/O
    1. 全缓冲 全缓冲指的是系统在填满标准IO缓冲区之后才进行实际的IO操作；注意，对于驻留在磁盘上的文件来说通常是由标准IO库实施全缓冲。
    2. 行缓冲 在这种情况下，标准IO在输入和输出中遇到换行符时执行IO操作；注意，当流涉及终端的时候，通常使用的是行缓冲。
    3. 无缓冲 无缓冲指的是标准IO库不对字符进行缓冲存储；注意，标准出错流stderr通常是无缓冲的。
15. C库头文件
    1. `<stdio.h>`
16. C库函数强调
    1. `FILE *fopen(const char * filename, const char* mode);`
        1. r 读
        2. w 清空写
        3. a 追加
        4. r+ 读写
        5. w+ 清空读写
        6. a+ 不清空读写
    2. `FILE *fdopen(int fildes, const char *mode)`根据已打开的文件描述符创建一个流
    3. `int fclose(FILE *stream)`
    4. `int getc(FILE *fp)`：是预定义宏，无函数副作用，更快
    5. `int fgetc(FILE *fp)`：常用，在用到函数副作用/函数指针用
    6. `int putc(int c, FILE *fp)`
    7. `int fputc(int c, FILE *fp)`
    8. `int fputs(const char *s, FILE * stream);`
    9. `char *fgets(char *s, int size, FILE *stream)`常用，最多从流中读取并存储size-1个字符，并最后添加一个\0
    10. `int fseek(FILE *stream, long int offset, int whence)`，whence的取值：0是文件首、1是不跳转、2是文件尾向前。
    11. `long ftell(FILE *stream)`当前位置到文件开头位置
    12. `void rewind(FILE *stream)`跳到开头
    13. `int fflush(FILE *stream)`刷新文件流，将流里的数据立刻写入文件
    14. `int fileno(FILE *fp)`确认流使用的底层文件描述符
17. 进程：是一个正在执行的程序实例，是操作系统的执行单位，由执行程序、当前值、状态信息以及通过操作系统管理此进程执行情况的资源组成；包含有一个地址空间，在该空间内指定一个或多个线程，并分配这些线程所需的系统资源。
    1. 进程的启动都由其他进程启动：所有的进程都是由内核进程(PID=1)来启动的，往往都具有父子进程关系，init(1号进程)由内核本身启动，形成树状的层次结构。
    2. 进程的终止

1. 正常终止
    1. 从main返回
    2. 调用exit函数(库函数，做清理处理)
    3. 调用_exit函数(系统函数，立即进入内核)
2. 异常终止
    1. 调用abort函数
    2. 由一个信号终止

```
// 例子：打开文件夹
DIR *dp;
struct dirent * entry;
if((dp = opendir(dir)) == NULL)
   err_sys();
while((entry = readdir(dp)) != NULL){
   lstat(entry->d_name, &statbuf);
   if(S_ISDIR(statbuf.st_mode)){}
   else{}
}
closedir(dp);
```

18. 文件锁
    1. 记录锁：按记录加锁
    2. 劝告锁：
        1. 检查，加锁有应用程序自己控制
        2. 不会强制应用程序不允许访问，只是提醒
    3. 强制锁
        1. 检查，加锁由内核控制
        2. 影响 open() read() write()
    4. 共享锁：可以读
    5. 排它锁：读写均不可
19. 其他文件封锁命令
    1. 头文件: `#include<sys/file.h>`
    2. `int lockf(int fd, int cmd, off_t len)`
        1. cmd：指定的操作类型
            1. F_LOCK：给文件夹互斥锁。若已被加锁则阻塞直到成功
            2. F_TLOCK：同上，但不会阻塞，直接失败
            3. F_ULOCK：解锁
            4. F_TEST：测试是否上锁。未上锁则0，否则-1
        2. len：从当前位置开始要锁住多长
        3. 这个函数是对fcntl的一层封装
20. 操作系统内核
    1. 操作系统是一系列程序的集合，其中最重要的部分构成了内核
    2. 单内核/微内核
        1. 单内核是一个很大的进程，内部可以分为若干模块，运行时是一个独立的二进制文件，模块间通讯通过直接调用函数实现
        2. 微内核中大部分内核作为独立的进程在特权下运行，通过消息传递进行通信
    3. Linux内核的能力：内存管理，文件系统，进程管理，多线程支持，多处理支持，抢占式。

    4. Linux内核区别于其他UNIX商业内核的优点
        1. 单内核，模块支持
        2. 免费/开源
        3. 支持多种CPU，硬件支持能力非常强大
        4. Linux开发者都是非常出色的程序员.
        5. 通过学习Linux内核的源码可以了解现代操作系统的实现原理
21. 模块.ko 掌握
    1. 加载和释放模块的命令
        1. 底层命令
            1. `insmod <module.ko> [module parameters]`：装载一个模块，只有超级用户才可以使用
            2. `rmmod`：卸载一个模块
        2. 高层命令
            1. `modprobe`
            2. `modprobe -r`
    2. 理解模块之间的依赖关系(比如模块A需要引用模块B所导出的符号)：自动按需加载、卸载
        1. `moddep`
        2. `lsmod`：列举所有在内核中装载的模块，等价于cat /proc/modules
        3. `modinfo`
22. 最简单的内核模块例子

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
static int __init hello_init(void)
{
   // __init标记 该函数只在初始化期间使用。模块装载后，将函数占用的空间释放
   printk(KERN_INFO "Hello world\n");
   return 0;
}
static void __exit hello_exit(void)
{
   // __exit标记 该代码近用于模块卸载
   printk(KERN_INFO "Goodbye world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

23. 内核开发需要注意的：不能使用C库开发驱动程序、没有内存保护机制、小内核栈、并发考虑
24. 内核态程序和用户态程序区别
    1. 运行空间不同：内核空间 vs 用户空间
    2. 入口不同：module_init() vs main()
    3. 出口不同：module_exit() vs 无出口
    4. 运行方式不同：insmod命令载入 vs 直接运行
    5. 调试方式不同：kdbug、kdb、kgdb等 vs gdb
25. 驱动设备分类
    1. 字符设备 Character Driver->字符型文件
    2. 块设备 Block Driver->块文件
    3. 网络接口设备 Network Driver->Socket
26. 字符设备驱动程序的初始化加载过程

1. 申请设备号：每一个字符设备或块设备都有一个主设备号或次设备号
    1. 主设备号：表示一个特定的驱动程序
    2. 次设备号：表示使用该驱动程序的各设备
2. 定义文件操作结构体file_operations
3. 创建并初始化定义结构体cdev
    1. cdev结构体描述字符设备
    2. 该结构体是所有字符设备的抽象，其包含了大量字符设备所共有的特性。
4. 将cdev注册到系统，并和对应的设备号绑定
5. 在/dev文件系统中用mknod创建设备文件，并将该文件绑定到设备号上
    1. 设定设备号:`device=scull`
    2. 定义主设备号:`major=15`
    3. 用户可以通过访问`/dev/scull`来访问当前的驱动设备

2. 课外部分

1. 2019年底EulerOS被正式推送开源社区，命名为openEuler
    1. openEuler是一个开源、免费的Linux发行平台
    2. 支持x86、ARM、RISC-V等多种处理器架构
    3. 所有开发者、企业、商业组织都可以使用openEuler社区版本，也可以基于社区版本发布自己二次开发的操作系统版本。
2. openEuler：毕昇JDK
    1. **ARM64优化**：dmb指令消除等提升新能
    2. **快速序列化技术**：提升序列化，反序列化性能
    3. **GC优化**：让系统减少卡顿
    4. **SpecJBB** 提升20%
3. 线程间通信ITC
    1. 互斥机制：openEuler提供了"NUMA感知队列自旋锁"实现互斥机制，减小了NUMA体系结构中使用自旋锁的开销。
    2. 同步机制：openEuler中提供down原语不up原语，能够实现线程的同步运行。
4. 进程间通信IPC：openEuler增强了两种进程间通信机制，共享内存与消息传递机制
5. OpenEuler内存页说明
    1. 各级页表的表项大小为8B
    2. OpenEuler将标准大页封装成一个**伪文件系统提供给用户程序申请并访问。**
    3. OpenEuler采用LRU最近最久未使用策略实现页选择唤出。
    4. 页在未来被访问的概率只能预测，不能精准判断。
6. 鲲鹏处理器
    1. 鲲鹏处理器是基于ARMv8-64位RISC指令集开发的通用处理器
    2. 使用大量寄存器：通用X0-X30(31个，64位) + 特殊寄存器 + 系统寄存器
7. openEuler的增强
    1. openEuler对通用Linux操作系统作了增强
    2. 提供iSulad轻量级容器全场景解决方案
8. 为了充分发挥鲲鹏处理器的优势，openEuler在多核调用技术、软硬件协同、轻量级虚拟化、指令集优化和智能优化引擎等方面做了增强。
9. KAE(鲲鹏加速引擎，Kunpeng Accelerator Engine)插件，使能KunPeng硬件加速能力，包括
    1. 对称/非对称加密
    2. 数字签名
    3. 压缩解压缩等算法，用于加速SSL/TLS应用和数据压缩

3. 几个题目

1. Linux中的文件描述符和文件指针FILE *的区别什么？(9')
    1. 文件描述符：打开文件就会获得文件描述符，是很小的正整数。每个进程在PCB(Process Control Block)中保存着一份文件描述表，文件描述符就是这个文件描述符的索引，每个表项都有一个指向已打开文件的指针。
    2. 文件指针：C语言中使用文件指针作为I/O的句柄，文件指针指向进程用户区中的一个被称为FILE结构的数据结构。FILE结果包括一个缓冲区和一个文件描述符。而文件描述符是文件描述符表的一个索引，因此从某种意义上文件指针就是句柄的句柄
2. Linux设备中字符设备与块设备有什么主要的区别？请分别列举一些实际的设备说出它是哪一类设备(9')
    1. 字符设备：提供连续的数据流，应用程序可以顺序读取，通常不支持随机存取。相反，此类设备支持按字节/字符来读写数据。
    2. 块设备：应用程序可以随机访问设备数据，程序可自行确定读取数据的位置。硬盘是典型的块设备，应用程序可以寻址磁盘上的任何位置，并由此读取数据。此外，数据的读写只能以块(通常是512B)的倍数进行。与字符设备不同，块设备不支持基于字符的寻址。
    3. 字符设备有鼠标和键盘等
    4. 块设备有U盘、硬盘和磁盘等。
3. 为什么在Linux中引入makefile？和其他脚本的区别？特点?
    1. makefile可以维护程序的时序依赖关系，按目标编译
    2. 语法不同：通配符不同、执行方式不同、需要用\换行，不然会在多个进程中执行
    3. makefile的最主要特点是自动化编译，一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率
4. 编写两个简单的程序（fred.c, bill.c），将其编译为目标文件，并分别生成静态库和动态库。再编写程序调用之，说明库的使用。
    1. 生成静态链接库
        1. `gcc -c h.c -o h.o`
        2. `ar cqs libh.a h.o`：ar是生成库的命令，cqs是参数，libh.a是生成的静态链接库须以lib开头，h是库名，a表示是静态链接库，h.o是刚生成的目标文件
    2. 生成动态链接库
        1. `gcc -c h.c -o h.o`
        2. `gcc -shared -WI -o libh.so h.o`：生成动态链接库使用gcc来完成，-shared -WI是参数，libh.so是刚生成的静态链接库，必须以lib开头，h是库名，so表示动态链接库，h.o是刚生成目标文件。
    3. 将生成的libh.a，libh.so拷贝到/usr/lib或/lib下
    4. 编译带静态链接库的程序
        1. `gcc -c test.c -o test.o`
        2. `gcc test.o -o test -WI -Bstatic -lh`:-WI -Bstatic表示链接静态库，-lh中-l表示链接，h是库名即/usr/lib下的libh.a
    5. 编译带动态链接库的程序
        1. `gcc -c test.c -o test.o
        2. `gcc test.o -o test -WI -Bdynamic -lh`:-WI -Bdynamic表示链接动态库，-lh中-l表示链接，h是库名即/usr/lib下的libh.so
    6. 运行./test得到结果
5. 用户没有对/etc/passwd和/etc/shadow的写权限，为什么可以通过passwd命令修改口令
    1. /bin/passwd在文件模式字（st_mode）中设置一个特殊标志，该标志的含义是"当执行此文件是，将进程的有效用户ID设置为文件所有者的用户ID（st_uid）"，与此类似的，st_mode

中还可以设置另一位，使得将执行此文件的进程的有效组ID设置成文件的组所有者ID（st_gid）。 在文件模式字中这两位被称为设置用户ID和设置组ID

2. 若文件的所有者是root，而且设置了该文件的设置用户ID位，然后当该文件由另一个进程执行时，该进程有了root 的权限。 如/bin/passwd ，该程序是一个设置用户ID程序，普通用户使用passwd 命令更改登录口令时，shell会调用/bin/passwd ,此时shell具有root权限，所以可以修改 /etc/passwd 文件来更改用户登录口令

6. 以I/O为例，说明系统调用接口与库接口的异同

    1. 一般函数库的函数是不需要系统的服务的(除非涉及I/O操作)。系统调用是要求操作系统为用户提供进程，提供服务，通常是涉及系统的硬件资源和一些敏感的软件资源等。
    2. 函数库可以被理解为是开发人员与系统调用之间的中间层接口，保证安全可靠的调用系统调用。
    3. 从程序执行效率来看，系统调用的执行效率大多要比函数高，尤其是处理输入输出的函数。
        1. 数据量小：函数库的函数执行效率可能更好(先存入缓冲区，然后一次性刷入)。
        2. 数据量大：函系统调用可以处理输入输出的数据量超过文件系统定义的尺寸大小的文件。
    4. 函数库往往比系统调用有更好的程序可迁移性。
    5. 函数库比系统调用在更高层次。

函数库调用	系统调用
在所有的 ANSI C 编译器版本中，C 库函数是相同的	各个操作系统的系统调用是不同的
它调用函数库中的一段程序（或函数）	它调用系统内核的服务
与用户程序相联系	是操作系统的一个入口点
在用户地址空间执行	在内核地址空间执行
它的运行时间属于“用户时间”	它的运行时间属于“系统”时间
属于过程调用，调用开销较小	需要在用户空间和内核上下文环境间切换，开销较大
在 C 函数库 libc 中有大约 300 个函数	在 UNIX 中大约有 90 个系统调用
典型的 C 函数库调用：system fprintf malloc	典型的系统调用：chdir fork write brk；

7. IPC机制：信号、管道、信号量、共享内存、消息队列、套接字

8. 重定向是Linux中的重要机制。请描述Linux重定向的用途、使用方法、典型案例，并简要描述其实现机制

    1. 重定向的用途：
    2. 使用方法：Linux中可以使用shell的重定向符号
        1. `<`：输入重定向，改变命令的输入，后面指定输入内容(文件名)
        2. `>`：输出重定向，将前面输出的部分输入到后面的文件，并清空文件内容。
        3. `>!`：同上，强制覆盖。
        4. `<<`：追加输入重定向：后面跟字符串，用来表示输入结束
        5. `>>`：追加输出重定向：把前面输出的东西追加到后面的文件尾部，不清除文件的内容
        6. `2>`：错误重定向：将错误的信息输入到后面的文件中，会删除文件原有的内容
        7. `2>>`：错误追加重定向：将错误的信息追加到后面的文件中，不会删除文件原有的内容
        8. 例子
            1. 将标准输出流重定向到1.txt：`echo "linux" >1.txt`
            2. 将标准输入流重定向到1.txt：`cat<1.txt`

3. 实现机制：主要通过使用dup2系统调用通过明确指定目标描述符来把一个文件描述符复制为另一个。

9. 使用C的库函数，编写一个函数`void bindiff(char *file1, char *file2, char *fileo)`，将文件从file1、file2对应的路径读取并逐字节比对，将相同的字节删除到fileo对应的文件。(25')

```c
#include<stdio.h>
void bindiff(char *file1, char *file2, char *fileo);
int main()
{
    bindiff("1.txt","2.txt","3.txt");
}
void bindiff(char *file1, char *file2, char *fileo)
{
    FILE * fp1 = 0,*fp2=0,*fpo=0;
    char ch1,  ch2;
    fp1 = fopen(file1,"r");
    fp2 = fopen(file2,"r");
    fpo = fopen(fileo,"w");
    while (1)
    {
        ch1 = (char)fgetc(fp1);
        ch2 = (char)fgetc(fp2);
        if (feof(fp1) || feof(fp2))
        {
            break;
        }
        if (ch1 == ch2)
        {
            fputc(ch2,fpo);
        }
    }
    fclose(fp1);
    fclose(fp2);
    fclose(fpo);
}
```

10. 用系统调用实现输出给定文件夹中所有文件的名字 用空格隔开，并在文件夹及文本文件的输出后注上"（文件夹）"、"（文本文件）"

```c
// dirent要了解
struct dirent {
   long d_ino;
   off_t d_off;
   unsigned short d_reclen;
   unsigned char d_type;
   char d_name [NAME_MAX + 1];
}
// stat要了解
struct stat {
   mode_t    st_mode;    /*file type & mode 低9位保存权限信息，之后为sticky、SGID、SUID，[12:17]保存了文件类型*/
   ino_t     st_ino;     /*inode number (serial number)*/
   dev_t     st_rdev;    /*device number (ile system)*/,
   nlink_t   st_nlink;   /*link count 硬链接计数 */
   uid_t     st_uid;     /*user ID of owner*/
   gid_t     st_gid;     /*group ID of owner*/
```

```
    off_t     st_size;    /*size of file, in bytes*/
    time_t    st_atime;   /*time of last access*/
    time_t    st_mtime;   /*time of last modification*/
    time_t    st_ctime;   /*time of last file status change*/
    long      st_blksize; /*Optimal block size for I/O*/
    long      st_blocks;  /*number 512-byte blocks allocated*/
}

mode_t 的 参考如下
S_IFMT      0170000   //掩码，过滤st_mode中除文件类型以外的信息
S_IFSOCK    0140000   //套接字
S_IFLNK     0120000   //符号链接(软链接)
S_IFREG     0100000   //普通文件
S_IFBLK     0060000   //块设备
S_IFDIR     0040000   //目录文件
S_IFCHR     0020000   //字符设备
S_IFIFO     0010000   //管道
S_ISUID     0004000   //设置用户ID
S_ISGID     0002000   //设置组ID
S_ISVTX     0001000   //粘住位
S_IRWXU     00700     //掩码，过滤st_mode除文件所有者权限以外的信息
S_IRUSR     00400     //用户读权限
S_IWUSR     00200     //用户写权限
S_IXUSR     00100     //用户执行权限
S_IRWXG     00070     //掩码，过滤st_mode除所属组权限以外的信息
S_IRGRP     00040     //读权限
S_IWGRP     00020     //写权限
S_IXGRP     00010     //执行权限
S_IRWXO     00007     //掩码，过滤st_mode除其他人权限以外的信息
S_IROTH     00004     //读权限
S_IWOTH     00002     //写权限
S_IXOTH     00001     //执行权限
```

```
#include <unistd.h>
#include <dirent.h>
#include <sys/type.h>
#include <sys/stat.h>
#include <stdio.h>

int main(){
   DIR *dp;
   struct dirent * entry;
   if((dp = opendir(dir)) == NULL)
      err_sys();
   while((entry = readdir(dp)) != NULL){
      lstat(entry->d_name, &statbuf);
      switch(statbuf->st_mode & S_IFmt){
         case S_IFDIR:
            printf("%s （文件夹）", entry->d_name);
            break;
         default:
            print("%s （文本文件）", entry->d_name);
            break;
      }
   }
   closedir(dp);
}
```

11. Shell 获得用户输入的100个整数，并输出其最大值，最小值，总和。

```
read number
max=$number
min=$number
sum=$number
for((i=0;i<99;i++))
do
   read number

   if [ $number -gt $max ]
   then
      max=$number
   fi

   if [ $min -gt $number ]
   then
      min=$number
   fi

   sum=$(($sum+$number))
done
echo $max
echo $min
echo $max
```

12. Shell 查看当前目录下的子目录并且只显示子目录
    1. `ls -F | grep /$`
    2. `ls -l | grep "^d"`
13. Shell 查找一个错误代码EPERM(宏定义)在Linux系统头文件中的定义并显示:`grep ERERM *.h`
14. Linux comm 命令用于比较两个已排过序的文件。这项指令会一列列地比较两个已排序文件的差异，并将其结果显示出来，如果没有指定任何参数，则会把结果分成 3 列显示
    1. 第 1 列仅是在第 1 个文件中出现过的列
    2. 第 2 列是仅在第 2 个文件中出现过的列
    3. 第 3 列则是在第 1 与第 2 个文件里都出现过的列。
    4. 若给予的文件名称为 - ，则 comm 指令会从标准输入设备读取数据。
15. sleep命令可以用来将目前动作延迟一段时间。
16. 使用系统调用实现函数half(a, b)将a文件的一般拷贝到b文件，且b仅包含a文件的后一半的内容

```
#include<unistd.h>

int main(){
   int fa = open("a.txt", O_RDONLY);
   int fb = open("b.txt", O_WRONLY|O_TRUNC);
   int fa_length = lseek(fa, SEEK_END, 0);
   lseek(fa, SEEK_SET, fa_length/2);
   int i = 0;
   char content[fa_length/2];
   int length = 0;
   do{
      length = read(fa, content, 1024);
      write(fb, content, length);
   }while(length == 1024);
   close(fa);
```

```
    close(fb);
}
```