Linux Programming review

文字题

boot loader工作

1. pass boot parameters to the linux kernel like device information
2. optionally load an initial root disk
3. boot other operating system
4. Common boot loader like
    1. LILO(linux loader) - configures the MBR accroding to a configure file
    2. GRUB(grand unified boot loader) - understand file system stucture, installed in MBR
        - MBR - main boot record, including boot loader and partition table

apt-get 原理

1. Ubuntu采用集中式软件仓库机制，apt从/etc/apt/sources.list中找到镜像站点地址，告知Ubuntu可以访问的镜像站点地址，拥有一个索引清单
2. 从package.gz中获取所有包信息，apt在本地存了一份软件包的信息索引，可以检索列表或是检测依赖
3. 在安装的时候安装包以及相关依赖。

文件类型

- 文件是数据的集合
- 文件结构：字节流、记录序列（Record Sequence）、记录树（Record Tree）。Linux下为字节流。
- Linux的文件类型
    - 普通文件（-）：纯文本文件、二进制文件、数据格式文件
    - 字符设备文件（c, character）：与设备进行交互的文件。按字符进行I/O。如：终端文件（ttyN）
    - 块设备文件（b, block）：同字符设备文件，但按块进行I/O。如：硬盘
    - 套接字文件（s, socket）：表示一个socket连接。可以通过这个文件来与连接的对方（peer）进行通信。
    - 管道文件/FIFO文件（p, pipe)：用于进程间通信。一个进程可以从这个文件中读取另一个进程此前写入的数据
    - 符号链接文件（l, link）
    - 目录文件（d, directory）
- 目录结构
    - Linux中所有的目录均包含在一个统一的、虚拟的统一文件系统（Unified File System）中。
    - 物理设备被抽象为文件，挂载到指定的挂载点。没有Windows下的盘符的概念
    - 根目录下各个文件夹的作用：
        - /bin：系统必要的命令的二进制文件。包含了会被会被系统管理者和用户使用的命令。大部分常用的命令都在这里。
        - /boot：Boot Loader相关的静态文件。包含了所有需要在系统引导阶段使用的文件（如内核镜像等）。

        - /dev：设备对应的虚拟文件。
        - /etc：系统和软件的配置文件。
        - /lib：必要的共享库文件（如.so）或内核模块。
        - /media：外部设备通用挂载点的父目录。
        - /mnt：临时文件系统的挂载点的父目录。
        - /opt：额外的应用软件包安装目录
        - /sbin：只有管理员可以使用的命令的二进制文件。是与系统相关的基本命令，如shutdown，reboot等
        - /srv：系统提供的有关服务的数据
        - /tmp：临时文件
        - /usr：Unix System Resources，不是user的简写。用于存放共享、只读的数据。子目录包括/bin，/etc，/lib，/sbin，/tmp等，与根目录下对应的目录对比，这些目录是給后来安装的软件的使用的（而不是系统自带的）。还有/include，/src等文件夹，存放系统编程所需的头文件和源码等。
        - /home：用户的家目录的父目录
        - /root：root用户的家目录
- 文件权限
    - 分为三个层次：所有者，所有者所在组，其他用户
    - 每个层次又分三个类型：读、写、执行
    - 更改权限：参见chmod
    - 如果是权限掩码，文件不可以设置执行位，文件夹可以，所以文件是666-umask, 文件夹是777-umask

进程

进程是一个正在执行的程序实例。由程序体，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

`echo $$` : the pid of current process

除了1号进程是操作系统自己启动的，别的都是父子关系的进程-树状结构。一个进程的结束：自己完成执行terminate, 终止信号

权限和粘滞位

rwx, **SUID, SGID, Sticky bit**

如果`ls –l /usr/bin/passwd`会看到owner的权限是rws而非rwx

- SUID - set user id 在执行时设置user ID S_ISUID(04000)
    - 只对二进制可执行文件有效。在执行命令时，间接临时拥有root权限，间接修改/etc/passwd完成修改自己密码的权限
- SGID - set group id 在执行时设置group ID S_ISGID(02000)
- S_ISVTX(01000) 粘滞位
    - 仅对目录有效，对文件无效。使用者在该目录下建立文件或目录时，仅自己与root有权利删除新建的文件或目录。/tmp是drwxrwxrwt

可以用chmod u+s setuid; g+s setgid ；o+t sticky标志

典型的SUID或SGID命令是`su`和`sudo`

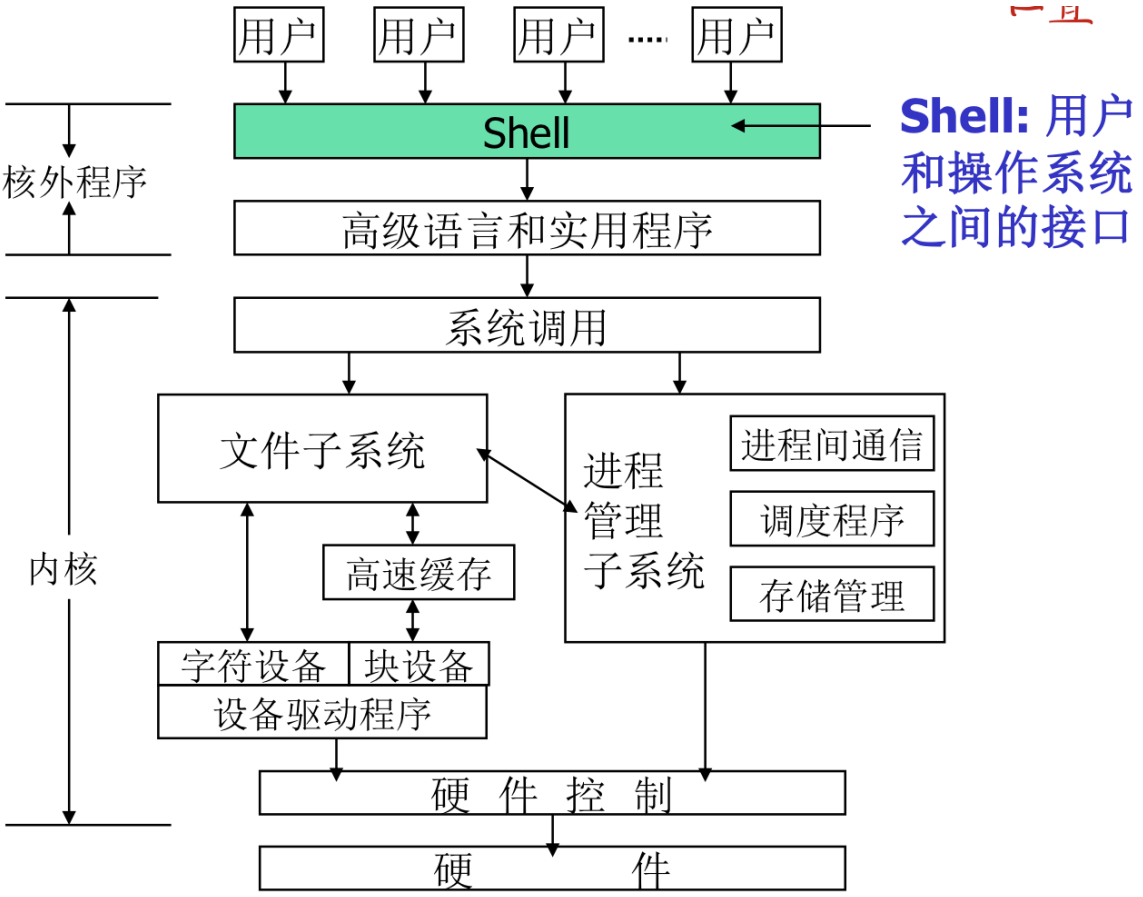
文件锁

锁的作用：几个进程同时操作一个文件

- 记录锁 可以锁定文件的部分区域甚至字节
- 劝告锁 检查，加锁由应用程序自己控制，无需检查
- 强制锁 检查，加锁由内核控制，影响open, read, write等，每次操作检查锁是否违背规则
- 共享锁 读锁，不能加排他锁
- 排他锁 写锁，不能加读锁或写锁

在挂载的时候会加锁，如**MS_MANDLOCK**加锁允许在文件上执行强制锁。

UNIX系统结构



Command

1. ls, ls -l(长信息), -R（递归显示子目录), -a(所有文件), -h(human-readable)
    1. ls -l : FileType, permissions, linkCounter(一个文件夹有., ..所以至少两个，还有里面子目录的数量), owner, group, size, modificationTime, name
2. mkdir, rmdir, ls, pwd, cd
3. touch, cp, mv, ln, rm, cat, more/less
    1. `for f in *.png; do mv -n "$f" "${f/-0}"; done` mv重命名文件
4. 权限
    1. chmod rwx, +-=, ugoa
    2. chown change owner 更改文件属主与属组
    3. chgrp change group 更改所有组
5. ps, pstree, `jobs – fg – bg`, kill -9/-15, `nohup` - run a command and ignoring hangup signals
6. `nice` 改变即将执行的程序的调度优先级， `renice` 改变正在运行进程的niceness值

文件操作

- 列出目录内容：ls

- 创建特殊文件：mkdir, mknod, mkfifo
- 文件操作：cp, mv, rm
- 修改文件属性：chmod, chown, chgrp, touch
- 查找文件：find
- 字符串匹配：grep
- 其他：pwd, cd, ar, file, tar, more, less, head, tail, cat

进程操作

- ps
- kill
- jobs
- fg
- bg
- nice

其他

- who
- whoami
- passwd
- su
- uname
- man

重定向

标准输入：0，stdin　标准输出：1，stdout　标准错误：2,stderr

`<`是输入重定向，`>`是输出重定向，实现原理是通过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外一个

- `<` 输入重定向
- `>` 输出重定向
- `>>` 输入重定向追加
- `<<` 一般是`<<tag`，将开始标记tag和结束标记tag之间的内容作为输入
- `2>` 把错误信息写入到file，如果合并stdout和stderr，可以`command > file 2>&1`

管道

一个进程的输出作为另一个进程的输入 |

环境变量

echo, env, set, export

PATH=$PATH:.

正则表达式

- find
- sed
- grep

即时文档

```
cat >>filename.txt <<!CATINPUT!
hello, this is a here document
!CATINPUT!
```

在最后输入`!CATINPUT!`作为一个end of file的输入信息

Shell

shell是用户和操作系统之间的接口，作为核外文件而存在

引号

- 单引号内所有字符都是字面量（不会被转义或求值）
- 双引号内$、``（反引号）会被求值，\会被转义

执行脚本文件

1. sh script_file
2. chmod + x script_file(chown / chgrp), ./script_file
3. source script_file / .script_file

其中1和2都是新建一个shell进程然后运行，3是使用当前的shell进程

用户环境

环境变量修改 export， env & set command

- env 当前用户的变量
- set 显示当前shell的所有变量
- export 显示当前导出成环境变量的shell变量

变量

调用脚本时，如果带有参数，会产生一些额外的变量

- $#：参数个数
- $0：脚本程序名
- $1，$2...：按调用时指定的顺序排列的参数
- $*：全部参数连接成的一个字符串，以环境变量$IFS的第一个字符分隔
- $@：全部参数组成的列表（就是数据结构的那个列表）

条件测试

- `test expression`或`[ expression ]`（注意方括号内部两侧都必须要有空格）。这两种方式是一个命令调用，通过命令的退出值表明真伪。0表示真，1表示假。使用举例：

  ```
  $ test 1 -lt 2
  $ echo $?
  0
  $ test 1 -gt 2
  $ echo $?
  1
  ```

  比较运算（注意关系算符和操作数之间都要有空格）：

  - 字符串比较：

        1. str1 = str2
        2. str1 != str2
        3. -z str：字符串是否为空
        4. -n str：字符串是否不为空
    - 算术比较：expr1 *op* expr2，下面列举的是op
        1. -eq：相等
        2. -ne：不等
        3. -gt：大于
        4. -ge：大于等于
        5. -lt：小于
        6. -le：小于等于
    - 文件测试：op file，下面列举的是op
        1. -e file：是否存在
        2. -d file：是否是目录
        3. -f file：是否是常规文件
        4. -s file：文件长度是否不为0
        5. -r file：是否可读
        6. -w file：是否可写
        7. -x file：是否可执行
    - 逻辑操作：
        1. ! expr：逻辑取反
        2. expr1 -a expr2：逻辑且
        3. expr1 -o expr2：逻辑或
- 算术拓展：`$((expression))`。这是一个求值表达式，可以赋给变量。0表示假，1表示真

一些语句

condition, itemlist等记得**加分号**

```
if [ expression ]
then
    statement
elif [ expression ]
then
    statement
else
    statement
fi
```

```
case str in
    str1 | str2) statement;;
    str3 | str4) statement;;
    *)  default;;
esac
```

```
for var in list; do
# 这个list可以是$(ls *.cpp);
    statement
done
```

```
while condition; do
# condition: [ "$quit" != "y" -o "$quit" != "Y" ];
    statement
done
```

```
until condition; do
    statement
done
```

```
select item in itemlist; do
    statement
done
```

函数

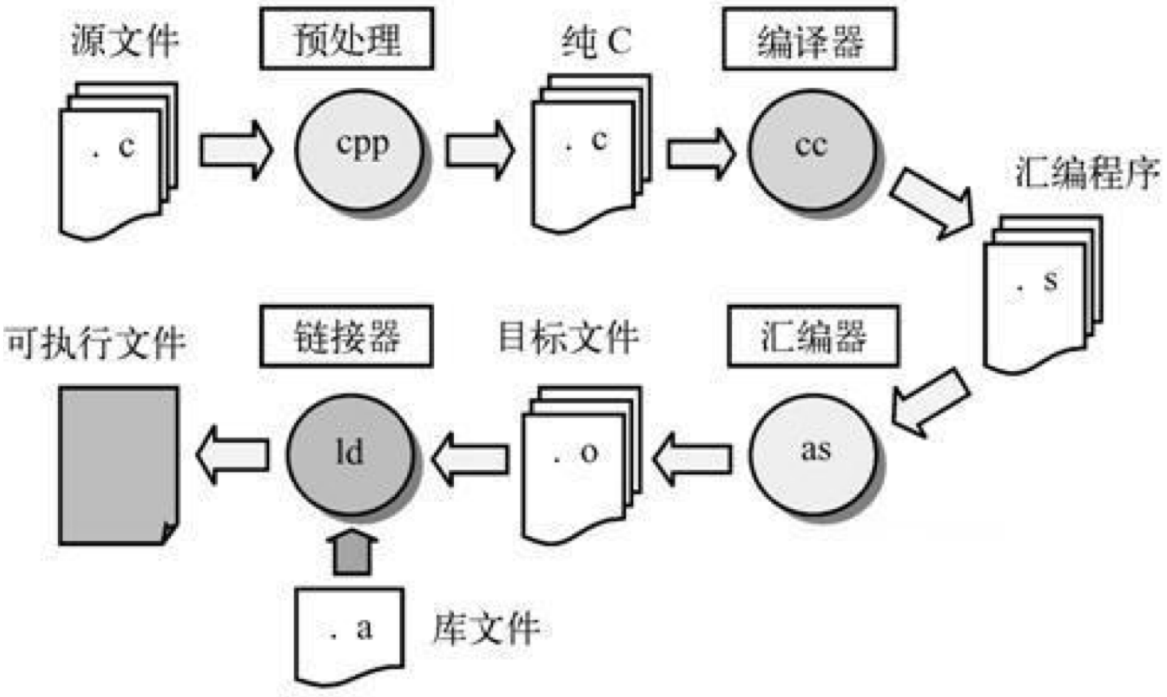
函数用`$1` `$2`来表示参数，比如下面那个函数的参数就是msg和default 1

```
if yesno "Continue installation? [n]" 1; then
    DO_YES_CONDITION;
else
    DO_NO_CONDITION;
fi
```

用`$?`获得函数返回值

Linux Programming Prerequisite

编译连接过程



1. 遇到.c, .h文件，用gcc进行预处理生成.i文件
2. .i文件gcc编译生成.a文件
3. assembly汇编生成.o文件
4. 和静态链接库用ld链接生成可执行文件

gcc

用于编译、连接，**命令和参数需要记忆**，要找命令里的错误

- -E：只进行预处理
- -S：只进行预处理、编译
- -c：预处理、编译、汇编
- -o [output_file]：指定输出文件名
- -g：产生符号文件（可用于调试工具）
- -On：优化等价，n可以取0,1,2,3
- -Wall：显示所有警告信息
- -Idir：指定额外的头文件搜索路径
- -Ldir：指定额外的库文件搜索路径
- -lname：指定链接时搜索的库文件（name要去掉lib，如要链接pthread，pthread的静态库名为libpthread.a，则参数写成-lpthread）
- -DMACROT[=DEFN]：定义宏

make & makefile

- makefile描述模块间的依赖关系
- make根据makefiile对程序进行管理和维护，make判断被维护文件的时序关系
  - makefile用于自动编译和链接，makefile中记录了所有文件的信息，make会在链接的时候决定需要重新编译哪些文件
  - make [-f filename] [targetname]

makefile规则结构

```
# target ... : prerequistes
#    command

hello : main.o kbd.o
    gcc -o hello main.o kbd.o
main.o : main.c def.h
    cc -c main.c
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
clean :
    rm edit main.o kbd.o
```

makefile执行顺序

1. 当前目录查找makefile或Makefile
2. 查找文件中第一个目标文件target如hello
3. 如果hello不存在或hello依赖的文件修改时间比hello新，执行后面的文件
4. 如果hello依赖的.o文件不存在，在文件中找到.o文件的依赖性，生成.o文件
5. make根据.o文件的规则生成.o文件，用.o生成hello文件

伪目标

例如make里的install, clean等，只是一个标签，无法生成他的依赖关系和决定是否要执行，只能通过显式的指明这个目标才能让其生效。伪目标不能和文件名重名。

伪目标可以作为默认目标，如果放在第一个就算

也可以用

```
.PHONY : clean
clean:
    rm *.o temp
```

多目标

多个目标同时依赖于一个文件，生成的命令大致相似就可以用自动化变量`$@`，$@是目标的完整名称

```
bigoutput littleoutput : text.g
    generate text.g -$(subst output,,$@) > $@

# 等价于
bigoutput : text.g
    generate text.g -big > bigoutput
littleoutput : text.g
    generate text.g -little > littleoutput
```

系统

文件系统：用户空间（用户程序，进程） -> 系统空间（VFS -> EXT2， FAT等文件系统）

四种对象

- super block：超级块。一个超级块对应一个文件系统。
- inode：索引节点。一个实际存在的文件实体只有一个inode。inode对象全系统共用。
- dentry：目录项。一个目录项对应一个dentry，就是ls -a列出来的每一项就是一个dentry。dentry中有指向inode的指针。多个dentry可以对应同一个inode。dentry对象全系统共用。
- file：文件对象。一个打开了的文件对应一个file。file中有指向dentry的指针。文件对象是进程私有的（会以copy-on-write的方式与子进程共享）。

硬链接和符号链接

- Hard link：-> 和原来的文件没区别
    - 不同的文件名对应一个inode
    - 不能跨越文件系统
    - 对应系统调用link
- Symbolic link： -> 快捷方式
    - 存储链接文件的文件名实现链接，而非inode
    - 可跨越文件系统
    - 对应系统调用symlink

系统调用 & 库函数

- 系统调用：Linux内核的对外接口，用户程序和内核之间唯一的接口，提供最小接口
- 库函数：依赖于系统调用，提供较复杂功能，如标准I/O
- 他们都以C函数的形式出现

基础IO系统调用

文件读写

open/create, close, read, write, lseek

open/create

flags可以是`O_CREAT|O_RDWR`

- flags: file access mode
  - O_RDONLY read only
  - O_WRONLY write only
  - O_RDWR read write
  - O_APPEND
  - O_TRUNC 从头写
  - O_CREAT 文件不存在就创建
- mode: 如果创建新文件，那么那个文件的权限就是mode
  - 五位数字，都是00开头，后面3个就是对应的rwx
  - S_IRUSR 就是00400 read by owner
  - S_IWGRP 00020 write by group
  - S_IXOTH 00001 execute by other
  - S_IRWXU read write execute by owner
  - mode和umask mode & ~umask 普通文件是666封顶，文件夹是777封顶
- `int open(const char* pathname, int flags)`
- `int open(const char* pathname, int flags, mode_t mode)`
- `int creat(const char* pathname, mode_t mode)`
- 如果成功返回文件描述符，失败返回-1

close

`int close(int fd);` 成功返回0， 失败返回1

read/write

- `ssize_t read(int fd, void* buf, size_t count)` 返回读到的字节数，到EOF为0，出错-1
- `ssize_t write(int fd, void* buf, size_t count)` 返回写的字节数，失败为-1

lseek

重定位read/write的文件，给他们一个offset

`off_t lseek(int fd, off_t offset, int whence)`

- SEEK_SET 相对**文件头**偏移+offset个byte
- SEEK_CUR 相对**当前位置**+offset个byte
- SEEK_END 文件**末尾**+offset

dup/dup2

dup复制一个文件描述符，返回新的文件描述符，dup2复制oldfd到newfd, newfd对应的文件将被关闭，如果失败就返回-1

- `int dup(int oldfd)`
- `int dup2(int oldfd, int newfd)`

fcntl

控制一个文件描述符，如果成功就依赖于cmd，出错就是-1

- `int fcntl(int fd, int cmd)`
- `int fcntl(int fd, int cmd, long arg)`
- `int fcntl(int fd, int cmd, struct flock* lock)`

对于cmd处理

- F_DUPFD：复制文件描述符，返回新的文件描述符
- F_GETFD/F_SETFD：获取/设置文件描述符标识（目前只有close-on-exec，表示子进程在执行exec族命令时释放对应的文件描述符）。
- F_GETFL/F_SETFL：获得/设置文件状态标识（open/creat中的flags参数），目前只能更改O_APPEND，O_ASYNC，O_DIRECT，O_NOATIME，O_NONBLOCK
- F_GETOWN/F_SETOWN：管理I/O可用相关的信号。获得或设置当前文件描述符会接受SIGIO和SIGURG信号的进程或进程组编号

文件属性

- `int stat(const char* filename, struct stat* buf)`
- `int fstat(int fd, struct stat* buf)`
- `int lstat(const char* file_name, struct stat* buf)`

获取文件的属性。最后一个是遇到符号链接时，能取到被链接的文件的属性（其他的只能取到链接文件自己的属性）

```
struct stat {
    mode_t st_mode;
    ino_t st_ino;
    dev_t st_rdev;
    nlink_t st_nlink;
    uid_t st_uid;
    gid_t st_gid;
    off_t st_size;
    time_t st_atime;
    time_t st_mtime;
    time_t st_ctime;
    long st_blksize;
    long st_blocks;
}

// 使用方法
struct stat getFile = {};
int res_stat = stat(filename, &getFile);
if(res_stat==-1){
    exit(1);
}
mode_t mode = getFile.st_mode;
if(S_ISDIR(mode)){
    exit(1);    // isdir
}
size_t size = getFile.st_size;
```

time_t是时间戳也就是long

- 判断文件类型方法

- S_ISREG()
- S_ISDIR()
- S_ISCHAR()
- S_ISBLK()
- S_ISFIFO()
- S_ISLNK()
- S_ISSOCK()

权限处理

成功0失败-1

- `int chmod(const char* path, mode_t mode)`
- `int fchmod(int fd, mode_t mode)`
- `int chown(const char* path, uid_t owner, gid_t group)`
- `int fchown(int fd, uid_t owner, gid_t group)`
- `int lchown(const char* path, uid_t owner, gid_t group)`
- `mode_t umask(mode_t mask)`
  - 为文件更改存取权限屏蔽字，默认为022，返回之前的值

链接

- `int link(const char* oldpath, const char* newpath)`
- `int unlink(const char* path)`
- `int sumlink(const char* oldpath, const char* newpath)`
- `int readlink(const char* path, char* buf, size_t bufsiz)`

文件夹处理

增删改

- `int mkdir(const char* pathname, mode_t mode)`
- `int rmdir(const char* pathname)`
- `int chdir(const char* path)` 更改当前工作目录
- `int fchdir(int fd)`
- `char* getcwd(char* buf, size_t size)` 获得当前工作路径，成功为buf，出错就NULL

读

- `DIR* opendir(const char* name)`
- `int closedir(DIR* dir)`
- `struct dirent* readdir(DIR* dir)` read next dir files
- `off_t telldir(DIR* dir)` 返回当前目录流读取位置，离开头偏移量的距离
- `void seekdir(DIR* dir, off_t offset)` set the position of the next readdir()

读取所有文件夹下的文件

```
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)){
        // ...
    }
}
closedir(dp)
```

读取一半的文件并写入另一个文件

```
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
int main(){
    int fd_a,fd_b,nread;
    char buf[10];
    struct stat f_stat;
    fd_a=open("a.txt",O_RDONLY);
    fd_b=open("b.txt",O_WRONLY);
    int st=stat("a.txt",&f_stat);
    int cno_a=lseek(fd_a,f_stat.st_size/2,SEEK_SET);
    int cno_b=lseek(fd_b,0,SEEK_END);
    int n;
    while((n=read(fd_a,buf,10))>0){
        write(fd_b,buf,10);
    }
    close(fd_a);
    close(fd_b);
    return 0;
}
```

内核 & 驱动

内核

- 操作系统是一系列程序的几何，最重要的部分构成了内核
- 单内核/微内核
  - 单内核是一个很大的进程，内部分为若干模块，运行时是一个独立的二进制文件，模块间通信通过直接调用函数实现
  - 微内核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讯
- Linux内核的能力
  - 内存管理，文件系统，进程管理，多线程支持，抢占式，多处理支持
- 区别于其他UNIX内核的优点：单内核模块支持；免费/开源；支持多种CPU

建立初始化程序

- initrd : `mkinitrd /boot/initrd.img $(uname –r)`
- initramfs : `mkinitramfs –o /boot/initrd.img $(uname –r)`
- `update–initramfs –u`

加载模块

- 操作模块
  - insmod install mod
  - rmmod remove mod
- 高层模块
  - modprobe 智能的向内核中加载模块
  - modprobe -r 卸载模块

内核模块和应用程序的区别

	C程序(用户态)	内核模块(内核态)
运行	用户空间	内核空间
入口	main	module_init()指定
出口	无	module_exit()指定
运行	直接运行	insmod
调试	gdb	kdbug，kdb，kgdb等

- 不能用C库开发驱动程序
- 没有内存保护机制
- 小内核栈
- 并发上的考虑

驱动类型

1. 字符设备 Character Driver
2. 块设备 Block Driver
3. 网络接口设备 Network Driver

字符设备驱动程序

- 对上接口
  - read
  - write
  - flush
  - ioctl
- 两个基本结构： file结构体，inode结构体

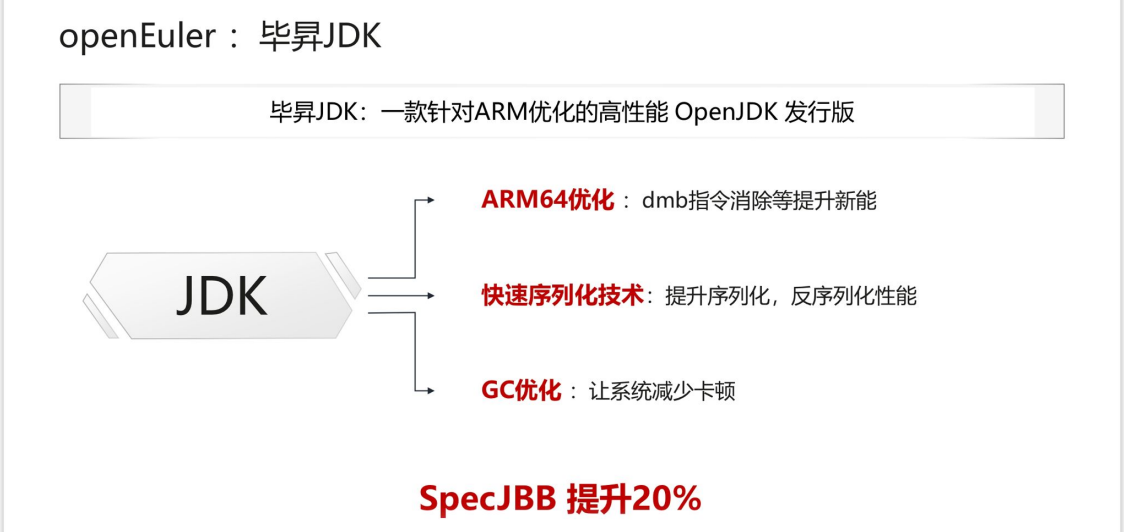
字符设备驱动程序的初始化加载过程

- 申请设备号
  - 一个字符设备或块设备都有主设备号和次设备号
    - 主设备号：特定的驱动程序
    - 次设备号：使用该驱动的各设备

- 定义文件操作结构体file_operations
- 创建并初始化定义结构体cdev
  - 在linux内核中用cdev描述字符设备
- 将cdev注册到系统，和对应的设备号绑定
  - `int cdev_add(struct cdev* dev, dev_t num, unsigned int count)`
  - `void cdev_del(struct cdev* dev)`
- 实现file_operations中的各个函数
- 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
  - 定义设备名，主设备号，`mknod /dev/${device}0 c $major 0`

华为

1. 2019年底EulerOS正式被推送开源社区，命名为openEuler
2. openEuler是什么?
   - openEuler是一个开源、免费的linux发行平台
   - 支持x86、ARM、RISC-V等多种处理器架构
   - 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。



3. 线程间通信ITC
   - 互斥机制主要使用自旋锁来实现。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。
   - 同步机制主要使用**信号量**来实现。openEuler中提供**down原语与up原语**，能够实现线程的同步运行
   - openEuler增强了两种进程间通信机制：**共享内存**与**消息传递机制**
4. openEuler内存页相关说明
   - 各级页表的页表项大小为**8B**
   - 将标准大页封装为一个**伪文件系统**（hugetlbfs）提供给用户程序申请并访问
   - openEuler采用Least Recently Used （**LRU**）最近最久未使用策略实现页选择换出
   - 页在未来被访问的概率只能预测，不能精准判断
5. 鲲鹏处理器是基于ARMv8-64位RISC指令集开发的通用处理器，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
6. openEuler对通用Linux操作系统作了增强
7. 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

8. openEuler提供“**NUMA感知队列自旋锁**”实现互斥机制以减小NUMA体系结构中使 用自旋锁的开销
9. openEuler通过提供鲲鹏加速引擎（Kunpeng Accelerator Engine，KAE）插件，使能Kunpeng硬件加速能力，包括：
    - **对称/非对称加密**
    - **数字签名**
    - **压缩解压缩**等算法，用于加速SSL/TLS应用和数据压缩