

辅助学习资料

参考书 1: 《Unix 环境高级编程》W.Richard Stevens [美]

本讲课堂义作为 APUE 的引导。适合初学 Linux 的学员。

《TCP/IP 详解》(3 卷), 《UNIX 网络编程》(2 卷)

参考书 2: 《Linux 系统编程》RobertLoVe [美]

参考书 3: 《Linux/UNIX 系统编程手册》Michael Kerrisk [德]

参考书 4: 《Unix 内核源码剖析》青柳隆宏[日]

业内知名: 《Linux 内核源代码情景分析》、《Linux 内核设计与实现》、《深入理解 Linux 内核》

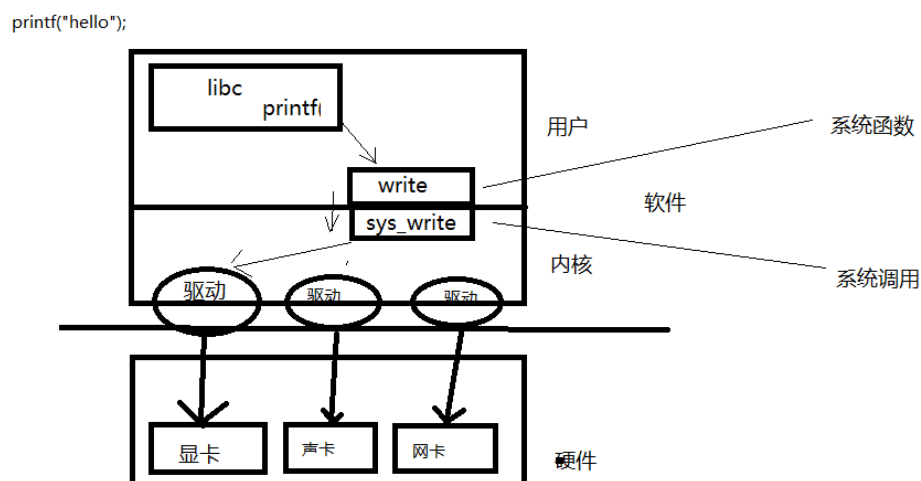
文件 IO

系统调用

什么是系统调用:

由操作系统实现并提供给外部应用程序的编程接口。(Application Programming Interface, API)。是应用程序同系统之间数据交互的桥梁。

C 标准函数和系统函数调用关系。一个 helloworld 如何打印到屏幕。





C 标准库文件 IO 函数。

fopen、fclose、fseek、fgetc、fputs、fread、fwrite.....

r 只读、 r+读写

w 只写并截断为 0、 w+读写并截断为 0

a 追加只写、 a+追加读写

open/close 函数

函数原型：

```
int open(const char *pathname, int flags);
```

```
int open(const char *pathname, int flags, mode_t mode);
```

```
int close(int fd);
```

常用参数



O_RDONLY、O_WRONLY、O_RDWR

O_APPEND、O_CREAT、O_EXCL、O_TRUNC、O_NONBLOCK

创建文件时，指定文件访问权限。权限同时受 umask 影响。结论为：

文件权限 = mode & ~umask

使用头文件：<fcntl.h>

open 常见错误：

1. 打开文件不存在
2. 以写方式打开只读文件(打开文件没有对应权限)
3. 以只写方式打开目录

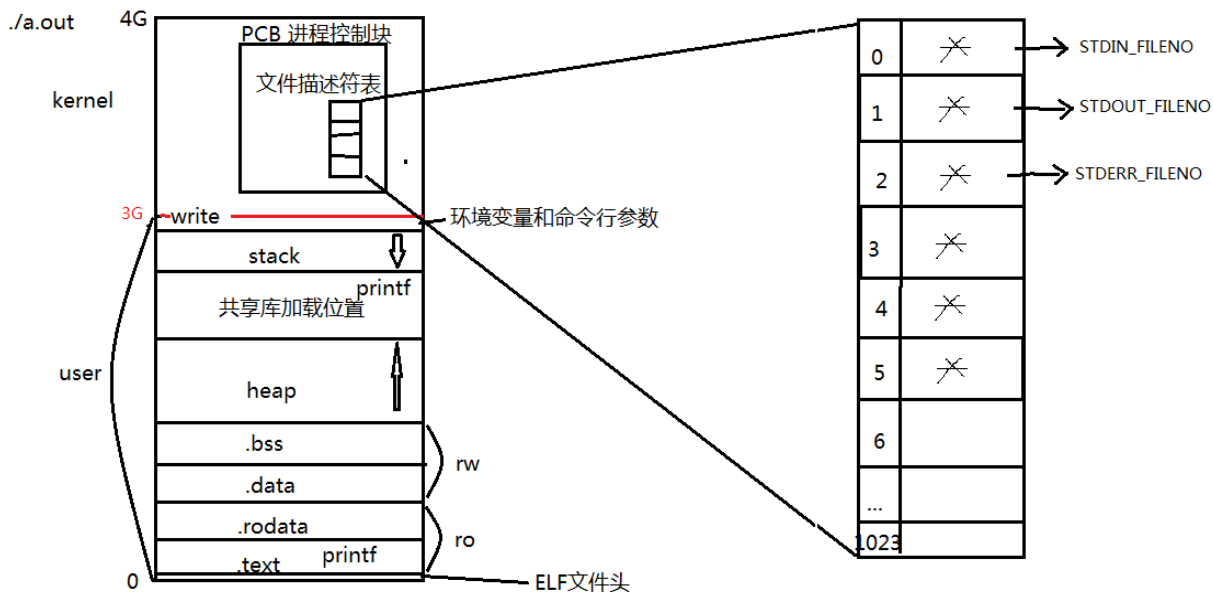
文件描述符：



PCB 进程控制块

可使用命令 locate sched.h 查看位置：
/usr/src/linux-headers-3.16.0-30/include/linux/sched.h

struct task_struct { 结构体



文件描述符表

结构体 PCB 的成员变量 `file_struct *file` 指向文件描述符表。

从应用程序使用角度，该指针可理解记忆成一个字符指针数组，下标 0/1/2/3/4...找到文件结构体。

本质是一个键值对 0、1、2...都分别对应具体地址。但键值对使用的特性是自动映射，我们只操作键不直接使用值。

新打开文件返回文件描述符表中未使用的最小文件描述符。

```

STDIN_FILENO    0
STDOUT_FILENO   1
STDERR_FILENO   2
    
```

最大打开文件数

一个进程默认打开文件的个数 1024。

命令查看 `ulimit -a` 查看 open files 对应值。默认为 1024

可以使用 `ulimit -n 4096` 修改

当然也可以通过修改系统配置文件永久修改该值，但是不建议这样操作。

`cat /proc/sys/fs/file-max` 可以查看该电脑最大可以打开的文件个数。受内存大小影响。

FILE 结构体

主要包含文件描述符、文件读写位置、IO 缓冲区三部分内容。



```
struct file {  
    ...  
    文件的偏移量;  
    文件的访问权限;  
    文件的打开标志;  
    文件内核缓冲区的首地址;  
    struct operations * f_op;  
    ...  
};
```

查看方法:

(1) /usr/src/linux-headers-3.16.0-30/include/linux/fs.h

(2) lxr: 百度 lxr → lxr.oss.org.cn → 选择内核版本(如 3.10) → 点击 File Search 进行搜

索

- 关键字: “include/linux/fs.h” → Ctrl+F 查找 “struct file {”
- 得到文件内核中结构体定义
- “struct file_operations” 文件内容操作函数指针
- “struct inode_operations” 文件属性操作函数指针

read/write 函数

```
ssize_t read(int fd, void *buf, size_t count);  
ssize_t write(int fd, const void *buf, size_t count);
```

read 与 write 函数原型类似。使用时需注意: read/write 函数的第三个参数。

练习: 编写程序实现简单的 cp 功能。

程序比较: 如果一个只读一个字节实现文件拷贝, 使用 read、write 效率高, 还是使用对应的标库函数(fgetc、fputc)效率高呢?

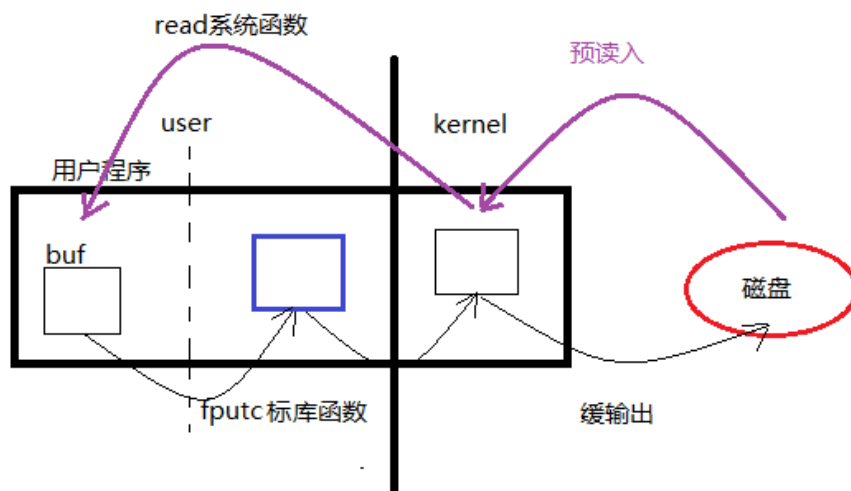
strace 命令

shell 中使用 strace 命令跟踪程序执行, 查看调用的系统函数。

缓冲区

read、write 函数常常被称为 Unbuffered I/O。指的是无用户及缓冲区。但不保证不使用内核缓冲区。

预读入缓输出



错误处理函数:

错误号: errno

perror 函数: **void perror(const char *s);**
strerror 函数: **char *strerror(int errnum);**

查看错误号:

/usr/include/asm-generic/errno-base.h
/usr/include/asm-generic/errno.h

```
#define EPERM      1  /* Operation not permitted */
#define ENOENT     2  /* No such file or directory */
#define ESRCH     3  /* No such process */
#define EINTR     4  /* Interrupted system call */
#define EIO       5  /* I/O error */
#define ENXIO     6  /* No such device or address */
#define E2BIG     7  /* Argument list too long */
#define ENOEXEC   8  /* Exec format error */
#define EBADF     9  /* Bad file number */
#define ECHILD    10 /* No child processes */
#define EAGAIN    11 /* Try again */
#define ENOMEM    12 /* Out of memory */
#define EACCES    13 /* Permission denied */
```



```
#define EFAULT    14 /* Bad address */
#define ENOTBLK   15 /* Block device required */
#define EBUSY     16 /* Device or resource busy */
#define EXIST     17 /* File exists */
#define EXDEV     18 /* Cross-device link */
#define ENODEV    19 /* No such device */
#define ENOTDIR   20 /* Not a directory */
#define EISDIR    21 /* Is a directory */
#define EINVAL    22 /* Invalid argument */
#define ENFILE    23 /* File table overflow */
#define EMFILE    24 /* Too many open files */
#define ENOTTY    25 /* Not a typewriter */
#define ETXTBSY   26 /* Text file busy */
#define EFBIG     27 /* File too large */
#define ENOSPC    28 /* No space left on device */
#define EPIPE     29 /* Illegal seek */
#define EROFS     30 /* Read-only file system */
#define EMLINK    31 /* Too many links */
#define EPIPE     32 /* Broken pipe */
#define EDOM      33 /* Math argument out of domain of func */
#define ERANGE    34 /* Math result not representable */
```

阻塞、非阻塞

读常规文件是不会阻塞的，不管读多少字节，`read` 一定会在有限的时间内返回。从终端设备或网络读则不一定，如果从终端输入的数据没有换行符，调用 `read` 读终端设备就会阻塞，如果网络上没有接收到数据包，调用 `read` 从网络读就会阻塞，至于会阻塞多长时间也是不确定的，如果一直没有数据到达就一直阻塞在那里。同样，写常规文件是不会阻塞的，而向终端设备或网络写则不一定。

现在明确一下阻塞（Block）这个概念。当进程调用一个阻塞的系统函数时，该进程被置于睡眠（Sleep）状态，这时内核调度其它进程运行，直到该进程等待的事件发生了（比如网络上接收到数据包，或者调用 `sleep` 指定的睡眠时间到了）它才有可能继续运行。与睡眠状态相对的是运行（Running）状态，在 Linux 内核中，处于运行状态的进程分为两种情况：

正在被调度执行。CPU 处于该进程的上下文环境中，程序计数器（`eip`）里保存着该进程的指令地址，通用寄存器里保存着该进程运算过程的中间结果，正在执行该进程的指令，正在读写该进程的地址空间。

就绪状态。该进程不需要等待什么事件发生，随时都可以执行，但 CPU 暂时还在执行另一个进程，所以该进程在一个就绪队列中等待被内核调度。系统中可能同时有多个就绪的进程，那么该调度谁执行呢？内核的调度算法是基于优先级和时间片的，而且会根据每个进程的运行情况动态调整它的优先级和时间片，让每个进程都能比较公平地得到机会执行，同时要兼顾用户体验，不能让和用户交互的进程响应太慢。

阻塞读终端:	【block_readtty.c】
非阻塞读终端	【nonblock_readtty.c】
非阻塞读终端和等待超时	【nonblock_timeout.c】

注意，阻塞与非阻塞是针对文件而言的。而不是 read、write 等的属性。read 终端，默认阻塞读。

总结 read 函数返回值：

1. 返回非零值： 实际 read 到的字节数
2. 返回-1： 1): errno != EAGAIN (或!= EWOULDBLOCK) read 出错
2): errno == EAGAIN (或== EWOULDBLOCK) 设置了非阻塞读，并且没有数据到达。
3. 返回 0: 读到文件末尾

lseek 函数

文件偏移

Linux 中可使用系统函数 lseek 来修改文件偏移量(读写位置)

每个打开的文件都记录着当前读写位置，打开文件时读写位置是 0，表示文件开头，通常读写多少个字节就会将读写位置往后移多少个字节。但是有一个例外，如果以 O_APPEND 方式打开，每次写操作都会在文件末尾追加数据，然后将读写位置移到新的文件末尾。lseek 和标准 I/O 库的 fseek 函数类似，可以移动当前读写位置（或者叫偏移量）。

回忆 fseek 的作用及常用参数。 SEEK_SET、SEEK_CUR、SEEK_END

int fseek(FILE *stream, long offset, int whence); 成功返回 0；失败返回-1

特别的：超出文件末尾位置返回 0；往回超出文件头位置，返回-1

off_t lseek(int fd, off_t offset, int whence); 失败返回-1；成功：返回的值是较文件起始位置向后的偏移量。

特别的：lseek 允许超过文件结尾设置偏移量，文件会因此被拓展。

注意文件“读”和“写”使用同一偏移位置。

【lseek.c】

lseek 常用应用：

1. 使用 lseek 拓展文件：write 操作才能实质性的拓展文件。单 lseek 是不能进行拓展的。
一般：write(fd, "a", 1);



od -tcx filename 查看文件的 16 进制表示形式
od -tcd filename 查看文件的 10 进制表示形式

2. 通过 lseek 获取文件的大小: lseek(fd, 0, SEEK_END); 【lseek_test.c】

【最后注意】: lseek 函数返回的偏移量总是相对于文件头而言。

fcntl 函数

改变一个【已经打开】的文件的 访问控制属性。
重点掌握两个参数的使用，F_GETFL 和 F_SETFL。
【fcntl.c】

ioctl 函数

对设备的 I/O 通道进行管理，控制设备特性。(主要应用于设备驱动程序中)。

通常用来获取文件的【物理特性】(该特性，不同文件类型所含有的值各不相同)
【ioctl.c】

传入传出参数

传入参数:

const 关键字修饰的 指针变量 在函数内部读操作。 char *strcpy(const char *src, char *dst);

传出参数:

1. 指针做为函数参数
2. 函数调用前，指针指向的空间可以无意义，调用后指针指向的空间有意义，且作为函数的返回值传出
3. 在函数内部写操作。

传入传出参数:

1. 调用前指向的空间有实际意义
2. 调用期间在函数内读、写(改变原值)操作
3. 作为函数返回值传出。



扩展阅读：

关于虚拟 4G 内存的描述和解析：

一个进程用到的虚拟地址是由内存区域表来管理的，实际用不了 4G。而用到的内存区域，会通过页表映射到物理内存。

所以每个进程都可以使用同样的虚拟内存地址而不冲突，因为它们的物理地址实际上是不同的。内核用的是 3G 以上的 1G 虚拟内存地址，

其中 896M 是直接映射到物理地址的，128M 按需映射 896M 以上的所谓高位内存。各进程使用的是同一个内核。

首先要分清“可以寻址”和“实际使用”的区别。

其实我们讲的每个进程都有 4G 虚拟地址空间，讲的都是“可以寻址”4G，意思是虚拟地址的 0-3G 对于一个进程的用户态和内核态来说是可以访问的，而 3-4G 是只有进程的内核态可以访问的。并不是说这个进程会用满这些空间。

其次，所谓“独立拥有的虚拟地址”是指对于每一个进程，都可以访问自己的 0-4G 的虚拟地址。虚拟地址是“虚拟”的，需要转化为“真实”的物理地址。

好比你有你的地址簿，我有我的地址簿。你和我的地址簿都有 1、2、3、4 页，但是每页里面的实际内容是不一样的，我的地址簿第 1 页写着 3 你的地址簿第 1 页写着 4，对于你、我自己来说都是用第 1 页（虚拟），实际上用的分别是第 3、4 页（物理），不冲突。

内核用的 896M 虚拟地址是直接映射的，意思是只要把虚拟地址减去一个偏移量（3G）就等于物理地址。同样，这里指的还是寻址，实际使用前还是要分配内存。而且 896M 只是个最大值。如果物理内存小，内核能使用（分配）的可用内存也小。