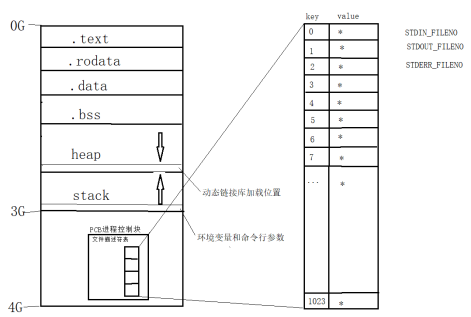
**PCB 进程控制块（在内核区存着）（本质上是结构体）**

 可 使 用 命 令 locate sched.h 查 看 位 置 ：

 /usr/src/linux-headers-3.13.0-32/include/linux/sched.h

 struct task\_struct { 结构体



（1）text段-代码段

text段存放程序代码，运行前就已经确定（编译时确定），通常为只读。

（2）rodata段（read-only-data）-常量区

rodata段存储常量数据，比如程序中定义为const的全局变量，#define定义的常量，以及诸如“Hello World”的字符串常量。只读数据，存储在ROM中。

注意：

const修饰的全局变量在常量区；const修饰的局部变量只是为了防止修改，没有放入常量区。

编译器会去掉重复的字符串常量，程序的每个字符串常量只有一份。

（3）data段

data存储已经初始化的全局变量，属于静态内存分配。（注意：初始化为0的全局变量还是被保存在BSS段）

static声明的变量也存储在数据段。

（4）bss段

bss段存储没有初值的全局变量或默认为0的全局变量，属于静态内存分配。

执行期间必须将bss段内容全部设为0。

（5）stack段-栈

stack段存储参数变量和局部变量，由系统进行申请和释放，属于静态内存分配。

stack的特点是先进先出，可用于保存/恢复调用现场。

（6）heap-堆

heap段是程序运行过程中被动态分配的内存段，由用户申请和释放（例如malloc和free）。

申请时至少分配虚存，当真正存储数据时才分配物理内存；释放时也不是立即释放物理内存，而是可能被重复利用。

**文件描述符表**

结构体 PCB 的成员变量 struct files\_struct \*files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。

0 标准输入 STDIN\_FILENO

1 标准输出 STDOUT\_FILENO

2 标准错误 STDERR\_FILENO

阻塞、非阻塞：O\_NONBLOCK 普通文件没有这一说，说的是设备文件、网络文件

**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。

**read/write函数**

ssize\_t read(int fd, void\* buf, size\_t count);

参数：

fd：文件描述符；

buf：数据缓冲区，存储我们读到的数据

count：缓冲的大小

返回值：

成功：读到的字节数，0：读到了文件的末尾

失败：-1， 设置errno O\_NONBLOCK

ssize\_t write(int fd, const void\* buf, size\_t count);

参数：

fd：文件描述符；

buf：待写出数据的缓冲区

count：数据大小

返回值：

成功：写入的字节数

失败：-1， 设置errno

**练习：编写程序实现简单的 cp 功能。**

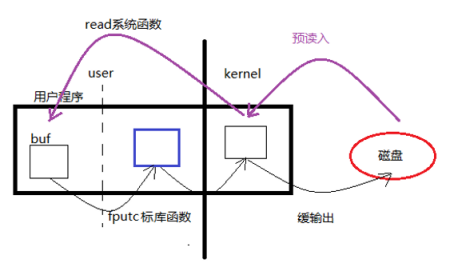
程序比较：如果一个只读一个字节实现文件拷贝，使用 read、write 效率高，还是使用对应的标库函数(fgetc、fputc)效率高呢？

首先应该明确缓冲区的概念：

**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**



答案：如果要比对两种写入方式，应该统一变量，假设每次只写入一个字节，因为read、write 函数 每次写一个字节都要从用户区跨越到内核去，这是一个相对较慢的过程，虽然库函数 也是调用系统函数r/w，但是库函数先往缓冲区上写数据凑够一定数量之后 一次性写入内核，这样的话相对较快。

**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。

**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

**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。

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，阻塞与非阻塞是对于文件而言的。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

**fcntl函数：**

int fcntl(int fd, int cmd, ... /\* arg \*/ );

fr = fcntl(STDIN\_FILENO, F\_GETFL);

fr |= O\_NONBLOCK;

fcntl(STDIN\_FILENO, F\_SETFL, fr);

F\_GETFL：获取文件属性

F\_SETFL：设置文件属性 （还需要传将要设置的文件属性）

**lseek函数：**

off\_t lseek(int fd, off\_t offset, int whence)

参数：

fd：待操作的文件的文件描述符

offset：偏移量

whence：偏移的起始位置（参照物） SEEK\_SET | SEEK\_CUR | SEEK\_END

返回值：

成功：参照起始位置的偏移量

失败：-1 设置errno

应用：

1、文件的读写使用同一偏移量

2、获取文件大小

3、拓展文件大小：要想真正拓展文件的大小，必须要引起IO操作

例如：lseekd(fd,199,SEEK\_END);

write(fd,"\0",1);（这部就是在引起io操作）

**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(cnost 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。