基于操作系统编程技术复习资料

考试范围：
第一讲：前言~第四讲：进程
题型：选择（10 分）+判断（10 分)+大题（80 分）：简答+代码填空（重点函数的参数意义要知道）。。。

第一讲

- Linux 的诞生：1991 年 10 月，芬兰的计算机系大二学生 Linus 开发了 Linux
- Linux 发展的五大支柱：UNIX 操作系统、MINIX 操作系统、GNU 计划、POSIX 标准、Internet 网络
- Linux 的近况：Ubuntu、SUSE、Fedora、DebianRed 、Hat、Slackware
- Ubuntu 默认没有安装编译环境
  - #apt-get install build-essential
- g++的基本用法（代码 1.1）
  - #g++ test.cpp
  - #g++ -o test test.cpp
  - #g++ -c test.cpp 生成目标文件 test.o
  - #g++ -o test test.cpp -g
  - #g++ -O -o test test.cpp 优化
  - #g++ -S test.cpp 产生汇编代码 test.s
  - #g++ -E test.cpp > my.txt 只激活预处理，将结果保存在 my.txt 中
  - #g++ -I../gtest/include test.cpp 指定头文件路径
  - #g++ -L../gtest -lgtest test.cpp 指定库的路径
  - #g++ test.cpp -DOK=2 设宏 OK 为 2
  - #g++ test.cpp -DOK 定义宏 OK
- 调试：gdb

程序编译的过程：预处理阶段，编译阶段，汇编阶段，链接阶段。

第二讲 文件的操作

■ 文件 I/O

- 文件的基本操作（打开、定位、读写、关闭）

文件操作基本顺序

- 打开 open
- 创建 creat
- 定位 lseek
- 读 read
- 写 write
- 关闭 close

✧ open 函数（重点）

- 用于打开或者创建一个文件
- 函数原型
  - #include<fcntl.h>

- int open(const char* pathname, int oflag, ...)
- 参数
  - 第一个参数 pathname：要打开或者创建的文件名
  - 第二个参数 oflag：用于指定文件打开模式、标志等信息。
- 第二个参数 oflag：
  - Linux 头文件已经为文件打开模式、标志等定义了若干的宏
  - oflag 需要指定这些宏
  - 宏定义在/usr/include/bits/fcntl.h 中
  - 在该头文件中，只读打开标志被定义为
  - #define O_RDONLY 00
- oflag：
  - 文件打开模式标志

以下三个标志必须指定一个且只能指定一个

- O_RDONLY ： 只读打开
- O_WRONLY ： 只写打开
- O_RDWR ： 读写打开

  - 其他文件标志

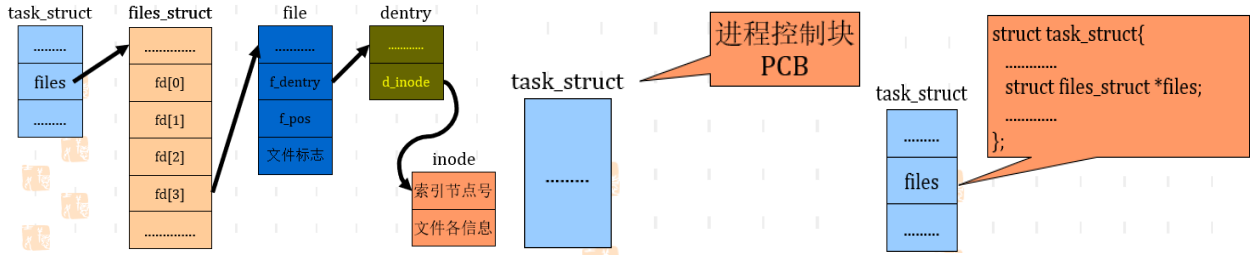
下面的标志是可以选择的，可通过 C 语言的或运算与文件打开标志进行组合

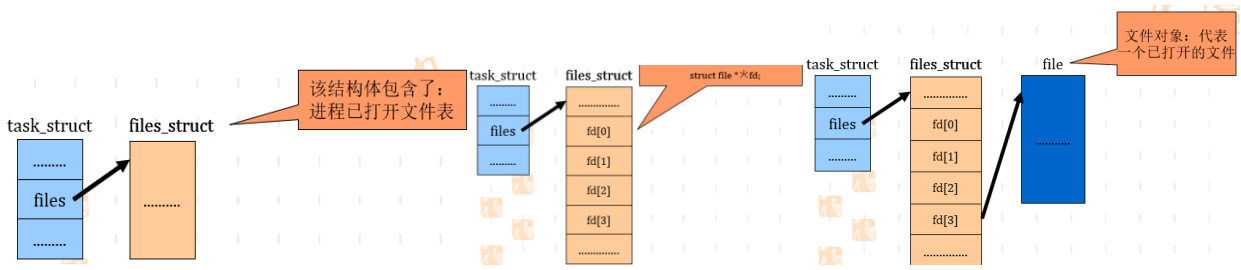
- O_APPEND：每次写的数据都添加到文件尾
- O_TRUNC：若此文件存在，并以读写或只写打开，则文件长度为 0
- O_CREAT：若文件不存在，则创建该文件。此时，open 函数需要第三个参数，用于指定该文件的访问权限位（后面描述）
- O_EXCL：若同时指定了 O_CREAT 标志，而文件已经存在，则会出错。可用于测试文件是否存在

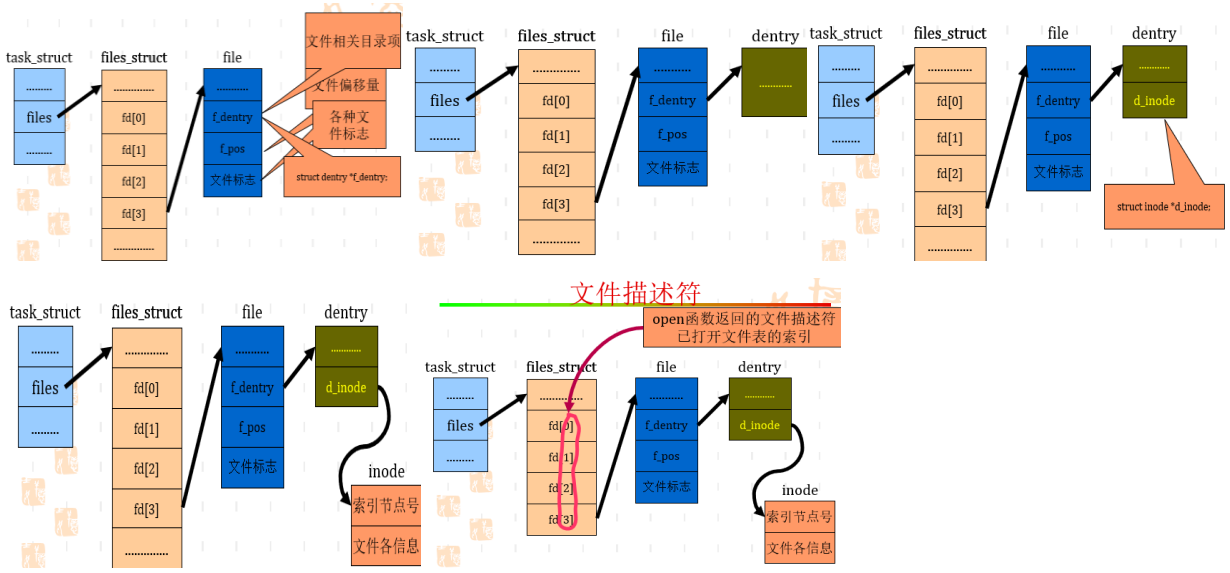
- 返回值
  - int open(const char* pathname, int oflag, …)
  - 返回值：整型数据
    - 成功时，返回文件描述符
    - 出错时，返回-1
- 需要了解进程打开文件时，内核创建或涉及到的一系列数据结构

进程打开文件的内核数据结构

-







- 出错处理(2.1)
  - errno.h 头文件中，定义了 errno：当 API 调用出错时，errno 说明出错的具体原因
  - 可简单地将 errno 理解成整型数据
  - 出错信息转换成可读字符串

```
#include<string.h>
char* strerror(int errno);
```

  - 以前的定义：extern int errno;
  - 多线程环境：

```
extern int * __errno_location();
#define errno (*__errno_location())
```

✧ perror 函数

- perror 函数根据当前的 errno，输出一条出错信息
- 函数原型

```
#include<stdio.h>
void perror(const char* msg);
```

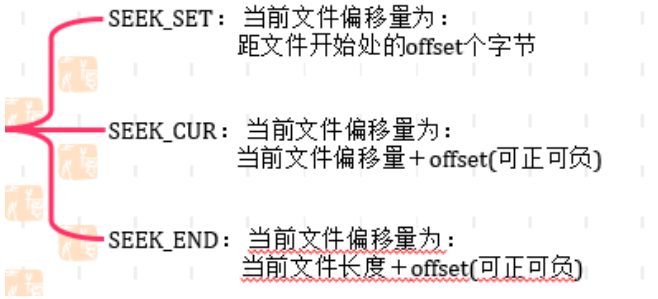
- 该函数输出：

msg 指向的字符串：errno 对应的出错信息

✧ lseek 函数

- lseek 函数用于修改当前文件偏移量
- 当前文件偏移量的作用：规定了从文件什么地方开始进行读、写操作
- 通常，读、写操作结束时，会使文件偏移量增加读写的字节数
- 当打开一个文件时，偏移量被设置为 0
- 函数原型：
  - off_t lseek(int filedes, off_t offset, int whence)
- 参数
  - 第一个参数 filedes：open/creat 函数返回的文件描述符
  - 第二个参数 offset：
    - 相对偏移量：需结合 whence 才能计算出真正的偏移量
    - 类型 off_t：32 位机是 32 位数据类型，64 位是 64 位

- 参数
  - 第三个参数Whence：该参数取值是三个常量之一



  - 返回值：
    - 若成功，返回新的文件偏移量
    - 若出错，返回-1
  - 获得当前的偏移量
    - off_t CurrentPosition;
    - CurrentPosition = lseek(fd, 0, SEEK_CUR);
  - lseek 操作并不引起任何 I/O 操作，只是修改内核中的记录
  - 使用 lseek 修改文件偏移量后，当前文件偏移量有可能大于文件的长度
  - 在这种情况下，对该文件的下一次写操作，将加长该文件
  - 这样文件中形成了一个空洞。对空洞区域进行读，均返回 0

✧ read 函数

✧ 很多情况下，read 实际读出的字节数都小于要求读出的字节数
- 读普通文件，在读到要求的字节数之前，就到达了文件尾端
- 当从终端设备读时，通常一次最多读一行
- 当从网络读时，网络中的缓冲机构可能造成 read 函数返回值小于所要求读出的字节数
- 某些面向记录的设备，如磁带，一次最多返回一个记录

✧ write 函数

✧ write 出错的原因
- 磁盘满
- 没有访问权限
- 超过了给定进程的文件长度限制

✧ close 函数

✧ 当 clsoe 函数关闭文件时，会释放进程加在该文件上的所有记录锁

✧ 内核会对进程打开文件表、文件对象、索引节点表项等结构进行修改，释放相关的资源

✧ 当进程退出时，会关闭当前所有已打开的文件描述符

什么是文件描述符？->已打开文件的索引，再通过索引找到已打开文件。

索引节点：

- 文件系统索引节点的信息，存储在磁盘上
- 当需要时，调入内存，填写 VFS 的索引节点（即 inode 结构）
- 每个文件都对应了一个索引节点
- 通过索引节点号，可以唯一的标识文件系统中的指定文件

文件描述符：

- 文件描述符是已打开文件的索引，通过该值可以在 fd_array 表中检索相应的文件对象
- 文件描述符是一个非负的整数
- 文件描述符 0、1、2 分别对应于标准输入、标准输出、标准出错，在进程创建时，已经打开。

I/O 效率(重点)

- 程序中，影响效率的关键：BUFFSIZE 的取值
- 原因
  - Linux 文件系统采用了某种预读技术
  - 当检测到正在进行顺序读取时，系统就试图读入比应用程序所要求的更多数据
  - 并假设应用程序很快就会读这些数据
  - 当 BUFFSIZE 增加到一定程度后，预读就停止了
- 如何提高 I/O 效率
- 慢在哪里？
  - 整个磁盘操作的过程是什么
  - 每调用一次 read/write，就陷入一次内核
  - SSD 的情况
- 改进的思路
  - 尽量使磁头顺序移动，如 LSF
  - 利用缓存，减少磁盘 I/O、read/write 调用次数
  - 有时轮询比中断好

文件共享（重点、难点、考试必考(15~20)）

- 不同进程打开不同的文件

  操作系统的基本原理告知我们，每个进程都有自己的进程控制块。因此，在图中，进程 A 和进程 B 都有自己的 task_struct 结构。

  在两个独立进程各自打开不同文件的情况下，进程 A 和进程 B 都有自己独立的文件表，文件对象。只是共享了索引节点

  进程 a 在文件描述符 3 上打开了文件，而进程 B 是在文件描述符 4 上打开该文件
- 不同进程打开同一个文件

  操作系统的基本原理告知我们，每个进程都有自己的进程控制块。因此，在图中，进程 A 和进程 B 都有自己的 task_struct 结构。

  在两个独立进程各自打开同一个文件的情况下，进程 A 和进程 B 都有自己独立的文件表，文件对象。只是共享了索引节点

  进程 a 在文件描述符 3 上打开了文件，而进程 B 是在文件描述符 4 上打开该文件

  每个进程都有自己的当前文件偏移量

  在完成每个 write 后，当前文件偏移量即增加所写的字节数

  如果用 O_APPEND 标志打开了一个文件，则该标志存储在 file 结构体中。每次执行写操作时，当前偏移量首先被设置为文件长度
- 父子进程共享文件对象的情况

  进程间共享文件还有一种形式是共享文件对象。即共享 file 结构体。

  进程 AB 有各自的 PCB 和文件表。进程 A 的 fd_array [3] 和 fd_array[4]指向了同一个文件对象，当然也就共享了索引节点

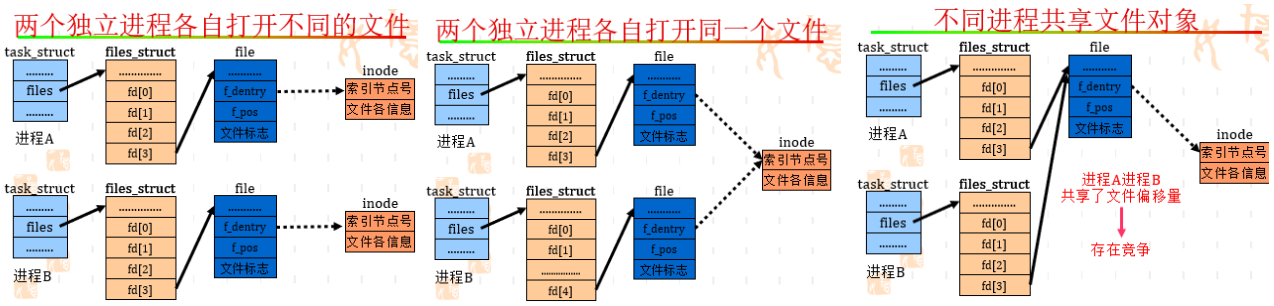
  这种情况通常是调用 fork 函数之后，父子进程之间对文件的共享

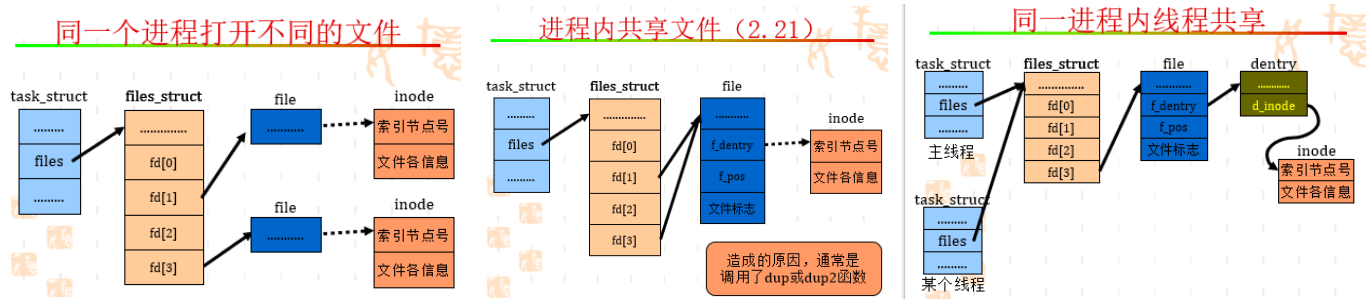
  由于文件偏移量存储在 file 对象的 f_pos 字段，所以两个进程共享。a 调用了 lseek 设置为 50，进程 b 就要从 50 处开始读写
- 同一个进程打开不同的文件

  现在只有一个进程，也就只有一个 pcb 和文件表

  说明两个表项指向同一个文件对象，两表象共享了共同一个文件对象和索引节点

- 同一个进程内的共享（dup 等）
  现在只有一个进程，也就只有一个 pcb 和文件表
  说明两个表项指向同一个文件对象，两表象共享了共同一个文件对象和索引节点
- 同一个进程内线程的共享





dup 函数

- 用于复制一个已经存在的文件描述符
- 函数原型
  - int dup(int filedes);
- 返回值
  - 成功返回新的文件描述符
  - 出错返回-1
- 参数
  - filedes：文件描述符
- dup 的含义是复制文件描述符，返回最小的文件描述符

dup 函数的作用：使新老文件描述符，都指向同一个文件对象

dup2 函数

- 用于复制一个已经存在的文件描述符
- 函数原型
  - int dup2(int filedes, int filedes2);
- dup 和 dup2 的区别
  - dup 返回的新文件描述符一定是当前可用描述符中的最小值
  - dup2 则将文件描述符复制到指定位置，即将 filedes 复制到 filedes2
  - 如果 filedes2 已经打开，dup2 则先将其关闭；若 filedes2 等于 filedes，则直接返回 filedes，而不关闭

● 其他重要 I/O 函数（重点/难点）

✧ sync、fsync、fdatasync 函数

- 通常 Linux 实现在内核中设有缓冲区高速缓存或页面高速缓存
- 大多数的磁盘 I/O 都通过缓冲区进行
- 当将数据写入文件时，内核通常先将数据复制到某一个缓冲区中
- 如果该缓冲区满或者内核需要重用该缓冲区，则将该缓冲排入到输出队列
- 等到其达到队首时，才进行实际的磁盘读写操作
- 延迟写

- 延迟写优点
  - 减少了磁盘读写次数
- 延迟写缺点
  - 降低了文件内容的更新速度
  - 当系统发生故障，高速缓冲区中的内容可能丢失
- 解决办法
  - 对缓冲区进行清理，希望将数据写入到磁盘
  - sync、fsync、fdatasync 起到了刷缓存的作用

- sync
  - 原型：
    ```
    void  sync();
    ```
  - 将所有修改过的缓冲区排入写队列，然后就返回
  - 并不等待实际的写磁盘操作结束
- sync 函数针对的是所有修改过的缓冲区，并不仅仅针对某个被修改过的文件
- 通常称为 update 的系统守护进程会周期性地调用 sync 函数，即保证定期冲洗内核缓冲区

- fsync
- 函数原型：
- ```
  int fsync(int filedes);
  ```
- 参数与返回值：
  - filedes：文件描述符
  - 返回值：成功返回 0，出错返回-1
- fsync 函数只对文件描述符 filedes 指定的单一文件起作用
- 并且等待写磁盘操作结束后才返回

- fdatasync
- 函数原型
- ```
  int fdatasync(int filedes);
  ```
- 参数与返回值
- filedes：文件描述符
- 返回值：成功返回 0，出错返回-1
- fdatasync 和 fsync 类似，但它只影响文件的数据部分；而 fsync 不仅影响文件的数据，还同步更新文件的属性。
-

	sync	fsync	fdatasync
是否等待写	否	是	是
是否更新数据	是	是	是
是否更新属性	是	是	否

✧ fcntl 函数(作用：赋值，获得文件标志和设置文件标志)

- 用于改变已经打开文件的性质
- 函数原型
  - int fcntl(int filedes, int cmd, .../* int arg */);
- 返回值
  - 成功时，返回值依赖于第二个参数 cmd
  - 出错时，返回-1
- 参数

  第一个参数 filedes：已打开文件的文件描述符

- 第二个参数 cmd 的五种取值方式：
  - 复制一个现存的描述符（cmd=F_DUPFD）
  - 获得/设置文件描述符标记（cmd=F_GETFD 或 F_SETFD）
  - 获得/设置文件状态标志（cmd=F_GETFL 或 F_SETFL）
  - 获得/设置异步 I/O 信号接收进程（cmd=F_GETOWN 或 F_SETOWN）
  - 获得/设置记录锁（cmd=F_GETLK,F_SETLK 或 F_SETLKW）
  - F_DUPFD
  - 复制文件描述符 filedes，与 dup(2)类似
  - fcntl 返回新文件描述符
  - 新描述符是尚未打开的各描述符中，大于或等于第三个参数值中，各值的最小值
- fcntl 函数与 dup、dup2 函数均用于复制文件描述符，即使不同的文件描述符指向同一个文件对象
- dup(filedes)等价于 fcntl(filedes, F_DUPFD, 0);
- dup2(filedes, filedes2)不完全等价于 close(filedes2);fcntl(filedes, F_DUPFD, filedes2);
- fcntl 与 dup2 不完全等价
- dup2 是一个原子操作，而 close 与 fcntl 则包括两个函数调用。
- 在 close 和 fcntl 之间可能被信号打断
- dup2 与 fcntl 之间有某些不同的 errno
- cmd 五种取值方式
  - F_DUPFD
  - F_GETFD
    - 将文件描述符 filedes 对应的标志，作为返回值返回。
    - 当前只定义了一个文件描述符标志 FD_CLOEXEC
  - F_SETFD
    - 设置文件描述符 filedes 对应的标志。新标志按照第三个参数设置。
  - F_GETFL
    - fcntl 函数返回文件描述符 filedes 对应的文件状态标志
    - 文件状态标志包括：
    - O_RDONLY
    - O_WRONLY
    - O_RDWR
    - O_APPEND
    - O_NONBLOCK　　非阻塞方式
    - O_SYNC　　等待写方式
    - O_ASYNC　　异步方式(仅 4.3+BSD)
  - F_SETFL
    - 将 fcntl 函数的第三个参数，设置为文件状态标志
    - 可以更改的标志包括：O_APPEND、O_NONBLOCK、O_SYNC、O_ASYNC
  - F_GETOWN
    - 获取当前接收 SIGIO 和 SIGURG 信号的进程 ID 或进程组 ID
  - F_SETOWN
    - 设置接收 SIGIO 和 SIGURG 信号的进程 ID 或进程组 ID
    - 第三个参数 arg：
      - arg>0 时，表示一个进程 ID
      - arg<0 时，其绝对值表示一个进程组 ID

✧ **ioctl 函数**

- I/O 操作的杂物箱

- 其实现的功能往往和具体的设备有关系
- 设备可以自定义自己的 ioctl 命令
- 操作系统提供了通用的 ioctl 命令
- ioctl 类似于 windows 的 DeviceIoControl 函数

■ 文件和目录

✧ ext2 文件系统在磁盘的组织

- ext2 文件系统是 Linux 土生土长的文件系统
- ext2 是 ext(Extended File System)的完善，因此，ext2 为 The Second Extended File System
- ext2 文件系统加上日志支持，即 ext3
- ext2 和 ext3 在磁盘上的布局大致相同，只是 ext3 多出了一个特殊的 inode，用于记录文件系统日志
- 同一个文件系统中，block 的大小都是相同的
- 不同的文件系统，block 的大小可以不同
- 典型的 block 大小为 1k 或者 4k
- 若干块聚集在一起，形成一个块组
- 分区被划分成若干个块组
- 每个块组所包括的块个数相同
- 超级块
  - 每个块组都包含有一个相同的超级块
  - 超级块重复的主要目的：灾难恢复
  - 超级块用于存放文件系统的基本信息
  - s_magic：ext2 文件系统标识 0xef53
  - s_log_block_size：可由它得出块大小
  - 块组包括的块个数、包括的索引节点个数，总的块个数
- 组描述符
  - bg_block_bitmap：指向块位图
  - bg_inode_bitmap：指向索引节点位图
  - bg_inode_table：指向索引节点表
- 块位图
  - 当某个 bit 为 1，表示该 bit 对应的数据块被占用
  - 当某个 bit 为 0，表示该 bit 对应的数据块未被占用
- 索引节点位图
  - 当某个 bit 为 1，表示该 bit 对应的索引节点被占用
  - 当某个 bit 为 0，表示该 bit 对应的索引节点未被占用
- 索引节点表
  - 索引节点表由若干个索引节点组成
  - 一个索引节点对应了一个文件（目录也是一种文件）
  - 每个索引节点都有一个编号，这个编号是全局的，从 1 开始计数
- 数据块

✧ stat、fstat、lstat 函数

- stat 函数
  - 用于获取有关文件的信息结构
  - 函数原型

```
int stat(const char* restrict pathname,
```

struct stat* restrict buf);

- restrict 关键字 C99 标准引入的
- 只能用于限定指针
- 表明指针是访问一个数据对象的唯一且初始的方式
- 实际上是用于指示编译器对代码进行优化的

- 参数与返回值
- 第一个参数 pathname：文件名，需要获取该文件的信息
- 第二个参数 buf：stat 函数将 pathname 对应的文件信息，填入 buf 指向的 stat 结构中
- 返回值：0 成功；-1 出错

fstat 函数、lstat 函数

- 用于获取有关文件的信息结构
- 函数原型
- int stat(const char* restrict pathname,
- struct stat* restrict buf);
- int fstat(int filedes, struct stat *buf);
- int lstat(const char* restrict pathname,
- struct stat* restrict buf);
- stat 通过文件名返回文件的信息，fstat 通过文件描述符。
- lstat 返回符号链接本身的信息，stat 返回符号链接所引用的文件信息。

✧ 文件的基本性质（考试重点、难点）

文件类型（考点）

- UNIX 或 Linux 系统中的常见文件类型有：
  - **普通文件**
  - **目录文件**
  - **字符特殊文件 提供对设备不带缓冲的访问**
  - **块特殊文件 提供对设备带缓冲的访问**
  - **FIFO 文件 用于进程间的通信，命名管道**
  - **套接口文件 用于网络通信**
  - **符号链接 使文件指向另一个文件**
- **使用如下的宏，判断文件类型**
  - **普通文件 S_ISREG()**
  - **目录文件 S_ISDIR()**
  - **字符特殊文件 S_ISCHR()**
  - **块特殊文件 S_ISBLK()**
  - **FIFO 文件 S_ISFIFO()**
  - **套接口文件 S_ISSOCK()**
  - **符号连接 S_ISLINK()**

用户 ID 和组 ID（考试重点）

- 第一种 ID：
  - Linux 是一个多用户的操作系统。每个用户都有一个 ID，用以唯一标识该用户。这个 ID，被称为 UID。
  - 每个用户都属于某一个组，组也有一个 ID。这个 ID，被称为组 ID，GID。
- 第二种 ID：文件所有者相关
  - 文件所有者 ID：拥有某文件的用户的 ID

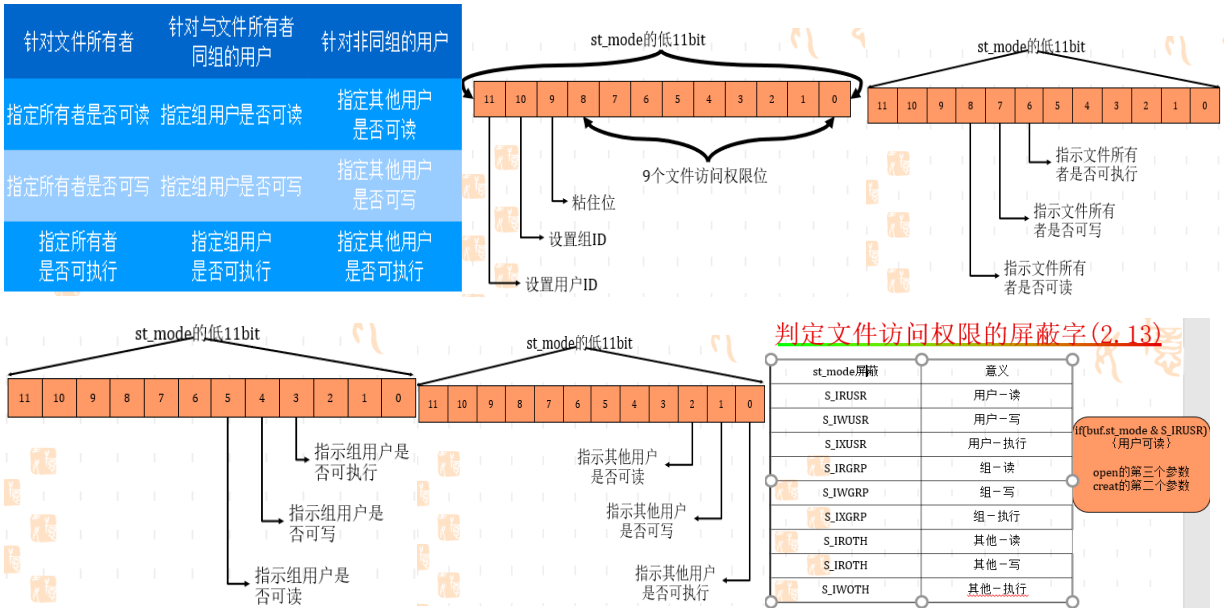
- 文件所有者组 ID：拥有某文件的用户所属组的 ID
- **第三种 ID**：实际用户 ID 和实际组 ID（重点）
  - 进程的实际用户 ID：运行该进程的用户的 ID
  - 进程的实际组 ID：运行该进程的用户所属的组 ID
- **第四种 ID**：有效用户 ID 和有效组 ID(重点)
  - 进程的有效用户 ID：用于文件访问权限的检查（大多数情况下有效用户/组 ID=实际用户/组 ID）
  - 进程的有效组 ID：
- 设置用户 ID 位和设置组 ID 位（考点）
  - 在可执行文件的权限标记中，有一个“设置用户 ID 位”
  - 若该位被设置，表示：执行该文件时，进程的有效用户 ID 变为文件的所有者
  - 对于设置组 ID 位类似
- 通过命令行设置用户 ID 位（考点）
  - chmod u+s filename　　chmod g+s filename
  - chmod u-s filename　　chmod g-s filename
- **第五种 ID：（重点）**
  - 保存的设置用户 ID
  - 保存的设置组 ID
  - 上述两者在执行一个程序时包含了有效用户 ID 和有效组 ID 的副本

文件访问权限（考点（st_mode））

为什么要访问权限？

Linux 是一个多用户的操作系统，从安全或者隐私考虑，通常一个文件并不是所有用户都能够访问的。

访问权限针对三种不同的用户：



- 当打开一个任意类型的文件时，对该文件路径名中包含的**每一个目录都应具有执行许可权；**
- **读许可权允许读目录**，获得该目录中所有文件名的列表。
- 目录具有执行权，表示可以搜索该目录（或进入该目录）；
- 为了在 open 函数中对一个文件指定 O_TRUNC 标志，必须对该文件具有写许可权；
- 为了在一个目录中创建一个新文件，必须对该目录具有写许可权和执行许可权；
- 为了删除一个文件，必须对包含该文件的目录具有写许可权和执行许可权，对该文件本身则不需要有读、写许可权；
- 如果用 6 个 exec 函数中的任何一个执行某个文件，都必须对该文件具有执行许可权。

- 进程访问文件时，内核就进行文件**存取许可权**测试。这种测试可能涉及到文件的所有者 ID、进程有效 ID 以及进程的添加组 ID。两个所有者 ID 是文件的性质，而有效 ID 与添加组 ID 是进程的性质：
  - 若进程的有效用户 ID 是 0，则允许存取；
  - 若进程的有效用户 ID 等于文件的所有者 ID（即该进程拥有文件）
    - **若所有者存取许可权被设置，则允许存取**
    - **否则拒绝存取**
  - 若进程的有效组 ID 或进程的添加组 ID 之一等于文件组 ID：
    - **若组存取许可权被设置，则允许存取**
    - **否则拒绝存取**
  - 若其他用户存取许可权被设置，则允许存取，否则拒绝存取
- 综上所述，内核按顺序执行上述 4 步测试。
- 若进程拥有此文件，则按用户存取许可权批准或拒绝该进程对文件的存取－不查看组存取许可权。
- 相类似，若进程并不拥有该文件，但进程属于某个适当的组，则按组存取许可权批准或拒绝该进程对文件的存取－不查看其他用户的许可权。

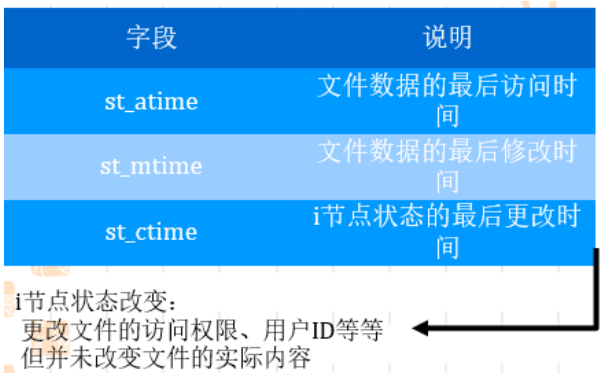
为什么需要设置用户/组 ID？（考点）

- 每个用户都可以使用 passwd 命令修改密码
- passwd 命令需要修改/etc/passwd 文件
- ls -l /etc/passwd
- 该文件属于超级用户，非超级用户无修改权限
- ls -l /usr/bin/passwd
- passwd 设置了设置用户 ID 位

新文件和目录的所有权

- 新文件的所有者 ID：即创建该文件的进程的有效用户 ID
- 新文件的组 ID：两种方式
  - 创建该文件的进程的有效组 ID
  - 新文件所在目录的组 ID
- Linux 中的处理
  - 取决于新文件所在目录的设置组 ID 是否被设置
  - 若设置，新文件的组 ID 即目录的组 ID

文件时间



utime 函数

- 用于更改一个文件的访问时间、修改时间
- 函数原型
  - int utime(const char* pathname,
    const struct utimbuf *times);

- 参数与返回值
  - 返回值：0 成功；-1 出错
  - 第一个参数 pathname：文件名，即需要修改时间属性的文件
- 参数与返回值
  - 第二个参数 times：指向 utimbuf 结构的常指针

```
struct utimbuf {
    time_t actime;     //访问时间
    time_t modtime;    //修改时间
};
```

  - 当 times＝NULL 时，使用当前时间更改文件的最后访问时间、最后修改时间
  - 否则，使用 utimbuf 中相关字段进行修改

✧ 修改文件属性的函数

- access 函数
  - 用于按实际用户 ID 和实际组 ID 进行存取许可权测试
  - 注意：此时不管设置用户 ID 和设置组 ID
- umask 函数
  - 用于为进程设置文件方式创建屏蔽字，即参与指定文件的访问权限
  - umask(0)
    - 未设置任何屏蔽字，creat 或 open 相关参数即指定了文件的访问权限
  - umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)
    - 禁止所有组和其他用户的存取许可权
    - 即使 open/creat 设置了也无用
- chmod 和 fchmod 函数
  - chmod 函数用于改变现有文件的存取许可权
  - 进程的有效用户 ID 等于文件的所有者，或者进程具有超级用户权限，才能改变文件的许可权位。
  - chmod 在下列条件下自动清除两个许可权位

    如果试图设置普通文件的粘住位（S_ISVTX），而且又没有超级用户权限，则 mode 中的粘住位自动被关闭。这意味着只有超级用户才能设置普通文件的粘住位
  - chmod 在下列条件下自动清除两个许可权位
    - 新创建文件的组 ID 可能不是调用进程所属的组（新文件的组 ID 可能是父目录的组 ID），
    - 如果新文件的组 ID 不等于进程的有效组 ID，以及进程没有超级用户权限，
    - 那么设置-组-ID 位自动关闭。这就防止了用户创建一个设置-组-ID 文件，而该文件是由并非该用户所属的组拥有的。
  - Fchmod 函数用于改变现有文件的存取许可证
- chown、fchown 和 lchown 函数
  - chown 函数用于更改文件的用户 ID 和组 ID
  - lchown 更改符号链接本身的所有者，而不是符号链接所指向的文件
  - 超级用户进程可以更改文件的用户 ID
  - 非超级用户进程更改文件 ID
    - 进程拥有该文件（有效用户 ID 等于文件的所有者 ID）
    - owner 等于文件的用户 ID，group 等于进程的有效组 ID 或进程的添加组 ID 之一
  - 你可以修改你所拥有的文件的组 ID，但只能改到你所属于的组
  - 若非超级用户进程调用成功后，文件的设置用户 ID 和设置组 ID 都被清除
- truncate、ftruncate 函数用于改变文件的长度

  truncate 函数
  - 当文件以前的长度>length 时，则超过 length 以外的数据将不复存在

- 当文件以前的长度<length 时，在文件以前长度到 length 之间，将形成空洞，读该区域，将返回 0

✧ 硬链接与符号链接（硬链接是考试重点）

- link、unlink、remove、rename 函数

  link 函数用于创建一个指向现存文件的连接
  - 创建一个新目录项 newpath，它引用现存文件 existingpath。
  - 若 newpath 已经存在，则返回出错
  - 增加被引用文件的连接技术

  unlink 函数、remove 函数
  - remove 函数删除一个现存目录项，并将 pathname 所引用的文件的连接计数减 1。
  - 如果该文件还有其他连接，则仍可以通过其他连接存取该文件的数据。
  - 如果是最后一个连接，则内核还要删除文件的内容。但，如果有进程打开了该文件，其内容也不能删除。
  - 当进程关闭一个文件时，内核首先检查打开该文件的进程计数，如果为 0，然后内核检查其连接计数，如果也为 0，那么删除该文件内容。
  - 要解除对文件的连接，必须对包含该目录项的目录具有写和执行许可权。
  - 如果 pathname 是符号连接，那么 unlink 的是符号连接文件本身，而不是该连接所引用的文件。
  - unlink 是系统调用，而 remove 是库函数。remove 的参数为普通文件时等价于 unlink，为目录时等价于 rmdir。

  rename 函数用于更名文件或目录
  - 如果 oldname 是一个文件而不是目录，那么为该文件更名。如果 newname 已存在，而且是一个目录，则出错，如果不是目录，则先将该目录项删除，然后将 oldname 更名为 newname
  - 如果 oldname 是一个目录，那么为该目录更名。如果 newname 已存在，则它必须引用一个目录，而且该目录应当是空目录，此时，内核先将其删除，然后将 oldname 更名为 newname。另外，当为一个目录更名时，newname 不能包含 oldname 作为其路径前缀
  - 作为一个特例，如果 oldname 和 newname 引用同一文件，则函数不做任何更改而成功返回。
  - 应对包含两个文件的目录具有写和执行许可权。
- 符号连接
  - 符号连接是对一个文件的间接指针。与硬连接不同的是，硬连接文件直接指向文件的 i 节点，并增加文件的引用计数。但符号连接是一种特殊类型的文件，其文件内容是被连接文件的路径名
  - 创建符号连接
    - ln -s a.txt syma.txt
  - 对于符号连接的处理，有些系统调用直接处理符号连接文件本身，
  - 而有些系统调用则跟踪符号连接文件到其所指向的文件。
  - 如 chown、remove、unlink 等就直接处理符号连接文件，
  - 而大多数系统调用则跟随符号连接，如 chmod、open、stat 等

- symlink、readlink 函数

  symlink 函数用于创建符号连接。

  readlink 函数用于读符号文件本身的内容。

✧ 目录操作

- mkdir、rmdir 函数

  mkidir 函数用于创建目录； rmdir 函数用于删除目录

- 读目录
  opendir 函数
  - 用于打开目录
  - 函数原型：
    - DIR* opendir(const char* pathname);
  - 返回值和参数
    - 返回值：返回打开目录的索引结构，出错返回 NULL
    - pathname：要打开的目录名

  读目录的基本操作
  - 打开目录（opendir）
  - 逐一读出目录项（readdir、rewinddir）
  - 关闭目录（closedir）
- chdir、fchdir、getcwd 函数
  - chdir、fchdir 用于改变进程的当前工作目录，getcwd 函数用于返回当前工作目录的绝对路径。

第三讲 标准 IO

- 运行出错的原因
  - 动态库导出函数的变形
  - 查看动态库导出的函数
    - #nm libtest.so
  - f 函数实际上在动态库中的名字是：
    - _Z1fv
- 库的编写注意事项
  - 导出函数的名称
  - 函数调用约定
  - 结构体对齐
  - 谁分配谁释放
- 函数的导出名
  - 在动态库的实现文件中函数的名称，与动态库导出的函数名称可能不同
  - 使用 extern “C”使的导出的函数名称和实际名称一致（示例 3.3）
    - extern “C”：告诉编译器按 C 语言的方式设定函数的导出名
    - 不同的编译器、不同的语言，对函数名的修改都有可能不同
- 函数调用约定
  - C 语言调用约定
    - void __cdecl f(int a, int b); VC 环境
    - void f(int a, int b) __attribute__((cdecl)) g++环境
  - f 被表示成_f
  - 从右至左，将参数压入堆栈
  - 函数调用者负责压入参数和堆栈平衡
  - 标准调用约定
    - void __stdcall f(int a, int b); VC 环境
  - f 被表示成_f@8；8 表示参数的字节数
  - 从右至左，将参数压入堆栈
  - 函数内负责堆栈平衡
  - 快速调用约定
    - void __fastcall f(int a, int b); VC 环境

- 由寄存器传送参数，用 ecx 和 edx 传送参数列表中前两个双字或更小的参数，剩下的参数仍然从右至左压入堆栈
- 函数内负责堆栈平衡
- C++类成员函数的调用约定：thiscall
- this 指针存放于 ecx 寄存器中
- 参数从右至左压入堆栈

✧ 结构体大小：（考点（第 12 次课））

```
struct A
{
 int i;
};//sizeof(A)=4bit
```

✧ 标准 I/O 库

- 为什么要设计标准 I/O 库？
  - ➢ 直接使用 API 进行文件访问时，需要考虑许多细节问题
  - ➢ 例如：read、write 时，缓冲区的大小该如何确定，才能使效率最优
- 标准 I/O 库封装了诸多细节问题，包括缓冲区分配

✧ 打开流

- fopen 函数用于打开一个指定的文件，返回值是 FILE 结构指针。

type：指定文件的读、写方式

Type	说明
r或rb	为读而打开
w或wb	使文件长度为0，或为写而创建
a或ab	添加；为在文件尾写而打开，或为写而创建
r+或r+b或rb+	为读和写而打开
w+或w+b或wb+	使文件长度为0，或为读和写而打开
a+或a+b或ab+	为在文件尾读和写而打开或创建

限制	r	w	a	r+	w+	a+
文件必须存在	√			√		
删除文件以前内容		√			√	
流可以读	√			√	√	√
流可以写		√	√	√	√	√
流只在尾端处写			√			√

- freopen 函数：在一个特定的流上打开一个指定的文件，如若该流已经打开了，则先关闭该流。
  - freopen 函数主要用于重定向
  - 即将 fp 重定向到 pathname 指定的文件中
- fdopen 函数：取一个现存的文件描述符，并使一个标准 I/O 流与该描舒符相结合，返回文件指针。
  - fdopen 常用于由创建管道和网络通信通道函数返回的描述符。
  - 这些特殊类型的文件，不能用 fopen 打开
  - 因此必须先调用设备专用函数以获得一个文件描述符
  - 然后再用 fdopen 使一个标准 I/O 流与该描述符相关联
  - 对于 fdopen 函数，type 参数的意义稍由区别
  - 因为该描述符已被打开，所以 fdopen 为写而打开并不截短该文件
  - 标准 I/O 添写方式，也不能用于创建该文件（因为如若一个描述符引用一个文件，则该文件一定已经存在）
- fclose 函数：关闭一个打开了的流，成功返回 0，出错返回 EOF
  - 在该文件被关闭之前，刷新缓存中的输出数据。缓存中的输入数据被丢弃，如果标准 I/O 库已经为该流自动分配了一个缓存，则释放此缓存。
- setbuf、setvbuf 函数用于设置缓冲类型

参数和返回值

- fp：fopen 函数的返回值
- buf：用户提供的文件缓冲区，其长度为 BUFSIZ
  - ◆ 若 buf 为 NULL，则为无缓冲

◆ 若 buf 不为 NULL，则为全缓冲

函数	mode	buf	缓存及长度	缓存的类型
setbuf		nonnull	长度为BUFSIZ的用户缓存	全缓存
		NULL	（无缓存）	不带缓存
setvbuf	_IOFBF	nonnull	长度为size的用户缓存	全缓存
		NULL	合适长度的系统缓存	
	_IOLBF	nonnull	长度为size的用户缓存	行缓存
		NULL	合适长度的系统缓存	
	_IONBF	忽略	无缓存	不带缓存

- 流的概念

  刷新一个流：

  - 强制标准 I/O 库进行系统调用，以将数据传递到内核
  - 函数原型
    - int fflush(FILE *fp);
  - 参数和返回值
    - fp：文件指针；若 fp=NULL，则刷新所有输出流
    - 成功返回 0，出错返回 EOF

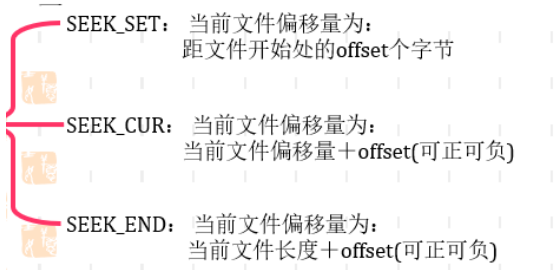
- 缓冲
  - 标准 I/O 库提供缓冲的目的：尽可能减少使用 read、write 调用的次数，以提高 I/O 效率。
  - 通过标准 I/O 库进行的读写操作，数据都会被放置在标准 I/O 库缓冲中中转。
  - **缓冲类型：**
    - **全缓冲**
      - 在填满标准 I/O 缓冲区后，才进行实际 I/O 操作（例如调用 write 函数）
      - 调用 fflush 函数也能强制进行实际 I/O 操作
    - **行缓冲**
      - 在输入和输出遇到换行符时，标准 I/O 库执行 I/O 操作
      - 因为标准 I/O 库用来收集每一行的缓存的长度是固定的，所以，只要填满了缓存，即使没有遇到新行符，也进行 I/O 操作
    - **行缓冲**
      - 终端（例如标准输入和标准输出），使用行缓冲
    - **不带缓冲**
      - 标准 I/O 库不对字符进行缓冲存储
      - 标准出错是不带缓冲的，为了让出错信息尽快显示出来

- 流的定向
  - 对于 ASCII 字符集，一个字符用一个字节表示
  - 对于国际字符集，一个字符可用多个字节表示
  - 流的定向决定了所读、写的字符是单字节还是多字节的
  - Fwide 函数用于改变流的定向
  - 函数原型：int fwide(FILE *fp, int mode);
  - 参数与返回值
    - mode<0，字节定向
    - mode>0，宽定向
    - mode=0，返回当前流的定向

✧ 定位流

- 类似于 lseek 函数，即指定从文件的什么地方开始进行读写
- 通常有两种方法定位标准 I/O 流
  - ftell、fseek 函数。
  - fgetpos、fsetpos 函数。

- 后者是 ANSI C 引入的。程序要移植到非 unix 类操作系统，应使用后者。
- ftell 函数用于获取当前文件偏移量，成功返回当前文件偏移量，出错返回-1
- fseek 函数用于设置当前文件偏移量，成功返回 0，出错返回非 0（offset 是相对偏移量：需结合 whence 才能计算出真正的偏移量）
  - 第三个参数Whence：该参数取值是三个常量之一



- rewind 函数用于将文件偏移量设置到文件的起始位置
- fgetpos 函数用于获取当前的文件偏移量，成功返回 0，出错返回非 0
- fsetpos 函数用于设置文件偏移量，成功返回 0，出错返回非 0

✧ 读写流

■ 对流有三种读写方式

- 每次读写一个字符

  输入函数 getc,fgetc,getchar 函数，成功返回预读字符，若已处于文件尾或出错返回 EOF

  输出函数 putc,fputc,putchar 函数，成功返回 c，出错返回 EOF
  - getchar()等同于 getc(stdin)
  - getc 通常是宏，fgetc 是函数
  - putchar(c)等同于 putc(c, stdout)
  - putc 通常是宏，fputc 是函数
  - 不管出错还是到达文件尾，都是返回 EOF。如何区分？
  - 调用 ferror 或 feof
    - int ferror(FILE *fp);
    - int feof(FILE *fp);
    - 当遇到文件结束符时，feof 返回真，ferror 返回假
    - 当出错时，feof 返回假，ferror 返回真
  - 在大多数实现中，为每个流在 FILE 对象中维持了两个标志
    - 出错标志
    - 文件结束标志
  - 调用 clearerr 清楚这两个标志
    - void clearerr(FILE *fp);
- 每次读写一行

  输入函数 fgets 函数，成功返回 buf，读到文件尾或出错返回 EOF。

  输出函数 fputs 函数，成功返回非负值，出错返回 EOF，null 符不写入文件。
  - fgets 函数一直读到下一个新行符为止，但是不超过 n-1 个字符
  - buf 缓存以 null 字符结尾
  - 若读到下一个新行符，会超过 n-1 个字符，则只会返回一个不完整的行，缓存总是以 null 字符结尾。
  - 下一次的 fgets 调用会继续读该行。
- 每次读写任意长度的内容（直接 IO，返回读/写的对象数）

➢ 每次 I/O 操作读写某种数量的对象，而每个对象具有指定的长度

✧ 格式化输出

■ 格式化输出函数
- int printf(const char* format, ...);
- int fprintf(FILE *fp, const char *format, ...);

- int sprintf(char *buf, const char *format, ...);
- printf 将格式化数据写到标准输出
- fprintf 写至指定的流
- sprintf 写入数组 buf 中
- 前两者返回输出字符数，后者返回写入数组的字符数
- sprintf 将格式化的字符送入数组 buf 中，并在该数组的尾端自动加入一个 null 字符，但该字节不包括在返回值中
- sprintf 函数可能会造成由 buf 指向的缓冲区的溢出，调用者要确保缓冲区足够大
  - int snprintf(char* buf, size_t n, const char* format, ……);
- 缓冲区长度为 n，超过缓冲区尾端的任何字符都会被丢弃
- 格式化输入函数
  - int scanf(const char* format, ...);
  - int fscanf(FILE *fp, const char *format, ...);
  - int sscanf(char *buf, const char *format, ...);

✧ 临时文件

- 创建临时文件可以用以前介绍的 creat，然后立即 unlink，也可以用以下方法：
- char *tmpnam (char *ptr);
- FILE *tmpfile (void); /* 返回文件指针 *、/
- char *tempnam (const char *directory, const char *prefix);
- 第一个函数产生一个与现在文件名不同的有效路径名字符串（每次调用均不同）。
- 第二个函数创建一个临时二进制文件（wb+），在关闭该文件或程序结束时，该文件自动删除。
- 第三个函数是第一个函数的变体，在产生路径名时，指定其目录和文件名前缀。

第四讲 进程

进程环境

✧ 进程的启动和终止

进程的终止

- 8 种方式使进程终止
- 正常终止
  - 从 main 返回
  - 调用 exit
  - 调用_exit 或_Exit
  - 最后一个线程从其启动例程返回
  - 最后一个线程调用 pthread_exit
- 异常终止
  - 调用 abort：产生 SIGABRT 信号
  - 接到一个信号并终止
  - 最后一个线程对取消请求做出响应
- 三个终止函数：exit、_Exit、_exit（考试重点）
- 函数原型：
  - void exit(int status);
  - void _Exit(int status);
  - void _exit(int status);
- exit 函数执行一个标准 I/O 库的清理关闭操作（为所有打开流调用 fclose 函数）后，进入内核

- _Exit、_exit 函数立即进入内核
- exit 等函数的参数
  - status：进程的终止状态
- 程序演示查看进程终止状态
  - echo $?

atexit 函数

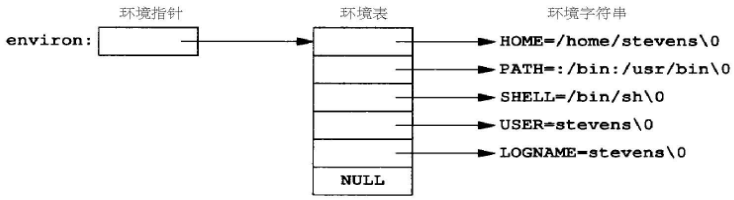
- 当进程终止时，程序可能需要进行一些自身的清理工作，如资源释放等等
- atexit 函数提供了进行这样工作的机会
- 它允许用户注册若干终止处理函数，当进程终止时，这些终止处理函数将会被自动调用
- 用于注册用户提供的终止处理函数
- 函数原型
  - int atexit(void (*func)(void));
- 参数
  - func：函数指针，返回值为 void，无参
- 返回值
  - 成功返回 0，出错返回非 0 值

注意：先注册的函数，后被运行。

调用_exit 函数并不会触发终止处理函数

✧ 环境表和环境变量

- 每个进程都会接收到一张环境表
- 通过 environ 找到环境表
  - extern char **environ;
- 环境字符串：name=value



- 访问环境变量的方法
  - 直接使用 environ
  - 使用 getenv 和 putenv 等函数
- getenv 函数用于获取环境变量值
- 函数原型
  - char* getenv(const char *name);
- 返回与 name 关联的 value 的指针，若未找到则返回 NULL
- 返回的指针是指向新分配的内存，还是环境表中存在的值？
- 设置环境变量
- 三种方法：
  - putenv
  - setenv
  - unsetenv
- putenv 函数将形式为 name=value 的字符串，放入环境表中；若 name 已经存在，则先删除其原来的定义。
- 函数原型：
  - int putenv(char *str);
- setenv 函数原型：

- int setenv(const char* name, const char* value, int rewrite);
- setenv 将环境变量 name 的值设置为 value。
- 若 name 已经存在
  - rewrite != 0，则删除其原先的定义
  - rewrite == 0，则不删除其原先的定义
- unsetenv 函数用于删除某个环境变量
- 函数原型
  - int unsetenv(const char* name);
- 删除 name 的定义
- 疑问？
  - 前两个设置环境变量的函数，都给出了自己的缓冲区存放环境变量。在环境表中是否直接使用这些缓冲区，还是环境表自己分配了缓冲区？

✧ setjmp 和 longjmp 函数（难点）

（原理：保存了寄存器的值）

- setjmp 和 longjmp 函数实现函数之间的跳转
- 函数原型
  - int setjmp(jmp_buf env);
  - void longjmp(jmp_buf env, int val);
- setjmp 函数用于设置跳转的目的位置
- longjmp 函数进行跳转
- 参数与返回值
  - env：保留了需要返回的位置的堆栈情况
  - setjmp 的返回值：直接调用该函数，则返回 0；若由 longjmp 的调用，导致 setjmp 被调用，则返回 val（longjmp 的第二个参数）

各类变量的情况：

- 当调用 longjmp 函数后，在 main 中的各类变量的值是否改变回原来的值呢？
- 全局变量、静态变量、易失变量不受优化的影响
- 在优化的版本，自动变量和寄存器变量存储在寄存器中

变量回滚问题：

- 保证局部变量在 longjmp 过程中一直保存它的值的方法：把它声明为 volatile 变量。（适合那些在 setjmp 执行和 longjmp 返回之间会改变的变量）
- 存放在内存中的变量，将具有调用 longjmp 时的值，而在 CPU 和浮点寄存器中的变量则恢复为调用 setjmp 函数时的值
- 优化编译时，register 和 auto 变量都存放在寄存器中，而 volatile 变量仍存放在内存
- volatile 变量：一般在多线程中使用的比较多
  - 例如有一个 int x，有两个线程都要对其读写
  - 有些编译器或 CPU 会将 x 保存在寄存器中，读的时候直接读取寄存器中的内容，而不是真实的 x 在内存中的内容
  - 线程 1，对 x 进行加 1 操作，此时内存中 x 的值为 2
  - 线程 2 想读 x，结果从寄存器中读出 1
  - 给变量加上 volatile，指示程序每次读写变量都必须从内存中读取，不要进行缓存（寄存器）

自动变量的潜在问题：

- 问题：
  - open_data 函数返回后，它在栈上所使用的空间将由下一个被调用函数所占用
  - 但是标准 I/O 库仍使用位于栈上的 databuf 缓冲区
  - 存在冲突和混乱
- 解决办法：

- 使用全局存储空间
- 使用静态存储空间
- 从堆中分配

- 资源必须是：
- RLIMIT_CPU.CPU 时间限制（以秒为单位）。 当进程达到软限制时，将发送 SIGXCPU 信号。
- RLIMIT_DATA .进程数据段的最大大小（初始化数据、未初始化数据和堆）。
- RLIMIT_FSIZE.进程可能创建的文件的最大大小。 尝试将文件扩展到超出此限制，会导致发送 SIGXFSZ 信号。
- RLIMIT_LOCKS.此过程可能建立的组合 flock（） 锁和 fcntl（） 租约数的限制。
- RLIMIT_MEMLOCK.使用 mlock（） 和 mlockall（） 锁定到 RAM 的虚拟内存的最大字节数。
- RLIMIT_NOFILE.指定大于此过程可打开的最大文件描述符编号的值 1。
- RLIMIT_NPROC.可为调用进程的实际用户 ID 创建的最大进程数。
- RLIMIT_STACK.进程堆栈的最大大小（以字节为单位）。 达到此限制后，将生成 SIGSEGV 信号。
- 三个规则控制资源限制的变化：
- 任何进程都可以将软限制更改为小于或等于其硬限制的值。
- 任何过程都可以将其硬限制降低为大于或等于其软限制的值。
- 只有超级用户进程才能提高硬性限制。

■ 进程控制

✧ 进程标识符

- 每个进程都有一个非负整型表示的唯一进程 ID
- 进程 ID 总是唯一的
- 当进程终止后，其 ID 值可以重用
- 查看进程情况
  - $ps -ef | less

✧ fork 等函数（考点）

- 一个进程可以调用 fork 函数创建一个新进程
- 新进程被称为子进程
- 函数原型
  - pid_t fork(void);
- 返回值
  - fork 函数调用一次，但是返回两次
  - 在子进程中返回 0，在父进程中返回子进程 ID，出错返回-1
  - 通过返回值，可以确定是在父进程还是子进程中
- 子进程和父进程继续执行 fork 调用之后的指令
- 子进程是父进程的副本
  - 子进程获得父进程数据空间、堆和栈的副本
  - 父子进程并不共享这些存储空间
  - 父子进程共享正文段（只读的）
- 为了提高效率，fork 后不并立即复制父进程空间，采用了 COW（Copy-On-Write）
  - 当父子进程任意之一，要修改数据段、堆、栈时，进行复制操作，但仅复制修改区域
- 为什么 write 调用的输出只有一次，而 printf 调用的输出出现了两次？
  - write 调用是不带用户空间缓冲的。在 fork 之前调用 write，其数据直接写到了标准输出上
  - 标准 I/O 库是带缓冲的，当标准输出连接到终端设备时，它是行缓冲，否则为全缓冲。

- 当 printf 输出到终端设备时，由于遇到换行符，因此缓冲被刷。子进程的数据空间中无缓冲内容
- 当重定向到文件时，变为全缓冲。fork 后，子进程的数据空间中也有内容。所以输出两次

- 子进程中，变量的值改变了；而父进程中，变量的值没有改变。原因？
- 在使用 fork 函数时，一定要牢记子进程复制了父进程的地址空间
  - 父进程在 fork 之前 new 了一个对象，子进程需要 delete 它吗？
  - 父进程在 fork 之前 open 的文件，子进程需要 close 文件描述符吗？
- fork 的一个特性：父进程的所有打开文件描述符，都被复制到子进程中。

父子进程文件共享：

- 父子进程对同一文件使用了一个文件偏移量
- 上例中，父进程等待了子进程两秒钟，所以他们的输出才没有混乱；否则有可能出现乱序
- 文件描述符的常见处理方式
  - 父进程等待子进程完成。父进程无需对描述符做任何处理，当子进程终止后，文件偏移量已经得到了相应的更新
  - 父子进程各自执行不同的程序段，各自关闭文件描述符

fork 函数常见用法：

- 一个父进程希望复制自己，使父子进程同时执行不同的代码段
  - 网络服务程序中，父进程等待客户端的服务请求，当请求达到时，父进程调用 fork，使子进程处理该次请求，而父进程继续等待下一个服务请求到达
- 一个进程要执行一个不同的程序
  - 子进程从 fork 返回后，立即调用 exec 执行另外一个程序

vfork 函数

- vfork 与 fork 的函数原型相同，但两者的语义不同
- vfork 用于创建新进程，而该新进程的目的是 exec 一个新程序（执行一个可执行的文件）
- 由于新程序将有自己的地址空间，因此 vfork 函数并不将父进程的地址空间完全复制到子进程中。
- 子进程在调用 exec 或 exit 之前，在父进程的地址空间中运行
- vfork 函数保证子进程先执行，在它调用 exec 或者 exit 之后，父进程才可能被调度执行

✧ exit 函数

- 不管进程如何终止，最后都会执行内核中的同一段代码：为相应进程关闭所有打开描述符，释放内存等等
- 若父进程在子进程之前终止了，则子进程的父进程将变为 init 进程，其 PID 为 1；保证每个进程都有父进程
- 当子进程先终止，父进程如何知道子进程的终止状态（exit(5)）
  - 内核为每个终止子进程保存了终止状态等信息
  - 父进程调用 wait 等函数，可获取该信息
- 当父进程调用 wait 等函数后，内核将释放终止进程所使用的所有内存，关闭其打开的所有文件
- 对于已经终止、但是其父进程尚未对其调用 wait 等函数的进程，被称为僵尸进程
- 程序演示(4.14 后台启动) ps
  - Defunct：死了的
- 对于父进程先终止，而被 init 领养的进程会是僵尸进程吗？
  - init 对每个终止的子进程，都会调用 wait 函数，获取其终止状态

✧ wait 等函数

wait 函数：

- 当一个进程正常获知异常终止时，内核就向其父进程发送 SIGCHLD 信号
- 父进程可以选择忽略该信号，也可以提供信号处理函数
- 系统的默认处理方式：忽略该信号
- wait 函数可用于获取子进程的终止状态
- 函数原型
  - pid_t wait(int *statloc);
- 参数与返回值
  - statloc：可用于存放子进程的终止状态
  - 返回值：若成功返回终止进程 ID，出错返回-1
- 调用 wait 函数之后，进程可能出现的情况
  - 如果所有子进程都还在运行，则阻塞等待，直到有一个子进程终止，wait 函数才返回
  - 如果一个子进程已经终止，正等待父进程获取其终止状态，则 wait 函数会立即返回
  - 若进程没有任何子进程，则立即出错返回
- 注意：若接收到信号 SIGCHLD 后，调用 wait，通常 wait 会立即返回
- 参数 statloc
  - statloc 可以为 NULL，表明父进程不需要子进程的终止状态。为了防止子进程成为僵尸或者需等待子进程结束
  - 若 statloc 不是空指针，则进程终止状态就存放在它指向的存储单元中
- statloc 指向的存储单元，存放了诸多信息，可以通过系统提供的宏获取

获取终止状态的宏：

宏	说明
WIFEXITED(status)	若为正常终止子进程返回的状态，则为真。对于这种情况可以执行WEXITSTATUS(status)，取子进程传递给exit、_exit、_Exit参数的低8位
WIFSIGNALED(status)	若为异常终止子进程返回的状态，则为真。对于这种情况可执行WTERMSTG(status)，取使子进程终止的信号编号
WIFSTOPPED(status)	若为当前暂停子进程的返回的状态，则为真。对于这种情况，可执行WSTOPSIG(status)，取使子进程暂停的信号编号
WIFCONTINUED(status)	若在作业控制暂停后已经继续的子进程返回了状态，则为真

waitpid 函数：

- 如果一个进程有几个子进程，那么只要有一个子进程终止，wait 就返回
- 如何才能等待一个指定的进程终止？
  - 调用 wait，然后将其返回的进程 ID 和所期望的进程 ID 进行比较
  - 如果 ID 不一致，则保存该 ID，并循环调用 wait 函数，直到等到所期望的进程 ID 为止
  - 下一次又想等待某一特定进程时，先查看已终止的进程列表，若其中已有要等待的进程，则无需再调用 wait 函数
- waitpid 函数可用于等待某个特定的进程
- 函数原型
  - pid_t waitpid(pid_t pid, int *statloc, int options);
- 参数和返回值
  - 成功返回进程 ID，失败返回-1
  - statloc：存放子进程终止状态
- 参数 pid
  - pid==-1：等待任一子进程，同 wait
  - pid>0：等待进程 ID 为 pid 的子进程
  - pid==0：等待其组 ID 等于调用进程组 ID 的任一子进程

- pid<-1：等待其组 ID 等于 pid 绝对值的任一子进程
- 参数 options：可以为 0，也可以是以下常量或运算的结果
  - WCONTINUED
  - WUNTRACED
  - WNOHANG：若 pid 指定的子进程并不是立即可用的，则 waitpid 不阻塞，此时其返回 0
- waitpid 函数提供了 wait 函数没有的三个功能：
  - waitpid 可等待一个特定的进程，而 wait 则返回任一终止子进程的状态
  - waitpid 提供了一个 wait 的非阻塞版本。有时用户希望取得一个子进程的状态，但不想阻塞
  - waitpid 支持作业控制

✧ exec 等函数

- 进程调用 exec 等函数用于执行另一个可执行文件
- 当进程调用一种 exec 函数时，该进程执行的程序完全替换为新程序
- 而新程序则从其 main 函数开始执行
- exec 并不创建新进程，所以前后的进程 ID 并未改变，exec 只是用一个全新的程序替换了当前进程的正文、数据、堆和栈段

exec 类函数

- execl　　execv　　execle
- execve　　execlp　　execvp
- 六个函数开头均为 exec，所以称为 exec 类函数
- l：表示 list，即每个命令行参数都说明为一个单独的参数
- v：表示 vector，命令行参数放在数组中
- e：调用者提供环境表
- p：表示通过环境变量 PATH，查找执行文件
- 通常，只有 execve 是内核的系统调用，其他 5 个都是库函数

✧ 更改用户 ID 和组 ID（考试的重点和难点）

- 系统的权限检查是基于用户 ID 或组 ID
- 当程序需要增加特权，或需要访问当前并不允许访问的资源时，需要更换自己的用户 ID 或组 ID
- 可以用 setuid 设置实际用户 ID 和有效用户 ID；setgid 设置实际组 ID 和有效组 ID
  - int setuid(uid_t uid);
  - int setgid(gid_t gid);
- 改变用户/组 ID 的规则
  - 若进程具有超级用户权限，则 setuid 将实际用户 ID、有效用户 ID、保存的设置用户 ID 设置为 uid
  - 若进程没有超级用户权限，但 uid 等于实际用户 ID 或保存的设置用户 ID，则 setuid 只将有效用户 ID 设置为 uid，不改变实际用户 ID 和保存的设置用户 ID
  - 若以上条件不满足，返回-1，errno 设为 EPERM
- 只有超级用户进程可以更改实际用户 ID
  - 实际用户 ID 是在用户登录时，由 login 程序设置的
  - login 是一个超级用户进程，当它调用 setuid 时，会设置所有三个用户 ID
- 仅当对程序文件设置了设置用户 ID 位时，exec 才会设置有效用户 ID。任何时候都可以调用 setuid，将有效用户 ID 设置为实际用户 ID 或保存的设置用户 ID
- 保存的设置用户 ID 是由 exec 复制有效用户 ID 而得来的
- **例子：man 联机手册：（理解）**
  - 当 man 需要对其手册页进行访问时，又需要将其有效用户 ID 改为 man
  - man 调用 setuid(man)

- 实际用户 ID＝我们的用户 ID
- 有效用户 ID＝man
- 保存的设置用户 ID＝man

➢ 由于 setuid 的参数等于保存的设置用户 ID，所以 setuid 可以成功修改有效用户 ID
➢ 这就是保存的设置用户 ID 的作用

✧ system 函数

- 用于执行一个 shell 命令
- 函数原型
  - int system(const char* cmdstring);
- cmdstring：shell 命令
- system 是通过 fork、exec、waitpid 等实现的，因此有三种返回值
  - 即 fork 失败，exec 失败，waitpid 失败

✧ 进程会计

- 进程会计记录
  - 包含命令名，CPU 时间总量，用户 ID 和组 ID，启动时间等等

■ 进程关系

✧ 进程组

- 每个进程除了有一个进程 ID 外，还属于一个进程组
- 进程组是一个或多个进程的集合。通常，它们与同一作业关联，可以接收来自同一终端的各种信号。
- 每个进程组有一个唯一的进程组 ID
- 每个进程组都可以有一个组长进程；组长进程的标识是：其进程组 ID 等于组长进程 ID
- 只要进程组中还有一个进程存在，则进程组就存在，与组长进程存在与否无关
- 从进程组创建开始，到其中最后一个进程离开为止的时间区间，称为进程组的生存期
- 进程组中的最后一个进程可以终止，或者转移到另一个进程组

✧ 会话

- 会话是一个或多个进程组的集合
- 一次登录形成一个会话
- Shell 上的一条命令形成一个进程组
- proc1 | proc2 &（开两个终端，观察组长 会话 ID）
  - ➢ proc3 | proc4 | proc5 (命令 ps -xj;程
- pid_t setsid();
- 非组长进程调用此函数，会创建一个新会话，导致三件事：
  - ➢ 该进程变成新会话首进程，即会话首进程是创建该会话的进程
  - ➢ 该进程成为一个新进程组的组长进程
  - ➢ 该进程没有控制终端
- 组长进程调用此函数，返回出错-1
- pid_t getsid(pid_t pid);
- 返回会话首进程的进程组 ID，即会话 ID

setpgid 函数

- 用于加入一个现有的进程组或者创建一个新进程组
- 函数原型
  - int setpgid(pid_t pid, pid_t pgid);

- 该函数将进程 ID 为 pid 的进程的进程组 ID，设置为 pgid
- 若 pid=pgid，则 pid 代表的进程将变为进程组组长
- 若 pid=0，则使用调用者的进程 ID
- 若 pgid=0，则由 pid 指定的进程 ID 将用作进程组 ID
- 注意：一个进程只能为它自己或它的子进程设置进程组 ID。在子进程调用 exec 函数之后，父进程就不能再改变该子进程的进程组 ID

setpgrp 函数

- 用于获取调用进程的进程组 ID
- 函数原型
  - pid_t getpgrp();
- 返回值
  - 返回调用进程的进程组 ID

✧ 控制终端

- 会话和组还有其他一些特征：
- 会话**可以有**一个***控制终端***。
- 建立与控制终端连接的会话领导称为控制***过程***。
- 会话中的进程组可以分为单个前台**进程和一个或多个后台组**。
- 如果会话具有控制终端，则它具有**单个**前台进程组，会话中的所有其他组都是后台组。
- 每当我们键入终端的中断键（通常*删除*或 *Ctrl+C*）或退出密钥（通常为 Ctrl-+）时，都会导致中断信号或退出信号**发送到前台进程**组的所有进程。
- 如果终端接口检测到调制解调器断开，则挂断信号将发送到**控制进程**（会话领导）。
- 需要注意的是：会话中的前台或后台进程组是针对该会话是否拥有控制终端而言。也即，如果一个会话没有控制终端，则会话中所有进程组均属“后台”。