Linux系统编程-文件I/O

杨劲松 yjs@oldhand.org

2011.07.09

课程目标

- 掌握文件I/O相关的操作函数
  - open()/creat()/close()/read()/write()/lseek()…
- 掌握标准I/O相关的函数
  - fopen()/fclose()/fread()/fwrite()/fseek()…
- 了解UNIX标准
- 了解文件I/O和标准I/O的差异和用途

参考资料

- 亚嵌教育《Linux 系统编程》
  - 第1章 文件与I/O
- W.Richard Stevens《UNIX环境高级编程》
  - 第1章 UNIX基础知识
  - 第2章 UNIX标准化及实现
  - 第3章 文件I/O
  - 第5章 标准I/O库
- 《UNIX编程艺术》

学习方法(建议)

- 结合Linux内核的工作原理来理解
  - 系统函数正是内核提供给应用程序的接口
- 熟练掌握C语言
  - Linux内核是用C语言写的，我们在描述内核工作原理时必然要用“指针”、“结构体”、链表这些名词来组织语言，就像只有掌握了英语才能看懂英文书一样，只有学好了C 语言才能看懂我描述的内核工作原理
- 学会使用Man pages
- 参考W. Richard Stevens《UNIX环境高级编程》

内容提纲

- UNIX基础知识
- UNIX标准化及实现
- 文件I/O
- 标准I/O库
- I/O模型比较

1. UNIX基础知识

1. UNIX基础知识

- 文件和目录
- 输入和输出
- 程序和进程
- 出错处理
- 用户标识
- 信号
- UNIX时间值
- 系统调用和库函数

1.1 UNIX基础知识

- 文件系统
  - 树状结构
- 文件名
  - 命名规范
  - 长度限制
- 路径名
- 工作目录
  - 通常为程序启动时的目录
  - 可以使用chdir()更改
- 起始目录
  - /etc/passwd文件中配置

1.2 UNIX基础知识-输入和输出

- 文件描述符
  - 顺序分配的非负整数
  - 内核用以标识一个特定进程正在访问的文件
  - 其他资源(socket、pipe等)的访问标识
- 标准输入、标准输出和标准出错
  - 由shell默认打开，分别为0/1/2
- 不用缓存的I/O
  - 通过文件描述符进行访问
  - open()/read()/write()/lseek()/close()…
- 标准I/O
  - 通过FILE*进行访问
  - printf()/fprintf()/fopen()/fread()/fwrite()/fseek()/fclose()…

1.3 UNIX基础知识-程序和进程

- 程序(静态)
  - 存放在磁盘文件中的可执行文件
  - 通过exec函数族调用运行
- 进程(动态)
  - 程序的执行实例
  - 进程的标识(pid,ppid,…)
- 进程控制
  - fork()
  - exec函数族
  - wait()/waitpid()

1.4 UNIX基础知识-出错处理

- 全局错误码errno
  - 在errno.h中定义，全局可见
  - 错误值定义为“EXXX”形式，如EACCESS
- 处理规则
  - 如果没有出错，则errno值不会被一个例程清除，即只有出错时，才需要检查errno值
  - 任何函数都不会将errno值设置为0，errno.h中定义了所有常数都不为0
- 错误信息输出
  - strerror() - 映射errno对应的错误信息
  - perror() – 输出用户信息及errno对应的错误信息

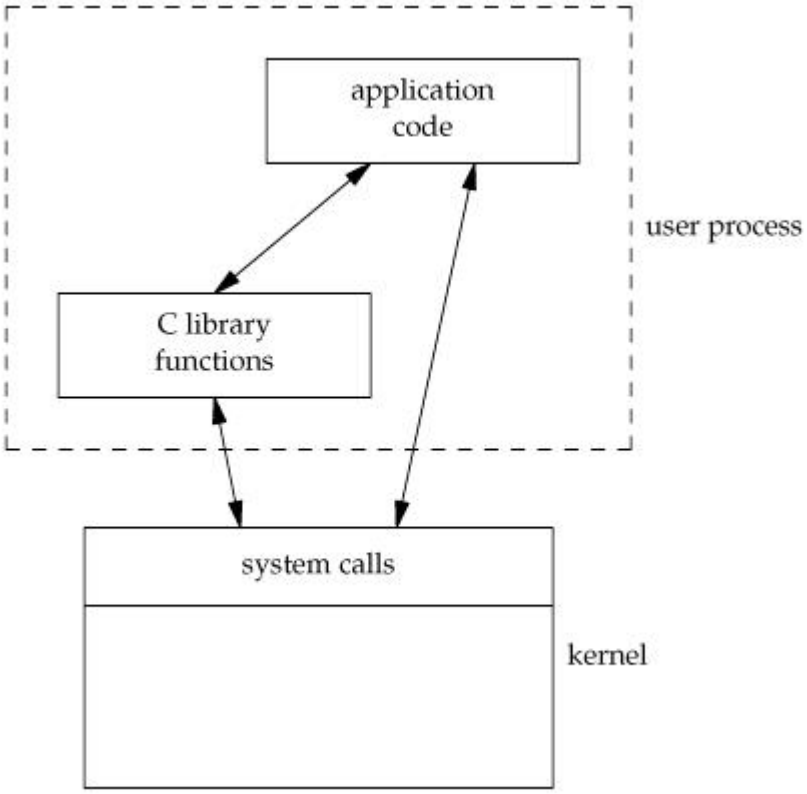
1.5 UNIX基础知识-用户标识

- 用户ID(uid)
  - 标识不同的用户，用户登录时通过/etc/passwd文件配置
  - 通过getuid()可以获取uid
  - 每个文件/目录都有相应的owner权限(-rwx------)
- 组ID(gid)及添加组ID
  - 通过组将多个有用户集中起来进行管理
  - 用户登录时通过/etc/passwd文件配置
  - 组ID与组名通过/etc/group文件配置
  - 每个文件/目录都有相应的group权限(----rwx---)
- 实际用户ID和有效用户ID
- 实际组ID和有效组ID

1.6 UNIX基础知识-系统调用和库函数

- 系统调用
  - 用户空间进程访问内核的接口
  - 把用户从底层的硬件编程中解放出来
  - 极大的提高了系统的安全性
  - 使用户程序具有可移植性
- 库函数
  - 库函数为了实现某个功能而封装起来的API集合
  - 提供统一的编程接口，更加便于应用程序的移植
- 系统调用与库函数的比较
  - 所有的操作系统都提供多种服务的入口点，通过这些入口点，程序向内核请求服务
  - 从执行者的角度来看，系统调用和库函数之间有重大的区别，但从用户的角度来看，其区别并不非常的重要。
  - 应用程序可以调用系统调用或者库函数，库函数则会调用系统调用来完成其功能。
  - 系统调用通常提供一个访问系统的最小界面，而库函数通常提供比较复杂的功能。

1.6.1 C库函数与系统调用之间的差别



1.6.2.1 文件I/O - 汇编程序的Hello world


```
 1 .data   # section declaration
 2
 3 msg:
 4         .ascii "Hello, world!\n"        # our dear string
 5         len = . - msg                   # length of our dear string
 6
 7 .text   # section declaration
 8
 9                         # we must export the entry point to the ELF linker or
10         .global _start  # loader. They conventionally recognize _start as their
11                         # entry point. Use ld -e foo to override the default.
12
13 _start:
14         # write our string to stdout
15         movl $len,%edx  # third argument: message length
16         movl $msg,%ecx  # second argument: pointer to message to write
17         movl $1,%ebx    # first argument: file handle (stdout)
18         movl $4,%eax    # system call number (sys_write)
19         int $0x80       # call kernel
20
21 # and exit
22         movl $0,%ebx    # first argument: exit code
23         movl $1,%eax    # system call number (sys_exit)
24         int $0x80       # call kernel
```

汇编与链接:

```
as -o hello.o hello.s
ld -o hello hello.o
```

1.6.2.2 文件I/O - C语言的Hello world


```c
#include <unistd.h>

char msg[14] = "Hello, world!\n";
#define len 14

int main(void)
{
  write(1, msg, len);
  _exit(0);
}
```

反汇编(objdump -d)之后的代码:

```
08048384 <main>:
 8048384:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048388:       83 e4 f0                and    $0xfffffff0,%esp
 804838b:       ff 71 fc                pushl  0xfffffffc(%ecx)
 804838e:       55                      push   %ebp
 804838f:       89 e5                   mov    %esp,%ebp
 8048391:       51                      push   %ecx
 8048392:       83 ec 14                sub    $0x14,%esp
 8048395:       c7 44 24 08 0e 00 00    movl   $0xe,0x8(%esp)
 804839c:       00
 804839d:       c7 44 24 04 98 95 04    movl   $0x8049598,0x4(%esp)
 80483a4:       08
 80483a5:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 80483ac:       e8 f7 fe ff ff          call   80482a8 <write@plt>
 80483b1:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 80483b8:       e8 0b ff ff ff          call   80482c8 <_exit@plt>
 80483bd:       90                      nop
 80483be:       90                      nop
 80483bf:       90                      nop
```

1.6.2.3 文件I/O – 汇编与C代码比较

- 汇编代码中
  - .data段有一个标号msg，代表字符串“Hello, world!\n”的首地址，相当于C程序的一个全局变量。
  - 汇编程序中的len代表一个常量，它的值由当前地址减去符号msg所代表的地址得到，也就是字符串“Hello, world!\n”的长度。
  - 在_start中调了两个系统调用，第一个是write系统调用，第二个是_exit系统调用，在调write系统调用时，eax 寄存器保存着write 的系统调用号4，ebx、ecx、edx 寄存器分别保存着write 系统调用需要的三个参数：
    - ebx 保存着文件描述符，进程中每个打开的文件都有一个编号称为文件描述符，文件描述符1 表示标准输出，对应于C 标准I/O 库的stdout，
    - ecx 保存着输出缓冲区的首地址，
    - edx 保存着输出的字节数，write 系统调用把从msg 开始的len 个字节写到标准输出。
- C 代码中的write() 函数是系统调用的包装函数，其内部实现就是把传进来的三个参数分别赋给ebx、ecx、edx 寄存器，然后执行movl $4,%eax 和int $0x80两条指令

2. UNIX标准化及实现

（本节作为学员了解内容，学员可根据自身情况酌情学习）

2. UNIX标准化及实现

- UNIX标准化
- UNIX实现
- 限制
- 功能测试宏
- 基本系统数据类型
- 标准之间的冲突

2.1 UNIX标准化及实现-UNIX标准化

- ANSI C
  - 这一标准是ANSI(美国国家标准局)于1989年制定的C语言标准[ANSI 1989]。 后来被ISO(国际标准化组织)接受为标准，因此也称为 ISO C[ISO/IEC 9899:1990]。
  - ANSI C的目标是为各种操作系统上的C程序提供可移植性保证，而不仅仅限于UNIX。 该标准不仅定义了C编程语言的语发和语义，而且还定义了一个标准库。这个库可以根据 头文件划分为 15 个部分，其中包括：
    - 字符类型 (<ctype.h>)
    - 错误码 (<errno.h>)
    - 浮点常数 (<float.h>)
    - 数学常数 (<math.h>)
    - 标准定义 (<stddef.h>)
    - 标准 I/O (<stdio.h>)
    - 工具函数 (<stdlib.h>)
    - 字符串操作 (<string.h>)
    - 时间和日期 (<time.h>)
    - 可变参数表 (<stdarg.h>)
    - 信号 (<signal.h>)
    - 非局部跳转 (<setjmp.h>)
    - 本地信息 (<local.h>)
    - 程序断言 (<assert.h>)
    - …

2.1 UNIX标准化及实现-UNIX标准化

- ISO C
  - 1989年下半年，C程序设计语言的ANSI标准X3.159-1989得到批准，此标准已被采纳为国际标准ISO/IEC9899:1990
    - ANSI-美国国家标准学会(American National Standards Institute)
    - ISO-国际标准话组织(International Organization for) Standardization
    - IEC-国际电子技术委员会(International Electrotechnical Commission)
  - ISO C标准现在由ISO/IEC的C程序设计语言国际标准化工作组维护和开发，该工作组被称为ISO/IEC JTC1/SC22/WG14，简称WG14
  - ISO C标准的意图是提供C程序的可移植性，使其能适合于大量不同的操作系统，而不只是UNIX系统
  - ISO C标准不仅定义了C程序设计语言的语法和语义，还定义了其他标准库
  - 在1999年，ISO C标准被更新为ISO/IEC 9899:1999
    - 新标准显著改善了对进行数值处理的应用程序的支持
    - 增加了关键字restrict，告诉编译器，哪些指针引用是可以优化的，其方法是指明指针指向的对象，在函数中只通过该指针进行访问
  - GCC对ISO C99的支持情况：http://www.gnu.org/software/gcc/c99status.html

ISO C标准定义的头文件

头 文 件	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说 明
<assert.h>	•	•	•	•	验证程序断言
<complex.h>	•	•	•		支持复数算术运算
<ctype.h>	•	•	•	•	字符类型
<errno.h>	•	•	•	•	出错码（1.7节）
<fenv.h>		•	•		浮点环境
<float.h>	•	•	•	•	浮点常量
<inttypes.h>	•	•	•	•	整型格式转换
<iso646.h>	•	•	•	•	替代关系操作符宏
<limits.h>	•	•	•	•	实现常量（2.5节）
<locale.h>	•	•	•	•	局部类别
<math.h>	•	•	•	•	数学常量
<setjmp.h>	•	•	•	•	非局部goto（7.10节）
<signal.h>	•	•	•	•	信号(第10章)
<stdarg.h>	•	•	•	•	可变参数表
<stdbool.h>	•	•	•	•	布尔类型和值
<stddef.h>	•	•	•	•	标准定义
<stdint.h>	•	•	•		整型
<stdio.h>	•	•	•	•	标准I/O库(第5章)
<stdlib.h>	•	•	•	•	实用程序函数
<string.h>	•	•	•	•	字符串操作
<tgmath.h>		•			通用类型数学宏
<time.h>	•	•	•	•	时间和日期(6.10节)
<wchar.h>	•	•	•	•	扩展的多字节和宽字符支持
<wctype.h>	•	•	•	•	宽字符分类和映射支持

2.1 UNIX标准化及实现-UNIX标准化

- IEEE POSIX
  - POSIX是Portable Operating System Interface of Unix的缩写。由IEEE（Institute of Electrical and Electronic Engineering）开发，由ANSI和ISO标准化。
  - POSIX的诞生和Unix的发展是密不可分的，Unix于70年代诞生于Bell lab，并于80年代向美各大高校分发V7版的源码以做研究。UC Berkeley在V7的基础上开发了BSD Unix。后来很多商业厂家意识到Unix的价值也纷纷以Bell Lab的System V或BSD为基础来开发自己的Unix，较著名的有Sun OS，AIX，VMS。由于各厂家对Unix的开发各自为政，造成了Unix的版本相当混乱，给软件的可移植性带来很大困难，对Unix的发展极为不利。为结束这种局面，IEEE开发了POSIX，POSIX在源代码级别上定义了一组最小的Unix(类Unix)操作系统接口。
  - POSIX 表示可移植操作系统接口（Portable Operating System Interface，缩写为 POSIX 是为了读音更像 UNIX）。电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）最初开发 POSIX 标准，是为了提高 UNIX 环境下应用程序的可移植性。然而，POSIX 并不局限于 UNIX。许多其它的操作系统，例如 DEC OpenVMS 和 Microsoft Windows NT，都支持 POSIX 标准，尤其是 IEEE Std. 1003.1-1990（1995 年修订）或 POSIX.1，POSIX.1 提供了源代码级别的 C 语言应用编程接口（API）给操作系统的服务程序，例如读写文件。POSIX.1 已经被国际标准化组织（International Standards Organization，ISO）所接受，被命名为 ISO/IEC 9945-1:1990 标准。
  - POSIX 现在已经发展成为一个非常庞大的标准族，某些部分正处在开发过程中。表 1-1 给出了 POSIX 标准的几个重要组成部分。POSIX 与 IEEE 1003 和 2003 家族的标准是可互换的。除 1003.1 之外，1003 和 2003 家族也包括在表中。

POSIX标准定义的必需的头文件

头 文 件	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	说 明
<dirent.h>	•	•	•	•	目录项（4.21节）
<fcntl.h>	•	•	•	•	文件控制（3.14节）
<fnmatch.h>	•	•	•	•	文件名匹配类型
<glob.h>	•	•	•	•	路径名模式匹配类型
<grp.h>	•	•	•	•	组文件（6.4节）
<netdb.h>	•	•	•	•	网络数据库操作
<pwd.h>	•	•	•	•	口令文件（6.2节）
<regex.h>	•	•	•	•	正则表达式
<tar.h>	•	•	•	•	tar归档值
<termios.h>	•	•	•	•	终端I/O（第18章）
<unistd.h>	•	•	•	•	符号常量
<utime.h>	•	•	•	•	文件时间（4.19节）
<wordexp.h>	•	•		•	字扩展类型
<arpa/inet.h>	•	•	•	•	Internet定义（第16章）
<net/if.h>	•	•	•	•	套接字本地接口（第16章）
<netinet/in.h>	•	•	•	•	Internet地址族（16.3节）
<netinet/tcp.h>	•	•	•	•	传输控制协议定义
<sys/mman.h>	•	•	•	•	内存管理声明
<sys/select.h>	•	•	•	•	select函数（14.5.1节）
<sys/socket.h>	•	•	•	•	套接字接口（第16章）
<sys/stat.h>	•	•	•	•	文件状态（第4章）
<sys/times.h>	•	•	•	•	进程时间（8.16节）
<sys/types.h>	•	•	•	•	基本系统数据类型（2.8节）
<sys/un.h>	•	•	•	•	UNIX域套接字定义（17.3节）
<sys/utsname.h>	•	•	•	•	系统名（6.9节）
<sys/wait.h>	•	•	•	•	进程控制（8.6节）

2.1 UNIX标准化及实现-UNIX标准化

- Single UNIX Specification
  - Single UNIX Specification(SUS)由Open Group发布
    - Open Group成立于1996年，由两个业界社团X/Open和Open Software Foundation(OSF)合并而成
    - Single UNIX Specification的第1版(SUSv 1)由X/Open在1994年出版，包含大约1170个接口，也被称为spec1170
    - Single UNIX Specification的第2版(SUS v2)由Open Group在1997年出版
    - Single UNIX Specification的第3版(SUS v3)由Open Group在2001年出版，SUS v3的基本规范与IEEE标准1003.1-2001相同
  - Single UNIX Specification是POSIX.1标准的一个超集，定义了一些附加的接口，这些接口扩展了基本的POSIX.1规范所提供的功能
  - 相应的系统接口全集被称为X/Open系统接口(XSI, X/Open System Interface)
    - __XOPEN_UNIX符号常量标识了XSI扩展的接口
    - XSI还定义了实现必须支持POSIX.1的哪些可选部分才能认为是遵循XSI的，包括文件同步、存储映射文件、存储保护及线程接口
    - 只有遵循XSI的实现才能称为UNIX系统

2.1 UNIX标准化及实现-UNIX标准化

- X/Open XPG3
  - X/Open组织制订
  - X/Open Portability Guide(X/Open可移植性指南)
- FIPS
  - Federal Information Processing Standard，美国政府用户计算机系统采购的标准
  - 基于XPG和POSIX，美国政府定义了一系列的标准，目前FIPS 151-2 描述了开放系统的需求。

2.2 UNIX标准化及实现-UNIX实现

- SVR(System V Release)
  - 60年代中期，AT&T贝尔实验室组织开发了一个叫Multics的操作系统。1969年贝尔实验室从Multics的计划中撤出，由于科学计算研究中心的成员没有了计算环境，Ken Thompson、Dennis Ritchie和其他一些研究人员开发了一个基本的文件系统，该系统后来演化成了UNIX文件系统。
  - AT&T专利部门将UNIX用来做文本处理使UNIX获得了成功，UNIX也因短小精巧而出名。后来为UNIX开发出C语言后，UNIX便使用C语言实现。这一实现也是UNIX变成开放系统的重要原因。
  - AT&T（后来的UNIX系统实验室，现为Novell所有）开发了UNIX的后续版本，如系统III以及系统V的一些版本。系统V的两个最新版本系统V版本3 (SVR3.2）和系统V版本4（SVR4）在计算机操作系统中一直很流行。
- BSD(Berkeley Software Distribution)
  - 1978年，UNIX研究小组将UNIX发布工作交给了UNIX支持组，该组在1978年就发布了一个叫程序员工作台的UNIX内部版本。1983年USG又发布了系统V，随着AT&T的解体，系统V又走上了市场。
  - 在AT&T发展UNIX的同时，许多大学也在研究UNIX。Berkeley的California大学计算机科学研究组开发了UNIX的BSD版本，该组首先在PDP-11上开发了1BSD和2BSD，然后又在DEC的VAX计算机上开发了3BSD，后来发展为4.0BSD、4.1BSD、4.2BSD和4.3BSD，其中（尤其是4.2和4.3）的许多特色（包括一些源代码）被应用到商业产品中。
- Linux
  - Linux并非UNIX系统，称为“类UNIX系统”更合适一些

2.2 UNIX标准化及实现-限制

- 限制类型
  - 编译时间限制
  - 运行时间限制
- ISO C限制
  - 所有由ISO C定义的限制都是编译时间限制
  - 通常在头文件limits.h中定义
- POSIX限制
  - POSIX.1定义了很多涉及操作系统实现限制的常数
- XPG3限制
- FIPS 151-1要求

2.2.1 sysconf()/pathconf()/fpathconf()

- sysconf()
  - Get configuration information at runtime

```
#include <unistd.h>

long sysconf(int name);
```

- pathconf()/fpathconf()
  - Get configuration values for files

```
#include <unistd.h>

long fpathconf(int filedes, int name);
long pathconf(char *path, int name);
```

2.2.1 sysconf()的参数

限 制 名	说 明	*name*参数
ARG_MAX	exec函数的参数最大长度（字节数）	_SC_ARG_MAX
ATEXIT_MAX	可用atexit函数登记的最大函数个数	_SC_ATEXIT_MAX
CHILD_MAX	每个实际用户ID的最大进程数	_SC_CHILD_MAX
clock ticks/second	每秒时钟滴答数	_SC_CLK_TCK
COLL_WEIGHTS_MAX	在本地定义文件中可以赋予LC_COLLATE顺序关键字项的最大权重数	_SC_COLL_WEIGHTS_MAX
HOST_NAME_MAX	gethostname函数返回的主机名最大长度	_SC_HOST_NAME_MAX
IOV_MAX	readv或writev函数可以使用的iovec结构的最大数	_SC_IOV_MAX
LINE_MAX	实用程序输入行的最大长度	_SC_LINE_MAX
LOGIN_NAME_MAX	登录名的最大长度	_SC_LOGIN_NAME_MAX
NGROUPS_MAX	每个进程同时添加的最大进程组ID数	_SC_NGROUPS_MAX
OPEN_MAX	每个进程的最大打开文件数	_SC_OPEN_MAX
PAGESIZE	系统存储页长度（字节数）	_SC_PAGESIZE
PAGE_SIZE	系统存储页长度（字节数）	_SC_PAGE_SIZE
RE_DUP_MAX	当使用间隔表示法\{m,n\}时，regexec和regcomp函数允许的基本正则表达式的重复出现次数	_SC_RE_DUP_MAX
STREAM_MAX	在任一时刻每个进程的最大标准I/O流数，如若定义，则其值一定与FOPEN_MAX相同	_SC_STREAM_MAX
SYMLOOP_MAX	在解析路径名期间，可遍历的符号链接数	_SC_SYMLOOP_MAX
TTY_NAME_MAX	终端设备名长度，包括终止字符null	_SC_TTY_NAME_MAX
TZNAME_MAX	时区名的最大字节数	_SC_TZNAME_MAX

2.2.2 pathconf()/fpathconf()的参数

限 制 名	说 明	*name*参数
FILESIZEBITS	以带符号整型值表示在指定目录中允许的普通文件最大长度所需的最少位数	_PC_FILESIZEBITS
LINK_MAX	文件链接数的最大值	_PC_LINK_MAX
MAX_CANON	终端规范输入队列的最大字节数	_PC_MAX_CANON
MAX_INPUT	终端输入队列可用空间的字节数	_PC_MAX_INPUT
NAME_MAX	文件名的最大字节数（不包括终止字符null）	_PC_NAME_MAX
PATH_MAX	相对路径名的最大字节数，包括终止字符null	_PC_PATH_MAX
PIPE_BUF	能原子地写到管道的最大字节数	_PC_PIPE_BUF
SYMLINK_MAX	符号链接中的字节数	_PC_SYMLINK_MAX

pathconf和fpathconf的限制及*name*参数

3. 文件I/O

3. 文件I/O

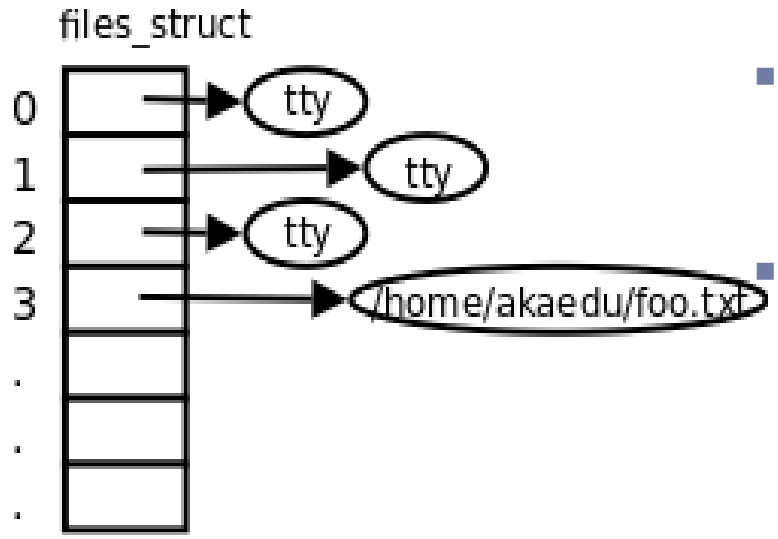
- 文件I/O介绍
- 文件描述符
- open()
- close()
- read()
- write()
- lseek()
- 作业

3.1 文件I/O - 介绍

- 文件I/O
  - 不带缓冲
    - 不带缓冲指的是每个read和write都调用内核中的相应系统调用
    - 不带缓冲的I/O函数不是ANSI C的组成部分，但是是POSIX和XPG3的组成部分
- 通过文件描述符来访问文件
- 文件I/O常用函数(系统调用)
  - open()/creat()
  - close()
  - read()
  - write()
  - lseek()

3.2 文件I/O – 文件描述符

- 每个进程在内核中都有一个task_struct 结构体来维护进程相关的信息，在Linux 内核中称为进程描述符(Process Descriptor)，而在操作系统理论中称为进程控制块(PCB，Process Control Block)。
- task_struct 中包含该进程当前打开的所有文件的信息，称为文件描述符表，在内核中用files_struct 结构体表示，其中的表项称为文件描述符(File Descriptor)，每个表项都包含一个指向已打开文件的指针，如下图所示。



- 在用户程序中文件描述符指的是文件描述符表的索引(即0、1、2、3 这些数字)，用int型变量来保存。
- 当调用open()打开一个现有文件或创建一个新文件时，内核分配一个新的文件描述符并返回给进程，当读写该文件时，文件描述符被作为参数传给read() 或write()。
- 使用C标准I/O函数时，调用fopen()打开文件，返回一个FILE *指针，当读写该文件时就传递这个FILE *指针。而fopen()/fputc()/fgetc()的底层实现就要调用open()/read()/write()。可见FILE 结构体中必然包含文件描述符，此外还包含缓冲区的相关信息，但FILE 指针是不透明的，我们不必关心这些信息在FILE 结构体中如何存储和表示。

3.3 文件I/O – open()/creat()

调用open()/creat()函数可以打开或者创建一个文件。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

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);
```

- open()和creat()调用成功返回文件描述符，失败返回-1，并设置errno。
- open()/creat()调用返回的文件描述符一定是最小的未用描述符数字。
- creat()等价于open(*pathname*, O_CREAT|O_WRONLY|O_TRUNC, *mode*)
- open()可以打开设备文件，但是不能创建设备文件，设备文件必须使用mknod()创建。

3.3 文件I/O – open()

<table>
<tr><td>原型</td><td colspan="3">int open(const char *pathname, int flags, mode_t mode);</td></tr>
<tr><td rowspan="9">参数</td><td>pathname</td><td colspan="2">被打开的文件名（可包括路径名）。</td></tr>
<tr><td rowspan="8">flags</td><td>O_RDONLY：只读方式打开文件。</td><td rowspan="3">这三个参数互斥</td></tr>
<tr><td>O_WRONLY：可写方式打开文件。</td></tr>
<tr><td>O_RDWR：读写方式打开文件。</td></tr>
<tr><td colspan="2">O_CREAT：如果该文件不存在，就创建一个新的文件，并用第三的参数为其设置权限。</td></tr>
<tr><td colspan="2">O_EXCL：如果使用O_CREAT时文件存在，则可返回错误消息。这一参数可测试文件是否存在。</td></tr>
<tr><td colspan="2">O_NOCTTY：使用本参数时，如文件为终端，那么终端不可以作为调用open()系统调用的那个进程的控制终端。</td></tr>
<tr><td colspan="2">O_TRUNC：如文件已经存在，并且以只读或只写成功打开，那么先全部删除文件中原有数据。</td></tr>
<tr><td colspan="2">O_APPEND：以添加方式打开文件，在打开文件的同时，文件指针指向文件的末尾。</td></tr>
<tr><td>mode</td><td colspan="2">被打开文件的存取权限，为8进制表示法。</td></tr>
</table>

3.3 文件I/O – open()-example1

Example1: 以所有者只写方式(参数O_WRONLY)打开文件(mode=644)，如果文件不存在则创建(参数O_CREAT)，如果文件存在则截短(参数O_TRUNC)：

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

本例中，mode可以直接赋值为0644。

3.3 文件I/O – open()-example2

Example2: 以所有者只写方式(参数O_WRONLY)打开锁文件 (mode=644)，如果文件不存在则创建(参数O_CREAT)，如果文件存在则 出错(参数O_CREAT|O_EXCL)：

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
```

- 本例中，mode可以直接赋值为0644。
- 本例可以作为处理某个临界资源的保护。

3.4 文件I/O – close()

调用close()函数可以关闭一个打开的文件。

```
#include <unistd.h>

int close(int fildes);
```

- 调用成功返回0，出错返回-1，并设置errno。
- 当一个进程终止时，该进程打开的所有文件都由内核自动关闭。
- 关闭一个文件的同时，也释放该进程加在该文件上的所有记录锁。

3.4 文件I/O – close()-example

example: 关闭一个已经打开的文件：

```
//mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
mode_t mode = 0644;
char *filename = TEST_FILENAME;

fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);

if (fd < 0)
{
  fprintf(stderr, "ERROR: Open file %s failed: %s\n", TEST_FILENAME,
                  strerror(errno));
  return -1;
}

if (close(fd) < 0)
{
  fprintf(stderr, "ERROR: close file %s failed: %s\n", TEST_FILENAME,
                  strerror(errno));
  return -1;
}
```

3.5 文件I/O – read()

调用read()函数可以从一个已打开的可读文件中读取数据。

```
#include <unistd.h>

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

- read()调用成功返回读取的字节数，如果返回0，表示到达文件末尾，如果返回-1，表示出错，通过errno设置错误码。
- 读操作从文件的当前位移量处开始，在成功返回之前，该位移量增加实际读取的字节数。
- buf参数需要有调用者来分配内存，并在使用后，由调用者释放分配的内存。

3.5 文件I/O – read()-example

example: 读取20字节的数据到缓冲区：

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
...
```

- 本例中，调用read()后，需要检查返回的结果bytes_read，根据bytes_read进行相应的处理。

3.6 文件I/O – write()

调用write()函数可以向一个已打开的可写文件中写入数据。

```
#include <unistd.h>

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

- write()调用成功返回已写的字节数，失败返回-1，并设置errno。
- write()的返回值通常与count不同，因此需要循环将全部待写的数据全部写入文件。
- write()出错的常见原因：磁盘已满或者超过了一个给定进程的文件长度限制。
- 对于普通文件，写操作从文件的当前位移量处开始，如果在打开文件时，指定了O_APPEND参数，则每次写操作前，将文件位移量设置在文件的当前结尾处，在一次成功的写操作后，该文件的位移量增加实际写的字节数。

3.6 文件I/O – write() – example

example: 将缓冲区的数据写入文件：

```
#include <sys/types.h>
#include <string.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_written;
int fd;
...
strcpy(buf, "This is a test\n");
nbytes = strlen(buf);

bytes_written = write(fd, buf, nbytes);
...
```

- 本例中，调用write()后，需要检查返回的结果bytes_written，根据bytes_written进行相应的处理，如果bytes_written小于nbytes，则需要循环将未写的数据写入文件。

3.7 文件I/O – lseek()

调用 lseek()函数可以显式的定位一个已打开的文件。

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

原型	off_t lseek(int fd,off_t offset,int whence);	
参数	fd：文件描述符。	
	offset：偏移量，每一读写操作所需要移动的距离，单位是字节的数量，可正可负（向前移，向后移）	
	whence (当前位置基点)：	SEEK_SET：当前位置为文件的开头，新位置为偏移量的大小。
		SEEK_CUR：当前位置为文件指针的位置，新位置为当前位置加上偏移量。
		SEEK_END：当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小。
返回值	成功：文件的当前位移	
	-1：出错	

3.7 文件I/O – lseek()

- 每个打开的文件都有一个与其相关的“当前文件位移量”，它是一个非负整数，用以度量从文件开始处计算的字节数。
- 通常，读/写操作都从当前文件位移量处开始，在读/写调用成功后，使位移量增加所读或者所写的字节数。
- lseek()调用成功为新的文件位移量，失败返回-1，并设置errno。
- lseek()只对常规文件有效，对socket、管道、FIFO等进行lseek()操作失败。
- lseek()仅将当前文件的位移量记录在内核中，它并不引起任何I/O操作。
- 文件位移量可以大于文件的当前长度，在这种情况下，对该文件的写操作会延长文件，并形成空洞。

3.7 文件I/O – lseek() - example

example: 创建一个有空洞的文件：

```
// FIXME: loop back to write rest data.
if (write(fd, buf1, strlen(buf1)) < strlen(buf1))
{
  fprintf(stderr, "buf1 write error.\n");
  return -1;
}

if (lseek(fd, 40, SEEK_SET) == -1)
{
  fprintf(stderr, "lseek() error.\n");
  return -1;
}

// FIXME: loop back to write rest data.
if (write(fd, buf2, strlen(buf2)) < strlen(buf2))
{
  fprintf(stderr, "buf2 write error.\n");
  return -1;
}
```

问题：创建有空洞的文件有何用途？

3.8 文件I/O – 作业

- 任务描述
  1. 使用非缓冲I/O方式，实现一个copy程序，该程序的第一个命令行参数为源文件，第二个命令行参数为目标文件，程序实现将源文件中的内容复制到目标文件。
  2. 命令行参数可以使用argv[1]访问第一个参数，argv[2]访问第二个参数
  3. 使用diff工具检查目标文件与源文件是否一致
- 实现思路
  1. 打开源文件
  2. 打开目标文件
  3. 循环读取源文件并写入目标文件
  4. 关闭源文件
  5. 关闭目标文件
- 存在哪些问题?

3.9 fcntl() – 操作文件描述符

```
#include <unistd.h>
#include <fcntl.h>

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

- 可以用*fcntl()*函数改变一个已打开的文件的属性，可以重新设置读、写、追加、非阻塞等标志（这些标志称为File Status Flag），而不必重新open 文件
- *fcntl()*函数和*open()*一样，也是用可变参数实现的，可变参数的类型和个数取决于前面的*cmd*参数
  - Duplicating a file descriptor
  - File descriptor flags
  - File status flags
  - Advisory locking
  - Mandatory locking(强制性锁)
  - Managing signals
  - Leases
  - File and directory change notification (dnotify)

3.10 ioctl()

- *ioctl()*用于向设备发控制和配置命令，有些命令也需要读写一些数据，但这些数据是不能用*read()*/*write()*读写的，称为Out-of-band数据
  - *read()*/*write()*读写的数据是in-band数据，是I/O操作的主体
    - 在串口线上收发数据通过*read()*/*write()*操作
    - A/D转换的结果通过*read()*读取
  - *ioctl()*命令传送的是控制信息，其中的数据是辅助的数据。
    - 而串口的波特率、校验位、停止位通过*ioctl()*设置
    - A/D转换的精度和工作频率通过*ioctl()*设置

```
#include <sys/ioctl.h>

int ioctl(int d, int request, ...);
```

- *d*是某个设备的文件描述符
- *request*是*ioctl()*的命令
- 可变参数取决于*request*，通常是一个指向变量或结构体的指针
- 若出错则返回-1，若成功则返回其他值，返回值也是取决于*request*

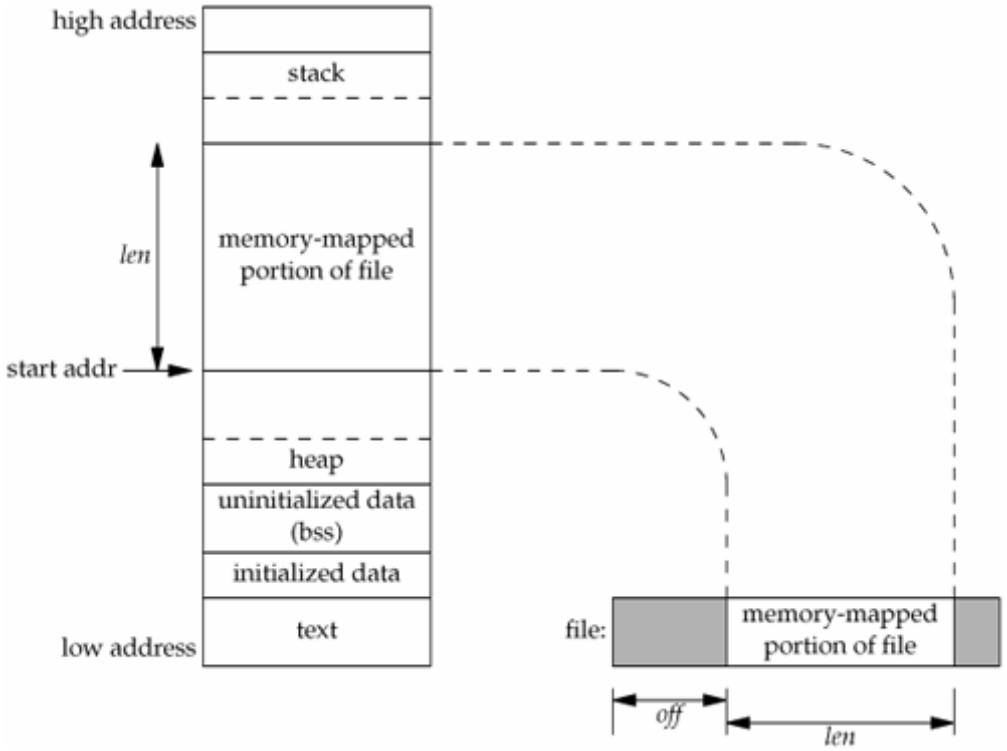
3.11 mmap()

```
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```

- *mmap()*可以把磁盘文件的一部分直接映射到内存，这样文件中的位置直接就有对应的内存地址，对文件的读写可以直接用指针来做而不需要*read()*/*write()*函数
- *start*参数
  - 如果*start*参数为NULL，内核会自己在进程地址空间中选择合适的地址建立映射。
  - 如果*start*不为NULL，则给内核一个提示，应该从什么地址开始映射，内核会选择*start*之上的某个合适的地址开始映射。
  - 建立映射后，真正的映射首地址通过返回值可以得到
- *length*参数是需要映射的那一部分文件的长度。
- *offset*参数是从文件的什么位置开始映射
  - 必须是页大小的整数倍（在32 位体系统结构上通常是4K）
- *fd*是代表该文件的描述符
- *prot*参数有四种取值：
  - PROT_EXEC 表示映射的这一段可执行，例如映射共享库
  - PROT_READ 表示映射的这一段可读
  - PROT_WRITE 表示映射的这一段可写
  - PROT_NONE 表示映射的这一段不可访问
- *flag*参数有很多种取值
  - MAP_SHARED 多个进程对同一个文件的映射是共享的，一个进程对映射的内存做了修改，另一个进程也会看到这种变化。
  - MAP_PRIVATE 多个进程对同一个文件的映射不是共享的，一个进程对映射的内存做了修改，另一个进程并不会看到这种变化，也不会真的写到文件中去。
  - 其它取值可查看*mmap(2)*
- 当进程终止时，该进程的映射内存会自动解除，也可以调用*munmap()*解除映射
- 返回值
  - 如果*mmap()*成功则返回映射首地址，如果出错则返回常数MAP_FAILED
  - *munmap()*成功返回0，出错返回-1。

3.11 mmap()



prot	说明
PROT_READ	数据可读
PROT_WRITE	数据可写
PROT_EXEC	数据可执行
PROT_NONE	数据不可访问

*prot*参数

flags	说明
MAP_SHARED	变动是共享的
MAP_PRIVATE	变动是私自的
MAP_FIXED	准确地解释 addr 参数

*flags*参数

4. 标准I/O

4. 标准I/O库

- 标准I/O库介绍
- 流和FILE对象
- stdin,stdout, stderr
- 缓存
- fopen()/freopen()/fdopen()
- fclose()
- 读写流
- 定位流
- 格式化I/O
- 临时文件(了解内容)
- 作业

4.1 标准I/O库-介绍

- 不仅在UNIX系统，在很多操作系统上都实现了标准I/O库
- 标准I/O库由ANSI C标准说明
- 标准I/O库处理很多细节，如缓存分配、以优化长度执行I/O等，这样使用户不必关心如何选择合适的块长度
- 标准I/O在系统调用函数基础上构造的，它便于用户使用

标准I/O库是由Dennis Ritchie在1975年左右编写的，它是由Mike Lesk编写的可移植I/O库的主要修改版本。

4.2 标准I/O库 – 流和FILE对象

- 标准I/O库的所有操作都是围绕流(stream)来进行的，在标准I/O中，流用FILE *来描述，通常称为文件指针
- FILE在头文件/usr/include/libio.h中定义

I/O模型	操作对象
文件I/O	文件描述符
标准I/O	FILE *

```
struct _IO_FILE {
  int _flags;           /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

4.3 标准I/O库 – stdin,stdout,stderr

- 标准I/O预定义3个流，他们可以自动地为进程所使用

标准输入	0	STDIN_FILENO	stdin
标准输出	1	STDOUT_FILENO	stdout
标准错误输出	2	STDERR_FILENO	stderr

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  printf("Hello world!\n");
  fprintf(stdout, "Hello world!\n");
  fprintf(stderr, "Hello world!\n");

  return 0;
}
```

4.4 标准I/O – 缓存(了解内容)

- 标准I/O提供了三种类型的缓存
  - 全缓存
    - 当填满I/O缓存后才进行实际I/O操作
  - 行缓存
    - 当在输入和输出中遇到新行符('\n')时，进行I/O操作
  - 不带缓存
    - 标准I/O库不对字符进行缓冲。例如stderr
- 使用stebuf()和setvbuf()可以更改缓存的类型
- 在任何时刻，可以使用fflush强制刷新一个流

说明: 除非有特殊需要，通常情况下标准I/O的缓冲方式不需要用户去修改。

4.5 标准I/O库 – 打开流

下列三个函数可用于打开一个标准I/O流：

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

- fopen()打开由*path*指定的一个文件
- freopen()在一个特定的流上(*fp*)打开一个指定的文件，若该流已经打开，则先关闭该流。一般用于将一个指定的文件打开为一个预定义的流(标准输入/标准输出/标准错误输出)
- fdopen()取一个现存的文件描述符，并使一个标准的I/O流与该描述符相结合。通常用于由管道等特殊的I/O类型所创建的文件描述符，这些特殊类型的文件不能直接用fopen()打开。

4.5 标准I/O - fopen() - mode参数

打开标准I/O流的mode参数：

r或rb	打开只读文件，该文件必须存在。
r+或r+b	打开可读写的文件，该文件必须存在。
w或wb	打开只写文件，若文件存在则文件长度清为0，即会擦些文件以前内容。若文件不存在则建立该文件。
w+或w+b或wb+	打开可读写文件，若文件存在则文件长度清为零，即会擦些文件以前内容。若文件不存在则建立该文件。
a或ab	以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。
a+或a+b或ab+	以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。

* 当给定“b”参数时，表示以二进制方式打开文件。

4.5 标准I/O - fopen() - mode参数

打开一个标准I/O流的六种不同方式：

打开一个标准I/O流的六种不同的方式

限　制	r	w	a	r+	w+	a+
文件必须已存在	•			•		
擦除文件以前的内容		•			•	
流可以读	•			•	•	•
流可以写		•	•	•	•	•
流只可在尾端处写			•			•

4.5 标准I/O - fopen() - example

example: 以读写方式打开文件test_file_3，如果该文件不存在，则创建。如果该文件已经存在，则长度截短为0。

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(int argc, char **argv)
{
  FILE *fp;

  if ((fp = fopen("test_file_3", "w+")) == NULL)
  {
    fprintf(stderr, "fopen() failed: %s\n", strerror(errno));
    return -1;
  }

  fclose(fp);

  return 0;
}
```

4.5 标准I/O库-fopen()-文件permission

- fopen()没有设定创建文件权限的参数，POSIX.1要求具有如下权限(0666或者-rw-rw-rw)：
  S_IRUSE|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
- 用户可以通过umask修改文件存取的权限，其结果为(0666 & ～umask)

```
[yjs@boss io]$ umask
0022
[yjs@boss io]$ ./test5
[yjs@boss io]$ ls -alh test_file_3
-rw-r--r--  1 yjs devel 0 Apr  6 10:17 test_file_3
[yjs@boss io]$ umask 0002
[yjs@boss io]$ rm -f test_file_3
[yjs@boss io]$ ./test5
[yjs@boss io]$ ls -alh test_file_3
-rw-rw-r--  1 yjs devel 0 Apr  6 10:17 test_file_3
[yjs@boss io]$
```

4.6 标准I/O库-fclose()

fopen()用于关闭一个已经打开的流:

```
#include <stdio.h>

int fclose(FILE *stream);
```

- fclose()调用成功返回0，失败返回EOF，并设置errno
- 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。
- 当一个进程正常终止时(直接调用exit函数，或从main函数返回)，则所有带未写缓存数据的标准I / O流都被刷新，所有打开的标准I / O流都被关闭。
- 在调用fclose()关闭流后对流所进行的任何操作，包括再次调用fclose()，其结果都将是未知的。

4.7 标准I/O库-读写流(非格式化I/O)

- 调用fopen()成功打开流之后，可在三种不同类型的非格式化I/O中进行选择，对其进行读、写操作:
  1. **每次一个字符的I/O**。一次读或写一个字符，如果流是带缓存的，则标准I/O函数处理所有缓存。
  2. **每次一行的I/O**。使用fgets()和fputs()一次读或写一行。每行都以一个新行符终止。当调用fgets()时，应说明能处理的最大行长。
  3. **直接I/O**。fread()和fwrite()函数支持这种类型的I/O。每次I/O操作读或写某种数量的对象，而每个对象具有指定的长度。这两个函数常用于从二进制文件中读或写一个结构。

4.7.1 标准I/O库-读写流-字符I/O-输入

以下三个函数可用于一次读一个字符:

```
#include <stdio.h>

int getc(FILE * stream);
int fgetc(FILE * stream);
int getchar(void);
```

- 三个函数的返回：若成功则为下一个字符，若已处文件尾端或出错则为EOF
- 函数getchar()等同于getc(stdin)
- 注意，不管是出错还是到达文件尾端，这三个函数都返回同样的值。为了区分这两种不同的情况，必须调用ferror()或feof()。

```
#include <stdio.h>

int feof(FILE * stream);
int ferror(FILE * stream);
void clearerr(FILE * stream);
```

4.7.1 标准I/O库-读写流-字符I/O-输出

以下三个函数可用于一次输出一个字符:

```
#include <stdio.h>

int putc(int c, FILE * stream);
int fputc(int c, FILE * stream);
int putchar(int c);
```

putchar(c)等价于putc(c,stdout)

4.7.1 标准I/O库-读写流-字符I/O-example

example: 循环从标准输入(stdin)逐个字符读入数据，并逐个字符显示到标准输出。

```c
#include <stdio.h>

int main(void)
{
  while (1)
    fputc(fgetc(stdin), stdout);

  return 0;
}
```

输出结果:

```
[yjs@boss io]$ ./test6
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
123
123
```

4.7.2 标准I/O库-读写流-行I/O-输入

下列两个函数提供每次输入一行的功能：

```
#include <stdio.h>

char *gets(char *s);
char *fgets(char *s, int size, FILE * stream);
```

- 两个函数返回：若成功则为buf，若已处文件尾端或出错则为null
- 这两个函数都指定了缓存地址，读入的行将送入其中。gets()从标准输入读，而fgets()则从指定的流读。
- 对于fgets()，必须指定缓存的长度n。此函数一直读到下一个新行符为止，但是不超过n-1个字符，读入的字符被送入缓存。该缓存以null字符结尾。如若该行，包括最后一个新行符的字符数超过n-1，则只返回一个不完整的行，而且缓存总是以null字符结尾。对fgets()的下一次调用会继续读该行。
- gets()是一个不推荐使用的函数，因为调用者在使用gets()时不能指定缓存的长度，这样就可能造成缓存越界（如若该行长于缓存长度），写到缓存之后的存储空间中，从而产生不可预料的后果。
- gets()与fgets()的另一个区别是，gets()并不将新行符存入缓存中。

4.7.2 标准I/O库-读写流-行I/O-输出

下列两个函数提供每次输出一行的功能：

```
#include <stdio.h>

int puts(const char *s);
int fputs(const char *s, FILE * stream);
```

- 两个函数返回：若成功则为非负值，若出错则为EOF
- 函数fputs()将一个以null符终止的字符串写到指定的流，终止符null不写出。注意，这并不一定是每次输出一行，因为它并不要求在null符之前一定是新行符。通常，在null符之前是一个新行符，但并不要求总是如此。
- puts()将一个以null符终止的字符串写到标准输出，终止符不写出。但是，puts()然后又将一个新行符写到标准输出。
- puts()并不像它所对应的gets()那样不安全。但是我们还是应避免使用它，以免需要记住它在
- 最后又加上了一个新行符。如果总是使用fgets()和fputs(),那么就会熟知在每行终止处我们必须自己加一个新行符。

4.7.2 标准I/O库-读写流-行I/O-example

example: 循环从标准输入(stdin)逐行读入数据，并逐行字符显示到标准输出，每次读取的最大长度为20字节。

```
#include <stdio.h>

#define MAXLINE 20

int main(void)
{
  char line[MAXLINE];

  while (fgets(line, MAXLINE, stdin) != NULL && line[0] != '\n')
  {
    fputs("result: ", stdout);
    fputs(line, stdout);
  }

  return 0;
}
```

输出结果:

```
[yjs@boss io]$ ./test7
012345678901234567890123456789012345
result: 01234567890123456789result: 0123456789012345
```

4.7.3 标准I/O库-读写流-二进制I/O

下列两个函数以执行二进制I/O(direct I/O)操作:

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE * stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE * stream);
```

- 两个函数的返回：读或写的对象数
- 对于二进制数据我们更愿意一次读或写整个结构。
- 为了使用getc()或putc()做到这一点，必须循环读取整个结构，一次读或写一个字节。(效率低)
- fputs()在遇到null字节时就停止，而在结构中可能含有null字节，所以不能使用每次一行函数实现这种要求。如果输入数据中包含有null字节或换行符，则fgets()也不能正确工作。(实现限制)

4.7.3 标准I/O库-读写流-二进制I/O-example

example1:读或写一个二进制数组，将一个浮点数组的第2至第5个元素写至一个文件上：

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(int argc, char **argv)
{
  FILE *fp;
  float data[10];

  int i;

  for (i = 0; i < 10; i++)
  {
    data[i] = i * 0.1;
  }

  if ((fp = fopen("test_file_4", "wb+")) == NULL)
  {
    fprintf(stderr, "Open file test_file_4 for binary write failed: %s\n", strerror(errno));
    return 0;
  }

  fprintf(stdout, "[DEBUG] sizeof(float) = %d\n", sizeof(float));

  if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
  {
    fprintf(stderr, "Binary write data to file test_file_4 failed: %s\n", strerror(errno));
    return 0;
  }

  fclose(fp);

  return 0;
}
```

4.7.3 标准I/O库-读写流-二进制I/O-example

example2:读或写一个结构体，将一个结构体写至一个文件上：

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>

#define NAMESIZE        16

struct
{
  short count;
  long total;
  char name[NAMESIZE];
} item;

int main(int argc, char **argv)
{
  FILE *fp;

  if ((fp = fopen("test_file_5", "wb+")) == NULL)
  {
    fprintf(stderr, "Open file test_file_4 for binary write failed: %s\n", strerror(errno));
    return 0;
  }

  memset(&item, 0, sizeof(item));

  item.count = 9;
  item.total = 74;
  // char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
  strncpy(item.name, "Richard Stallman", NAMESIZE);

  //fprintf(stdout, "[DEBUG] sizeof(float) = %d\n", sizeof(float));

  //if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
  if (fwrite(&item, sizeof(item), 1, fp) != 1)
  {
    fprintf(stderr, "Binary write data to file test_file_4 failed: %s\n", strerror(errno));
    return 0;
  }

  fclose(fp);

  return 0;
}
```

4.8 标准I/O库-定位流

- 定位标准I/O流的两种方式
  - ftell()和fseek(): 这两个函数自V7以来就存在了，但是它们都假定文件的位置可以存放在一个长整型中。
  - fgetpos()和fsetpos()。这两个函数是新由ANSI C引入的。它们引进了一个新的抽象数据类型fpos_t，它记录文件的位置。在非UNIX系统中，这种数据类型可以定义为记录一个文件的位置所需的长度。
- 需要移植到非UNIX系统上运行的应用程序应当使用fgetpos()和fsetpos()

4.8 标准I/O库-定位流-fseek()/ftell()/rewind()

fseek()/ftell()/rewind()函数原型：

```
#include <stdio.h>

int fseek(FILE * stream, long offset, int whence);
long ftell(FILE * stream);
void rewind(FILE * stream);
```

- ftell()用于取得当前的文件位置，调用成功则为当前文件位置指示，若出错则为-1L
- fseek()用户设定stream流的文件位置指示，调用成功返回0，失败返回-1，并设置errno
- fseek()的whence参数与lseek()中的whence参数定义相同。
- rewind()用于设定流的文件位置指示为文件开始，该函数调用成功无返回值。
- rewind()等价于(void)fseek(stream, 0L, SEEK_SET)

4.8 标准I/O库-定位流-fgetpos()/fsetpos()

fegetpos()/fsetpos()函数原型：

```
#include <stdio.h>

int fgetpos(FILE * stream, fpos_t * pos);
int fsetpos(FILE * stream, fpos_t * pos);
```

- 两个函数返回：若成功则为0，若出错则为非0
- fgetpos()将文件位置指示器的当前值存入由*pos*指向的对象中。在以后调用fsetpos()时，可以使用此值将流重新定位至该位置。

4.9 标准I/O库-格式化I/O

- 格式化输出函数原型:

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

4.9 标准I/O库-格式化I/O

- 格式化输入函数原型:

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

#include <stdarg.h>
int vscanf(const char *format, va_list ap);
int vsscanf(const char *str, const char *format, va_list ap);
int vfscanf(FILE *stream, const char *format, va_list ap);
```

4.10 标准I/O库-临时文件

- 标准I/O库提供了两个函数以帮助创建临时文件：
  - tmpnam()产生一个与现在文件名不同的一个有效路径名字符串。每次调用它时，它都产生一个不同的路径名，最多调用次数是TMP_MAX。TMP_MAX定义在<stdio.h>中。
  - tmpfile()创建一个临时二进制文件(类型wb+)，在关闭该文件或程序结束时将自动删除这种文件。

```
#include <stdio.h>

char *tmpnam(char *s);
FILE *tmpfile(void);
```

4.11 标准I/O – 作业

- 任务描述
  1. 使用标准I/O方式，实现一个copy程序，该程序的第一个命令行参数为源文件，第二个命令行参数为目标文件，程序实现将源文件中的内容复制到目标文件。
  2. 命令行参数可以使用argv[1]访问第一个参数，argv[2]访问第二个参数
  3. 使用diff工具检查目标文件与源文件是否一致
- 实现思路
  1. 使用fopen()打开源文件
  2. 使用fopen()打开目标文件
  3. 循环读取(fread())源文件并写入(fwrite())目标文件
  4. 关闭源文件(fclose())
  5. 关闭目标文件(fclose())
- 存在哪些问题?

5. I/O模型比较

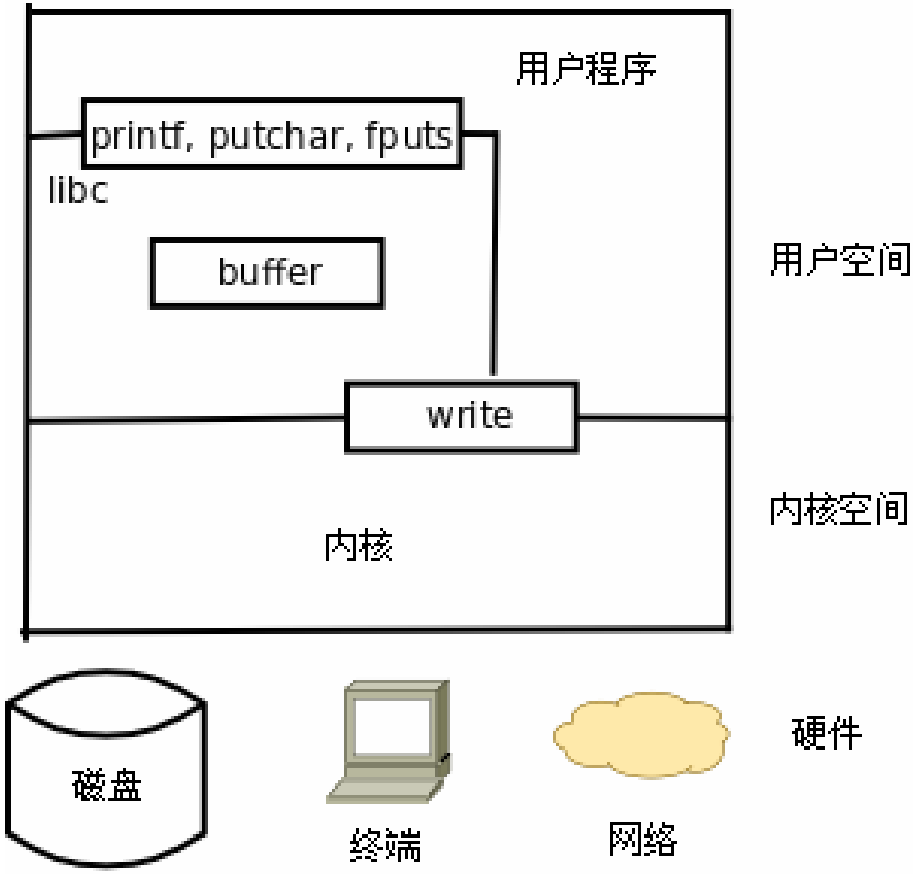
5.1 两种I/O比较

I/O模型	文件I/O	标准I/O
缓冲方式	非缓冲I/O	缓冲I/O
操作对象	文件描述符	流(FILE *)
打开	open()	fopen()/freopen()/fdopen()
读	read()	fread()/fgetc()/fgets()…
写	write()	fwrite()/fputc()/fputs()…
定位	lseek()	fseek()/ftell()/rewind()/fsetpos()/fgetpos()
关闭	close()	fclose()

5.2 库函数与系统调用的层次关系

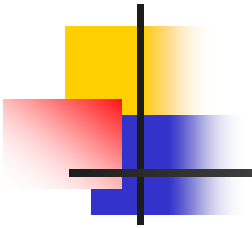
- fopen(3)
  - 调用open(2)打开指定的文件，返回一个文件描述符（就是一个int 类型的编号），分配一个FILE 结构体，其中包含该文件的描述符、I/O 缓冲区和当前读写位置等信息，返回这个FILE 结构体的地址。
- fgetc(3)
  - 通过传入的FILE *参数找到该文件的描述符、I/O 缓冲区和当前读写位置，判断能否从I/O 缓冲区中读到下一个字符，如果能读到就直接返回该字符，否则调用read(2)，把文件描述符传进去，让内核读取该文件的数据到I/O 缓冲区，然后返回下一个字符。注意，对于C 标准I/O 库来说，打开的文件由FILE *指针标识，而对于内核来说，打开的文件由文件描述符标识，文件描述符从open 系统调用获得，在使用read、write、close 系统调用时都需要传文件描述符。
- fputc(3)
  - 判断该文件的I/O 缓冲区是否有空间再存放一个字符，如果有空间则直接保存在I/O 缓冲区中并返回，如果I/O 缓冲区已满就调用write(2)，让内核把I/O 缓冲区的内容写回文件。
- fclose(3)
  - 如果I/O 缓冲区中还有数据没写回文件，就调用write(2)写回文件，然后调用close(2)关闭文件，释放FILE 结构体和I/O 缓冲区。

5.2 库函数与系统调用的层次关系



5.3 如何选择I/O模型

- 用 Unbuffered I/O函数每次读写都要进内核，调一个系统调用比调一个用户空间的函数要慢很多，所以在用户空间开辟I/O缓冲区还是必要的，用C标准I/O 库函数就比较方便，省去了自己管理I/O 缓冲区的麻烦。
- 用C标准I/O库函数要时刻注意I/O缓冲区和实际文件有可能不一致，在必要时需调用fflush(3)。
- 我们知道UNIX的传统是Everything is a File，I/O函数不仅用于读写常规文件，也用于读写设备，比如终端或网络设备。在读写设备时通常是不希望有缓冲的，例如向代表网络设备的文件写数据就是希望数据通过网络设备发送出去，而不希望只写到缓冲区里就算完事儿了，当设备接收到数据时应用程序也希望第一时间被通知到，所以网络编程通常直接调用Unbuffered I/O函数。



Let's DO it!

Thanks for listening!

☺