Linux之IO系统编程

wuzhh

一、概述

1.1 LinuxI/O体系结构

- 设备类型：外部设备，包括键盘，鼠标，硬盘，内存，打印机等
- 总线系统：总线类型：PCI、ISA、SBus、SCSI、并口串口、AMBA
- 文件系统

1.2 I/O操作过程

- 打开文件：一个应用程序通过要求内核打开相应文件，宣告它要访问要给I/O设备；内核返回要给非负整数（描述符号）
- 改变文件位置：对于每个打开的文件，内核保持一个文件位置k，初始为0，这个文件位置是从文件投开始的偏移量，通过执行seek操作显式地设置当前位置为k（文件位置指针）
- 读写文件：读从文件拷贝n > 0个字节到存储器；写：从存储器拷贝n > 0字节到文件
- 关闭文件：通知内核关闭文件，作为响应，内核释放文件打开时创建的数据结构

1.3 GNU Linux I/O操作类别

- 文件流的标准输入输出
- 底层输入输出
- 文件系统接口
- 管道及FIFO（先入先除队列）
- Socket
- 底层终端接口（tty）

1.4 主要数据结构

1.4.1 FD

对于内核而言，所有打开文件都由文件描述符引用

--- 文件描述符是一个**非负整数**，当打开一个现存文件或创建一个新文件时，内核向进程返回一个文件描述符

--- 当读写一个文件时，用open/creat返回的**文件描述符fd**标识该文件，将其作为**参数**传送给read/write

--- 在POSIX。1应用程序中，文件描述符为常数0、1和2分别代表STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO，即标准输入、标准输出和标准出错输出，这些常数一般在头文件`<unistd.h>`中定义

--- 文件描述符**范围**为0 ~ OPEN_MAX，不同位数的系统，表示范围不同

1.4.2 File

文件结构体代表一个打开的文件，系统中的每个打开的文件在内核空间都有一个关联的struct file。它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。在内核创建和驱动源码中，struct file的指针通常被命名为file或filp。

1.4.3 Files Structure

文件描述符(file_struct)是操作系统用来管理文件的数据结构,当我们创建一个进程时，会创建文件描述符表，进程控制块PCB中的fs指针指向文件描述符表，当我们创建文件时，会为指向该文件的指针FILE* 关联一个文件描述符并添加在文件描述符表中。在文件描述符表中fd相当于数组的索引，FILE*相当于数组的内容吗，指向一个文件结构体

1.5 底层输入输出

- 打开（建立）和关闭文件：open、close
- 读写文件：read、write
- 设置文件位置：lseek
- 文件符号和流
- 快速聚集I/O
- 内存映射I/O
- 同步I/O操作
- 文件锁定
- 中断驱动输入

1.6 标准输入输出

1.6.1 标准IO库简介

- Linux调用层次结构
- Linux内核功能的划分
- 系统调用的过程
- 添加新的系统调用：修改系统调用表 -> 添加系统调用对应的内核函数声明 -> 添加系统调用对应的内核函数定义

  arm例子：1、系统调用表：更新`/arch/arm/kernel/calls.S`

  2、系统调用对应的内核函数声明：更新`/include/linux/syscalls.h`

  3、添加系统调用对于的内恶化函数定义：在内核源码目录树中添加或是指定一个源文件，然后在该文件中添加

1.6.2 常用函数

fopen、fread、fwrite、fclose、fflush、fseek

1.6.3 lowlevel和std的选择

1.7 文件锁定

1.7.1 文件锁概述

- Linux中软件、硬件资源都是文件（一切皆文件），文件在多用户环境中是可共享的
- 文件锁是用于解决资源的共享使用的一种机制：当多个用户需要共享一个文件时，Linux通常采用的方法是给文件上锁，来避免共享的资源产生竞争的状态

文件锁定类型

强制锁 mandatory lock：是由内核执行的锁，当一个文件被上锁进行写入操作的时候，内核将阻止其他任何文件对其进行读写操作。采用强制性锁对性能的影响很大，每次读写操作都必须检查是否有锁存在

建议锁 Advisory lock：要求每个使用上锁文件的进程都要检查是否有锁存在，并且尊重已有的锁。在一般情况下，内核和系统都不使用建议性锁，它们依靠程序员遵守这个规定。

1.7.2 文件锁结构

file_lock

```
struct flock
    {
        short l_type;/*F_RDLCK, F_WRLCK, or F_UNLCK*/
        off_t l_start;/*相对于l_whence的偏移值，字节为单位*/
        short l_whence;/*从哪里开始：SEEK_SET, SEEK_CUR, or SEEK_END*/
        off_t l_len;/*长度，字节为单位; 0 意味着缩到文件结尾*/
        pid_t l_pid;/*returned with F_GETLK*/
    };
/* 锁类型： F_RDLCK(读共享锁)， F_WRLCK(写互斥锁),和F_UNLCK(对一个区域解锁)
 * 锁开始： 锁位置(l_whence),相对于l_whence要锁或者解锁的区域开始位置(l_start)
 * 锁长度： 要锁的长度，字节计数(l_len) 0表示锁至文件结尾
 * 锁拥有者：记录锁的拥有进程ID，这个进程可以阻塞当前进程，仅F_GETLK形式返回
*/
```

1.7.3 文件锁函数调用方法

fcntl

`int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );` 头文件：`<unistd.h> <fcntl.h>`

参数filedes：要锁定的文件描述符
参数cmd：F_GETLK，得到锁；F_SETLK，设置锁；F_SETLKW，设置锁并返回
参数结构体：指向一个flock结构指针（文件锁结构）

```
struct flock
    {
        short l_type;/*F_RDLCK, F_WRLCK, or F_UNLCK*/
        off_t l_start;/*相对于l_whence的偏移值，字节为单位*/
        short l_whence;/*从哪里开始：SEEK_SET, SEEK_CUR, or SEEK_END*/
        off_t l_len;/*长度，字节为单位; 0 意味着缩到文件结尾*/
        pid_t l_pid;/*returned with F_GETLK*/
    };
/* 锁类型： F_RDLCK(读共享锁)， F_WRLCK(写互斥锁),和F_UNLCK(对一个区域解锁)
 * 锁开始： 锁位置(l_whence),相对于l_whence要锁或者解锁的区域开始位置(l_start)
 * 锁长度： 要锁的长度，字节计数(l_len) 0表示锁至文件结尾
 * 锁拥有者：记录锁的拥有进程ID，这个进程可以阻塞当前进程，仅F_GETLK形式返回
*/
```

lock

1.7.4 线程锁定

1.8 文件和目录

1.8.1 文件和目录的维护

- chmod：用来修改某个目录或文件的访问权限（改变权限）
- chown：用于修改文件（或目录）的所有者，除此之外，这个命令也可以修改文件（或目录）的所属组
- unlink：用于删除单个文件的命令行实用程序
- mkdir/rmdir：创建/删除指定目录

1.8.2 扫描目录

- opendir：打开一个目录，并返回指向该目录的句柄，供后续操作使用
- readdir：读取目录，获取目录下所有文件的名称以及对应 inode 号
- closefir：用于关闭处于打开状态的目录，同时释放它所使用的资源

1.9 /proc文件系统

包含内存管理、系统进程特征数据、文件系统、设备驱动程序、系统总线、电源管理、终端、系统控制参数等

二、文件IO

是从用户空间角度考虑的输入与输出

从内核读取数据或从文件中读取数据交input（**read函数**）；写数据到内核/文件叫output（**write函数**）

定位要对内核进行读写的文件（**open函数**）；　　关闭文件（**close函数**）

open和close函数以及touch命令的实现

2.1 open

open -- 打开或创建一个文件

`int open(const char *pathname, int flags, mode_t mode)` 在`fcntl.h`文件中声明，函数的作用；创建或打开某个文件

返回值：成功返回文件描述符（非负的正实数，即文件ID号）；出错返回-1

参数`*pathname`：包含文件名和路径

参数`flags`：　打开文件的方式

参数`mode`：　创建文件的权限

flag内容表：

flag	功能
`O_RDONLY`	只读
`O_WRONLY`	只写
`O_RDWR`	读写
`O_CREAT`	创建一个文件
`O_EXCL`	如果使用`O_CREAT`时文件存在，则可返回错误消息，这一参数测试文件是否存在
`O_TRUNC`	打开文件（会把已经存在的内容给删除）
`O_APPEND`	追加方式打开文件（不会把已经存在的内容给删除）

`ls -lai`命令 每一行的第一个数字串为文件的ID号（inode）

ID号的规律：从0开始累加，程序进行时（进程），内核会自动打开三个文件描述符0，1，2，分别对应标准输入、输出和出错，这样在程序中，每打开一个文件，文件描述符值从3开始累加

open函数创建文件时的**实际权限**是：`mode & ~(umask)` 例：`B111 111 111 & ~(B000 010 010) = B111 101 101`

`umask`：**掩码 （`umask`命令即可查看对应的掩码）**

2.2 write

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

参数：写到哪里去/写什么/写多少个；返回值：实际写的字节数

参数`fd`：向哪个文件写； 参数`*buf`：向这个文件中写入什么内容； 参数`count`：向这个文件中写多少个

2.3 read

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

参数：从哪个文件读/读到哪里去/想要读多少个； 返回值：实际读的字节数(从"指针指向的地方开始读)

2.4 lseek

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

功能：调整读写的位置指针的位置，函数头文件路径：`sys/type.h unistd.h` 返回值：成功：文件当前的位置；出错：-1

参数`fd`：要调整的文件的文件描述符；

参数`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）

参数`whence`：当前位置的基点，有三个标志，

`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

读写位置指针：

```
1 //
```

2.5 close

`int close(int fd);` 关闭指定文件，成功返回0，出错返回-1

参数：fd文件描述符

三、标准IO

文件IO：直接调用内核提供的系统调用函数，头文件是`unistd.h`

标准IO：间接调用系统调用函数，头文件是`stdio.h`

标准IO的相关函数，不仅可以读写普通文件，也可以向标准的输入或标准的输出中读写

输入输出相关的函数，都是和标准的输入（键盘），标准的输出（显示器）

`getchar(),putchar()` --- 一个字符

`gets(buf),puts(buf)` --- 一串字符

`scanf(),printf()` --- 一个字符一串字符都可以

3.1 三个缓存的概念（数组）

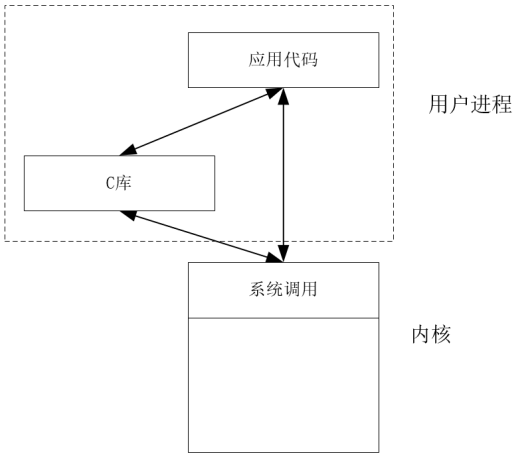
用户空间的缓存(user_buffer)：我们程序中的缓存，就是想从内核读写的缓存（数组）

内核空间的缓存(kernel_buffer)：每打开一个文件，内核在内核空间中也会开辟一块缓存

文件IO中的**写**即是将用户空间中的缓冲写到内核空间的缓存中　（用户空间 -> 内核空间）

文件IO中的**读**即是将内核空间的缓存写到用户空间中的缓存中　（内核空间 -> 用户空间）

库缓存(lib_buffer)：文件IO的库函数中存在的缓存（如C库中）




```c
/* printf满足一定条件：遇到\n时，即会将库缓存的内容写到内核中，即调用了系统调用函数
 * 库函数写满时，也会调用系统调用函数
 *
*/
//无法打印出hello linux
#include"stdio.h"
#include"unistd.h"
int main()
{
    char buf[] = "hello linux";
    printf("%s",buf);
    while(1);
    return 0;
}

//可以打印出hello linux
#include"stdio.h"
#include"unistd.h"
int main()
{
    char buf[] = "hello linux\n";
    printf("%s",buf);
    while(1);
    return 0;
}
//库函数写满时（1024），可以打印出hello linux
#include"stdio.h"
#include"unistd.h"
int main()
{
    char buf[] = "hello linux";
    int i = 0;
    while(i <= 95)  //可不断调试数字，测出可以打印的临界次数
```

```
    {
        printf("%s",buf);
    }
    while(1);
    return 0;
}
```

3.2 标准IO相关函数

文件IO	标准IO
open	fopen
close	fclose
lseek	fseek，rewind
read/write	读写函数较多（分三类，全缓存、行缓存和无缓存）

3.3 FILE

`FILE *fopen(const char *path, const char *mode);`

返回值：FILE*：文件流指针；类似于文件IO的文件描述符
FILE定义：`struct_IO_FILE`，在路径`/usr/include/libio.h`中，包含读写缓存的首地址、大小、位置指针等

fopen函数创建一个文件权限默认设定的权限都是`0666`，最终生成的**文件的权限为：**`0666 & ~(umask)`

标准的输入流：`stdin  0`
标准的输出流：`stdout  1`
标准的出错流：`stderr  2`

mode

r/rb	打开只读文件，该文件必须存在
r+/r+b	打开可读写的文件，该文件必须存在
w/wb	打开只写文件，若文件存在则文件长度清为0，即会擦写文件以前内容。若文件不存在则建立该文件
w+/w+b/wb	打开可读写文件，若文件存在则文件长度清为零，即会擦写文件以前内容。若文件不存在则建立该文件
a/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+/a+b/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

- b：二进制文件
- r：只读方式打开文件，文件不存在则创建
- w/a：只写方式打开文件，文件不存在则创建；区别：w等价于`O_TRUNC`(打开文件 会把原来内容删除)，a等价于`O_APPEND`（追加）

- +：读写方式打开文件，文件必须存在

```c
//fopen函数的用法
#include"stdio.h"
int main(int argc, char *argv[])
{
    FILE *fp;   //定义文件流指针fp
    fp = fopen(argv[1], "w+"); //创建并且以读写方式打开文件
    if(fp == NULL)
    {
        printf("creat file %s sucess failure\n", argv[1]);
        return -1;
    }
    printf("creat file %s sucess\n", argv[1]);
    fclose(fp);         //关闭文件
    return 0;
}
```

`int fclose(FILE *stream)` 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，刷新缓存中的数据，如果标准I/O库已经为该自动分配了一个缓存，则释放此缓存

也就是说**当程序执行了`fclose()`，该函数会尝试将内容从库中的缓存写到内核缓存中**

3.4 读写函数

三种读写函数：

一类：行缓存 遇到新行符（\n）即调用系统调用函数

读：`fgets(), gets(), printf(), fprintf(), sprintf()`
写：`fputs(), puts(), scanf()`

单个字符的读写

读：`fgetc(), getc(), getchar()`
写：`fputc(), putc(), putchar()`

二类：无缓存　只要用户调这个函数，就会将其内容写道内核中

stderr

三类：全缓存　只有写满缓存再调用系统调用函数

读：`fread`
写：`fwrite`

3.4.1 行缓存的读写函数

`char *fgets(char *s, int size, FILE *stream)` 返回值：成功返回s（缓存的地址），若处于文件尾端或出错则为null

第一个参数`*s`：缓存，即读到哪里去
第二个参数`size`：读多少字节
第三个参数`*stream`：从什么地方读

```c
//示例
#include "stdio.h"
int main(int argc, char *argv[])
```

```
{
    FILE *fp;
    char buf[] = "hello linux\n";
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
        printf("open file a.c failure\n");
        return -1;
    }
    printf("open file a.c sucess\n");
    fputs(buf, fp);
    fclose(fp);
    return 0
}
```

`char *fput(const char *s, FILE *stream)` 返回值：成功则非负值，若出错则为EOF

第一个参数`*s`：写什么内容

第二个参数`*stream`：写到哪里去

```
//示例 这里注意读写位置指针 注意是否需要调整读写位置指针
/*以下示例无法读到hello linux*/
#include "stdio.h"
int main(int argc, char *argv[])
{
    FILE *fp;
    char buf[] = "hello linux\n";
    char readbuf[128] = {0};
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
        printf("open file a.c failure\n");
        return -1;
    }
    printf("open file a.c sucess\n");
    fputs(buf, fp);
    fgets(readbuf, 128, fp);
    printf("readbuf:%s\n", readbuf);
    fclose(fp);
    return 0
}
```

3.4.2 刷新缓存函数

`int fflush(FILE *fp)`

把库函数中的缓存的内容强制写到内核中

```
//示例，可读到hello linux
#include "stdio.h"
int main(int argc, char *argv[])
{
    FILE *fp;
    char buf[] = "hello linux";
    char readbuf[128] = {0};
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
```

```
        printf("open file a.c failure\n");
        return -1;
    }
    printf("open file a.c sucess\n");
    fputs(buf, fp);
    fflush(fp);
    while(1);
    fclose(fp);
    return 0
}
```

3.4.3 调整读写位置指针函数

`fseek()` 参数与 `lseek()` 是一样的但是返回值不一样；`lseek()` 返回值是：当前文件的位置指针值；`fseek()` 成功返回0，失败返回-11

`rewind(FILE *fp)` 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（相当于 `fseek()` 函数的子功能）

`rewind()` 等价于 `(void)fseek(fp 0, SEEK_SET)`

`ftell(FILE *fp)` 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

3.4.4 行缓存的读写函数gets和puts

`char *gets(char *s)` 读
`int puts(const char *s)` 写

`gets()` 和 `fgets()` 区别：

- `gets()` 不指定缓存的长度，可能会造成缓存越界（如若该行长于缓存长度），写到缓存之后的存储空间中，结果不可预料
- `gets()` 只能从标准输入中读
- `gets()` 与 `fgets()` 的另一个区别是：`gets()` 并**不将新行符存入缓存中**，`fgets` **将新行符存入缓存中**（即会包含键盘输入的enter键）

`puts()` 与 `fputs` 区别

- `puts()` 只能向标准转出中写
- `puts()` 输出时会添加一个新行符；`fputs` 不会添加

3.4.5 fprintf、printf、sprintf

`int fprintf(FILE *stream, "字符串格式")`

fprintf 可以**输出到文件**，也可以输出到显示器
printf **只能**输出到显示器中

`int sprintf(str *, "字符串格式")`

输出内容到一个字符串中

```
//fprintf、printf示例 example
#include "stido.h"
int main()
{
    FILE *fp;
    int i = 10;
```

```
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
        printf("open a.c failure\n");
        return -1;
    }
    //printf("open a.c sucess\n");
    fprintf(fp, "open a.c sucess i = %d\n", i);
    fclose(fp);
    return 0;
}

//spritnf example
#include "stido.h"
int main()
{
    int i = 10;
    char buf[128] = {0};
    sprintf(buf, "i = %d", i);
    printf("buf = %s\n", buf);
    return 0;
}
```

3.4.6 一个字符串读写函数 fgetc 和 fputc

`int fgetc(FILE *fp)` 将文件中的内容一个字符输出到显示器，到文件结尾时返回EOF（不是行缓存函数）

功能：从文件中读取一个字符； 参数：文件流； 返回值：正确为读取的字符，到文件结尾或出错时返回EOF

`char *fgets(char *s, int size, FILE *stream)` 返回值：成功则为s（缓存的地址），若处于文件结尾或读错则为null

`int fputc(int c, FILE *fp)` 输入一个字符到文件中，成功则返回输入的字符，出错返回EOF（存在缓存，但不是行缓存函数）

功能：写一个字符到文件中； 参数：第一个参数为要写的字符，第二个参数为文件流；

返回值：成功返回输入的字符，出错返回EOF

```
//fputc example
#include "stido.h"
int main()
{
    FILE *fp;
    int i = 10;
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
        printf("open a.c failure\n");
        return -1;
    }
    printf("open a.c sucess\n");

    fputc('a', fp);
    fputc('\n', fp);    //检查当有'\n'时是否会写入
    fflush(fp); //强制将缓存刷新进内核，运行可以写入
```

```
    fclose(fp);
    return 0;
}

//fgetc example
#include "stido.h"
int main()
{
    FILE *fp;
    int ret;
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
        printf("open a.c failure\n");
        return -1;
    }
    printf("open a.c sucess\n");

    fputc('a', fp);     //同样要注意读写指针
    rewind(fp);         //将读写指针调整到文件开头
    ret = fgetc(fp);
    printf("ret = %c\n", ret);
    /*
    ret = fgetc(fp);
    printf("ret = %d\n", ret);      //检查再读时是否错误，EOF==?-1
    */
    fclose(fp);
    return 0;
}
```

3.4.7 feof、ferror、clearerr

`int feof(FILE *stream)` 功能：判断是否已经到文件结束

参数：文件流；返回值：到文件结束，返回为非0，没有则返回0

`int ferror(FILE *stream)` 功能：判断是否读写错误

参数：文件流；返回值：是读写错误，返回为非0， 不是则返回0

`void clearerr(FILE *stream)` 功能：清除流错误

参数：文件流

```
//feof and ferror example
#include "stido.h"
int main()
{
    FILE *fp;
    int ret;
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
        printf("open a.c failure\n");
        return -1;
    }
    printf("open a.c sucess\n");

    fputc('a', fp);     //同样要注意读写指针
    rewind(fp);         //将读写指针调整到文件开头
```

```
    ret = fgetc(fp);
    printf("ret = %c\n", ret);
    ret = fgetc(fp);
    printf("ret = %d\n", ret);      //检查再读时是否错误，EOF==?-1 end or read err
    //clearerr(fp);     //清除流错误
    printf("feof = %d, ferror = %d\n", feof(fp), ferror(fp));

    fclose(fp);
    return 0;
}
```

3.4.8 cat命令

将文件的内容输出到显示器

```
//cat example
/* 实现过程
 * 内核驱动程序读取文件的内容到内核缓存中，用户空间通过read函数将内核缓存中的数据读到用户空间去，再通过write函数写到内核空间的
 * 缓存，然后内核通过驱动程序写到显示器上来
 */
#include "stido.h"
#include "unistd.h"
#include "fcntl.h"
#include "string.h"
int main(int argc, char *argv[])
{
    FILE *src_fp;
    int read_ret;       //定义变量存储读到的值
    if(argc < 2)
    {
        printf("please input src file\n");
        return -1;
    }
    src_fp = fopen(argv[1], "r");//打开要读的文件，以只读方式打开
    if(src_fp == MULL)
    {
        printf("open src file %s failure\n",  argv[1]);
        return -2;
    }
    printf("open src file %s sucess\n", argv[1]);
    //start read write
    while(1)        //要考虑什么时候读完退出
    {
        read_ret = fgetc(src_fp);       //读
        if(feof(src_fp))                //检查是否读完文件
        {
            printf("read file %s end\n", argv[1]);
            break;              //读完 跳出循环
        }
        fputc(read_ret, stdout);        //写 写到哪里去
    }
    fclose(src_fp);
    return 0;
}
```

3.4.9 全缓存的两个函数fread、fwrite

`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)` （有缓存 但不是行缓存）

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

功能：全缓存读写函数； 返回值：实际读写的单元数

参数`*ptr`：写的内容

参数`size`：写的内容中，每个单元所占的字节数

参数`nmemb`：写的内容中，有多少个单元数

参数`*stream`：写到哪里去

```c
//example fwrite
#include "stido.h"
int main(int argc, char *argv[])
{
    FILE *fp;
    char buf[] = "hello linux";
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
        printf("open a.c failure\n");
        return -1;
    }
    printf("open a.c sucess\n");
    fwrite(buf, sizeof(char), sizeof(buf), fp);

    fclose(fp);
    return 0;
}

//example fread
#include "stido.h"
int main(int argc, char *argv[])
{
    FILE *fp;
    char buf[] = "hello linux";
    char readbuf[128] = {0};
    fp = fopen("./a.c", "w+");
    if(fp == NULL)
    {
        printf("open a.c failure\n");
        return -1;
    }
    printf("open a.c sucess\n");
    fwrite(buf, sizeof(char), sizeof(buf), fp);
    rewind(fp);     //调整读写指针 或调用fseek(fp, 0, SEEK_SET);

    fread(readbuf, sizeof(char), sizeof(readbuf), fp);
    printf("readbuf %s\n", readbuf);
    fclose(fp);
    return 0;
}
```

3.4.10 四类读写函数的效率对比

fgetc和read效率对比

```c
//fgetc 函数
#include "stdio.h"
#include "unistd.h"
#include "fcntl.h"
#include "string.h"
int main(int argc, char *argv[])
{
    FILE *src_fp, *des_fp;
    int read_ret;
    if(argc < 3)
    {
        printf("please input src and des file\n");
        return -1;
    }
    src_fp = fopen(argv[1], "r");   //打开源文件
    if(src_fp == NULL)
    {
        printf("open src file %s failure\n", argv[1]);
        return -2;
    }
    printf("open src file %s sucess\n", argv[1]);
    des_fp = fopen(argv[2], "w");  //打开目标文件 以写方式打开
    if(des_fp == NULL)
    {
        printf("open des file %s failure\n", argv[2]);
        return -3;
    }
    printf("open des file %s sucess\n", argv[2]);
    //start read write
    while(1)
    {
        read_ret = fgetc(src_fp);//从源文件开始读
        if(feof(src_fp))//读到文件结尾时结束
        {
            printf("read file %s end\n", argv[1]);
            break;
        }
        fputc(read_ret, des_fp);//写到目标文件中去
    }
    fclose(src_fp);
    fclose(des_fp);
    return 0;

}
/* 查看程序运行时间命令：time 命令
 * time ./fgetc a.c b.c
 */
//read 函数
#include "stdio.h"
#include "unistd.h"
#include "fcntl.h"
#include "string.h"
int main(int argc, char *argv[])
```

```
{
    int src_fd, des_fd;      //文件描述符
    int read_ret;
    char buf[128] = {0};     //读缓存定义
    if(argc < 3)
    {
        printf("please input src and des file\n");
        return -1;
    }
    src_fd = fopen(argv[1], O_RDONLY);  //以只读方式打开
    if(src_fd < 0)
    {
        printf("open src file %s failure\n", argv[1]);
        return -2;
    }
    printf("open src file %s sucess\n", argv[1]);
    des_fd = open(argv[2], O_CREAT | O_WRONLY, 0777);  //打开目标文件 以写方式打开
    if(des_fd < 0)
    {
        printf("open des file %s failure\n", argv[2]);
        return -3;
    }
    printf("open des file %s sucess\n", argv[2]);
    //start read write
    while(1)
    {
        read_ret = read(src_fd, buf, 128);   //从源文件开始读 读到buf去，读128个
        write(des_fd, buf, read_ret);   //写到目标文件中去，写的内容，写多少个（读多少写多少）
        if(read_ret < 128)//读到文件结尾时结束
        {
            printf("read file %s end\n", argv[1]);
            break;
        }
    }
    close(src_fd);
    close(des_fd);
    return 0;
}
/* 对read来讲，只存在用户空间的缓存，如果调用read函数，就会将用户空间的缓存写到内核里面去；
 * 对fgetc来讲，在库函数中本身就有一个缓存，当写满这个缓存时，再调用系统调用函数read写到内核里面去
 * 所以read在内核空间的时间要多于fgetc，由于fgetc是一个字符的读写的过程，所以在用户空间所花费的时间要多于read
 */
```

fread和fgets效率对比

```
//fgets 函数
#include "stdio.h"
#include "unistd.h"
#include "fcntl.h"
#include "string.h"
int main(int argc, char *argv[])
{
```

```c
    FILE *src_fp, *des_fp;  //源文件流和目标文件流
    int read_ret;
    char readbuf[128] = {0};
    if(argc < 3)
    {
        printf("please input src and des file\n");
        return -1;
    }
    src_fp = fopen(argv[1], "r");   //打开源文件
    if(src_fp == NULL)
    {
        printf("open src file %s failure\n", argv[1]);
        return -2;
    }
    printf("open src file %s sucess\n", argv[1]);
    des_fp = fopen(argv[2], "w");  //打开目标文件 以写方式打开
    if(des_fp == NULL)
    {
        printf("open des file %s failure\n", argv[2]);
        return -3;
    }
    printf("open des file %s sucess\n", argv[2]);
    //start read write 开始读写
    while(1)
    {
        fgets(readbuf, 128, );//读， 三个参数，写什么，写多少个，写到哪里去
        if(feof(src_fp))//读到文件结尾时结束
        {
            printf("read file %s end\n", argv[1]);
            break;
        }
        fputc(readbuf, des_fp);//写到目标文件中去 写什么，写到哪里去
    }
    fclose(src_fp);
    fclose(des_fp);
    return 0;

}
/* 查看程序运行时间命令：time 命令
 *
 */
//fread 函数
#include "stdio.h"
#include "unistd.h"
#include "fcntl.h"
#include "string.h"
int main(int argc, char *argv[])
{
    int src_fd, des_fd;     //文件描述符
    int read_ret;
    char buf[128] = {0};    //读缓存定义
    if(argc < 3)
    {
        printf("please input src and des file\n");
        return -1;
    }
    src_fd = fopen(argv[1], O_RDONLY);  //以只读方式打开
    if(src_fd < 0)
```

```
    {
        printf("open src file %s failure\n", argv[1]);
        return -2;
    }
    printf("open src file %s sucess\n", argv[1]);
    des_fd = open(argv[2], O_CREAT | O_WRONLY, 0777);  //打开目标文件 以写方式打开
    if(des_fd < 0)
    {
        printf("open des file %s failure\n", argv[2]);
        return -3;
    }
    printf("open des file %s sucess\n", argv[2]);
    //start read write
    while(1)
    {
        read_ret = fread(readbuf, 1, 128, src_fp);   //读到哪里去，读单元的字节数，多少个单元，从哪里读

        if(read_ret < 128)//读到文件结尾时结束
        {
            printf("read file %s end\n", argv[1]);
            break;
        }
        fwrite(readbuf, 1, read_ret, des_fp);   //写的内容，单元字节数，写多少个单元，写到哪里去
    }
    fwrite(readbuf, 1, read_ret, des_fp);   //保证最后一次跳出循环后也能写进去
    close(src_fd);
    close(des_fd);
    return 0;
}
```

效率总结：

fread、fwrite > fgets、fputs > fgetc、fputc > read、write

四、库函数的制作

Linux操作系统支持的函数库分为：

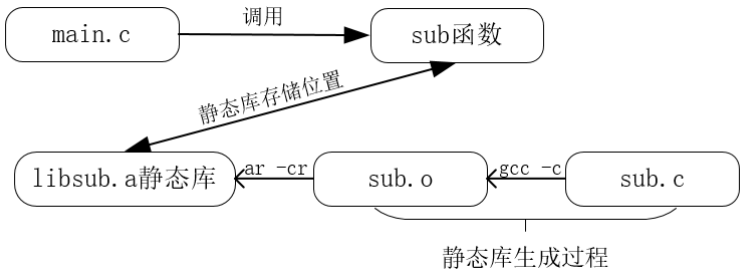
静态库，libxxx.a, **在编译时**就将库编译进可执行程序中
　优点：程序的运行环境中不需要外部的函数库　　；　　缺点：可执行程序大

动态库，又称共享库，libxxx.so， **在运行时**将库加载到可执行程序中
　优点：可执行程序小　　；　　缺点：程序运行环境中必须提供相应的库

函数库目录：`/lib /usr/lib`
查看命令：在`/usr/lib`目录下输入`ls -l *.a` / `ls -l *.so`

4.1 静态库制作

流程图



静态库生成过程

```c
//example
/*同一目录下操作，目录下有文件main.c sub.c*/

/*main.c*/
#include "stdio.h"
int sub(int a, int b); //声明函数，不然会显示warning：implicit declaration of function ‘sub’[-Wimplicit-function-declaration]
int main()
{
    int ret;
    int x, y;
    x = 10;
    y = 5;
    ret = sub(x, y);    //x - y 注意这里调用sub函数
    printf("ret = %d\n", ret);
    return 0;
}
/*此时编译main.c 显示undefined reference to 'sub'*/

/*sub.c*/
int sub(int x, int y)
{
    return (x - y);
}

/*静态库制作过程，同一个目录路径下*/
//第一步，输入命令：gcc -c -o sub.o sub.c     此时，将.c文件生成目标文件，目录下生成目标文件sub.o
//第二步，输入命令：ar -cr -o libsub.a sub.o 此时，将目标文件生成静态库，名称位libsub.a，目录下生成libsub.a文件
//第三步，输入命令：gcc main.c -L. -lsub      此时，正确编译，输出文件a.out
//第四步，输入命令：./a.out                   即运行该文件，输出结果，ret = 5
```

生成目标文件：`gcc -c file.c`

静态函数库创建命令ar

`ar -cr libfile.a file.o`

`-c`：create的意思

`-r`：replace的意思，表示当插入的模块file.o已经存在`libfile.a`中，则覆盖。反之ar显示一个错误消息

操作静态库的实例：

情况1：如果从别处得到一个静态库`libunknown.a`，想知道其中包含哪些模块

命令：`ar -t libunknown.a`

静态库的编译：`gcc -o main main.c -L. -lfile` 编译 `main.c` 就会把静态函数库整合进 `main`

其中：`-L`：指定静态函数库的位置供查找，注意 `L` 后面还有 `.`， 表示静态函数库在本目录下查找

`-l`：指定了静态函数库名，由于静态函数库的命名方式是 `lib***.a`，其中的lib和.a忽略

4.2 动态库制作

```
//实例
/*同一目录下操作，目录下有文件main.c sub.c*/
//第一步，输入命令：gcc -c -o sub.o sub.c          此时，将.c文件生成目标文件，目录下生成目标文件sub.o
//第二步，输入命令：gcc -shared -fpic -o libsub.so sub.o
//                                                 此时，将目标文件生成动态库，名称位libsub.so，目录下生成libsub.so文件
//第三步，输入命令：gcc -o main main.c -L. -lsub   此时，正确编译，目录下生成可执行文件main
//第四步，输入命令：./main                         即运行该文件，但输出结果无法运行 no such file or directory
//原因：linux运行时，默认在路径为`/lib`或`/usr/lib`中寻找库，但实际上运行所需要的库不在该目录下
```

生成目标文件：`gcc -c file.c`

`gcc -shared -fpic -o libfile.so file.o` 生成名称为 `libaddsub.so` 的动态函数库 `file.o` 为目标文件

`-fpic`：产生位置无关代码

`-shard`：生成共享库

`gcc -o out main.c -L. -lfile`

此时还不能立即运行输出文件(./out)在动态函数库使用时，会查找 `/usr/lib` `/lib` 目录下的动态函数库，而此时生成的库不在里面

处理方法：

- `libaddsub.so` 放到 `/usr/lib` 或 `/lib` 中去 即拷贝转移 `mv libsub.so /usr/lib`
- 假设 `libfile.so` 在 `/home/linux/file` 利用环境变量找到库所在路径 `export LD_LIBRARY_PATH=/home/linux/addsub:$LD_LIBRARY_PATH`
- 在 `/etc/ld.so.conf` 文件中加入生成的库的目录，然后 `/sbin/ldconfig` 运行修改的脚本 `ldconfig /etc/ld.so.conf`

  `/etc/ld.so.conf` 为非常重要的目录，里面存放的是链接器和加载器搜索共享库时要检查的目录，默认从 `/usr/lib /lib` 中读取的，因此，可将库的目录加入到该文件夹中即可顺利运行

五、目录IO

对目录的IO操作（读写操作）
文件IO和标准IO（对文件的读写操作）

目录IO与文件IO函数的比较

目录IO	文件IO
opendir 只能打开目录，mkdir创建目录	open
readdir 读目录	read
rewinddir 调整位置指针 telldir seekdir	rewind ftell fseek
closedir 关闭目录	close

5.1 opendir、mkdir、closedir、readdir

5.1.1 opendir

`DIR *opendir(const char *pathname)` 参数：打开目录以及路径；该函数不能创建目录
返回值：成功返回目录流指针，出错返回NULL

5.1.2 mkdir

`int mkdir(const char *path, mode_t mode)` path：为欲创建的目录文件的路径；mode：为该目录的访问权限；
返回值：若目录创建成功，则返回0；否则返回-1 生成的目录权限和*umsak*有关系

```c
//example mkdir
#include "stdion.h"
#include "sys/types.h"
#include "dirent.h"
int main()
{
    int ret;
    ret = mkdir("./mydir", 0777);//创建目录的权限和umask有关系
    if(ret < 0)
    {
        printf("creat mydir failure\n");
        return -1;
    }
    printf("creat mydir sucess\n");
    return 0;
}
/* gcc -o mymkdir mymkdir.c
 * ./mymkdir mydir
 */

//opendir example
/* open返回值文件描述符，fopen返回文件流指针，作用：用来区别进程里面多个不同的文件
 * opendir返回值为目录流
*/
#include "stdion.h"
#include "sys/types.h"
#include "dirent.h"
int main()
{
    int ret;
    DIR *dp;        //定义目录流指针
    ret = mkdir("./mydir", 0777);//创建目录的权限和umask有关系
    if(ret < 0)
```

```
    {
        printf("creat mydir failure\n");
        return -1;
    }
    printf("creat mydir sucess\n");
    dp = opendir("./mydir");    //打开当前目录下的dir
    if(dp == NULL)
    {
        printf("open mydir failure\n");
        return -2;
    }
    printf("open mydir sucess\n");
    closedir(dp);
    return 0;
}
```

5.1.3 readdir

`struct dirent *readdir(DIR *dr)` 参数：目录流指针

返回值：成功返回为`struct dirent`指针，若在目录尾或出错则返回NULL；`struct dirent`定义在头文件`dirent.h`中

```
struct dirent{      //结构体内容
    ino_t d_ino;    //inode号 Linux底下的文件唯一标识符，文件不同inode号不同 类型为长整型
    char d_name[NAME_MAX + 1];  //文件名
}
//readdir example
#include "stdion.h"
#include "sys/types.h"
#include "dirent.h"
int main(int argc, char *argv[])
{
    int ret;
    DIR *dp;        //定义目录流指针
    struct dirent *dir;
    ret = mkdir("./mydir", 0777);//创建目录的权限和umask有关系
    if(argc < 0)
    {
        printf("please input open directory name\n");
        return -1;
    }
    dp = opendir(argv[1]);  //打开当前目录下的dir
    if(dp == NULL)
    {
        printf("open mydir failure\n");
        return -2;
    }
    printf("open mydir sucess\n");
    while(1)
    {
        dir = readdir(dp);  //参数为读哪个目录
        if(dir != NULL) //当返回值为NULL节点已经遍历完成最后指向空
        {
            printf("inode = %ld, name = %s\n", dir->d_ino, dir->d_name);
        }
```

```
        else
            break;
    }

    closedir(dp);
    return 0;
}
/* 目录文件的内容包括子文件和子目录信息，这些信息的存放方式是通过/链表/的形式来存放，每个链表都有一个节点，一个节点就只包含一
 * 个子文件或者子目录，有多个子文件或子目录，那么节点也对应增加，每次读完一个节点，指针久指向下一个，所以，需要/遍历节点/
 *
*/
```

5.1.4 rewinddir

`void rewinddir(DIR *dr)` 重置读取目录的位置为开头(指针的位置)　参数：目录流指针

`long telldir(DIR *dirp)` 参数：目录流指针　返回值：目录流当前位置（指针位置）

`void seekdir(DIR *dirp, long loc)` 类似文件定位函数fseek()，在目录流上设置下一个readdir()操作的位置 参数：目录流指针和偏移量

```
//readdir example
#include "stdion.h"
#include "sys/types.h"
#include "dirent.h"
int main(int argc, char *argv[])
{
    int ret;
    DIR *dp;        //定义目录流指针
    struct dirent *dir;
    ret = mkdir("./mydir", 0777);//创建目录的权限和umask有关系
    if(argc < 0)
    {
        printf("please input open directory name\n");
        return -1;
    }
    dp = opendir(argv[1]);  //打开当前目录下的dir
    if(dp == NULL)
    {
        printf("open mydir failure\n");
        return -2;
    }
    printf("open mydir sucess\n");
    while(1)
    {
        dir = readdir(dp);  //参数为读哪个目录
        /*seekdir(dp, 0);  等价于 rewinddir(dp);*/
        rewinddir(dp);      //调整目录流指针为开头，那么程序就一直打印第一个子目录对应的节点信息，进入死循环
        if(dir != NULL) //当返回值为NULL节点已经遍历完成最后指向空
        {
            printf("inode = %ld, name = %s\n", dir->d_ino, dir->d_name);
        }
        else
            break;
```

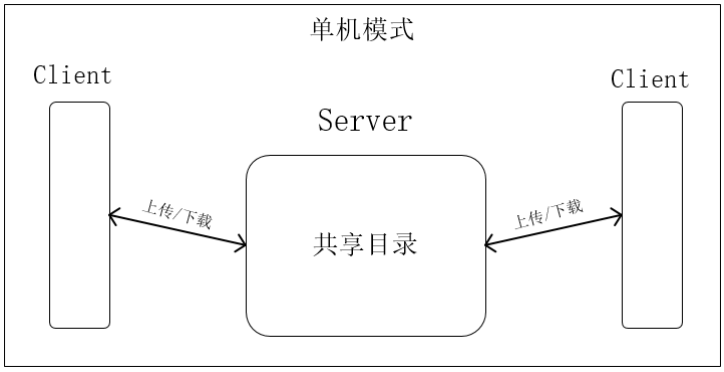
```
    }

    closedir(dp);
    return 0;
}

///telldir
#include "stdion.h"
#include "sys/types.h"
#include "dirent.h"
int main(int argc, char *argv[])
{
    int ret;
    DIR *dp;         //定义目录流指针
    long loc;        //用于返回位置
    struct dirent *dir;
    ret = mkdir("./mydir", 0777);//创建目录的权限和umask有关系
    if(argc < 0)
    {
        printf("please input open directory name\n");
        return -1;
    }
    dp = opendir(argv[1]);  //打开当前目录下的dir
    if(dp == NULL)
    {
        printf("open mydir failure\n");
        return -2;
    }
    printf("open mydir sucess\n");
    while(1)
    {
        dir = readdir(dp);  //参数为读哪个目录
        loc = telldir(dp);  //获取当前目录流指针的位置
        printf("loc = %ld\n", loc);      //打印当前目录流指针的位置 注意位置不是连续的过程（链表）
        if(dir != NULL) //当返回值为NULL节点已经遍历完成最后指向空
        {
            printf("inode = %ld, name = %s\n", dir->d_ino, dir->d_name);
        }
        else
            break;
    }
    closedir(dp);
    return 0;
}
```

5.1.5 closedir

`int close(DIR *dr)` 参数：目录流指针； 返回值：成功返回0，出错返回-1

六、总结实例

单机模式下的文件上传送和下载



1. 输入服务器的地址：路径和目录名
2. 列出服务器中有哪些文件：opendir readdir
3. 输入从服务器下载的文件名 或 上传文件到服务器的文件名
4. 文件下载　　或 文件上传

    文件IO：open、read、write、close

    标准IO：fopen、fputs、fgets　fputc、fgetc、fread、fwrite、fclose

```c
#include "stdion.h"
#include "sys/types.h"
#include "dirent.h"
#include "string.h"
#include "unistd.h"
#include "fcntl.h"
int main()
{
    DIR *dp;
    int src_fd, des_fd;     //源文件 目的文件
    int fd, ret;            //ret用于返回读了多少个
    struct dirent *dir;
    char server[128] = {0};
    char file[128] = {0};   //缓存文件名接收
    char buf[128] = {0};    //读写文件内容的缓存
start:              //start标号
    printf("please input server PATH and Directory name\n");
    scanf("%s", server);    //键盘输入服务器的地址，存放到变量server中
    //second list server files
    if(dp == NULL)
    {
        printf("open server: %s failure\n", server);
        goto start; //跳转到start标号处运行
    }
    printf("open server:%s sucess\n", server);
    while(1)
    {
        dir = readdir(dp);
        if(dir == NULL)
            break;
        else
        {
            printf("inode = %ld\t file_name = %s\n",  dir->d_ino, dir->d_name);
        }
    }
    //three
    printf("\n please input down file\n");
    scanf("%s", file);  //获取要下载的文件名
```

```
39
40      //four
41      src_fd = open(strcat(strcat(server, "/"), file), O_RDONLY);//打开文件的名
    称 路径
42      if(src_fd < 0)
43      {
44          printf("open download file:%s\n", file);
45          return -1;
46      }
47      printf("open download file: %s\n", file);
48      des_fd = open(file, O_CREAT | O_WRONLY, 0777);
49      if(des_fd < 0)
50      {
51          printf("creat file:%s failure\n", file);
52      }
53      printf("creat file:%s sucess\n", file);
54
55      while(1)
56      {
57          ret = read(src_fd, buf, 128);
58          if(ret < 128)
59          {
60              break;
61          }
62          write(des_fd, buf, ret);    //写到哪里去，写什么，写多少个
63      }
64      write(des_fd, buf, ret);//跳出循环后再写一次 完成文件的拷贝
65      close(src_fd);
66      close(des_fd);
67      closedir(dp);
68      return 0;
69  }
70  /* 文件下载过程：服务器打开一个文件（源文件） -> 客户端新建一个文件（目标文件） -> 从源文
    件读到内容然后写到目标文件
71   *
72   * 命令:
73   * gcc client.c -> ./a.out -> <输入目录>
74  */
```

随笔

内核要为应用程序服务，应用程序如果没有内核服务，则应用程序功能非常单一
内核是一个稳定的代码，同时要为多个用户空间的程序服务，防止用户空间的某些用户程序使内核代码崩溃或其他问题

内核向用户空间提供的接口（函数），在这些接口函数中加上一些保护，这样会使符号接口函数的应用提供服务，同时保护内核

驱动就是为了识别设备，然后控制设备