IO 进程线程

第一天：IO概念，标准IO,

第二天：标准IO，文件IO

第三天：文件IO，库

第四天：进程

第五天：线程

第六天：线程的同步互斥

第七、八天：进程间的通信

如何学：记函数的功能以及函数的名字，将白天的练习还有代码，晚上敲两遍

1. IO

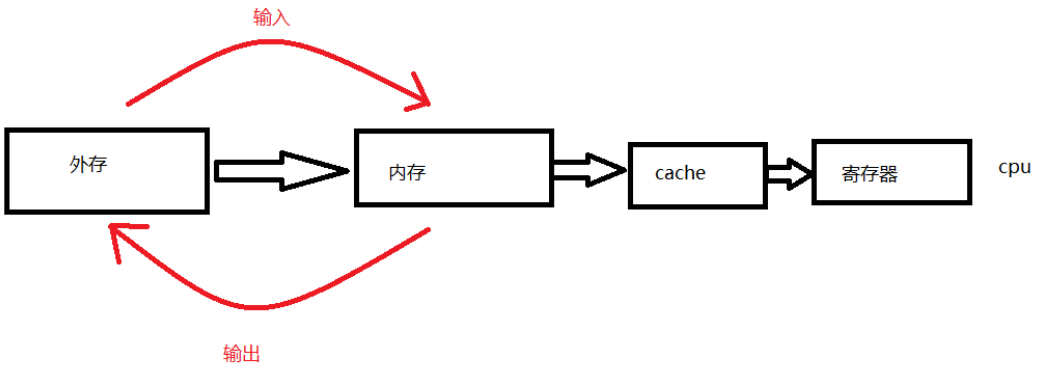
【1】什么是IO

i：input　输入　　从外存设备输入到内存中

o：output 输出　　从内存输出到外存设备中

外部存储设备：硬盘、磁盘

内存：DDR4



总结：IO就是数据从硬盘到内存，内存到硬盘的流动。

【2】IO函数分类

1. 文件IO

文件IO是由操作系统提供的基本IO函数，与操作系统绑定，又称之为系统调用.

例子：

windows	Linux
file_read	read

注意：

1. 文件IO是与操作系统绑定的，所以不同的操作系统有不同的文件IO函数。所以文件IO的移植性更低
2. 文件IO涉及到用户空间到内核空间的切换，cpu模式的切换，C代码调用汇编指令等等操作，是一种**耗时操作**。

2. 标准IO

   根据ANSI标准，是对文件IO的二次封装，printf scanf

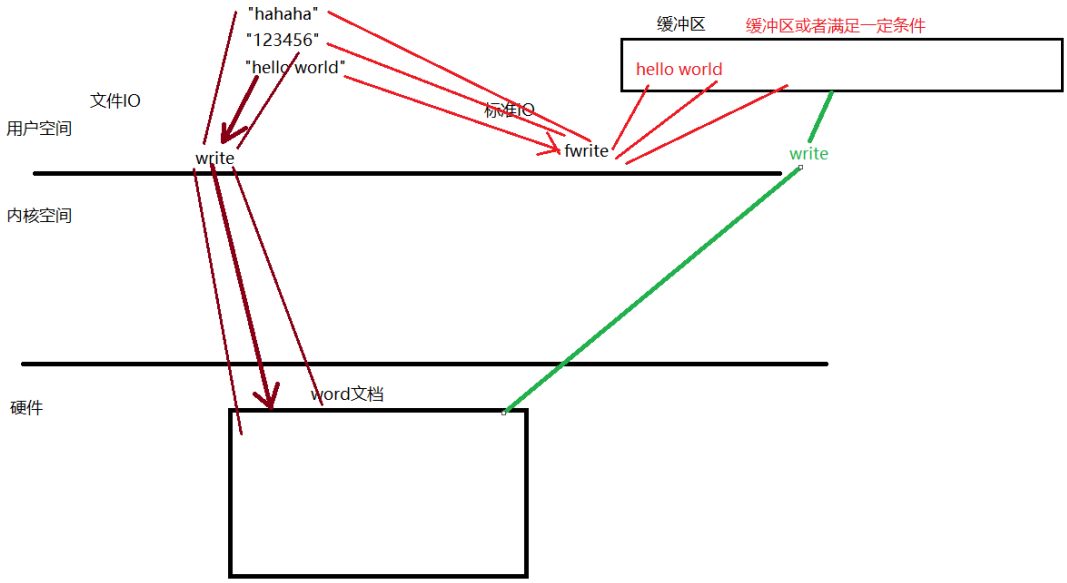
   标准IO是对文件IO的二次封装，所以最终标准IO还是会去调用文件IO

```
scanf:

if(OS == windows)
{
    file_read();    //windows的文件IO
}
else if(OS == Linux)
{
    read();         //Linux的文件IO
}
```

注意：

1. 标准IO的移植性更高。
2. 提高输入输出的效率；

   设置了一个缓冲区，缓冲区满或者满足一定条件后，调用文件IO，陷入内核空间，由内核完成对硬件的操作，大大减少了对文件IO的调用。



2. 标准IO

【1】流和流指针

流（stream）:将数据一个一个的移入缓冲区，或者移出缓冲区的形式叫做字节流。

流指针（FILE*）：每打开一个文件，都会在用户空间中申请一片空间（缓冲区）。

管理这片空间的变量都存储在FILE结构体中，该结构体由系统定义好的，我们直接使用即可。

1. 查看FILE结构体

> 1）sudo apt-get install ctags
>
> 2）cd /usr/include/
>
> 3）sudo ctags -R
>
> 4）在家目录下有个隐藏文件：vim .vimrc 打开隐藏文件
>
> 在结尾或者开头添加 set tags+=/usr/include/tags
>
> 追代码：
>
> 左键选中要追的代码
>
> ctrl +]
>
> ctrl + 鼠标左键
>
> 返回：
>
> ctrl + t
>
> ctrl + 鼠标右键

```
vi -t FILE          查看系统定义好的数据类型，宏
 typedef struct _IO_FILE FILE;

鼠标移动到_IO_FILE，按下ctrl + ]
```

```
struct _IO_FILE {
  char* _IO_buf_base;    /* Start of reserve area. */ 缓冲区的起始地址
  char* _IO_buf_end;     /* End of reserve area. */    缓冲区结束地址

  int _fileno;           文件描述符，在文件IO的时候讲解.
}

int* ptr = 0x10;
ptr+1 == 0x14

short int* ptr = 0x10;
ptr+1 == 0x12;
```

```
int* ptr = 0x14;
ptr-1 == 0x10
```

2. man手册 -->函数原型

查看：函数的功能参数返回值

```
       1   可执行程序或 shell 命令
       2   系统调用(内核提供的函数)     文件IO函数
       3   库调用(程序库中的函数)       库函数，标准IO函数

man 3 printf
```

3 .特殊的流指针

在main函数运行之前，系统会自动帮我们打开三个流指针

```
FILE* stdin   标准输入流指针    从终端文件读取数据时候使用
FILE* stdout  标准输出流指针    将数据打印到终端文件时候使用的流指针
FILE* stderr  标准错误输出流指针
```

【2】标准IO函数

```
fopen   /  fclose
fprintf /  fscanf
fputc   /  fgetc
fputs   /  fgets
fwrite  /  fread
fseek
```

1）fopen

```
功能：打开一个文件：
头文件：
       #include <stdio.h>

       FILE *fopen(const char *pathname, const char *mode);
参数：
    char *pathname：需要打开的文件的路径以及文件名；
    char *mode：打开方式
       r       以读的方式打开文件，
                 如果文件不存在，则函数运行失败；
```

```
                如果文件存在，则打开成功；

        r+      以读写的方式打开文件，
                如果文件不存在，则函数运行失败；
                如果文件存在，则打开成功

        w       以写的方式打开文件，
                如果文件不存在，则创建文件；
                如果文件存在，则清空文件；

        w+      以读写的方式打开文件，
                如果文件不存在，则创建文件；
                如果文件存在，则清空文件；

        a       以追加的方式写；
                如果文件不存在，则创建文件；
                如果文件存在，则以追加的方式写入；

        a+      以读以及追加写的方式打开文件；
                如果文件不存在，则创建文件；
                如果文件存在，则以追加的方式写入;或者从文件开头读取；

返回值：
    成功，返回流指针；
    失败，返回NULL，同时更新错误码errno；

 不同的错误，代码会设置不同的错误编号errno是不同的；
```

2）perror

```
功能：通过errno打印对应的错误信息；
头文件：
        #include <stdio.h>

        void perror(const char *s);
参数：
    char *s：用于提示的字符串；

        #include <errno.h>
        int errno;       /* Not really declared this way; see errno(3) */

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

3）fclose

功能：关闭文件；
头文件：

```
#include <stdio.h>

int fclose(FILE *stream);
```

参数：
FILE *stream：指定要关闭的文件的流指针；
返回值：
成功，返回0；
失败，返回EOF(其实就是-1)，更新errno；

4）fprintf

功能：将数据格式化输出到文件中；
头文件：

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

参数：
FILE *stream：指定要输出到哪个文件中，则填对应的流指针.
const char *format：标准格式化；
...:不定参数，不定数据类型不定数据个数；
返回值：
成功，返回被打印的字节个数，其实就是大于0；
失败，返回负数，其实就是小于0；

将数据以字符形式放入文件，如数字64会变成字符6和4

5）fscanf

功能：从指定文件中格式化输入数据到内存中；
头文件：

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

参数：
FILE *stream：指定要从哪个文件中读取数据，则填对应的流指针.
const char *format：标准格式化；
...:不定参数，不定数据类型不定数据个数；
返回值：
成功，返回成功读取到的数据个数，注意不是字节数；
失败，或者读取到文件结尾，返回EOF，同时更新errno；

练习

> 自行写一个usr.txt文件，文件中每一行都存储一个用户名和密码，中间用空格隔开，要求：
>
> 1. 从终端获取用户名和密码
> 2. 与文件存储的用户名密码比较
> 3. 如果用户名不存在，则输出用户不存在

> 4. 如果密码错误，则输出密码错误
> 5. 如果均正确，则输出登录成功

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

int main(int argc, const char *argv[])
{
    FILE* fp = fopen("./usr.txt", "r");
    if(NULL == fp)
    {
        perror("fopen");
        return -1;
    }

    char name[10] = "";
    char passwd[10] = "";
    printf("请输入用户名和密码>>>");
    scanf("%s %s", name, passwd);

    char f_name[10] = "";
    char f_passwd[10] = "";
    int ret = 0;

    while(1)
    {
        ret =fscanf(fp, "%s %s", f_name, f_passwd);
        if(ret < 0)
        {
            printf("用户名不存在\n");
            break;
        }
        //  printf("f_name = %s f_passwd = %s\n", f_name, f_passwd);

        //比较用户名
        if(strcmp(name, f_name) != 0)
            continue;       //如果用户名不相等，则直接读取下一行，不需要比较密码

        //能运行到当前位置，则说明用户名相同
        //则需要去比较密码
        if(strcmp(passwd, f_passwd) == 0)
        {
            printf("登录成功\n");
        }
        else
        {
            printf("密码输入错误\n");
        }

        break;
    }

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

6）fputc

功能：向指定文件中输出单个字符；
头文件：

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putchar(int c);  putchar('a'); putchar(10); putchar(97);
```

参数：
int c：指定要输出的字符对应的字符形式，或者整型，例如‘a’,或者97
FILE *stream：指定要输出到哪个文件中，填对应的流指针；
返回值：
成功，返回成功输出的字符对应的整型形式；
失败，返回EOF(-1);

7）fgetc

功能：从文件中获取单个字符；
头文件：

```
#include <stdio.h>

int fgetc(FILE *stream);
int getchar(void);       char c = getchar();
```

参数：
FILE *stream：指定要从哪个文件中读取数据，则填对应的流指针.
返回值：
成功，返回成功读取到的字符对应的整型；
失败或者读取到文件结尾，返回EOF;

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    //打开一个文件，以读的方式打开文件
    FILE* fp = fopen("./01_fopen.c", "r");
    if(NULL == fp)
    {
        perror("fopen");
        return -1;
    }

    char c ;
    while(1)
    {
        c = fgetc(fp);
        if(EOF == c)
            break;

        printf("%c", c);
    }

    fclose(fp);

    return 0;
```

```
}
```

练习

> 1. 要求用fputc和fgetc拷贝一个文件，例如将01.c的内容拷贝到02.c中

```
#include <stdio.h>

#define ERR_MSG(msg)  do{\
    printf("line:%d\n", __LINE__); \
    perror(msg);\
}while(0)


int main(int argc, const char *argv[])
{
    if(argc < 2)
    {
        printf("请外部传参输入要拷贝的文件名\n");
        return -1;
    }

    //打开一个文件，以读的方式打开文件
    FILE* fp = fopen(argv[1], "r");
    if(NULL == fp)
    {
        //printf("%d %s %s\n", __LINE__, __func__, __FILE__);
        ERR_MSG("fopen");
        return -1;
    }

    //以写的方式打开目标文件
    FILE* fp_w = fopen("./copy.c", "w");
    if(NULL == fp_w)
    {
        ERR_MSG("fopen");
        return -1;
    }

    char c ;

    while((c=fgetc(fp)) != EOF)
    {
        fputc(c, fp_w);
    }
    printf("拷贝完毕\n");

    fclose(fp);

    return 0;
}
```

> Linux操作系统，用编辑器打开文件保存退出后，会自动补上一个'\n'
>
> windows操作系统，用编辑器打开文件保存退出后，会自动补上一个'\r'
>
> unix操作系统，用编辑器打开文件保存退出后，会自动补上一个'\r\n'

作业

> 1. 用fgetc实现，计算一个文件有几行，要求封装成函数，用命令行传参，

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

int get_fileLine(FILE* fp)
{
    int c = 0, line = 0;
    while( (c = fgetc(fp)) != EOF)
    {
        if('\n' == c)
            line++;
    }

    return line;
}


int main(int argc, const char *argv[])
{
    if(argc < 2)
    {
        fprintf(stderr, "请在命令行输入文件路径以及文件名\n");
        return 0;
    }

    FILE* fp = fopen(argv[1], "r");
    if(NULL == fp)
    {
        perror("fopen");
        return -1;
    }

    int line = get_fileLine(fp);
    printf("line = %d\n", line);
    fclose(fp);
    return 0;
}
```

8）缓冲区

注意只有标准IO才有缓冲区，文件IO是没有的。

i.全缓冲

操作对象

对普通文件进行从操作：用fopen函数手动打开的文件，创建的缓冲区全部都是全缓冲。

大小：

4096byte = 4k

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    FILE* fp = fopen("fullBuf.txt", "w");
    if(NULL == fp)
    {
        perror("fopen");
        return -1;
    }

    fputc('a', fp);
    //由于操作系统优化，只申请不操作的话不会真正的把缓冲区申请出来。
    printf("%ld\n", fp->_IO_buf_end - fp->_IO_buf_base);

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

缓冲区的刷新机制

1. 缓冲区满
2. 用fflush函数强制刷新

```
        #include <stdio.h>

        int fflush(FILE *stream);

fflush(fp);
```

3. 调用fclose关闭流指针
4. 主函数调用return退出程序
5. 调用exit函数退出程序

```
功能：退出进程；
        #include <stdlib.h>

        void exit(int status);
参数：
    int status：进程退出状态值，目前随便填一个整型数即可；

exit(0);
```

ps：

1. 读写转换。

ii.行缓冲

操作对象：

标准输入流指针（stdin） 标准输出流指针(stdout)

大小：

1024byte = 1K

```c
#include <stdio.h>

int main(int argc, const char *argv[])
{
    //fprintf(stdout, "hello world\n");
    printf("hello world\n");
    printf("%ld\n", stdout->_IO_buf_end - stdout->_IO_buf_base);
    return 0;
}
```

缓冲区的刷新机制

1. 缓冲区满
2. 用fflush函数强制刷新

```
        #include <stdio.h>

        int fflush(FILE *stream);

fflush(stdout);
```

3. 调用fclose关闭流指针
4. 主函数调用return退出程序
5. 调用exit函数退出程序

```
功能：退出进程；
        #include <stdlib.h>

        void exit(int status);
参数：
    int status：进程退出状态值，目前随便填一个整型数即可；

exit(0);
```

6. 遇到'\n'字符刷新缓冲区;

ps:

1. 读写转换。

iii.无缓冲

操作对象:

标准错误输出流指针(stderr)

大小：0

9）fputs

```
功能：将字符串输出到指定的文件中；
头文件：
        #include <stdio.h>

        int fputs(const char *s, FILE *stream);
参数：
    char *s：指定要输出的字符串首地址；
    FILE *stream：；
返回值：
    成功，返回非负数；
    失败，返回EOF；
```

10）fgets

```
功能：从指定的文件中获取字符串；
头文件：
        #include <stdio.h>

        char *fgets(char *s, int size, FILE *stream);
参数：
    char *s：该指针指向的内存空间会存储获取到的字符串；
    int size：指定要获取多少个字节的数据;最多会读取size-1个字节;
                期间遇到新的一行或者文件结尾会停止读取;在有效字符串的结尾会自动补上'\0';
    FILE *stream:;
返回值：
    成功，返回字符串首地址，其实就是s参数；
    失败或者文件读取完毕，返回NULL；
```

```
 char str[4] = "";
 fgets(str, 10, stdin);          //终端输入 abcdefgh\n
A. abcd后面乱码  B.abc   C.段错误  D.abcdefgh\n
```

作业

1. 用fgets和fputs实现文件的拷贝
2. 用fgets实现计算一个文件有几行。

```
int get_fileLine(FILE* fp)
{
    char str[100] = "";
    int line = 0;
    while(fgets(str, sizeof(str), fp) != NULL)
    {
        //由于fgets遇到'\n'字符会停止读取，所以'\n'字符肯定在'\0'字符之前
        //所以只需要判断'\0'字符之前是否是'\n'即可，如果是，则算一行
        if(str[strlen(str)-1] == '\n')
            line++;

    }
    return line;
}
```

11）fwrite

功能：会将数据转换成二进制数据写入到文件中。以二进制存入文件，如数字2有四个字节，会把他拆成4个一字节存入，所以会乱码
其实就是将数据拆分成一个一个的字节，并将该字节对应的字符形式写入到文件中；
头文件：

```
#include <stdio.h>

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

参数：
void *ptr：指定要写入到文件中的数据首地址,参数是void*类型，所以该函数可以写入任意类型的数据到文件中；
size_t size：每个数据的大小，字节为单位；
假设每个数据是int类型，则size==4；
假设每个数据是char类型，则size==1；
假设每个数据是自定义结构体类型，则size==sizeof(结构体类型)；
size_t nmemb：指定要输出的数据个数；
FILE *stream：指定要输出到哪个文件中；
返回值：
成功，返回成功输出的数据个数，其实就是nmemb参数中的内容；
失败，返回小于nmemb或者等于0；

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    FILE* fp = fopen("fwrite.txt", "w");

    if(NULL == fp)
```

```
    {
        perror("fopen");
        return -1;
    }

    int a[3] = {48, 49, 50};
    size_t res = 0;

    //以数组中的每个元素为单位，每个元素都是整形，大小为4，个数3个
    res = fwrite(a, 4, 3, fp);
    printf("res=%ld\n", res);

    char c = 10;
    fwrite(&c, 1, 1,fp);

    //将整个数组作为整体，大小为12即sizeof(a)，个数是1个
    res = fwrite(a, sizeof(a), 1, fp);
    printf("res=%ld\n", res);

    fclose(fp);

    return 0;
}
```

12）fread

功能：从文件中读取二进制数据，ps:以fwrite形式写入，就以fread形式读取；
头文件：

```
#include <stdio.h>

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

参数：

```
void *ptr：该指针指向的内存空间中会存储读取到的数据；
size_t size：每个数据的大小，字节为单位；
            假设每个数据是int类型，则size==4；
            假设每个数据是char类型，则size==1；
            假设每个数据是自定义结构体类型，则size==sizeof(结构体类型)；
size_t nmemb：指定要获取的数据个数；
FILE *stream：指定要从哪个文件中获取数据；
```

返回值：

```
成功，返回成功读取到的数据个数，其实就是nmemb参数中的内容；
失败或者读取到文件结尾，返回小于nmemb或者等于0；
```

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

typedef struct
{
    int a;
    char b;
} Node;          //Node是一个数据类型，是自定义的结构体类型
```

```c
int main(int argc, const char *argv[])
{
    FILE* fp = fopen("fwrite.txt", "r");
    if(NULL == fp)
    {
        perror("fopen");
        return -1;
    }
    size_t res = 0;
    Node n;
    while(1)
    {
        memset(&n, 0, sizeof(n));
        res = fread(&n, sizeof(Node), 1, fp);
        printf("res=%ld a=%d b=%c\n", res, n.a, n.b);
    }

    fclose(fp);

    return 0;
}
```

13）fseek

```
功能：修改文件偏移量；
头文件：
        #include <stdio.h>

        int fseek(FILE *stream, long offset, int whence);
参数：
    FILE *stream：指定要修改哪个文件的偏移量，填入对应的流指针；
    long offset：距离第三个形参whence的偏移量；（往后偏用正数，往前偏用负数）
    int whence：
        SEEK_SET    文件开头
        SEEK_CUR    文件当前位置
        SEEK_END    文件结尾；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；


功能：修改文件偏移量到文件开头位置 void rewind(FILE *stream); 相当于fseek(fp, 0 ,
SEEK_SET);
```

注意：

1. 当文件偏移量在文件开头位置时，不允许继续往前偏移了；
2. 当文件偏移量在文件结尾位置时，允许继续往后偏移;且偏移后往文件中写入数据，则文件会自动将偏移部分补上0（^@）

14）ftell

```
功能：获取文件当前位置距离文件开头的偏移量；
头文件：
        #include <stdio.h>

        long ftell(FILE *stream);
参数：
    FILE *stream：指定要获取哪个文件的偏移量，填入对应的流指针；
返回值：
    成功，返回文件当前位置距离文件开头的偏移量；
    失败，返回-1，更新errno；


计算文件大小：
   //修改文件偏移量到文件结尾
   fseek(fp, 0, SEEK_END);
   long off = ftell(fp);
   printf("%ld\n", off);
```

【3】时间相关的函数

1）time

```
功能：获取1970-1-1至今的秒数；
头文件：
        #include <time.h>

        time_t time(time_t *tloc);
参数：
    time_t *tloc：该指针指向的内存空间中，会存储秒数；
                    填NULL；
返回值：
    1970-1-1至今的秒数
```

2）localtime

```
功能：将1970-01-01至今的秒数转换成日历格式；
头文件：
        #include <time.h>

        struct tm *localtime(const time_t *timep);
参数：
    time_t *timep：指定要转换成日历格式的秒数对应的首地址；
返回值：
    成功，返回存储日历格式的结构体指针；
    失败，返回NULL；

            struct tm {
                int tm_sec;     /* Seconds (0-60) */        秒
                int tm_min;     /* Minutes (0-59) */        分
```

```
                int tm_hour;   /* Hours (0-23) */        时
                int tm_mday;   /* Day of the month (1-31) */     日
                int tm_mon;    /* Month (0-11) */        月 = tm_mon+1
                int tm_year;   /* Year - 1900 */         年 = tm_year+1900
                int tm_wday;   /* Day of the week (0-6, Sunday = 0) */   星期
                int tm_yday;   /* Day in the year (0-365, 1 Jan = 0) */  一年的第几
天
                int tm_isdst;  /* Daylight saving time */
            };
```

```c
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, const char *argv[])
{
    /*
    time_t t = time(NULL);
    printf("%ld\n", t);
    */

    time_t t2;
    while(1)
    {
        system("clear");    //让C代码执行shell指令

        time(&t2);
    //  printf("%ld\n", t2);

        struct tm* info = NULL;
        info = localtime(&t2);

        printf("%d-%02d-%02d %02d-%02d-%02d\r", \
                info->tm_year+1900, info->tm_mon+1, info->tm_mday, \
                info->tm_hour, info->tm_min, info->tm_sec);
        fflush(stdout);

        sleep(1);
    }

    return 0;
}
```

作业

> 要求创建一个time.txt，存储内容格式如下：
>
> [1] 2022-07-28 17:15:06
>
> [2] 2022-07-28 17:15:07
>
> [3] 2022-07-28 17:15:08
>
> ctrl + c退出程序，过一会儿之后重新启动程序

> [1] 2022-07-28 17:15:06
>
> [2] 2022-07-28 17:15:07
>
> [3] 2022-07-28 17:15:08 <----------
>
> [4] 2022-07-28 17:16:31
>
> [5] 2022-07-28 17:16:32

```c
#include <stdio.h>
#include <time.h>
#include <unistd.h>

int get_fileLine(FILE* fp)
{
    int c = 0, line = 0;
    while( (c = fgetc(fp)) != EOF)
    {
        if('\n' == c)
            line++;
    }

    return line;
}


int main(int argc, const char *argv[])
{
    //以a+的方式打开文件
    FILE* fp = fopen("./time.txt", "a+");
    if(NULL == fp)
    {
        perror("fopen");
        return -1;
    }

    int line = get_fileLine(fp);
    printf("line = %d\n", line);

    time_t t;
    struct tm* info = NULL;

    while(1)
    {
        line++;

        t = time(NULL);
        info = localtime(&t);

        fprintf(fp, "[%02d] %d-%02d-%02d %02d-%02d-%02d\n", line,\
                info->tm_year+1900, info->tm_mon+1, info->tm_mday, \
                info->tm_hour, info->tm_min, info->tm_sec);
        fflush(fp);

        sleep(1);

    }
```

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

3. 文件IO

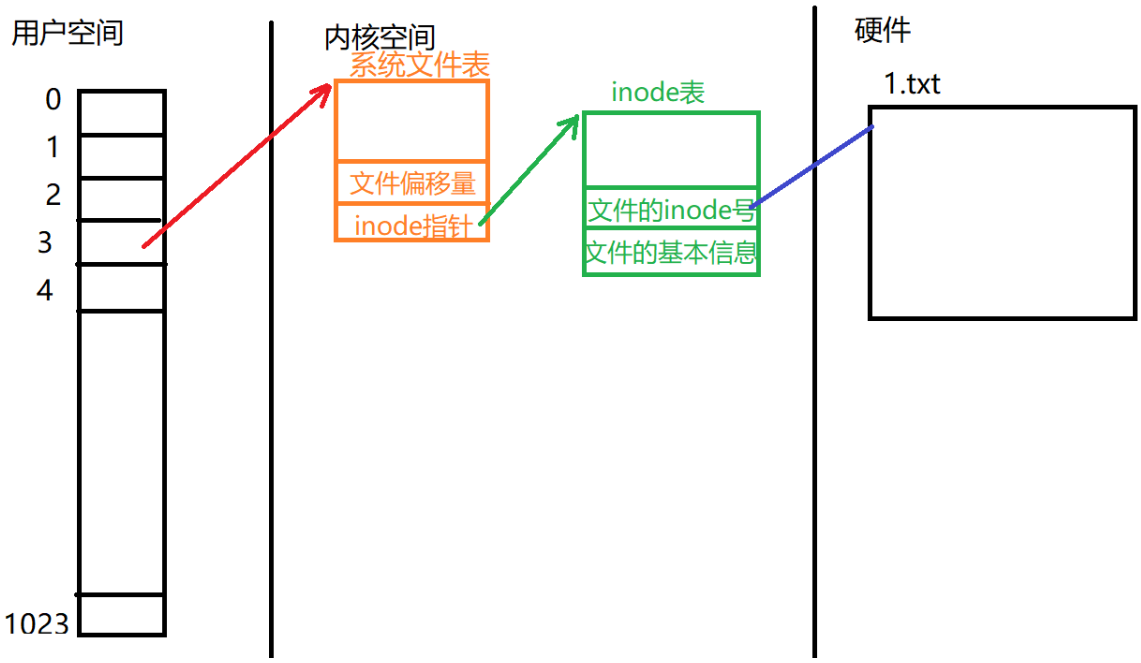
1. 只有标准IO才有缓冲区，文件IO没有缓冲区。
2. 文件IO与操作系统绑定，不同的操作系统，文件IO函数是不同的。
3. 标准IO是通过流指针维护一个文件，文件IO是通过**文件描述符**来维护一个文件的。

【1】文件描述符

1. 文件描述符的概念

1. 当尝试打开一个文件的时候，系统会自动给这个文件编上一个编号，这个编号就是文件描述符，通过文件描述符来描述这个文件。
2. 每个程序文件描述符的总量默认是1024个。

文件描述符的本质：数组下标，数组由系统申请，我们打开文件后最终会拿到数组下标，数组下标就是文件描述符;



2. 特殊的文件描述符

特殊的流指针	特殊的文件描述符	文件描述符对应的数值
FILE* stdin	stdin->_fileno	0
FILE* stdout		1
FILE* stderr		2

1. 注意文件描述符的总量默认是1024个，
2. 所以描述符总量是有限的，在不使用的情况下记得关闭;

【2】文件IO的函数

```
open / close
read / write
lseek
```

1）open

```
功能：打开一个文件；
头文件：
        #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);
参数：
    char *pathname：指定要打开的文件对应的路径及名字；
    int flags：
        O_RDONLY        只读方式打开
        O_WRONLY        只写方式打开
        O_RDWR          读写方式打开
     ---以上三种必须包含一种，且只能包含一种------；
        O_APPEND        追加写的方式打开
        O_CREAT         如果文件不存在，则创建一个普通文件
        O_TRUNC         如果文件存在，并且以写的方式打开，则会清空文件；

        如果需要组合多个flags则需要用按位或（|）连接
        "w"：O_WRONLY|O_CREAT|O_TRUNC
        "w+"：O_RDWR|O_CREAT|O_TRUNC;

    mode_t mode：当要创建文件的时候，需要指定文件的权限，可以填对应的八进制权限。例如0664
0775 0777;
                当flags中包含了O_CREAT或者O_TMPFILE的时候，必须填mode参数，
                如果flags中没有包含上述选项，则会忽略mode参数.

返回值：
    成功，返回新的文件描述符；
    失败，返回-1，更新errno;
```

> 请写出fopen的打开方式分别对应open打开方式的flags组合

fopen() mode	open() flags
r	O_RDONLY
w	O_WRONLY | O_CREAT | O_TRUNC
a	O_WRONLY | O_CREAT | O_APPEND
r+	O_RDWR
w+	O_RDWR | O_CREAT | O_TRUNC
a+	O_RDWR | O_CREAT | O_APPEND

2）umask

```
the mode of the created file is (mode & ~umask).
文件创建时候的真实权限是 (mode & ~umask)
```

i. 什么是umask

文件权限掩码，会影响文件创建时候的权限

ii.umask的值

```
终端输入 umask  --》得到结果为0002
```

```
mode: 0777 ---> 111 111 111
umask:0002 ---> 000 000 010

mode & ~umask  = 111 111 111 & ~000 000 010
               = 111 111 111 & 111 111 101
               = 111 111 101 --> 0775

当umask设置为0002的时候，其他用户肯定没有写权限.
```

iii. 设置umask的值

1. 直接在终端输入 umask 0。但是只在设置终端有效
2. 用umask函数设置

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

        mode_t umask(mode_t mask);
umask(0);
```

3）close

```
功能：关闭文件；
头文件：
        #include <unistd.h>

        int close(int fd);
参数：
    int fd：指定要关闭的文件对应的文件描述符；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

4）write

```
功能：将数据输出到文件中；
头文件：
        #include <unistd.h>

        ssize_t write(int fd, const void *buf, size_t count);
参数：
    int fd：指定要写到哪个文件中，填对应的文件描述符；
    void *buf：该指针指向的内存空间中会存储要输出的数据首地址，可以输出任意类型数据；
    size_t count：指定要输出多少个字节；注意填多少字节，就会写入多少个字节，所以要注意不要越界写入
返回值：
    成功，返回成功输出的数据字节数；
    失败，返回-1，更新errno；
```

5）read

```
功能：从指定文件中读取数据；
头文件：
        #include <unistd.h>

        ssize_t read(int fd, void *buf, size_t count);
参数：
    int fd：指定从哪个文件中读取，填对应的文件描述符；
    void *buf：该指针指向的内存空间中会存储读取到的数据，可以读取任意类型数据；
    size_t count：指定要读取多少个字节；
返回值：
    >0，成功，返回成功读取到的数据字节数；
    =0，读取到文件结尾；
```

```
    =-1，失败，返回-1，更新errno;


注意：
    1.read函数遇到'\n'字符等特殊字符是不会停止读取的.
    2.read函数不会自动补上'\0'
```

作业

> 1.用read和write拷贝一张图片

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

int main(int argc, const char *argv[])
{
    //以读的方式打开源文件
    int fd_r = open("./1.png", O_RDONLY);
    if(fd_r < 0)
    {
        perror("open");
        return -1;
    }

    //以写的方式打开目标文件
    int fd_w = open("copy.png", O_WRONLY|O_CREAT|O_TRUNC, 0777);
    if(fd_w < 0)
    {
        perror("open");
        return -1;
    }

    char buf[20] = "";
    ssize_t res = 0;
    //读一次写一次，直到文件读取完毕，跳出循环
    while(1)
    {
        bzero(buf, sizeof(buf));
        res = read(fd_r, buf, sizeof(buf));
        if(0 == res)
        {
            printf("文件读取完毕\n");
            break;
        }
        else if(res < 0)
        {
            perror("read");
            return -1;
        }

        //注意：读多少个，就应该写入多少个
```

```
        write(fd_w, buf, res);
    }

    //关闭文件
    close(fd_w);
    close(fd_r);
    return 0;
}
```

6）lseek

```
功能：修改文件偏移量；
头文件：
        #include <sys/types.h>
        #include <unistd.h>

        off_t lseek(int fd, off_t offset, int whence);
参数：
    int fd：指定要修改文件偏移量的文件对应的文件描述符；
    off_t offset：距离第三个形参whence的偏移量；（往后偏用正数，往前偏用负数）
    int whence：
        SEEK_SET    文件开头
        SEEK_CUR    文件当前位置
        SEEK_END    文件结尾；
返回值：
    成功，返回修改后文件当前位置距离文件开头的偏移量；
    失败，返回(off_t) -1，同时更新errno；


计算文件大小：
   //修改文件偏移量到文件结尾
   fseek(fp, 0, SEEK_END);
   long off = ftell(fp);
   printf("%ld\n", off);

相当于
    off_t size = lseek(fd, 0, SEEK_END);
    printf("%ld\n", size);
```

【3】文件属性相关的函数

1）stat

```
功能：获取文件属性；
头文件：
        #include <sys/types.h>
        #include <sys/stat.h>
        #include <unistd.h>

        int stat(const char *pathname, struct stat *statbuf);
参数：
```

```
    char *pathname：指定要获取哪个文件的属性，填上对应的路径以及名字；
    struct stat *statbuf：获取出来的属性，会被存储到该指针指向的内存空间中；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；

            struct stat {
                dev_t     st_dev;         /* ID of device containing file */
                ino_t     st_ino;         /* Inode number */                    文件的
inode号
                mode_t    st_mode;        /* File type and mode */              文件的
类型以及权限
                nlink_t   st_nlink;       /* Number of hard links */            文件的
硬链接数
                uid_t     st_uid;         /* User ID of owner */                文件所
属用户的uid号
                gid_t     st_gid;         /* Group ID of owner */               文件所
属用户的gid号
                dev_t     st_rdev;        /* Device ID (if special file) */
                off_t     st_size;        /* Total size, in bytes */            文件大
小
                blksize_t st_blksize;     /* Block size for filesystem I/O */
                blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

                /* Since Linux 2.6, the kernel supports nanosecond
                   precision for the following timestamp fields.
                   For the details before Linux 2.6, see NOTES. */

                struct timespec st_atim;  /* Time of last access */             文件最
后一次被访问的时间
                struct timespec st_mtim;  /* Time of last modification */       文件最
后一次被修改的时间
                struct timespec st_ctim;  /* Time of last status change */      文件最
后一次改变状态的时间

            #define st_atime st_atim.tv_sec      /* Backward compatibility */
            #define st_mtime st_mtim.tv_sec
            #define st_ctime st_ctim.tv_sec
            };
```

2）提取文件的权限

```
mode_t    st_mode;        /* File type and mode */          文件的类型以及权限
其中低9bit存储了文件的权限
```

将mode按位&上对应的权限，判断结果是否为0，如果不为0，则代表有对应权限。如果为0，则没有对应权限。

```
mode: 100664 ---》 0664 ---> 110 110 100 ---> rw- rw- r--

               110 110 100
            &  100 000 000
             --------------------
```

```
                100 000 000 ---> 0400 结果不为0，则有读权限，打印 r

                110 110 100
            &   010 000 000
            ------------------
                010 000 000 ---> 0200 结果不为0，则有写权限，打印w

                110 110 100
            &   001 000 000
            ----------------
                000 000 000 ---> 0 结果为0，则没有可执行权限，打印-
```

```
void get_filePermission(mode_t m)
{
    int i = 0;
    for(i=0; i<9; i++)
    {
        if(((0400>>i) & m) == 0)
        {
            putchar('-');
            continue;
        }

        //能运行到当前位置，则说明有权限
        //则需要判断打印是r w x

        switch(i%3)
        {
        case 0:
            putchar('r');
            break;
        case 1:
            putchar('w');
            break;
        case 2:
            putchar('x');
            break;
        }
    }
}
```

3）提取文件的类型

> 7种：bsp-lcd;

i. 通过宏函数提取文件类型

```
man 2 stat,找到
       st_mode
              This field contains the file type and mode.  See inode(7) for
further information.
man 7 inode:

           S_ISREG(m)  is it a regular file?     判断是否是普通文件 -
```

```
           S_ISDIR(m)  directory?                                   d
           S_ISCHR(m)  character device?                           c
           S_ISBLK(m)  block device?                               b
           S_ISFIFO(m) FIFO (named pipe)?                          p
           S_ISLNK(m)  symbolic link?  (Not in POSIX.1-1996.)      l
           S_ISSOCK(m) socket?  (Not in POSIX.1-1996.)             s
如果是该类型文件，则返回真
否则返回假;
```

```c
//提取文件的类型
void get_fileType(mode_t m)     //mode_t m = buf.st_mode
{
    if(S_ISREG(m))
        putchar('-');
    else if(S_ISDIR(m))
        putchar('d');
    else if(S_ISCHR(m))
        putchar('c');
    else if(S_ISBLK(m))
        putchar('b');
    else if(S_ISFIFO(m))
        putchar('p');
    else if(S_ISLNK(m))
        putchar('l');
    else if(S_ISSOCK(m))
        putchar('s');
}
```

ii. 通过提取对应bit获取文件类型

man 7 inode

```
st_mode & S_IFMT 提取文件类型
S_IFMT     0170000   bit mask for the file type bit field
            001 111 000 000 000 000
st_mode    040775
            000 100 000 111 111 101

            000 100 000 111 111 101     st_mode
          &  001 111 000 000 000 000     S_IFMT
          -------------------------------
            000 100 000 000 000 000  -->0040000
          将计算之后的数字与下列宏进行比较，得出文件类型。

100664          001 000 000 110 110 100
&0170000    &   001 111 000 000 000 000     S_IFMT
        ---------------------------------------
                001 000 000 000 000 000 --> 0100000
          将计算之后的数字与下列宏进行比较，得出文件类型。


           S_IFSOCK   0140000   socket
           S_IFLNK    0120000   symbolic link
           S_IFREG    0100000   regular file
           S_IFBLK    0060000   block device
```

```
S_IFDIR     0040000   directory
S_IFCHR     0020000   character device
S_IFIFO     0010000   FIFO
```

4）提取文件所属用户名

```
uid_t     st_uid;         /* User ID of owner */            文件所属用户的uid号
```

getpwuid函数

```
功能：将uid_t类型的uid转换成对应的用户名
        #include <sys/types.h>
        #include <pwd.h>

        struct passwd *getpwuid(uid_t uid);
参数：
    uid_t uid：指定要转换成用户名的uid号；
返回值：
    成功，返回结构体指针；
    失败，返回NULL；
            struct passwd {
                char   *pw_name;       /* username */
                char   *pw_passwd;     /* user password */
                uid_t   pw_uid;        /* user ID */
                gid_t   pw_gid;        /* group ID */
                char   *pw_gecos;      /* user information */
                char   *pw_dir;        /* home directory */
                char   *pw_shell;      /* shell program */
            };
```

5）提取文件所属组用户名

```
gid_t     st_gid;         /* Group ID of owner */           文件所属用户的gid号
```

getgrgid函数

```
功能：将gid_t类型的gid转换成对应的组用户名；
头文件：
        #include <sys/types.h>
        #include <grp.h>

        struct group *getgrgid(gid_t gid);
参数：
    gid_t gid：指定要转换成组用户名的gid号；
返回值：
    成功，返回结构体指针；
    失败，返回NULL；
            struct group {
                char   *gr_name;       /* group name */
                char   *gr_passwd;     /* group password */
                gid_t   gr_gid;        /* group ID */
```

```
            char   **gr_mem;         /* NULL-terminated array of pointers
                                       to names of group members */
        };
```

【4】目录相关的函数

1）opendir

```
功能：打开一个目录：
头文件：
        #include <sys/types.h>
        #include <dirent.h>

        DIR *opendir(const char *name);
参数：
    char *name：指定要打开的目录对应的路径以及目录名；
返回值：
    成功，返回目录指针；
    失败，返回NULL；
```

2）closedir

```
功能：关闭目录；
头文件：
        #include <sys/types.h>
        #include <dirent.h>

        int closedir(DIR *dirp);
参数：
    DIR *dirp：指定要关闭的目录对应的目录指针；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

3）readdir

```
功能：读取目录；
头文件：
        #include <dirent.h>

        struct dirent *readdir(DIR *dirp);
参数：
    DIR *dirp：指定要读取的目录对应的目录指针；
返回值：
    成功，返回结构体指针；
    读取到文件结尾，则返回NULL,但是不会更新errno,即errno == 0;
    失败，返回NULL,更新errno;
            struct dirent {
                ino_t          d_ino;       /* Inode number */
```

```
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

作业

> 要求输入目录的路径后，能够打印出指定路径下所有文件的详细信息，类似ls -l

5. 库

【1】库的概念

1. 库中存放的是二进制可执行文件; **库中存放的都是功能代码，没有主函数**

   注意：封装库的时候，不要把main函数放到库中！！！
2. **相比较于二进制可执行程序，库是不能单独使用的。**
3. 库中的可执行文件需要被载入到内存中使用。
4. 每个操作系统都有自己的库，互不兼容。

1）库的分类

1. 静态库
2. 动态库（共享库）

	linux	windows
静态库	libxxx.a	xxx.lib
动态库	libxxx.so	xxx.dll

2）库存在的意义

库是已经写好的，成熟的，可以复用的功能代码。我们写的很多代码都是依赖基础库的。封装好的库，可以提高工作效率。

【2】静态库

1. 静态库的原理

通过静态库封装的功能代码，在程序编译到**链接库**步骤的时候，会将函数**继承**到可执行程序中。生成的二进制可执行程序中会有静态库函数的源功能代码

优点：

1. 程序编译完成后与静态库没有任何关系，所以脱离静态库后能够正常运行。
2. 方便移植

缺点：

1. 由于库函数是继承过来的，所以可执行二进制程序会比较大。
2. 加载到内存中运行的时候，占用内存空间会比较大；
3. 程序的更新部署比较麻烦：

   由于完全脱离静态库，所以静态库中的函数如果有更新的话，程序需要重新编译后才可以使用新的函数。

2. 静态库的制作以及使用

1）制作指令

> -ESc .iso

```
gcc -c  func.c -o func.o          //一步完成前三个步骤：预处理，编译，汇编步骤
ar -crs libfunc.a func.o          //生成一个静态库，且静态库的名字func

如果有多个文件想要合成一个静态库：
ar -crs libfunc.a func1.o func2.o func3.o func4.o
```

静态库的命名规则：

以lib开头，很后面紧跟库的名字，以.a结尾：

例如：libxxx.a **注意：xxx才是库的名字**

> ar：创建静态库指令
>
> -c：创建静态库选项
>
> -r：将文件插入到静态库中或者替换静态库中的同名文件;
>
> -s：重置静态库索引

步骤：

1. 分文件：将mian函数与功能函数分离，例如创建一个func.c用于存储功能函数
2. 写一个头文件，例如：func.h，将功能函数的声明放在该头文件中。
3. 测试一下分文件是否正确：联合编译即可： gcc main.c func.c
4. 按照上述制作指令，将func.c制作成静态库。

2）使用方式

```
gcc  main.c -L库的路径 -l库的名字                          默认生成一个a.out可执行文件
gcc  main.c -o main -L库的路径 -l库的名字                  默认生成一个名字叫main的可执行文件
```

注意：

1. -L后面紧跟库的路径，中间没有空格
2. -l后面紧跟库的名字，中间没有空格

【3】动态库（共享库）

1. 动态库的原理

动态库封装的库函数，会把库函数的链接推迟到**程序运行**的时候。

当程序执行到库函数的时候，会到内存中查找该动态库函数，

如果内存中不存在该函数，则会将动态库函数加载到内存中。

如果内存中存在该函数，则直接调用该函数。

优点：

1. 由于只有在程序运行后，才会加载动态库函数，所以可执行程序会比较小。
2. 可执行程序运行的时候，占用的内存空间小，因为内存中如果存在该动态库函数，则不会再加载第二次。
3. 程序更新部署会更加方便

缺点：

1. 程序运行的时候，无法脱离动态库，如果没有找到动态库，程序会直接崩溃。

2. 动态库的制作以及使用

1）制作指令

```
gcc -c -fPIC func.c -o func.o        //完成预处理、编译、汇编
gcc -shared -o libxxx.so  func.o     //生成一个动态库，库的名字叫做xxx

如果有多个文件想要合成一个动态库：
gcc -shared -o libxxx.so func1.o func2.o func3.o func4.o
```

2）使用方式

```
gcc  main.c -L库的路径 -l库的名字                          默认生成一个a.out可执行文件
gcc  main.c -o main -L库的路径 -l库的名字                  默认生成一个名字叫main的可执行文件
```

注意：

1. -L后面紧跟库的路径，中间没有空格
2. -l后面紧跟库的名字，中间没有空格

```
ldd a.out            查看可执行二进制程序所依赖的动态库
    linux-vdso.so.1 (0x00007ffd5f1a9000)
    libxxx.so => not found
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe3256cc000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fe325cbf000)


    环境变量如果没有配置，系统会默认到/lib/ /usr/lib/下查找动态库，会导致找不到自定义的动态库;
```

```
ubuntu@ubuntu:02_lib.so $ gcc -c -fPIC  func.c -o func.o
ubuntu@ubuntu:02_lib.so $ gcc -shared -o libxxx.so func.o
ubuntu@ubuntu:02_lib.so $ gcc 01_stat.c -L./ -lxxx
ubuntu@ubuntu:02_lib.so $ ./a.out
./a.out: error while loading shared libraries: libxxx.so: cannot open shared object file: No such file or directory
ubuntu@ubuntu:02_lib.so $ ldd a.out
        linux-vdso.so.1 (0x00007ffcd6be3000)
        libxxx.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6f6c88b000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f6f6ce7e000)
ubuntu@ubuntu:02_lib.so $
```

3）配置环境变量

1）在 LD_LIBRARY_PATH 环境变量中，添加上动态库所在的绝对路径

```
echo $LD_LIBRARY_PATH
```

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:动态库的绝对路径                注意只要路径不要库名;
只在配置终端有效。

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/ubuntu/mydir/031/01_IO/3_lib/2_动态库/
```

2）将动态库移动到 /lib/ 或者 /usr/lib/

如果目录移动错误，会导致虚拟机崩溃，请先拍摄快照。

```
sudo mv libmyfunc.so /lib/
sudo mv libmyfunc.so /usr/lib/
```

3）在 /etc/ld.so.conf.d/ 目录下创建一个以 .conf 为拓展名的文件

```
cd /etc/ld.so.conf.d/
sudo vim my.conf
    将动态库的绝对路径添加到my.conf中   注意只要路径不要库名;
    文件中，一行只能放一条动态库的绝对路径;
    保存退出
sudo ldconfig    刷新环境变量
```

3.静态库与动态库的区别（重点！！）

6. 进程

【1】进程的概念

1. 什么是进程

1. 进程是程序的一次执行过程。进程是存在于内存中的，是一个抽象的概念。
2. 进程是独立的，可以被**cpu调度**的任务。

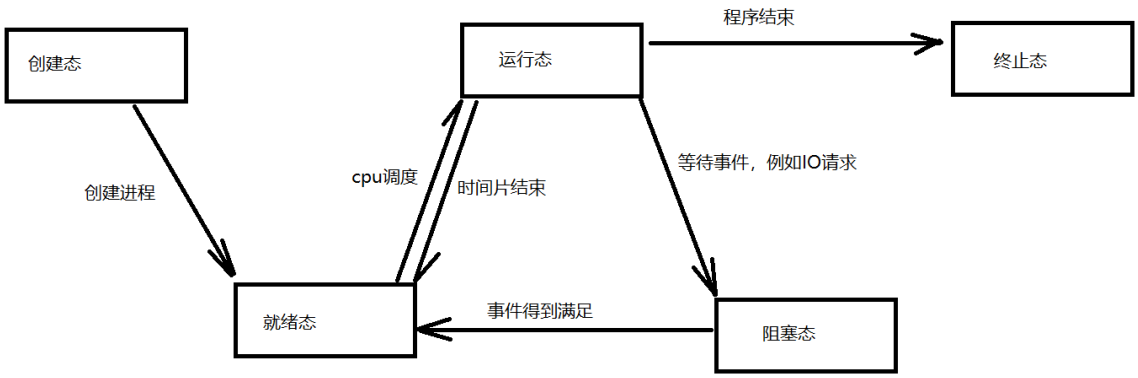
   Linux的进程调度机制：时间片轮询机制。
3. 进程在被调度的时候，系统会分配和释放各种资源（cpu资源，内存资源，进程调度块。。。。。）

2. 进程的五种状态（重点）

进程的五态图： 7态图会多两种状态，挂起阻塞态，挂起就绪态

运行：进程运行后，会出现五种状态，分别是创建态，就绪态，运行态，阻塞态，终止态。

运行态：运行态知识进程运行时候的一种状态而已。



3. 进程和程序的区别

程序是静态的，它是存储在硬盘上的可执行二进制文件。

进程是动态的，它是程序的一次执行过程，包括了进程的创建，调度，死亡等等状态，是存在内存中的。

4. 进程的内存管理（重点！！！）

1. 每个进程都会分配4G的内存空间。（虚拟内存空间）
2. 其中0~3G是用户空间使用，**每个进程都有自己独立的用户空间**，互不相关
3. 3G~4G的内核空间，该**内核空间是所有进程共享的**。

虚拟内存空间和物理内存空间

物理内存空间（物理地址）：是内存条上（硬件上）真正存在的存储空间。

虚拟内存空间（虚拟地址）：是程序运行后，系统分配的虚拟空间，使用的时候需要**从物理地址映射到虚拟地址上**。

为什么虚拟地址空间是4G

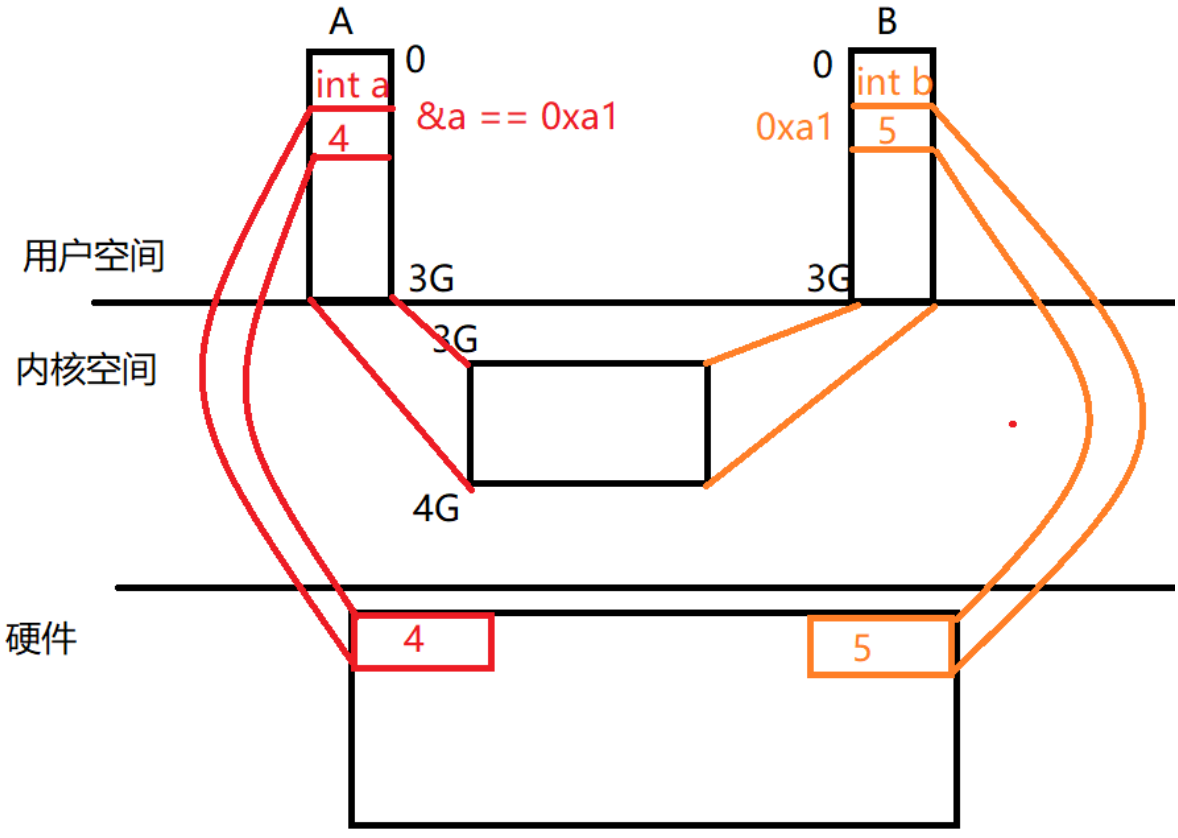
计算机中存储地址的肯定是指针变量

由于32位操作系统中，指针变量占4个字节，所以取值范围为[0, 2^32-1]-->[0, 4G-1]，所以32位操作系统指针变量的寻址范围是4G。

所以32位操作系统的虚拟地址空间是0~4G.

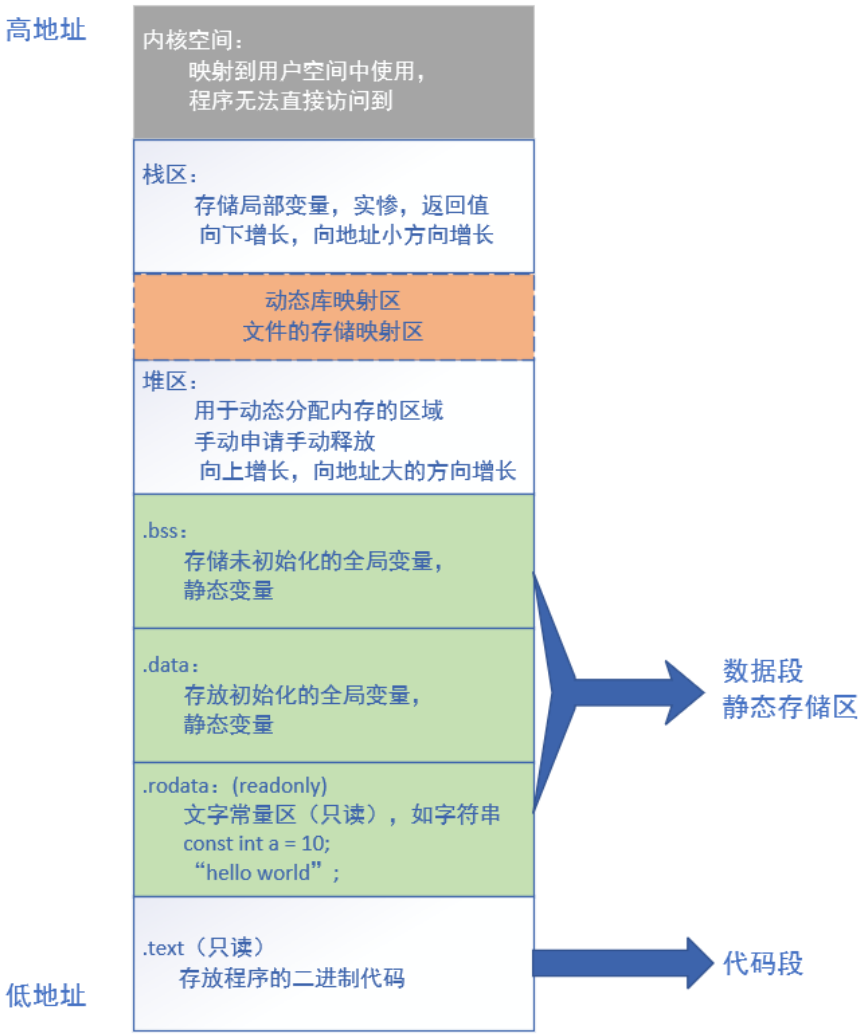
由于64位操作系统，指针变量占了8个字节，所以指针变量的取值范围为[0, 2^64-1]，由于这个数字太大，用不了这么多，则只取了前48位。

所以64位操作系统的寻址范围是[0, 2^48-1] --> 256TB



5. 进程是资源分配的最小单位（重点！！！）

1. 以进程为单位申请释放内存资源。如下图所示，栈区，堆区静态存储区，正文段，均处于用户空间，所有进程相互独立



2. 以进程为单位分配文件描述符：1024个
3. 以进程为单位分配cpu时间片。
4. 以进程为单位管理自己的虚拟地址空间，在需要使用的时候有物理地址映射到虚拟地址上。

6. 进程标识

操作系统会给每个进程分配一个id号，这个id号就是进程号

1. 主要的进程标识

   进程号：PID　　process id

   父进程号：PPID　　parent process id

2. PID号是进程的唯一标识
3. 操作系统刚启动的时候，会启动三个进程。

   PID：0　idle进程　　操作系统引导程序，创建1号，2号进程

   PID：1　init进程　　初始化内核的各个模块，当内核启动完毕后，用于收养孤儿进程（没有父进程的进程）

   PID：2 kthreadd进程　　用于进程间调度

7.进程相关的shell指令

1）ps -aux

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

功能：显示进程占资源的百分比

2）ps -ajx

```
  PPID      PID          PGID         SID     TTY          TPGID   STAT   UID
TIME COMMAND
  父进程id  当前进程id     进程组id     会话组id  所依赖的终端          进程的状态
```

进程组：若干个进程的集合，称之为进程组。默认情况下，新创建的进程会继承父进程的组id;

会话组：若干个进程组的集合，称之为会话组。默认情况下，新创建的进程会继承父进程的会话id;

STAT

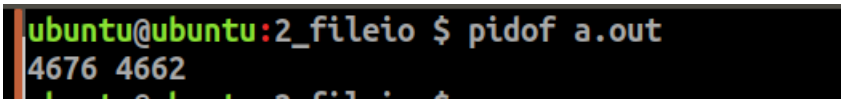
> man ps /stat

```
               D    不可被中断的阻塞状态
               R    运行状态
               S    可以被中断的阻塞状态
               T    被挂起的进程状态
               t    被追踪被调试时候的挂起状态;
               X    死亡态
               Z    僵尸状态，进程退出后，资源没有被父进程回收;

       For BSD formats and when the stat keyword is used, additional characters
may be displayed:

               <    高优先级的
               N    低优先级的
               L    有些页被锁进内存
               s    会话组组长
               l    多线程
               +    运行在前端
```

3）pidof

功能：根据进程名字查看进程的pid号

pidof 进程名字



4）pstree

功能：查看进程关系树

5）top

功能：实时显示进程状态

top -d 秒数

top -p pid

退出：按一下q

6）kill

kill -9 pid 根据pid进程号杀死指定进程

killall -9 进程名字 根据名字杀死进程。

【3】进程相关的系统调用

1）fork

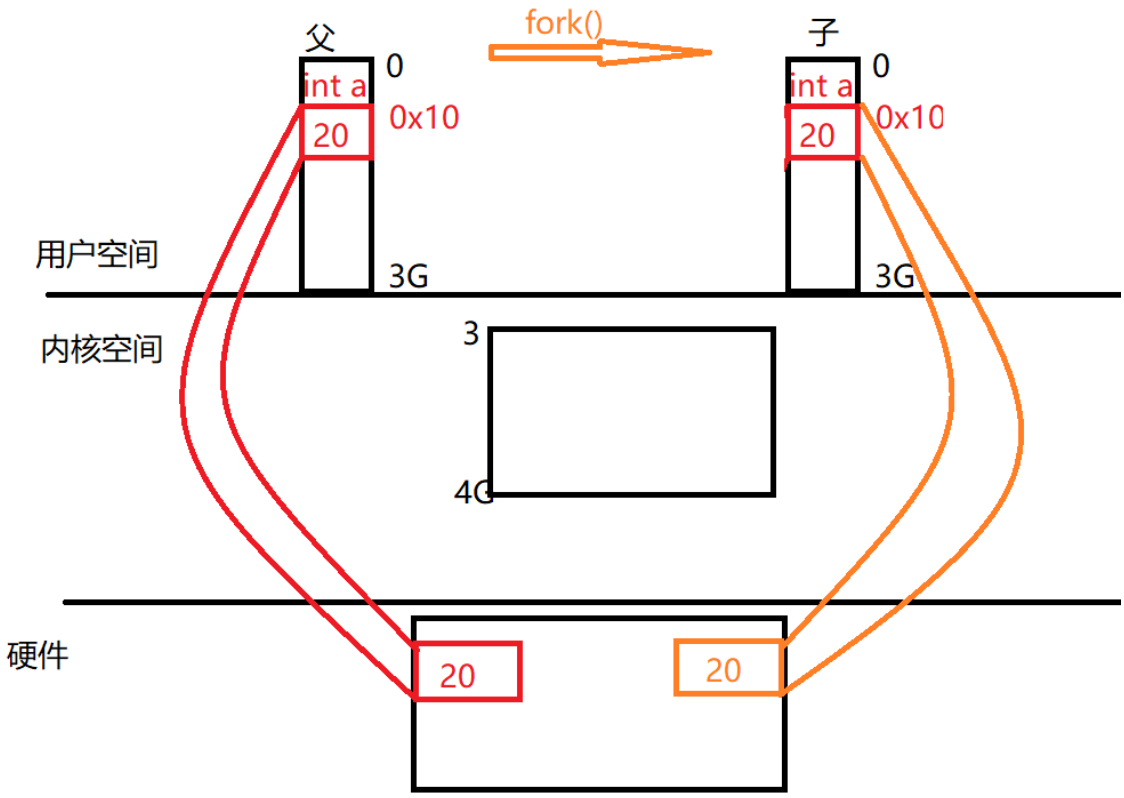
```
功能：创建一个子进程；
头文件：
        #include <sys/types.h>
        #include <unistd.h>

        pid_t fork(void);
参数：；
返回值：
     成功，在父进程中返回创建成功的子进程的PID；
           在子进程中返回0；
     失败，在父进程中返回-1，并且没有子进程被创建出来，更新errno；
```

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

int main(int argc, const char *argv[])
{
    int a = 20;
    printf("__%d__\n", __LINE__);
    //创建一个子进程
    pid_t pid = fork();

    //从当前位置往后，会有两个进程执行，一个是父进程，一个是子进程
    //父进程中fork函数的返回值为子进程的pid号，即pid>0
    //子进程fork函数的返回值为0，即pid == 0
```

```
    if(pid > 0)      //在父进程中该条件为真
    {
        //父进程会执行该段代码
        a = 10;
        printf("pid = %d a=%d &a=%p __%d__\n", pid, a, &a, __LINE__);
    }
    else if(0 == pid)   //在子进程中该条件为真
    {
        //子进程会执行该段代码
        sleep(1);   //让子进程休眠，主动放弃cpu资源
        printf("pid = %d a=%d &a=%p __%d__\n", pid, a, &a, __LINE__);
    }
    else if(pid < 0)
    {
        perror("fork");
        return -1;
    }

    while(1)
    {
        sleep(1);
    }
    return 0;
}
```

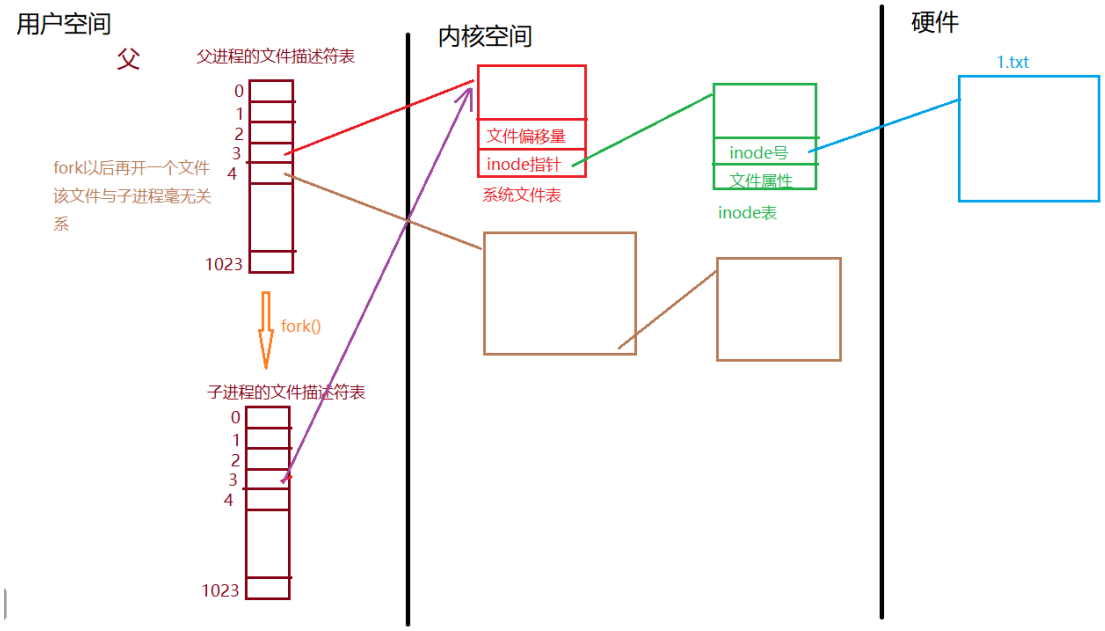
注意：

1. 父进程会拷贝一份资源给子进程，父子进程的资源是一致的，但是存在的内存空间肯定不一致。
2. 创建的子进程会执行fork以下的代码，不会执行fork以及fork以上的代码。
3. fork以后，父子进程谁先运行是不确定的，主要看时间片以及cpu的调度;
4. 子进程创建完毕的那一瞬间，父子进程的虚拟地址空间长得完全一致。但是映射的物理地址肯定不一致。
5. 进程之间相互独立，即使是父子进程用户空间也是相互独立的。



6. 子进程会拷贝父进程的文件描述符表。

   所以fork之前父进程打开了一个文件，fork之后的子进程不需要再打开这个文件就可以操作这个文件了。




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

int main(int argc, const char *argv[])
{
    int i = 0;
    while(i<4)
    {
        fork();
        i++;
    }

    while(1)
    {
        sleep(1);
    }
    return 0;
}
```

创建的进程个数为 2^4 = 16个

如果循环次数是n，则创建的进程个数是2^n个

> 1. 用文件IO拷贝一张图片，要求子进程先拷贝后半部分，父进程后拷贝前半部分。
> 2. 用文件IO拷贝一张图片，要求子进程拷贝后半部分，父进程拷贝前半部分。不允许使用sleep(1)函数。
>
>   说明父子进程不会相互干扰，即系统文件表不使用同一张即可。---》说明父进程开一次文件，子进程自己开一次文件。

```c
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
```

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

int main(int argc, const char *argv[])
{
    int fd_r = open("./1.png", O_RDONLY);
    if(fd_r < 0)
    {
        perror("open");
        return -1;
    }

    int fd_w = open("copy.png", O_WRONLY|O_CREAT|O_TRUNC, 0664);
    if(fd_w < 0)
    {
        perror("open");
        return -1;
    }

    off_t size = lseek(fd_r, 0, SEEK_END);
    printf("size=%ld\n", size);

    //创建子进程
    pid_t pid = fork();
    if(pid > 0)
    {
        sleep(2);

        //父进程执行，拷贝前半部分，所以要偏移到开头位置
        lseek(fd_r, 0, SEEK_SET);
        lseek(fd_w, 0, SEEK_SET);

        char c = 0;
        int i = 0;
        for(i=0; i<size/2; i++)
        {
            read(fd_r, &c, 1);
            write(fd_w, &c, 1);
        }
        printf("文件前半部分拷贝完毕\n");


    }
    else if(0 == pid)
    {
        //子进程执行，拷贝后半部分，所以要偏移到中间位置
        lseek(fd_r, size/2, SEEK_SET);
        lseek(fd_w, size/2, SEEK_SET);

        char c = 0;
        int i = 0;
        for(i=size/2; i<size; i++)
        {
            read(fd_r, &c, 1);
            write(fd_w, &c, 1);
        }
        printf("文件后半部分拷贝完毕\n");
    }
```

```
    else if(pid < 0)
    {
        perror("fork");
        return -1;
    }


    close(fd_r);
    close(fd_w);
    return 0;
}
```

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

int main(int argc, const char *argv[])
{
    //删除copy.png的文件，防止上次写入的内容，影响当前的写入
    remove("copy.png");

    //创建子进程
    pid_t pid = fork();

    int fd_r = open("./1.png", O_RDONLY);
    if(fd_r < 0)
    {
        perror("open");
        return -1;
    }

    int fd_w = open("copy.png", O_WRONLY|O_CREAT, 0664);
    if(fd_w < 0)
    {
        perror("open");
        return -1;
    }

    off_t size = lseek(fd_r, 0, SEEK_END);
    printf("size=%ld\n", size);

    if(pid > 0)
    {
        //父进程执行，拷贝前半部分，所以要偏移到开头位置
        lseek(fd_r, 0, SEEK_SET);
        lseek(fd_w, 0, SEEK_SET);

        char c = 0;
        int i = 0;
        for(i=0; i<size/2; i++)
        {
            read(fd_r, &c, 1);
            write(fd_w, &c, 1);
        }
        printf("文件前半部分拷贝完毕\n");
```

```
    }
    else if(0 == pid)
    {
        //子进程执行，拷贝后半部分，所以要偏移到中间位置
        lseek(fd_r, size/2, SEEK_SET);
        lseek(fd_w, size/2, SEEK_SET);

        char c = 0;
        int i = 0;
        for(i=size/2; i<size; i++)
        {
            read(fd_r, &c, 1);
            write(fd_w, &c, 1);
        }
        printf("文件后半部分拷贝完毕\n");
    }
    else if(pid < 0)
    {
        perror("fork");
        return -1;
    }


    close(fd_r);
    close(fd_w);
    return 0;
}
```

2）getpid / getppid

```
功能：获取当前进程/父进程的pid号；
头文件：
        #include <sys/types.h>
        #include <unistd.h>

        pid_t getpid(void);
        pid_t getppid(void);

返回值：
    成功获取到的进程号、父进程号；
```

3）_exit / exit

i. _exit

man 2 卷

```
功能：终止进程，清除进程使用的内存空间。销毁缓冲区，不会刷新缓冲区；
头文件：
        #include <unistd.h>

        void _exit(int status);
参数：
    int status：用来传递进程退出状态值;可以输入任意整型数据；
                该状态值会被 wait/waitpid函数接收；
```

注意：_exit不会刷新缓冲区，而是直接销毁缓冲区；

ii. exit

man 3 卷

```
功能：终止进程，清除进程使用的内存空间。会刷新缓冲区；
头文件：
        #include <stdlib.h>

        void exit(int status);
参数：
    int status：用来传递进程退出状态值;可以输入任意整型数据；
                该状态值会被 wait/waitpid函数接收；
```

4）wait / waitpid

i. wait

```
功能：阻塞等待，任意子进程退出，并回收其资源；（即回收僵尸进程）
头文件：
        #include <sys/types.h>
        #include <sys/wait.h>

        pid_t wait(int *wstatus);
参数：
    int *wstatus：该指针指向的内存空间中会存储子进程退出状态值；
                    可以选择填NULL，代表不接收子进程的退出状态值；
返回值：
    成功，返回被回收的子进程的pid号；
    失败，返回-1，更新errno；
```

注意：

1. wait函数可以回收**任意**子进程的资源，如果没有子进程，则wait函数不阻塞，立即返回-1，失败情况.
2. wait/waitpid的参数可以接收子进程的退出状态，但是该状态中不仅包含了exit函数传递的状态值，还有其他状态，需要通过宏函数提取：

   WEXITSTATUS(wstatus)---》该宏函数的算法原型为：`(((status) & 0xff00) >> 8)`

所以，exit/_exit传递的进程退出状态值，在wait函数的形参中，只能占第9位到第16位，总共8bit

3. 由于子进程的退出状态值在wait形参中只能占8bit，所以子进程只能传递2^8=256 --》[0, 255]个退出状态值。

ii. waitpid

```
功能：可以阻塞等待指定子进程退出;，并回收其资源;（即回收僵尸进程）
头文件：
        #include <sys/types.h>
        #include <sys/wait.h>

        pid_t waitpid(pid_t pid, int *wstatus, int options);
参数：
    pid_t pid：指定要回收的进程/进程组的id;
        < -1    等待指定进程组下的任意一个子进程退出，进程组id == pid参数的绝对值
        -1      等待当前进程下的任意一个子进程;
        0       等待当前进程组下的任意一个子进程;
        > 0     等待指定子进程退出，指定子进程的pid号 == pid参数;

    int *wstatus：该指针指向的内存空间中会存储子进程退出状态值;
                可以选择填NULL，代表不接收子进程的退出状态值;

    int options：
              填0，阻塞方式运行该函数，当指定的子进程没有退出，则当前函数阻塞，直到指定子进程退出，解除阻塞;
              WNOHANG：非阻塞形式运行该函数,即使指定的子进程没有退出，当前函数不阻塞，立即返回;
返回值：
    成功，返回成功回收到的子进程的PID的号;
         如果options参数指定为WNOHANG形式的时候，函数运行成功，但是指定子进程没有退出，则这个时候返回0;
    失败，函数运行失败，返回-1;更新errno;

注意：只能父收子，不能子收父；  只能上级收下级,下级不能收上级，并且同级之间也不能相互回收;
```

【4】Linux中的特殊进程

1. 孤儿进程

没有父进程的进程，称之为孤儿进程。孤儿进程会被init (1号)进程收养，脱离终端控制。

孤儿进程不能用ctrl+c退出，但是可以被kill -9杀死

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

int main(int argc, const char *argv[])
{
    pid_t pid = fork();
    if(pid > 0)
    {
```

```
    }
    else if(0 == pid)
    {
        while(1)
        {
            printf("this is child  %d %d\n", getppid(), getpid());
            sleep(1);
        }
    }
    else
    {
        perror("fork");
        return -1;
    }


    return 0;
}
```

2. 僵尸进程

子进程退出，父进程没有为子进程收尸，此时子进程会变成僵尸进程。

注意：

1. 僵尸进程只能被回收，不能再次被kill
2. 父进程退出，子进程的资源将由内核回收。
3. 僵尸进程是有危害的，因为僵尸进程中的部分资源没有被回收掉，它会占用进程的调度块，占用部分物理空间，占用进程号等等....

如何回收僵尸进程：

1. 退出父进程
2. 用wait/waitpid函数回收，缺点：用阻塞形式会导致父进程无法正常运行自己的任务，用非阻塞形式可能会回收不到。
3. 用信号的方式回收僵尸进程。

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

int main(int argc, const char *argv[])
{
    pid_t pid = fork();
    if(0 == pid)
    {
    }
    else if(pid > 0)
    {
        while(1)
        {
```

```
            printf("this is parent  %d %d\n", getpid(), pid);
            sleep(1);
        }
    }
    else
    {
        perror("fork");
        return -1;
    }


    return 0;
}
```

3. 守护进程（幽灵进程）

1. 守护进程脱离于终端并且在后台运行。
2. 守护进程的执行过程中的所有信息不会显示到终端上，并且不会被任何终端产生的终端信息所打断。
3. 守护进程独立于控制终端，且一般会周期性的执行某个任务或者等待处理某些事情。
4. 大多数服务器都是由守护进程实现的。

守护进程的创建

1. 创建孤儿进程：所有的工作都在子进程中执行，形式上脱离终端控制
2. 创建新的会话组。（如干个进程组的集合称之为会话组，默认情况下子进程会继承父进程的会话组id）

   子进程创建新的会话，就不再依附于父进程的会话组;

   setsid函数

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

   pid_t setsid(void);
   返回值：
       就是新建成功的会话组id号；

   创建新的进程组，成为进程组组长；
   创建新的会话组，成为会话组组长；
   完全脱离终端。
   ```

3. 修改当前孤儿进程的运行目录为不可卸载的文件系统。防止占用可以删除的文件夹运行，可以修改成运行到'/'根目录下，也可以换成其他不能被删除的目录下运行。

   chdir函数

```
功能：修改进程的运行目录；
头文件：
        #include <unistd.h>

        int chdir(const char *path);
char *path：指定要修改到哪个目录下运行


    chdir("/");
```

4. 重设文件权限掩码：umask

   umask(0);

5. 关闭所有文件描述符。

   子进程的文件描述符是从父进程位置拷贝过来，继承过来的文件描述符不会使用到，浪费系统资源。所以需要关闭。

   需要打印数据的时候，或者调试信息的时候，可以开一个文件，将数据打印到文件中。该文件一般放在/tmp目录下。

```
int i = 0;
for(i=0; i<1024; i++)
    close(i);
```

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

int main(int argc, const char *argv[])
{
    //创建孤儿进程
    pid_t pid = fork();
    if(pid > 0)
    {
    }
    else if(0 == pid)
    {
        //创建新的会话组
        pid_t sid = setsid();
    //  printf("sid = %d\n", sid);


        //修改运行目录到不可卸载的文件目录
        chdir("/");     //注意从当前位置往后，进程运行在根目录下

        //重设文件权限掩码
        umask(0);

        //关闭所有文件描述符
        for(int i=0; i<1024; i++)
            close(i);

        while(1)
        {
            //功能代码
```

```
            sleep(1);
        }
    }
    else
    {
        perror("fork");
        return -1;
    }


    return 0;
}
```

7. 线程

【1】概念

线程：是一个进程中并发执行多种任务的机制。

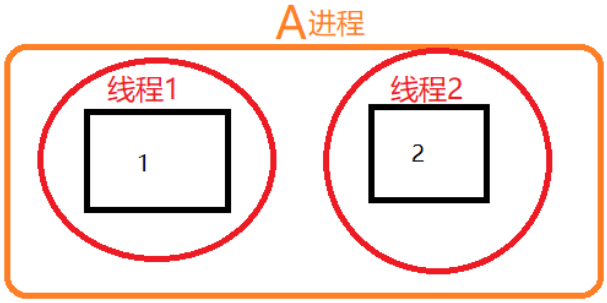
并发：单核cpu多个任务同时运行。（同时：伪同时，cpu以ns~ms级别的速度进行进程间线程间调度，切换进程和线程）

进程间的上下文切换：

上下文：运行一个进程所需要的所有资源。

上下文切换：从A进程运行到B进程的时候，cpu要切换所有资源，替换原有资源，这是一个**耗时操作**。

为了提高系统的性能，引入了一个轻量级进程的概念，称之为线程。将多个任务放在一个进程中执行。



进程是资源分配的最小单位

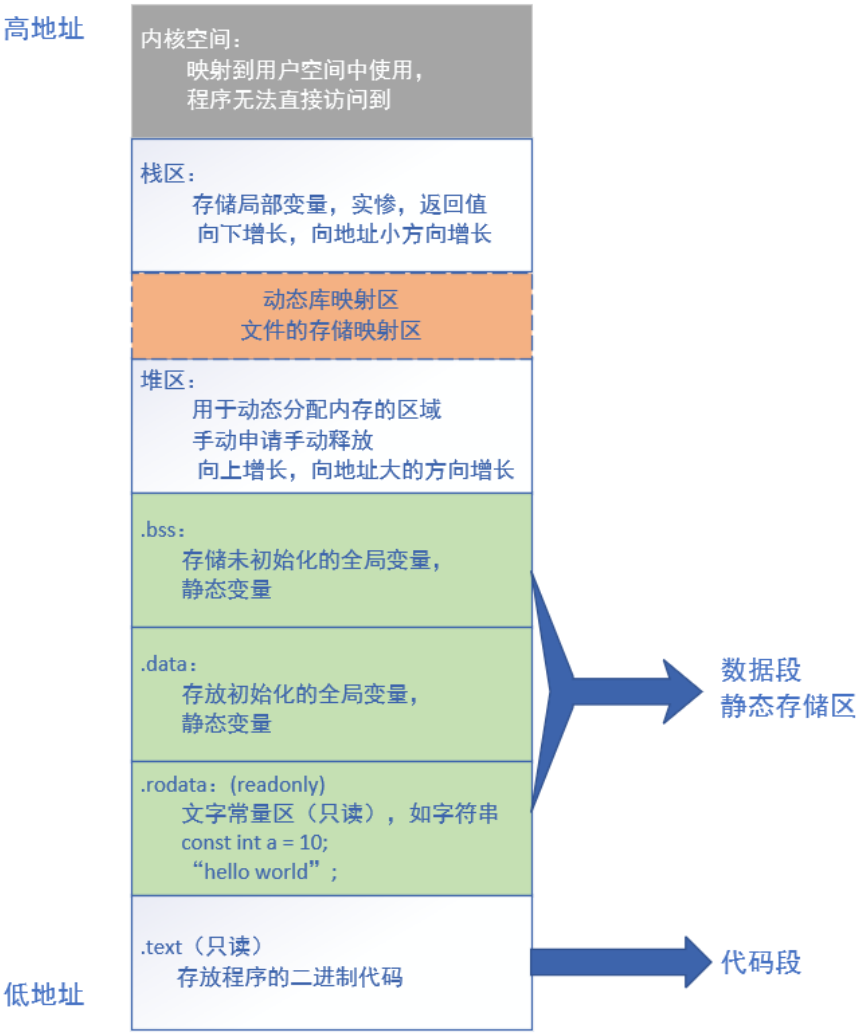
线程是任务运行的最小单位。

线程：属于进程，每一个进程至少需要一个线程作为指令执行体，线程运行在进程的空间内。

多线程：一个进程可以运行多个线程，所以一个进程可以运行多个任务。

【2】线程是任务运行的最小单位

同一个进程下的线程，共享其附属进程的所有资源。



【3】进程和线程的区别（重点！！！！）

1. **进程是资源分配的最小单位，线程是进程运行的最小单位。**
2. 进程与进程之间的用户空间是相互独立的，内核空间是所有进程共享的。

   因此，进程之间要实现数据传输，需要引入进程间通信机制（IPC）

   同一个进程下的线程，共享其附属进程的所有资源，所以线程之间不需要通信机制。

   因此，线程之间的通信需要注意同步互斥。
3. 创建子进程需要拷贝父进程的所有资源，创建多线程不需要，因为本身共享进程的资源。并且进程的上下文切换比线程的切换更耗时。

   所以多线程的效率比多进程高。
4. 进程之间相互独立，所以多进程的稳定性比多线程高。
5. 以进程为单位分配资源，所以多进程的资源量肯定比多线程高。

【4】线程常用的函数

```
Compile and link with -pthread.
gcc 编译的时候要加上 -pthread 选项;
gcc 01_pthread_create.c -pthread
```

1）pthread_create

```
功能：创建一个线程;
头文件：
        #include <pthread.h>

        int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                           void *(*start_routine) (void *), void *arg);
参数：
    pthread_t *thread：该指针指向的内存空间中会存储创建成功的线程的id号;
    pthread_attr_t *attr：线程的属性，采用默认属性，填NULL;
    void *(*sr) (void *)：回调函数，该函数指针可以指向线程执行体，返回值是void*类型，参数列表是void*类型的函数。例如：
                                void* callBack(void* arg)
                                {
                                }
                                数据类型 *指针变量名;    int *ptr;
    void *arg：传递给回调函数的参数;
返回值：
    成功，返回返回0;
    失败，返回错误码，非0;
```

注意：

1. 线程是依附于进程的，默认情况下如果进程结束，则该进程下的附属线程都会强制退出。
2. 分支线程退出不会倒置进程退出，所以不会强制让其他线程退出。
3. 创建问分支线程后，主线程和分支线程谁先运行不确定，根据时间片以及cpu的调度来决定;

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int a = 10;

//线程的执行体--->制定新创建的线程要执行什么任务
void* callBack(void* arg)   //void* arg =(void*) &b;
{
    //printf("解引用 %d\n",*arg); //错误的，因为arg是void*类型，所以解引用后不知道要访问几个字节
    //printf("解引用 %d\n",(int)*arg); //错误的，因为会先运行*arg, arg是void*类型，所以解引用后不知道要访问几个字节
    printf("解引用 %d\n",*(int*)arg);

    a = 20;
    while(1)
    {
        printf("this is pthread \n" );
        sleep(1);
    }
```

```
    return NULL;
}

int main(int argc, const char *argv[])
{
    int b = 99;

    //创建一个分支线程
    pthread_t tid;
    if(pthread_create(&tid, NULL, callBack, (void*)&b) != 0)
    {
        perror("pthread_create");
        return -1;
    }
    printf("分支线程创建成功\n");

    while(1)
    {
        printf("this is main function b=%d\n", b);
        sleep(1);
    }

    return 0;   //主线程退出，则进程退出，
}
```

作业

> 1. 修改标准IO时候写的时钟代码，要求输入'q'后，能够退出该程序。
>
> 思考题：
>
> 2. 要求创建两个线程，以及一个全局变量，char str[] = "123456";要求如下：
>
> 1）一个线程专门用于打印str;
>
> 2）另外一个线程专门用于倒置str字符串，不使用辅助数组。
>
> 3）要求打印出来的结果必须是123456或者654321，不能出现乱序情况。

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdlib.h>

void* callBack(void* arg)
{
    char c = 0;
    while(1)
    {
        scanf("%c", &c);
        while(getchar() != '\n');   //循环吸收垃圾字符到'\n'位置为止;

        if('q' == c)
            exit(0);    //退出进程
    }
```

```
}

int main(int argc, const char *argv[])
{
    pthread_t tid;
    if(pthread_create(&tid, NULL, callBack, NULL) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    time_t t2;
    while(1)
    {
        system("clear");    //让C代码执行shell指令

        time(&t2);
    //  printf("%ld\n", t2);

        struct tm* info = NULL;
        info = localtime(&t2);

        printf("%d-%02d-%02d %02d-%02d-%02d\r", \
                info->tm_year+1900, info->tm_mon+1, info->tm_mday, \
                info->tm_hour, info->tm_min, info->tm_sec);
        fflush(stdout);

        sleep(1);
    }

    return 0;
}
```

2）pthread_exit

```
功能：退出线程；
头文件：
        #include <pthread.h>

        void pthread_exit(void *retval);
参数：
    void *retval：该参数可以返回线程退出状态值，该状态值会被pthread_join函数接收；
                如果不想返回线程退出状态值可以填NULL；
```

线程退出后，线程所占用的一小部分资源没有被回收，类似僵尸进程的状态，可以用pthread_join函数接收。

3）pthread_join

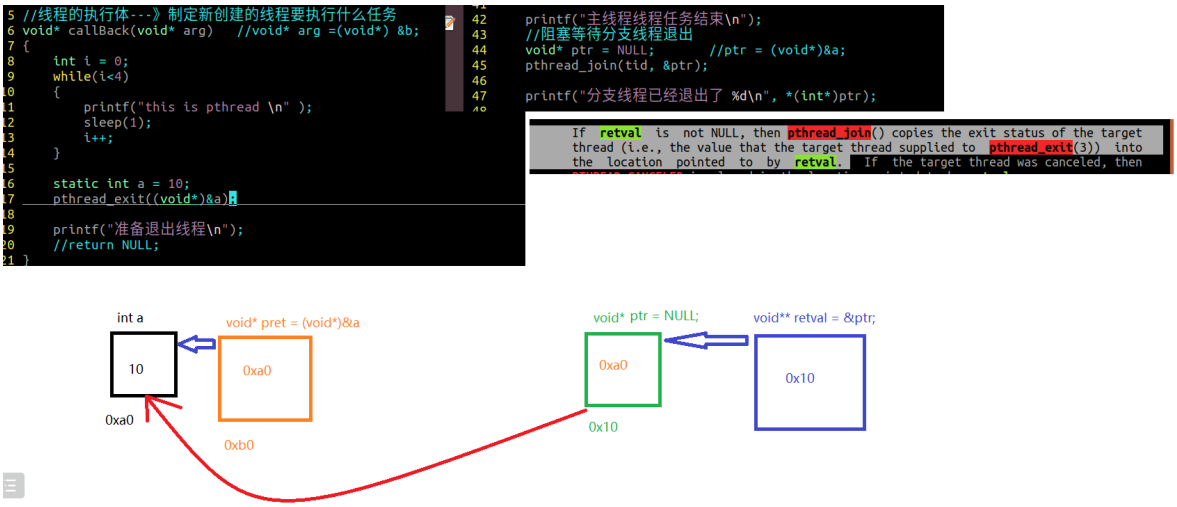
功能：阻塞等待，指定分支线程退出，并回收分支线程的资源；
头文件：

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

参数：
pthread_t thread：指定要回收的线程的tid号；
void **retval：该指针指向的内存空间中会存储线程退出状态值，即pthread_exit的参数；
如果不想接收线程的退出状态值，可以填NULL；
返回值：
成功，返回返回0；
失败，返回错误码，非0；

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

//线程的执行体--->制定新创建的线程要执行什么任务
void* callBack(void* arg)   //void* arg =(void*) &b;
{
    int i = 0;
    while(i<4)
    {
        printf("this is pthread \n" );
        sleep(1);
        i++;
    }

    static int a = 10;
    pthread_exit((void*)&a);

    printf("准备退出线程\n");
    //return NULL;
}

int main(int argc, const char *argv[])
{
    //创建一个分支线程
    pthread_t tid;
    if(pthread_create(&tid, NULL, callBack, NULL) != 0)
    {
        perror("pthread_create");
        return -1;
    }
    printf("分支线程创建成功\n");

    int i = 0;
    while(i<2)
    {
        printf("this is main function\n");
        sleep(1);
        i++;
    }

    printf("主线程线程任务结束\n");
    //阻塞等待分支线程退出
```

```
    void* ptr = NULL;       //ptr = (void*)&a;
    pthread_join(tid, &ptr);

    printf("分支线程已经退出了 %d\n", *(int*)ptr);

    return 0;   //主线程退出，则进程退出，
}
```



4) pthread_detach

功能：分离线程,分离后的线程退出后，会自动回收自己的资源. 且pthread_join函数无法回收该线程资源，不会阻塞；
头文件：

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

参数：
    pthread_t thread：指定要分离哪个线程；
返回值：
    成功，返回返回0；
    失败，返回错误码，非0；

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

//线程的执行体---》制定新创建的线程要执行什么任务
void* callBack(void* arg)   //void* arg =(void*) &b;
{
    int i = 0;
    while(i<4)
    {
        printf("this is pthread \n" );
        sleep(1);
        i++;
    }

    static int a = 10;
    pthread_exit((void*)&a);

    printf("准备退出线程\n");
```

```
    //return NULL;
}

int main(int argc, const char *argv[])
{
    //创建一个分支线程
    pthread_t tid;
    if(pthread_create(&tid, NULL, callBack, NULL) != 0)
    {
        perror("pthread_create");
        return -1;
    }
    //分离线程
    pthread_detach(tid);
    printf("分支线程创建成功\n");

    int i = 0;
    while(i<2)
    {
        printf("this is main function\n");
        sleep(1);
        i++;
    }

    printf("主线程线程任务结束\n");

    pthread_join(tid, NULL);    //分离tid线程，
    //pthread_join函数不在阻塞，且无法回收线程资源。

    printf("分支线程已经退出\n");

    return 0;   //主线程退出，则进程退出，
}
```

> 要求创建两个线程，以及一个全局变量，char str[] = "123456";要求如下：
>
> 1）一个线程专门用于打印str;
>
> 2）另外一个线程专门用于倒置str字符串，不使用辅助数组。
>
> 3）要求打印出来的结果必须是123456或者654321，不能出现乱序情况。

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

//临界资源
char str[] = "1234567";
int flag = 0;       //如果为0，则打印，如果为1则倒置

void* callBack1(void* arg)
{
    while(1)
    {
        /***临界区*********/
        if(0 == flag)
```

```
        {
            printf("%s\n", str);
            flag = 1;
        }
    }
    pthread_exit(NULL);
}

void* callBack2(void* arg)
{
    int i = 0;
    char temp = 0;
    while(1)
    {
        if(1 == flag)
        {
            for(i=0; i<strlen(str)/2; i++)
            {
                temp = str[i];
                str[i] = str[strlen(str)-1-i];
                str[strlen(str)-1-i] = temp;
            }
            flag = 0;
        }
    }
    pthread_exit(NULL);
}

int main(int argc, const char *argv[])
{
    pthread_t tid1,tid2;
    //创建一个线程用于打印
    if(pthread_create(&tid1, NULL, callBack1, NULL) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    //创建一个线程用于倒置
    if(pthread_create(&tid2, NULL, callBack2, NULL) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    //阻塞等待分支线程退出
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```

【5】线程的同步互斥机制（重点！！！）

临界资源（共享资源）：多个任务并发执行的时候，访问同一个资源，我们将这个资源称之为临界资源。

临界区：访问临界资源的代码，称之为临界区；

线程之间，如果要进行通信，需要引入线程的同步互斥机制，避免产生竞态，保证任何一个时刻，只有一个线程处理临界资源。保证临界区的完整性

线程同步互斥机制：

1. 互斥锁
2. 条件变量
3. 信号量

同步和异步的区别

1. 同步：当前线程执行任务会收到其他线程的影响，如果其他线程进入临界区，则当前线程会阻塞。如果当前线程进入临界区，其他线程会阻塞。

   例如：互斥锁，条件变量，以及信号量的初始值为1的情况均是同步情况。

2. 异步：当前线程执行的任务不会受到其他线程的影响，只会收到cpu调度机制的影响。

当多个线程需要访问同一个资源的时候，他们需要以某种特定的顺序来保证该资源只有在某一个特定时刻只能被一个线程访问，如果使用异步，程序的结果将不可预料。因此，在这种情况下，就必须对数据进行同步，即限制只能有一个进程访问资源，其他线程必须等待。

1. 互斥锁

1）工作原理

1. 对于要访问共享资源的线程，在访问共享线程之前，先做申请互斥锁操作。

   若申请互斥锁操作成功，则执行临界区代码，直到退出临界区，解开互斥锁。

   若申请互斥锁操作失败，则代表互斥锁资源被别的线程占用，则线程进入休眠等待互斥锁解开。

2. 互斥锁会将临界区锁住，保证临界区代码的完整性。

2）pthread_mutex_init

功能：创建并初始化互斥锁；
头文件：

```
#include <pthread.h>

pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex, const  pthread_mutexattr_t
 *mutexattr);
```

参数：
pthread_mutex_t *mutex：该指针指向的内存空间会存储申请后的互斥锁；
pthread_mutexattr_t *mutexattr：互斥锁属性，可以指定该互斥锁是用于线程的还是用于进程的。
一般填NULL，代表默认属性,且用户线程；
返回值：
成功，返回返回0；
失败，返回错误码，非0；

3）pthread_mutex_destroy

功能：销毁互斥锁
头文件：

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

参数：
pthread_mutex_t *mutex：指定要销毁的互斥锁的首地址；
返回值：
成功，返回0；
失败，返回非0；

4）pthread_mutex_lock

功能：上锁；
头文件：

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

参数：
pthread_mutex_t *mutex：指定要上锁的互斥锁的首地址；
返回值：
成功，返回0；
失败，返回非0；

5）pthread_mutex_unlock

```
功能：上锁；
头文件：
        #include <pthread.h>

        int pthread_mutex_unlock(pthread_mutex_t *mutex);
参数：
    pthread_mutex_t  *mutex：指定要解锁的互斥锁的首地址；
返回值：
    成功，返回0；
    失败，返回非0；
```

6）死锁

拥有互斥锁资源的任务， **没有释放锁资源**：

1. 持有互斥锁的任务，异常退出，没有释放锁资源
2. 同一互斥锁重复上锁
3. 互斥锁交叉嵌套

7）代码示例

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

//临界资源
char str[] = "1234567";
int flag = 0;        //如果为0，则打印，如果为1则倒置

void* callBack1(void* arg)
{
    while(1)
    {
        /***临界区**********/
        if(0 == flag)
        {
            printf("%s\n", str);
            flag = 1;
        }
    }
    pthread_exit(NULL);
}

void* callBack2(void* arg)
{
    int i = 0;
    char temp = 0;
    while(1)
    {
        if(1 == flag)
        {
```

```
            for(i=0; i<strlen(str)/2; i++)
            {
                temp = str[i];
                str[i] = str[strlen(str)-1-i];
                str[strlen(str)-1-i] = temp;
            }
            flag = 0;
        }
    }
    pthread_exit(NULL);
}

int main(int argc, const char *argv[])
{
    pthread_t tid1,tid2;
    //创建一个线程用于打印
    if(pthread_create(&tid1, NULL, callBack1, NULL) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    //创建一个线程用于倒置
    if(pthread_create(&tid2, NULL, callBack2, NULL) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    //阻塞等待分支线程退出
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```

练习

> 要求用两个线程拷贝一张图片，一个线程拷贝后半部分，另外一个线程拷贝前半部分。要求用文件IO实现。
>
> 不允许使用sleep函数

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


//需要传入到线程执行体的参数
struct msg
{
```

```
    int fd_r;
    int fd_w;
    off_t size;
};

//互斥锁
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


//拷贝前半部分
void* callBack_front(void* arg)     //void* arg = (void*)&info;
{
    struct msg info = *(struct msg*)arg;
    int fd_r = info.fd_r;
    int fd_w = info.fd_w;
    off_t size = info.size;

    /*******临界区***********/
    pthread_mutex_lock(&mutex);     //上锁

    //先将文件偏移量修改到开头
    lseek(fd_r, 0, SEEK_SET);
    lseek(fd_w, 0, SEEK_SET);

    int i = 0;
    char c;
    for(i=0; i<size/2; i++)
    {
        read(fd_r, &c, 1);
        write(fd_w, &c, 1);
    }

    printf("前半部分拷贝完毕\n");

    pthread_mutex_unlock(&mutex);   //解锁
    /*******临界区***********/

    pthread_exit(NULL);
}

void* callBack_end(void* arg)
{
    struct msg info = *(struct msg*)arg;
    int fd_r = info.fd_r;
    int fd_w = info.fd_w;
    off_t size = info.size;

    /*******临界区***********/
    pthread_mutex_lock(&mutex);     //上锁

    //先将文件偏移量修改到中间
    lseek(fd_r, size/2, SEEK_SET);
    lseek(fd_w, size/2, SEEK_SET);

    int i = 0;
    char c;
    for(i=size/2; i<size; i++)
    {
```

```
        read(fd_r, &c, 1);
        write(fd_w, &c, 1);
    }

    printf("后半部分拷贝完毕\n");
    pthread_mutex_unlock(&mutex);   //解锁
    /*******临界区***********/

    pthread_exit(NULL);

}



int main(int argc, const char *argv[])
{
    /*
    //创建并初始化互斥锁
    if(pthread_mutex_init(&mutex, NULL) != 0)
    {
        perror("pthread_mutex_init");
        return -1;
    }
    */


    //以读的方式打开源文件
    int fd_r = open("./1.png", O_RDONLY);
    if(fd_r < 0)
    {
        perror("open");
        return -1;
    }

    //以写的方式打开目标文件
    int fd_w = open("./copy.png", O_WRONLY|O_CREAT|O_TRUNC, 0664);
    if(fd_w < 0)
    {
        perror("open");
        return -1;
    }

    off_t size = lseek(fd_r, 0, SEEK_END);
    lseek(fd_r, 0, SEEK_SET);

    struct msg info;
    info.fd_r = fd_r;
    info.fd_w = fd_w;
    info.size = size;

    pthread_t tid_front, tid_end;
    //创建线程拷贝前半部分
    if(pthread_create(&tid_front, NULL, callBack_front, (void*)&info) != 0)
    {
        perror("pthread_create");
        return -1;
    }
    //创建线程拷贝后半部分
```

```
    if(pthread_create(&tid_end, NULL, callBack_end, (void*)&info) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    //主线程阻塞等待分支线程退出
    pthread_join(tid_front, NULL);
    pthread_join(tid_end, NULL);

    //销毁互斥锁
    pthread_mutex_destroy(&mutex);

    close(fd_r);
    close(fd_w);

    return 0;
}
```

2. 信号量（又称之为信号灯）

1）工作原理

1. 对于访问共享资源的线程，都去执行申请信号量的操作。

   当信号量的值大于0，则申请信号量成功，信号量的值-1；

   当信号量的值等于0，则申请信号量失败，线程进入休眠等待阶段。

2. PV操作：实现线程、进程同步互斥的有效方式

   P操作：申请信号量，信号量的值会-1，-1操作

   V操作：释放信号量，信号量的值为+1，+1操作

3. 互斥锁又称之为二值信号量（要么为0，要么为1），只允许一个线程进入临界区；

4. 信号量运行多个线程同时进入临界区，主要看信号量的初始值为多少。

2）sem_init

```
功能：信号量的创建，以及初始化；
头文件：
        #include <semaphore.h>

        int sem_init(sem_t *sem, int pshared, unsigned int value);
参数：
    sem_t *sem：该指针指向的内存空间，会存储申请好的信号量；
    int pshared：共享标识;用来决定信号灯是用于进程的同步互斥还是线程的同步互斥
                填0：用于线程
                非0：用于进程；
    unsigned int value：信号灯的初始值；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

3）sem_wait（P操作）

```
功能：申请信号量，即P操作，-1操作；
    当信号量的值大于0，则申请信号量成功，信号量的值-1；
    当信号量的值等于0，则申请信号量失败，线程进入休眠等待阶段；
头文件：
        #include <semaphore.h>

        int sem_wait(sem_t *sem);
参数：
    sem_t *sem：指定对哪个信号量做P操作；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

4）sem_post（V操作）

```
功能：释放信号量，V操作，+1；
头文件：
        #include <semaphore.h>

        int sem_post(sem_t *sem);
参数：
    sem_t *sem：指定要做V操作的信号量的首地址；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

5）sem_destroy

```
功能：销毁信号量；
头文件：
        #include <semaphore.h>

        int sem_destroy(sem_t *sem);
参数：
    sem_t *sem：
```

6）代码示例

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

```c
#include <semaphore.h>


//需要传入到线程执行体的参数
struct msg
{
    int fd_r;
    int fd_w;
    off_t size;
};

//信号量
sem_t sem;

//拷贝前半部分
void* callBack_front(void* arg)     //void* arg = (void*)&info;
{
    struct msg info = *(struct msg*)arg;
    int fd_r = info.fd_r;
    int fd_w = info.fd_w;
    off_t size = info.size;

    /*******临界区***********/
    sem_wait(&sem);     //P操作

    //先将文件偏移量修改到开头
    lseek(fd_r, 0, SEEK_SET);
    lseek(fd_w, 0, SEEK_SET);

    int i = 0;
    char c;
    for(i=0; i<size/2; i++)
    {
        read(fd_r, &c, 1);
        write(fd_w, &c, 1);
    }

    printf("前半部分拷贝完毕\n");

    sem_post(&sem);

    /*******临界区***********/

    pthread_exit(NULL);
}

void* callBack_end(void* arg)
{
    struct msg info = *(struct msg*)arg;
    int fd_r = info.fd_r;
    int fd_w = info.fd_w;
    off_t size = info.size;

    /*******临界区***********/
    sem_wait(&sem);     //p操作

    //先将文件偏移量修改到中间
    lseek(fd_r, size/2, SEEK_SET);
```

```
    lseek(fd_w, size/2, SEEK_SET);

    int i = 0;
    char c;
    for(i=size/2; i<size; i++)
    {
        read(fd_r, &c, 1);
        write(fd_w, &c, 1);
    }

    printf("后半部分拷贝完毕\n");
    sem_post(&sem);
    /********临界区***********/

    pthread_exit(NULL);

}



int main(int argc, const char *argv[])
{

    //创建并初始化信号量
    if(sem_init(&sem, 0, 1) < 0)
    {
        perror("sem_init");
        return -1;
    }

    //以读的方式打开源文件
    int fd_r = open("./1.png", O_RDONLY);
    if(fd_r < 0)
    {
        perror("open");
        return -1;
    }

    //以写的方式打开目标文件
    int fd_w = open("./copy.png", O_WRONLY|O_CREAT|O_TRUNC, 0664);
    if(fd_w < 0)
    {
        perror("open");
        return -1;
    }

    off_t size = lseek(fd_r, 0, SEEK_END);
    lseek(fd_r, 0, SEEK_SET);

    struct msg info;
    info.fd_r = fd_r;
    info.fd_w = fd_w;
    info.size = size;

    pthread_t tid_front, tid_end;
    //创建线程拷贝前半部分
    if(pthread_create(&tid_front, NULL, callBack_front, (void*)&info) != 0)
    {
```

```
        perror("pthread_create");
        return -1;
    }
    //创建线程拷贝后半部分
    if(pthread_create(&tid_end, NULL, callBack_end, (void*)&info) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    //主线程阻塞等待分支线程退出
    pthread_join(tid_front, NULL);
    pthread_join(tid_end, NULL);

    //销毁信号量
    sem_destroy(&sem);

    close(fd_r);
    close(fd_w);

    return 0;
}
```

练习

> 1.用信号量实现，创建两个线程A、B，以及一个全局变量char str[] = "123456"，在不考虑线程退出的情况下，要求如下：
>
> 1. A线程 循环 打印str字符串。
> 2. B线程 循环 倒置str字符串，不使用辅助数组。注意是循环倒置，要把字符串倒过来，倒回去。
> 3. 要求A线程打印出来的结果是有序的，例如："123456" 或者 "654321"，不允许出现"623451",,,等无序情况
> 4. 要求必须 打印线程执行一次，倒置线程执行一次，再打印线程执行一次，倒置线程执行一次，如此往复。
>
>   即临界区运行顺序为：A线程B线程A线程B线程ABABABAB

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>

//临界资源
char str[] = "1234567";

sem_t sem1,sem2;

void* callBack1(void* arg)
{
    while(1)
    {
        /***临界区**********/
        sem_wait(&sem1);    //P操作

        printf("%s\n", str);
```

```
        sleep(1);

        sem_post(&sem2);    //V操作

        /***临界区*********/
    }
    pthread_exit(NULL);
}

void* callBack2(void* arg)
{
    int i = 0;
    char temp = 0;
    while(1)
    {
        /***临界区*********/
        sem_wait(&sem2);    //P操作

        for(i=0; i<strlen(str)/2; i++)
        {
            temp = str[i];
            str[i] = str[strlen(str)-1-i];
            str[strlen(str)-1-i] = temp;
        }

        sem_post(&sem1);    //V操作
        /***临界区*********/
    }
    pthread_exit(NULL);
}

int main(int argc, const char *argv[])
{
    //信号量的创建以及初始化
    if(sem_init(&sem1, 0, 1) < 0)
    {
        perror("sem_init");
        return -1;
    }
    if(sem_init(&sem2, 0, 0) < 0)
    {
        perror("sem_init");
        return -1;
    }


    pthread_t tid1,tid2;
    //创建一个线程用于打印
    if(pthread_create(&tid1, NULL, callBack1, NULL) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    //创建一个线程用于倒置
    if(pthread_create(&tid2, NULL, callBack2, NULL) != 0)
    {
        perror("pthread_create");
```

```
        return -1;
    }


    //阻塞等待分支线程退出
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    //销毁互斥锁
    sem_destroy(&sem1);
    sem_destroy(&sem2);

    return 0;
}
```

作业

> 用信号量的方式，创建两个线程 A B
>
> 1. A线程读取文件中的内容
> 2. B线程打印A读取到的内容到终端，
> 3. 全部打印完毕后，结束进程；
> 4. 现象类似cat一个文件

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

#define N 128

//信号量
sem_t sem1, sem2;

void* callback_read(void* arg)      //void* arg = (void*)buf;
{
    char* ptr = (char*)arg;

    FILE* fp = fopen("./01_pthread_create.c", "r");
    if(NULL == fp)
    {
        perror("fopen");
        pthread_exit(NULL);
    }

    //读取文件中的数据
    while(1)
    {
        sem_wait(&sem1);

        bzero(ptr, N);
        if(fgets(ptr, N, fp) == NULL)
        {
            sem_post(&sem2);
            break;
```

```
        }
        sem_post(&sem2);
    }

    printf("文件读取完毕\n");
    pthread_exit(NULL);
}

void* callback_print(void* arg)     //void* arg = (void*)buf;
{
    char* ptr = (char*)arg;
    while(1)
    {
        sem_wait(&sem2);

        //由于读线程每次读取之前都会清空buf字符串，
        //只要判断buf中有无数据，如果没有数据则说明文件读完了
        if(strlen(ptr) == 0)
            break;

        printf("%s", ptr);
        fflush(stdout);

        sem_post(&sem1);
    }
    pthread_exit(NULL);
}

int main(int argc, const char *argv[])
{
    //初始化信号量
    if(sem_init(&sem1, 0, 1) < 0)
    {
        perror("sem_init");
        return -1;
    }
    if(sem_init(&sem2, 0, 0) < 0)
    {
        perror("sem_init");
        return -1;
    }


    char buf[N] = "";

    pthread_t tid_r, tid_p;
    //创建一个线程读取文件
    if(pthread_create(&tid_r, NULL, callback_read, (void*)buf) != 0)
    {
        perror("pthread_create");
        return -1;
    }

    //创建另外一个线程打印读取到的内容
    if(pthread_create(&tid_p, NULL, callback_print, (void*)buf) != 0)
    {
        perror("pthread_create");
        return -1;
```

```
    }

    //阻塞等待分支线程退出
    pthread_join(tid_r, NULL);
    pthread_join(tid_p, NULL);

    sem_destroy(&sem1);
    sem_destroy(&sem2);

    return 0;
}
```

3. 条件变量

1）工作原理

1. 将不访问共享资源的线程直接休眠，并设置一个唤醒条件，其他线程需要通过该唤醒条件才能唤醒该线程。该唤醒条件就称之为条件变量
2. 如果线程需要访问共享资源的时候，则通过其他线程唤醒当前线程即可。

2）pthread_cond_init

```
功能：创建并初始化条件变量；
头文件：
        #include <pthread.h>

        pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
        int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t
*cond_attr);
参数：
    pthread_cond_t *cond：该指针指向的内存空间中，会存储创建并初始化后的条件变量；
    pthread_condattr_t *cond_attr：条件变量属性，可以指定该条件变量是用于线程的还是用于进
程的。
                                        一般填NULL，代表默认属性,且用户线程；
返回值：
    成功，返回返回0；
    失败，返回错误码，非0；
```

3）pthread_cond_destroy

```
功能：销毁条件变量；
头文件：
        #include <pthread.h>

        int pthread_cond_destroy(pthread_cond_t *cond);
参数：
    pthread_cond_t *cond：指定要销毁的条件变量；
返回值：
    成功，返回返回0；
    失败，返回错误码，非0；
```

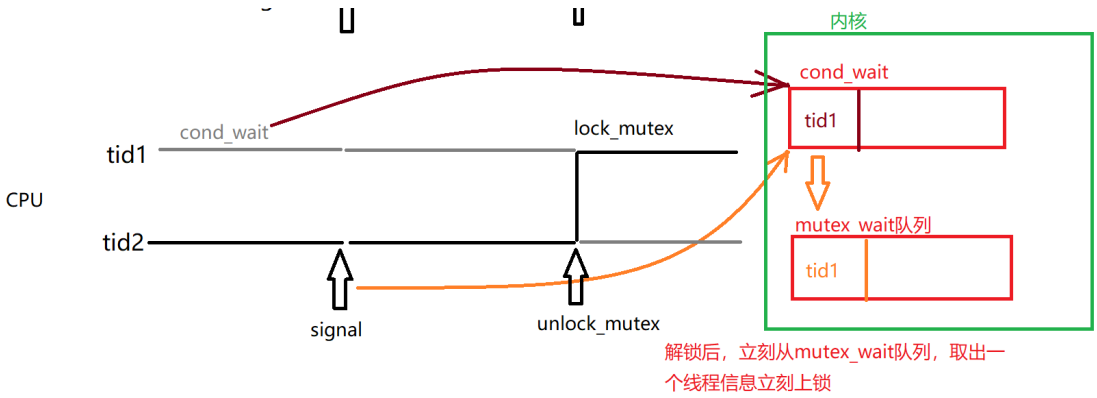
4）pthread_cond_wait

功能：让当前线程去休眠，并设置一个唤醒条件，即设置一个条件变量;我们一般称之为线程睡在该条件变量上。
头文件：

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

参数：
pthread_cond_t *cond：指定要设置的唤醒条件，传入一个条件变量的首地址；
pthread_mutex_t *mutex：互斥锁;休眠之前指定要解开的互斥锁；
返回值：
成功，返回返回0；
失败，返回错误码，非0；

步骤：
1.打开互斥锁，同时设置唤醒条件，并让当前线程进入休眠等待阶段。（原子操作：当前任务不会被线程，进程调度机制打断。）
2.当其他线程唤醒该条件变量的时候，线程会从cond_wait队列移动到mutex_wait队列，等待互斥锁解锁。
3.等互斥锁解锁后，如果上锁成功，则继续从当前位置继续往后执行。
如果上锁失败，则从mutex_wait队列重新回到cond_wait队列，等待下一次的signal.



5）pthread_cond_signal

功能：唤醒睡在指定条件变量上的线程；
头文件：

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

参数：
pthread_cond_t *cond：；

练习

> 创建ABC三个线程，要求ABC线程顺序运行，不考虑退出条件。
>
> 提示：用三个条件变量

8. 进程间的通信机制（IPC）

进程与进程之间的用户空间是相互独立的，内核空间是所有进程共享的。

因此，进程之间要实现数据传输，需要引入进程间通信机制（IPC），所以IPC肯定是操控内核的。

【1】进程间通信方式（重点）

inter process communication 简称为 IPC

1. 传统的进程间通信方式
   1）无名管道　　pipe
   2）有名管道　　fifo
   3）信号　　signal

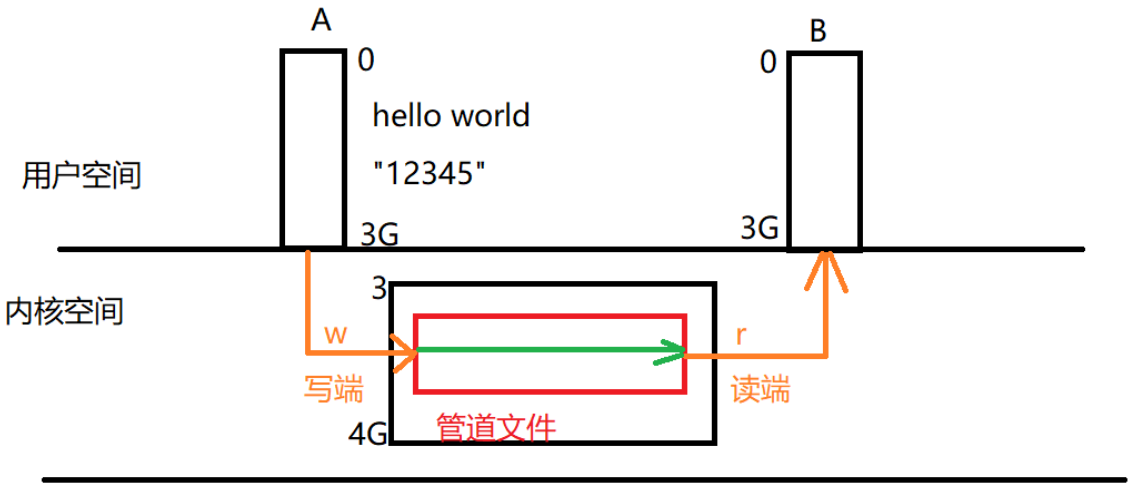
2. system V操作系统的IPC对象：**IPC对象！！**
   1）消息队列　　message queue
   2）共享内存　　shared memory
   3）信号灯集　　semaphore

3. 可用于跨主机传输
   1）套接字　　socket

【2】管道

1. 管道的原理

在进程的内核空间中（3~4G）空间中，会创建一个管道（该管道可以看做是一个特殊的文件），管道文件中的数据是直接保存在内核内存中的。



2. 管道特性

1. 管道可以看做是一个**特殊的文件**，一般文件存储在硬盘上，而管道文件存储在内核内存中。

2. 管道遵循先进先出的原则。
3. 对于管道的读操作是一次性的操作，如果对管道进行读操作，那么被读取的数据会从管道中删除。
4. 管道是一种半双工的通信方式。

> 单工：只能单向通信，A能发消息给B，B不能发消息给A.
>
> 半双工：同一时间只能单向通信。同一时间只能A发消息给B或者B发消息给A。
>
> 全双工：同一时间双方可以相互通信。

5. 管道的大小：65536byte = 64k。
6. 当从管道中读取数据的时候

   写端存在，没有关闭

   1）当管道中没有数据时候，read函数会阻塞；

   2）当管道中存储了10个数据，read函数读取5个，实际读取5个

   3）当管道中存储了5个数据，read函数读取10个，实际读取5个

   当所有写端均被关闭（父子进程均有写端）

   1）从管道中读取数据，先会正常的将管道中的所有数据读取完毕，管道中即使没有数据，read函数不会阻塞，且立即返回0;

7. 当向管道中写入数据的时候

   读端存在，没有关闭

   1）当管道写满后，写函数会阻塞。

   当所有读端均被关闭

   1）当所有读端均关闭后，只要向管道中写入数据，无论管道满还是不满，都会造成管道破裂，调用write函数的进程回收到一个管道破裂信号。

   2）管道破裂信号会导致进程退出：SIGPIPE

3. 无名管道（pipe）

1）无名管道的特性

1. 无名管道是没有名字的管道，即在 **文件系统中不可见的管道文件。**
2. 无名管道只能用于具有 **亲缘关系** 的进程间通信。（例如：父子进程，爷孙进程，叔侄进程等等）
3. 无名管道的读写，只能使用文件IO，例如read,write。但是不能使用lseek函数。
4. 当管道的读写端均关闭后，管道的内存空间会自动释放。

为什么无名管道只能用于具有 亲缘关系 的进程间通信?

由于无名管道在文件系统中不可见，所以两个没有亲缘关系的进程无法拿到同一根管道的读写端，所以无法通信。

因为子进程会克隆父进程的文件描述符表，所以子进程和父进程可以拿到同一根无名管道的读写端，所以可以通信。

2）pipe

```
功能：创建一个无名管道，同时会打开管道的读写端；
头文件：
        #include <unistd.h>

        int pipe(int pipefd[2]);
参数：
    int pipefd[2]：需要传入一个int类型数组，且数组的容量为2；
                    函数运行完毕后，该指针指向的数组中会存储两个文件描述符；
                    pipefd[0]：读端
                    pipefd[1]：写端；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    //由于父子进程如果想要通信，必须读写同一根管道
    //所以利用子进程会克隆父进程的文件描述符表的原理
    //我们在fork之前创建这个无名管道

    int pfd[2];
    if(pipe(pfd) < 0)
    {
        perror("pipe");
        return -1;
    }
    printf("无名管道创建成功 r:%d w:%d\n",pfd[0], pfd[1]);

    char buf[128] = "";
    ssize_t res = 0;

    pid_t pid = fork();
    if(pid > 0)
    {
        //父进程中运行
        printf("this is parent %d\n", getpid());
        while(1)
        {
            bzero(buf, sizeof(buf));
            scanf("%s", buf);
            getchar();

            //将数据写入到管道中
            res=write(pfd[1], buf, sizeof(buf));
            if(res < 0)
            {
                perror("write");
                return -1;
            }
```

```
            printf("写入成功 res=%ld\n", res);
        }
    }
    else if(0 == pid)
    {
        //子进程中运行
        printf("this is child %d\n", getpid());
        while(1)
        {
            bzero(buf, sizeof(buf));
            //读取管道中的数据，当管道中没有数据的时候read函数阻塞
            res = read(pfd[0], buf, sizeof(buf));
            if(res < 0)
            {
                perror("read");
                return -1;
            }
            printf("读取数据成功:%ld: %s\n", res, buf);
        }

    }
    else
    {
        perror("fork");
        return -1;
    }

    return 0;
}
```

作业

> 创建父子进程，实现父子进程的通话。
>
> 1）父进程先发送一句话给子进程，子进程接收打印。
>
> 2）子进程发送与句话给父进程，父进程接收后打印。
>
> 3）重复1）2）步骤即可。
>
> 4）当父进程或者子进程发送quit后，父子进程均要结束。

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, const char *argv[])
{
    //由于父子进程如果想要通信，必须读写同一根管道
    //所以利用子进程会克隆父进程的文件描述符表的原理
    //我们在fork之前创建这个无名管道

    int pfd1[2];            //父发送给子，子读
    if(pipe(pfd1) < 0)
```

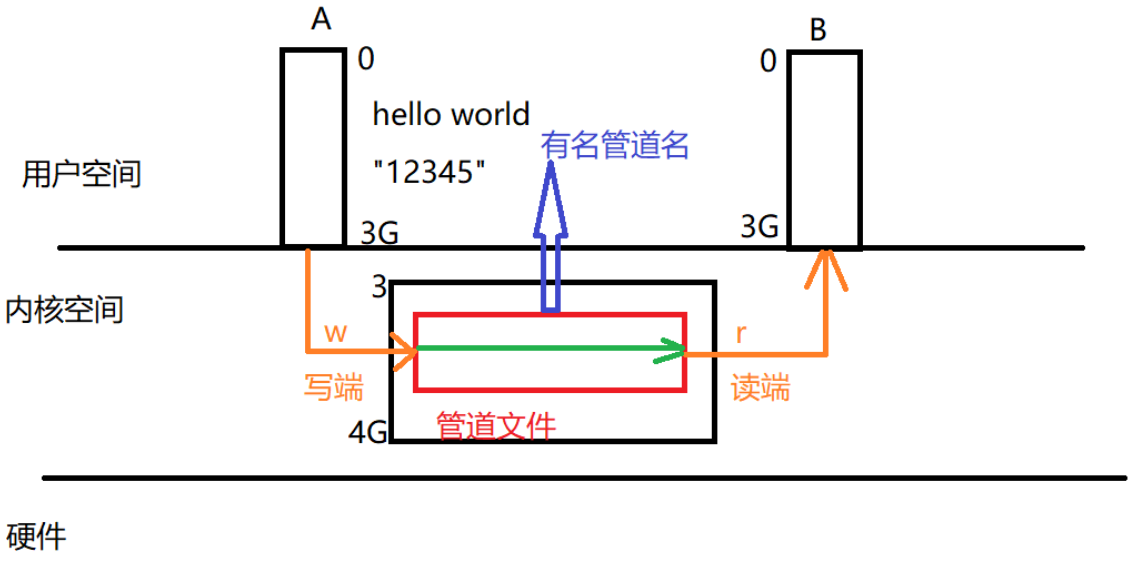
```
{
    perror("pipe");
    return -1;
}
printf("无名管道创建成功 r:%d w:%d\n",pfd1[0], pfd1[1]);

int pfd2[2];            //子发给父，父读取
if(pipe(pfd2) < 0)
{
    perror("pipe");
    return -1;
}
printf("无名管道创建成功 r:%d w:%d\n",pfd2[0], pfd2[1]);


char buf[128] = "";
ssize_t res = 0;

pid_t pid = fork();
if(pid > 0)
{
    //父进程中运行
    printf("this is parent %d\n", getpid());

    //关闭不需要的文件描述符
    close(pfd1[0]);
    close(pfd2[1]);

    while(1)
    {
        //父进程写入管道
        printf("父进程说>>>");
        bzero(buf, sizeof(buf));
        scanf("%s", buf);
        getchar();

        //将数据写入到管道中
        res=write(pfd1[1], buf, sizeof(buf));
        if(res < 0)
        {
            perror("write");
            return -1;
        }
        printf("父进程写入成功 res=%ld\n", res);
        if(strcasecmp(buf, "quit") == 0)    //忽略大小写的比较
        {
            break;
        }



        //父进程从管道中读取
        bzero(buf, sizeof(buf));
        res = read(pfd2[0], buf, sizeof(buf));
        if(res < 0)
        {
            perror("read");
            return -1;
```

```
            }
            else if(0 == res)
            {
                printf("写端关闭\n");
                break;
            }
            printf("父进程接收到：%s\n", buf);
            if(strcasecmp(buf, "quit") == 0)    //忽略大小写的比较
            {
                break;
            }
        }

        wait(NULL);     //阻塞等待子进程退出

        //退出前，关闭文件描述符
        close(pfd1[1]);
        close(pfd2[0]);


    }
    else if(0 == pid)
    {
        //子进程中运行
        printf("this is child %d\n", getpid());

        //关闭不需要的文件描述符
        close(pfd1[1]);
        close(pfd2[0]);

        while(1)
        {
            //子进程从管道中赌球数据
            bzero(buf, sizeof(buf));
            //读取管道中的数据，当管道中没有数据的时候read函数阻塞
            res = read(pfd1[0], buf, sizeof(buf));
            if(res < 0)
            {
                perror("read");
                return -1;
            }
            printf("子进程接收到:%ld：%s\n", res, buf);
            if(strcasecmp(buf, "quit") == 0)    //忽略大小写的比较
            {
                break;
            }



            //子进程往管道中发送数据
            bzero(buf, sizeof(buf));

            printf("子进程说>>>");
            scanf("%s", buf);
            getchar();
            if(write(pfd2[1], buf, sizeof(buf)) < 0)
            {
```

```
                perror("write");
                return -1;
            }
            printf("子进程发送成功\n");
            if(strcasecmp(buf, "quit") == 0)     //忽略大小写的比较
            {
                break;
            }

        }
        //退出前，关闭文件描述符
        close(pfd1[0]);
        close(pfd2[1]);
    }
    else
    {
        perror("fork");
        return -1;
    }

    return 0;
}
```

4. 有名管道（fifo）

有名管道：顾名思义，就是管道文件的名字在文件系统中可见，但是数据依然存在于内核内存中，是不可见的。



1）有名管道的特性

1. 有名管道文件的名字在文件系统中可见。
2. 有名管道可以使用在 **无亲缘关系的** 进程间通信
3. 有名管道的读写，必须使用文件IO函数，但是不能使用lseek函数。
4. 有名管道使用的时候，需要手动open。
5. 当管道的读写端均关闭后，释放其在内核的内存空间。

2）创建有名管道文件

1. 用shell指令创建

```
mkfifo 有名管道路径以及名字
    mkfifo myfifo
```

2. 用mkfifo函数创建

```
功能：创建有名管道文件；
        #include <sys/types.h>
        #include <sys/stat.h>

        int mkfifo(const char *pathname, mode_t mode);
参数：
    char *pathname：指定要创建的有名管道路径以及名字；
    mode_t mode：the  permissions of the created file are (mode & ~umask).
                真实权限为，mode&~umask，如果想保留原有权限，则直接将umask清0；
返回值：
    成功，返回0；
    失败，返回-1;更新errno； 当errno ==17即文件已经存在的错误时，该错误允许存在
```

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>

int main(int argc, const char *argv[])
{
    umask(0);

    if(mkfifo("./fifo", 0777) < 0)
    {
        //printf("%d\n", errno);
        //管道文件已经存在的错误，是一个合法错误，不做处理，即errno==17的情况允许存在
        if(17 != errno)         {
            perror("mkfifo");
            return -1;
        }
    }
    printf("fifo create success\n");


    return 0;
}
```

01_mkfifo.c 20,23

3）有名管道的使用

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

        int open(const char *pathname, int flags);
参数:
    char *pathname:指定要打开的文件路径以及文件名;
    int flags:
        O_RDONLY:只读
        O_WRONLY
        O_RDWR
     ---以上三种必须包含一种------
        O_NONBLOCK   非阻塞方式打开
```

1. flags == O_RDONLY

   open函数会阻塞，直到有另外一个进程以写的方式打开同一根FIFO文件，解除阻塞。

2. flags == O_WRONLY

   open函数会阻塞，直到有另外一个进程以读的方式打开同一根FIFO文件，解除阻塞。

3. flags == O_RDWR

   open函数不阻塞，读写端均被打开

4. flags == O_RDONLY | O_NONBLOCK

   open函数不阻塞，open函数运行成功，读端被打开。

5. flags == O_WRONLY | O_NONBLOCK

   open函数不阻塞的，open函数运行失败，此时写端打开失败。

4）写端代码示例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    umask(0);

    if(mkfifo("./fifo", 0777) < 0)
    {
        //printf("%d\n", errno);
        //管道文件已经存在的错误，是一个合法错误，不做处理，即errno==17的情况允许存在
        if(17 != errno)         {
            perror("mkfifo");
            return -1;
        }
    }
    printf("fifo create success\n");

    //以只写的方式打开有名管道文件，open函数阻塞
```

```
    int fd = open("./fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open");
        return -1;
    }
    printf("FIFO open writeonly success\n");


    //写入数据
    char buf[128] = "";
    ssize_t res = 0;
    while(1)
    {
        printf("请输入>>>");
        fgets(buf, sizeof(buf), stdin);     //从终端获取数据
        //由于fgets会拿到'\n',且拿完'\n'后，就停止当前次的读取。
        //因为实际获取字符串的时候，我们是不需要'\n',
        //所以需要把'\n'修改成'\0'
        buf[strlen(buf)-1] = '\0';

        res = write(fd, buf, sizeof(buf));
        if(res < 0)
        {
            perror("write");
            return -1;
        }
        printf("成功发送res=%ld个字节\n", res);
    }

    close(fd);

    return 0;
}
```

5）读端代码示例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    umask(0);

    if(mkfifo("./fifo", 0777) < 0)
    {
        //printf("%d\n", errno);
        //管道文件已经存在的错误，是一个合法错误，不做处理，即errno==17的情况允许存在
```

```
        if(17 != errno)         {
            perror("mkfifo");
            return -1;
        }
    }
    printf("fifo create success\n");

    //以只读的方式打开有名管道文件，open函数阻塞
    int fd = open("./fifo", O_RDONLY);
    if(fd < 0)
    {
        perror("open");
        return -1;
    }
    printf("FIFO open readonly success\n");


    //读取数据
    char buf[128] = "";
    ssize_t res = 0;
    while(1)
    {
        bzero(buf, sizeof(buf));
        res = read(fd, buf, sizeof(buf));
        if(res < 0)
        {
            perror("read");
            return -1;
        }
        else if(0 == res)
        {
            printf("对端退出\n");
            break;
        }
        printf("res=%ld:%s\n", res, buf);
    }

    close(fd);
    return 0;
}
```

练习

> 创建AB进程，实现AB进程的通话。
>
> 1）A进程先发送一句话给B进程，B进程接收打印。
>
> 2）B进程发送与句话给A进程，A进程接收后打印。
>
> 3）重复1）2）步骤即可。
>
> 4）当A进程或者B进程发送quit后，AB进程均要结束。

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

```c
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    umask(0);

    if(mkfifo("./fifo", 0777) < 0)
    {
        //printf("%d\n", errno);
        //管道文件已经存在的错误，是一个合法错误，不做处理，即errno==17的情况允许存在
        if(17 != errno)         {
            perror("mkfifo");
            return -1;
        }
    }
    printf("fifo create success\n");

    if(mkfifo("./fifo1", 0777) < 0)
    {
        //printf("%d\n", errno);
        //管道文件已经存在的错误，是一个合法错误，不做
        if(17 != errno)         {
            perror("mkfifo");
            return -1;
        }
    }
    printf("fifo1 create success\n");


    //以只写的方式打开有名管道文件，open函数阻塞
    int fd = open("./fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open");
        return -1;
    }
    printf("FIFO open writeonly success\n");

    //以读的方式打开fifo1
    int fd1 = open("./fifo1", O_RDONLY);
    if(fd < 0)
    {
        perror("open");
        return -1;
    }
    printf("FIFO1 open writeonly success\n");



    //写入数据
    char buf[128] = "";
    ssize_t res = 0;
    while(1)
    {
        printf("请输入>>>");
```

```
        fgets(buf, sizeof(buf), stdin);     //从终端获取数据
        //由于fgets会拿到'\n',且拿完'\n'后，就停止当前次的读取。
        //因为实际获取字符串的时候，我们是不需要'\n',
        //所以需要把'\n'修改成'\0'
        buf[strlen(buf)-1] = '\0';

        res = write(fd, buf, sizeof(buf));
        if(res < 0)
        {
            perror("write");
            return -1;
        }
        printf("成功发送res=%ld个字节\n", res);

        //从fifo1管道中读取数据
        bzero(buf, sizeof(buf));
        res = read(fd1, buf, sizeof(buf));
        if(res < 0)
        {
            perror("read");
            return -1;
        }
        else if(0 == res)
        {
            printf("对端退出\n");
            break;
        }
        printf("res=%ld:%s\n", res, buf);

    }

    close(fd);
    close(fd1);

    return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    umask(0);

    if(mkfifo("./fifo", 0777) < 0)
    {
        //printf("%d\n", errno);
        //管道文件已经存在的错误，是一个合法错误，不做处理，即errno==17的情况允许存在
        if(17 != errno)         {
            perror("mkfifo");
```

```
            return -1;
        }
    }
    printf("fifo create success\n");

    if(mkfifo("./fifo1", 0777) < 0)
    {
        //printf("%d\n", errno);
        //管道文件已经存在的错误，是一个合法错误，不做处理，即errno==17的情况允许存在
        if(17 != errno)         {
            perror("mkfifo");
            return -1;
        }
    }
    printf("fifo1 create success\n");


    //以只读的方式打开有名管道文件，open函数阻塞
    int fd = open("./fifo", O_RDONLY);
    if(fd < 0)
    {
        perror("open");
        return -1;
    }
    printf("FIFO open readonly success\n");


    //以写的方式打开fifo1
    int fd1 = open("./fifo1", O_WRONLY);
    if(fd < 0)
    {
        perror("open");
        return -1;
    }
    printf("FIFO1 open writeonly success\n");




    //读取数据
    char buf[128] = "";
    ssize_t res = 0;
    while(1)
    {
        bzero(buf, sizeof(buf));
        res = read(fd, buf, sizeof(buf));
        if(res < 0)
        {
            perror("read");
            return -1;
        }
        else if(0 == res)
        {
            printf("对端退出\n");
            break;
        }
        printf("res=%ld:%s\n", res, buf);
```

```
        //往fd1中写入
        bzero(buf, sizeof(buf));
        printf("请输入>>>");
        fgets(buf, sizeof(buf), stdin);      //从终端获取数据
        buf[strlen(buf)-1] = '\0';

        res = write(fd1, buf, sizeof(buf));
        if(res < 0)
        {
            perror("write");
            return -1;
        }
        printf("成功发送res=%ld个字节\n", res);

    }

    close(fd);
    close(fd1);
    return 0;
}
```

作业

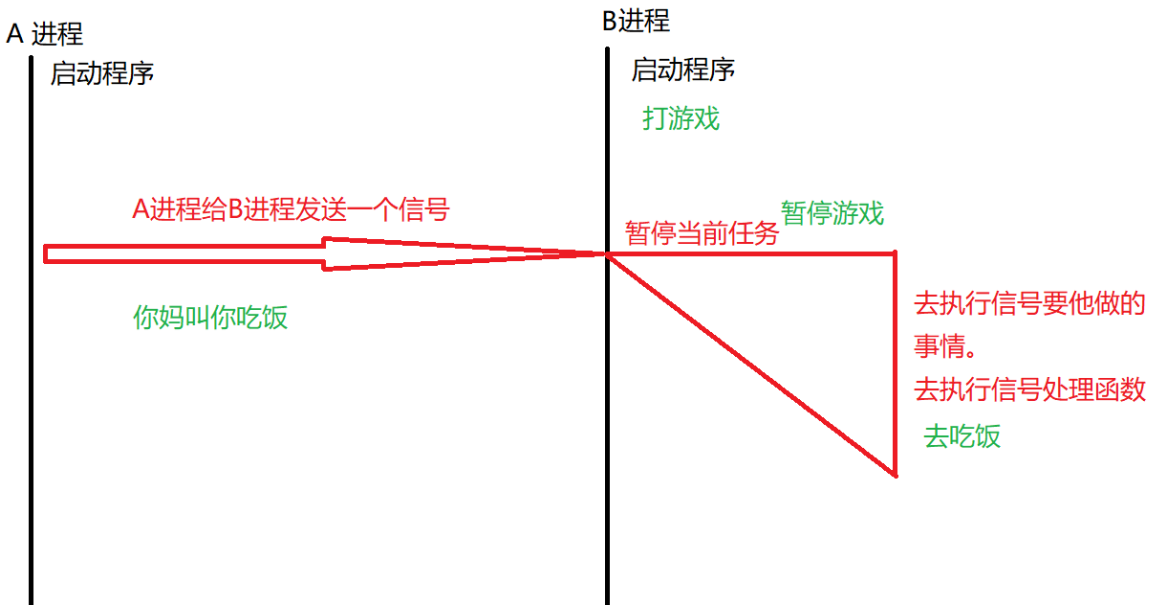
> 将上述练习修改成：AB进程能够随时收发形式
>
> 提示：用多线程，多进程，

【3】信号（signal）

1. 信号的概念

1）信号的原理

1. 信号是软件层次上对中断的一种模拟。
2. 是一种异步通信方式，AB进程各自运行各自的。



2）进程对信号的处理方式（重点！）

1. 执行默认操作（执行缺省操作）

   每个信号都规定了默认的信号处理函数，收到信号后，去执行默认的处理函数叫做执行默认操作。

2. 忽略信号

   对信号不做处理，但是有两个信号不能忽略 9) SIGKILL 19)SIGSTOP

3. 捕获信号

   将信号默认处理函数修改了，自己定义一个信号处理函数。当信号发生的时候，去执行信号的自定义处理函数。

   无法捕获 9) SIGKILL 19)SIGSTOP

3）常见的信号

```
kill -l          查看所有信号，    进程的64种死法

 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL      10) SIGUSR1
11) SIGSEGV      12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN      22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS       34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4   39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14  49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11  54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6   59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

1. **硬件能按出来的信号**

```
2) SIGINT           默认处理函数：退出进程    ctrl+c
3) SIGQUIT          默认处理函数：退出进程    ctrl+\
20) SIGTSTP         默认处理函数：挂起进程    ctrl+z
```

2. **无法被忽略、捕获的信号**

```
9) SIGKILL          默认处理函数：退出进程
19) SIGSTOP         默认处理函数：退出进程
```

3. **常见的信号**

```
11) SIGSEGV         段错误信号
13) SIGPIPE         管道破裂信号
14) SIGALRM         时钟信号，设置一个定时器；
17) SIGCHLD         子进程退出后，父进程会收到该信号；
```

4. **用户自定义处理函数的信号**

```
10) SIGUSR1          默认处理函数：退出进程
12) SIGUSR2          默认处理函数：退出进程
```

2. 信号相关的函数

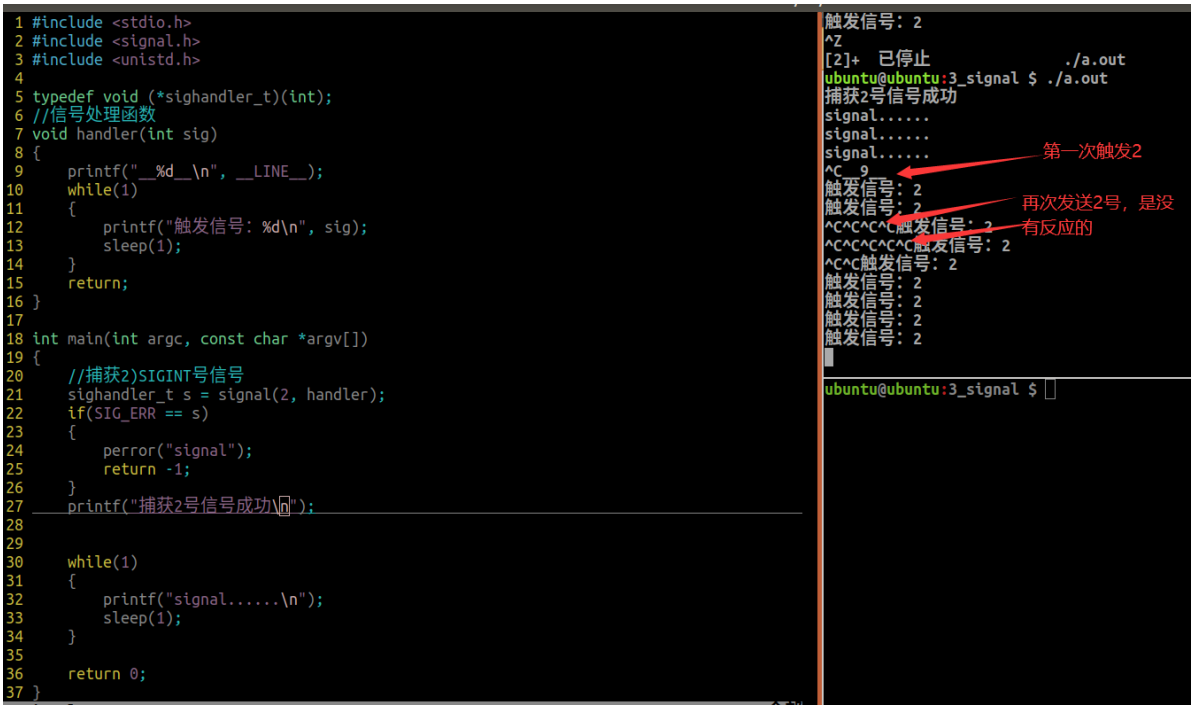
1）signal

```
void (*ptr)(int);        函数指针变量，变量名ptr
int *pa;

typedef：给数据类型重命名；
typedef 旧的类型名 新的类型名；
typedef int  int_t;                          int a ---> int_t a;
typedef int* pint;                           int* pa --> pint pa;

typedef void (*)(int)   sighandler_t;     void (*ptr)(int); ----> sighanler_t
ptr;
```

```
功能：捕获信号，为信号注册新的处理函数；
头文件：
        #include <signal.h>

        typedef void (*sighandler_t)(int);        //数据类型重命名，将void (*)(int) 类
型重命名成sighandler_t类型。

        sighandler_t signal(int signum, sighandler_t handler);
参数：
    int signum：指定要捕获的信号,可以填信号的编号，也可以填对应的宏，例如可以填2，也可以填
SIGINT；
    sighandler_t handler：回调函数，函数指针;该指针能指向，返回值是void类型，参数列表是int
类型的函数；
            1.SIG_IGN：忽略信号；
            2.SIG_DFL：执行默认操作；
            3.设置新的信号处理函数，函数的返回值是void类型，参数列表是int类型；
                void handler(int sig)
                {
                    //信号的新功能
                }
返回值：
    成功，返回该信号的上一个信号处理函数的首地址；
    失败，返回SIG_ERR,更新errno；
            #define SIG_ERR ((__sighandler_t)-1)
```

注意：

1. 当程序运行在一个信号处理函数中没有从该信号处理函数退出的时候，若此时发送该信号给当前程序，则当前程序不会再次触发该信号的信号处理函数。所以要注意，信号处理函数中**一定不要写死循环**。



2. 第一次捕获信号的处理函数时，返回值理论上应该返回默认处理函数的首地址，但是默认处理函数的首地址是无法获取，所以结果返回NULL；

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

typedef void (*sighandler_t)(int);
//信号处理函数
void handler(int sig)
{
    printf("触发信号：%d\n", sig);
    return;
}

int main(int argc, const char *argv[])
{
    //捕获2)SIGINT号信号
    sighandler_t s = signal(2, handler);
    if(SIG_ERR == s)
    {
        perror("signal");
        return -1;
    }
    printf("捕获2号信号成功\n");

    while(1)
    {
        printf("signal......\n");
        sleep(1);
    }

    return 0;
}
```

2）kill

```
功能：发送一个信号给指定的进程；
头文件：
        #include <sys/types.h>
        #include <signal.h>

        int kill(pid_t pid, int sig);
参数：
    pid_t pid：指定要将信号发送给哪个进程/进程组；
         pid > 0,     发送信号给指定的进程，进程的pid号 == 参数pid
         pid == 0,    发送信号给当前进程组下的所有进程；
         pid == -1,   发送信号给当前进程权限所能发送到的所有进程，除了1号进程；
         pid < -1,    发送信号给指定进程组下的所有金成功，进程组id == 参数pid的负数，-pid；

    int sig：指定要发送的信号；
         sig == 0，没有发送信号，但是回去检查pid指定的进程或者进程组是否存在，以及当前进程是否
有权限访问；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；


#include <signal.h>
int raise(int sig); 功能：给自身发送一个信号 相当于---》kill(getpid(), sig);
```

练习

> 启动AB进程，B进程通过10)号信号杀死A进程;

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>

int main(int argc, const char *argv[])
{
    //创建有名管道
    if(mkfifo("./fifo", 0777) < 0)
    {
        if(17 != errno)
        {
            perror("mkfifo");
            return -1;
        }
    }


    //以读写的方式打开
    int fd = open("./fifo", O_RDWR);
    if(fd < 0)
    {
```

```
        perror("open");
        return -1;
    }
    //写入当前进程的pid号
    pid_t pid = getpid();
    printf("%d\n", pid);

    if(write(fd, &pid, sizeof(pid)) < 0)
    {
        perror("write");
        return -1;
    }


    while(1)
    {
        printf("signal...\n");
        sleep(1);
    }
    return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main(int argc, const char *argv[])
{
    //创建有名管道
    if(mkfifo("./fifo", 0777) < 0)
    {
        if(17 != errno)
        {
            perror("mkfifo");
            return -1;
        }
    }

    //以只读方式打开
    int fd = open("./fifo", O_RDONLY);
    if(fd < 0)
    {
        perror("open");
        return -1;
    }

    pid_t pid = 0;
    if(read(fd, &pid, sizeof(pid)) < 0)
    {
        perror("read");
        return -1;
    }
    printf("%d\n", pid);
```

```
    sleep(2);
    kill(pid, 9);

    return 0;
}
```

3）alarm

```
功能：设置一个定时器，等时间到后，会给当前进程发送一个14） SIGALRM       时钟信号；
        该函数不是阻塞函数；
头文件：
       #include <unistd.h>

       unsigned int alarm(unsigned int seconds);
参数：
    unsigned int seconds：设置定时器时间，单位为秒；
                seconds == 0，取消设置好的定时器；
返回值：
    >0 ，返回上一次设置的时钟，没有走完的时间；
    =0，当前没有设置时钟；
```

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

typedef void (*sighandler_t)(int);
//信号处理函数
void handler(int sig)
{
    printf("触发信号：%d\n", sig);

    alarm(3);   //不会阻塞
    return;
}

int main(int argc, const char *argv[])
{
    //捕获14） SIGALRM号信号
    sighandler_t s = signal(14, handler);
    if(SIG_ERR == s)
    {
        perror("signal");
        return -1;
    }
    printf("捕获14号信号成功\n");


    alarm(3);   //不会阻塞

    while(1)
    {
        printf("signal,,,\n");
```

```
        sleep(1);
    }
    return 0;
}
```

3. 信号的方式回收僵尸进程（重点）

1. 子进程退出后会发送一个17)SIGCHLD信号给父进程，父进程可以通过捕获该信号，在新的处理函数中回收子进程的尸体。
2. 由于多个子进程同时退出的时候，会造成父进程收到的17号信号重叠，倒置多个子进程退出，只会触发一次17号信号的处理函数。

   所以多个子进程退出如果只运行一次wait/waitpid函数，会让僵尸进程收不干净。
3. 如果成功回收到僵尸进程，则再调用waitpid函数再收一次，如此往复，直到没有收到僵尸进程（返回值为0），或者函数运行失败（返回值为-1，当前父进程下没有子进程的情况）为止；

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <stdlib.h>

typedef void (*sighandler_t)(int);

void handler(int sig)
{
    //父进程执行回收僵尸进程的函数
    pid_t cpid = 0;
    //如果成功回收到僵尸进程，则再收一次
    //直到没有收到僵尸进程或者函数运行失败为止，结束循环
    /*
    while(1)
    {
        cpid = waitpid(-1, NULL, WNOHANG);
        if(cpid <= 0)
            break;
        printf("cpid = %d\n", cpid);
    }
    */

    while(waitpid(-1, NULL, WNOHANG) > 0);
}

int main(int argc, const char *argv[])
{
    //捕获17号SIGCHLD信号
    sighandler_t s = signal(SIGCHLD, handler);
    if(SIG_ERR == s)
    {
        perror("signal");
        return -1;
```

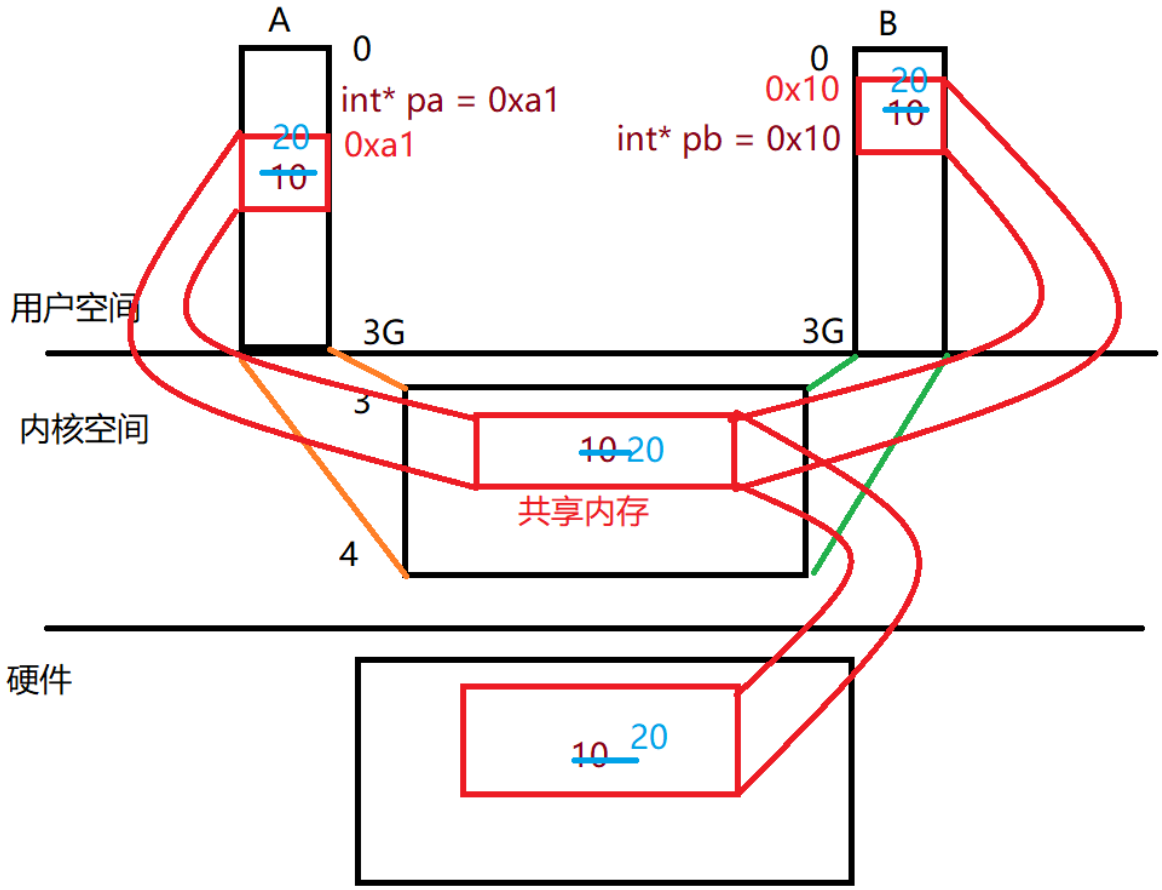
```
    }

    int i = 0;
    while(i < 100)
    {
        if(fork() == 0)      //如果是子进程，就退出
            exit(0);

        i++;
    }

    //能运行到当前位置的只有父进程
    while(1)
    {
        sleep(1);
    }

    return 0;
}
```

【4】共享内存（shared memory）

1. 概念

1）共享内存的原理

将同一物理地址通过共享内存分别映射到不同的进程中，进程只需要操作自己的用户空间中的虚拟地址空间就能实现对物理地址的操作。



2）共享内存的原理

1. 共享内存是 **最高效** 的进程间通信方式

   进程可以直接在用户空间通过操作地址，往共享内存中写入数据，读取数据，不需要任何数据拷贝;
2. 共享内存是在内核空间中创建，可以被映射到不同的用户空间中。
3. 共享内存可以被多个进程同时访问，所以需要引入进程的同步互斥机制：信号灯集
4. 一旦共享内存申请成功，就会独立于进程，就算进程退出，共享内存依然存在。需要手动删除，或者关机重启;

3）查看共享内存

```
ipcs
ipcs -m

删除共享内存
    ipcrm -m shmid
```

2. 共享内存的函数

1）ftok

```
功能：通过给定的文件路径char *pathname以及int proj_id，去计算键值，
        只要文件路径和proj_id不变，则key值不变;后序的函数通过key值找到的IPC对象就是同一个。
头文件：
        #include <sys/types.h>
        #include <sys/ipc.h>

        key_t ftok(const char *pathname, int proj_id);
参数：
    char *pathname：文件路径以及文件名;要求文件存在且可被访问即可;
    int proj_id：非0参数;
返回值：
    成功，返回key值;
    失败，返回-1，更新errno;
```

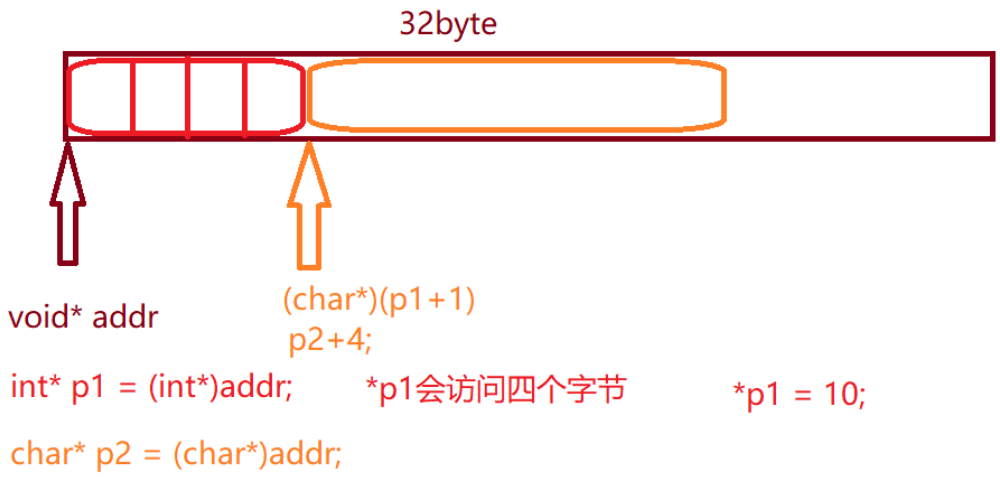
2）shmget

```
功能：创建共享内存，并返回共享内存的id号;
头文件：
        #include <sys/ipc.h>
        #include <sys/shm.h>

        int shmget(key_t key, size_t size, int shmflg);
参数：
    key_t key：通过key值找对应的共享内存，key值由ftok函数创建;
    size_t size：指定要创建多大的共享内存，以字节为单位;
    int shmflg：
        IPC_CREAT：创建共享内存，如果共享内存已经存在，则忽略该选项
        IPC_CREAT | IPC_EXCL：如果共享内存不存在则创建，如果存在则该函数运行失败;
        IPC_CREAT|0777
返回值：
    成功，返回共享内存的id号，shmid;
    失败，返回-1，更新errno;
```

3）shmat

```
功能：将共享内存映射到用户空间中;
头文件：
        #include <sys/types.h>
        #include <sys/shm.h>

        void *shmat(int shmid, const void *shmaddr, int shmflg);
参数：
    int shmid：指定要将哪个共享内存映射到用户空间;
    void *shmaddr：指定要映射到用户空间的什么位置;
                填NULL，让系统自动映射;
    int shmflg：
        0：可读可写;
        SHM_RDONLY：只读;
返回值：
    成功，返回共享内存成功映射到用户空间的地址;
```

```
    失败， (void *) -1，更新errno;

       int shmdt(const void *shmaddr);
```




```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    key_t key = ftok("/home/ubuntu/", 2);
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("key = %#x\n", key);

    //创建共享内存
    int shmid = shmget(key, 32, IPC_CREAT|0777);
    if(shmid < 0)
    {
        perror("shmget");
        return -1;
    }
    printf("shmid = %d\n", shmid);

    system("ipcs -m");


    //共享内存的映射
    void* addr = shmat(shmid, NULL, 0);
    if((void*)-1 == addr)
    {
        perror("shmat");
        return -1;
    }
```

```
    printf("addr=%p\n", addr);


    //先往共享内存中写入一个整形
    int a = 10;
    int* p1 = (int*)addr;
    *p1 = a;

    //再往四个字节后写入一个字符串
    char* p2 = (char*)addr+4;
    strcpy(p2, "hello world");

    system("ipcs -m");

    return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    key_t key = ftok("/home/ubuntu/", 2);
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("key = %#x\n", key);

    //创建共享内存
    int shmid = shmget(key, 32, IPC_CREAT|0777);
    if(shmid < 0)
    {
        perror("shmget");
        return -1;
    }
    printf("shmid = %d\n", shmid);

    system("ipcs -m");


    //共享内存的映射
    void* addr = shmat(shmid, NULL, 0);
    if((void*)-1 == addr)
    {
        perror("shmat");
        return -1;
    }
    printf("addr=%p\n", addr);


    int* pa = (int*)addr;
```

```
    printf("%d\n", *pa);

    printf("%s\n", (char*)(pa+1));


    system("ipcs -m");

    return 0;
}
```

4) shmdt

```
功能：断开映射；
头文件：
        #include <sys/types.h>
        #include <sys/shm.h>

        int shmdt(const void *shmaddr);
参数：
    void *shmaddr：需要断开映射的用户空间首地址；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

5) shmctl

```
功能：控制共享内存，常用于删除共享内存；
头文件：
        #include <sys/ipc.h>
        #include <sys/shm.h>

        int shmctl(int shmid, int cmd, struct shmid_ds *buf);
参数：
    int shmid：指定要操控的共享内存id号；
    int cmd：
         IPC_RMID：删除共享内存，只有共享内存在用户空间的所有映射均断开后，才能被删除；第三个参数填NULL；
         IPC_STAT：获取共享内存的属性，会存储到第三个参数指向的内存空间中；
         IPC_SET：设置共享内存的属性，要设置的内容也会存储到第三个参数指向的内存空间中；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

练习

> 创建两个进程A、B，以及一个共享内存，共享内存中存储char str[] = "123456"，在不考虑进程退出的情况下，要求如下：
>
> 1. A进程 循环 打印str字符串。
> 2. B进程 循环 倒置str字符串，不使用辅助数组。注意是循环倒置，要把字符串倒过来，倒回去。
> 3. 要求A进程打印出来的结果是有序的，例如："123456" 或者 "654321"，不允许出现"623451",,,等无序情况
>
> 提示：将flag + str一起写到共享内存中，当flag=0，打印 当flag=1 ,倒置

i.printf

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    //创建key
    key_t key = ftok("./", 'b');
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("key = %#x\n", key);


    //创建共享内存
    int shmid = shmget(key, 32, IPC_CREAT|0777);
    if(shmid < 0)
    {
        perror("shmget");
        return -1;
    }
    printf("shmid = %d\n", shmid);


    //映射到用户空间
    void* shmaddr = shmat(shmid, NULL, 0);
    if((void*)-1 == shmaddr)
    {
        perror("shmat");
        return -1;
    }
    printf("shmaddr = %p\n", shmaddr);

    //注意：不要直接操作shmaddr,因为容易将shmaddr的指向修改;
    //会造成内存泄露

    char* pa = (char*)shmaddr;
    *pa = 0;
```

```
    //写入字符串
    char* ptr =pa+1;
    strcpy(ptr, "hello");


    while(1)
    {
        if(0 == *pa)
        {
            printf("%s\n", ptr);
            *pa = 1;
        }
    }

    return 0;
}
```

ii.reserve

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main(int argc, const char *argv[])
{
    //创建key
    key_t key = ftok("./", 'b');
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("key = %#x\n", key);


    //创建共享内存
    int shmid = shmget(key, 32, IPC_CREAT|0777);
    if(shmid < 0)
    {
        perror("shmget");
        return -1;
    }
    printf("shmid = %d\n", shmid);


    //映射到用户空间
    void* shmaddr = shmat(shmid, NULL, 0);
    if((void*)-1 == shmaddr)
    {
        perror("shmat");
        return -1;
    }
    printf("shmaddr = %p\n", shmaddr);
```

```
    //注意：不要直接操作shmaddr,因为容易将shmaddr的指向修改;
    //会造成内存泄露

    char* pa = (char*)shmaddr;

    char* start, *end, temp;

    while(1)
    {
        start = pa+1;
        end = start+strlen(start)-1;

        if(1 == *pa)
        {
            while(start < end)
            {
                temp = *start;
                *start = *end;
                *end = temp;

                start++;
                end--;
            }

            *pa = 0;
        }
    }

    return 0;
}
```

【5】信号灯集（semaphore）

1.概念

信号灯：有称之为信号量，它可以实现不同进程或者给定进程中不同线程的同步互斥；

信号灯集：信号量的集合，一个信号灯集中有一个或者多个信号灯；

2）核心操作

P：申请信号量，信号量的值-1；

V：释放信号量，信号量的值+1；

wait for zero操作；

3）信号灯集的查看

```
ipcs
ipcs -s

删除信号灯集：
    ipcrm -s 信号灯集的id
```

2. 信号灯集的函数

1）ftok

```
功能：通过给定文件路径对应的id 和 proj_id 计算键值;只要文件路径和proj_id不变，
      则计算出来的key值就是不变的，则通过key值获取到的IPC对象就是同一个
头文件：
       #include <sys/types.h>
       #include <sys/ipc.h>

       key_t ftok(const char *pathname, int proj_id);
参数：
    char *pathname：文件的路径以及文件名，要求该文件存在且可以访问；
    int proj_id：非0参数，用户自定义；
返回值：
    成功，返回成功计算的key值：
    失败，返回-1，更新errno；

    key_t key = ftok("./", 2);
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("key = %#x\n", key);
```

2）semget

```
功能：创建一个信号灯集;信号灯集中的灯的初始值为0
头文件：
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/sem.h>

       int semget(key_t key, int nsems, int semflg);
参数：
    key_t key：ftok函数计算出来的键值；
    int nsems：信号灯集中信号灯的个数；
    int semflg：
        IPC_CREAT：创建信号灯集，如果信号灯集已经存在，则忽略该选项
        IPC_CREAT | IPC_EXCL：如果信号灯集不存在则创建，如果存在则该函数运行失败；
        IPC_CREAT|0777;
返回值：
```

```
    成功，返回信号灯集的id号，semid；
    失败，返回-1，更新errno；
```

3）semop

```
功能：对信号灯集中的信号灯进行：P(-)/V(+)，wait for zero操作；
头文件：
        #include <sys/types.h>
        #include <sys/ipc.h>
        #include <sys/sem.h>

        int semop(int semid, struct sembuf *sops, size_t nsops);
参数：
    int semid：信号灯集的编号；
    struct sembuf *sops：
         struct sembuf
         {
            unsigned short sem_num;  /* semaphore number */ 指定要控制的灯的编号，从0
开始
            short          sem_op;   /* semaphore operation */   P:负整数，例如-2，
会在信号灯原值基础上-2，如果不够减则阻塞。
                                                                  如果够减则
不阻塞
                                                                  V:正整数，例如+2，
会在信号灯的原值基础上加+2
                                                                  wait for zero：如
果信号灯的值为0，不阻塞，

如果信号灯的值不为0，则阻塞；

            short          sem_flg;  /* operation flags */    0：默认方式运行，阻塞方
式运行;该阻塞的时候阻塞；
                                                              IPC_NOWAIT：非阻塞方式,该阻
塞的时候，例如P操作不够减，也不会阻塞；
         };
    size_t nsops：指定要控制几个灯；

返回值：
    成功，返回0；
    失败，返回-1，更新errno；
```

4）semctl

```
功能：控制信号灯集；
头文件：
        #include <sys/types.h>
        #include <sys/ipc.h>
        #include <sys/sem.h>

        int semctl(int semid, int semnum, int cmd, ...);
参数：
```

```
    int semid：指定要控制哪一个信号灯集；
    int semnum：要控制信号灯集中的哪一个灯，灯的编号从0开始；
    int cmd：
        IPC_STAT：获取信号灯集的属性，存储到struct semid_ds结构体中；
        IPC_RMID：删除信号灯集； 所以信号灯的编号没有任何意义，可以随便填，不定参数可以不填；
        GETALL：获取信号灯集中，所有信号灯的值;所以信号灯的编号没有任何意义，可以随便填；
        SETALL：设置信号灯集中，所有信号灯的值;所以信号灯的编号没有任何意义，可以随便填；
        GETVAL：获取信号灯集中，指定信号灯的值；
        SETVAL：设置信号灯集中，指定信号灯的值；


    ...：不同的cmd有不同的类型形参；
           union semun {
               int              val;    /* Value for SETVAL */
               struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
               unsigned short  *array;  /* Array for GETALL, SETALL */
               struct seminfo  *__buf;  /* Buffer for IPC_INFO
                                           (Linux-specific) */
           };
返回值：
    成功，根据cmd的不同，返回值意义不同，例如GETVAL，返回值就位获取到的灯的值；
    失败，返回-1，更新errno；
```

i. SETALL

```
    //SETALL
    unsigned short setall[2] = {1, 2};
    if(semctl(semid, 0, SETALL, setall) < 0)
    {
        perror("semctl");
        return -1;
    }
    printf("SETALL success\n");
```

ii. GETALL

```
    //GETALL

    unsigned short getall[2] = {0};
    if(semctl(semid, 0, GETALL, getall) < 0)
    {
        perror("semctl");
        return -1;
    }
    printf("0:%d 1:%d\n", getall[0], getall[1]);
```

iii. SETVAL

```
//SETVAL
int setval = 5;
if(semctl(semid, 0, SETVAL, setval) < 0)
{
    perror("semctl");
    return -1;
}
printf("SETVAL success\n");
```

iv. GETVAL

```
//GETVAL
int getval = semctl(semid, 0, GETVAL);
if(getval < 0)
{
    perror("semctl");
    return -1;
}
printf("getval = %d\n", getval);
```

作业

> 1.一个进程循环打印字符串，另外一个进程循环倒置字符串；
>
> 要求打印一次，倒置一次

1）printf

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/sem.h>

int main(int argc, const char *argv[])
{
    //创建key值
    key_t key = ftok("./", 10);
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("%#x\n", key);
```

```c
    //创建信号灯集
    int semid = semget(key, 2, IPC_CREAT|0664);
    if(semid < 0)
    {
        perror("semget");
        return -1;
    }

    //设置信号灯的值
    unsigned short arr[2] = {0, 1};
    if(semctl(semid, 0, SETALL, arr) < 0)
    {
        perror("semctl");
        return -1;
    }

    //创建共享内存
    int shmid = shmget(key, 128, IPC_CREAT|0664);
    if(shmid < 0)
    {
        perror("shmget");
        return -1;
    }
    printf("shmid=%d\n", shmid);

    //映射到用户空间
    void* shmaddr = shmat(shmid, NULL, 0);
    if((void*)-1 == shmaddr)
    {
        perror("shmat");
        return -1;
    }
    printf("shmaddr = %p\n", shmaddr);

    //注意不要直接操作shmaddr;
    //因为一旦shmaddr的值修改后，会找不到共享内存首地址;

    char* str = (char*) shmaddr;

    strcpy(str, "123456");


    struct sembuf sop;

    while(1)
    {
        sop.sem_num = 1;    //控制1号灯
        sop.sem_op = -1;    //P操作;
        sop.sem_flg = 0;    //阻塞方式
        if(semop(semid, &sop, 1) < 0)
        {
            perror("semop");
            return -1;
        }

        printf("%s\n", str);

        sop.sem_num = 0;    //控制0号灯
```

```
        sop.sem_op = +1;    //v操作;
        sop.sem_flg = 0;    //阻塞方式
        if(semop(semid, &sop, 1) < 0)
        {
            perror("semop");
            return -1;
        }
    }

    return 0;
}
```

2) reserve

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/sem.h>

int main(int argc, const char *argv[])
{
    //创建key值
    key_t key = ftok("./", 10);
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("%#x\n", key);


    //创建信号灯集
    int semid = semget(key, 2, IPC_CREAT|0664);
    if(semid < 0)
    {
        perror("semget");
        return -1;
    }

    //创建共享内存
    int shmid = shmget(key, 128, IPC_CREAT|0664);
    if(shmid < 0)
    {
        perror("shmget");
        return -1;
    }
    printf("shmid=%d\n", shmid);

    //映射到用户空间
    void* shmaddr = shmat(shmid, NULL, 0);
    if((void*)-1 == shmaddr)
```

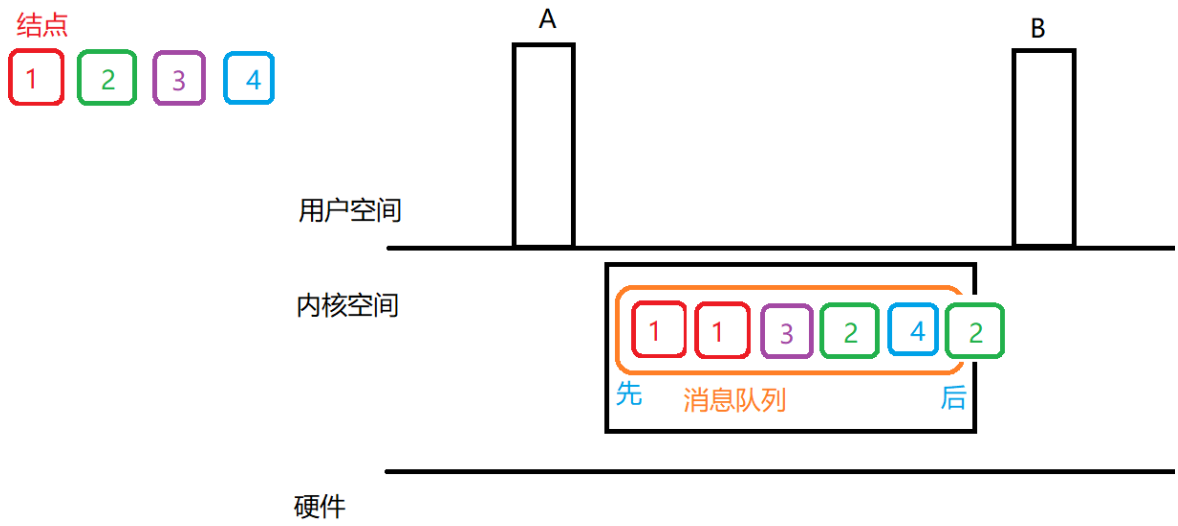
```
    {
        perror("shmat");
        return -1;
    }
    printf("shmaddr = %p\n", shmaddr);

    //注意不要直接操作shmaddr;
    //因为一旦shmaddr的值修改后，会找不到共享内存首地址;

    char* str = (char*) shmaddr;
    char* start = NULL;
    char* end = NULL;

    char temp = 0;

    struct sembuf sop;

    while(1)
    {
        sop.sem_num = 0;    //控制0号灯
        sop.sem_op = -1;    //p操作;
        sop.sem_flg = 0;    //阻塞方式
        if(semop(semid, &sop, 1) < 0)
        {
            perror("semop");
            return -1;
        }

        start = str;
        end = start + strlen(start)-1;

        while(start < end)
        {
            temp = *start;
            *start = *end;
            *end = temp;

            start++;
            end--;
        }

       sop.sem_num = 1;    //控制1号灯
       sop.sem_op = +1;    //V操作;
       sop.sem_flg = 0;    //阻塞方式
       if(semop(semid, &sop, 1) < 0)
       {
           perror("semop");
           return -1;
       }


    }

    return 0;
}
```

【6】消息队列（message queue）

1. 概念

1）消息队列的原理

消息队列是在内核空间中创建一个容器（队列），进程会将数据结点添加到队尾，或者从队列中读取结点，实现进程间通信。



2）消息队列的特点

1. 消息队列是面向记录的，其中消息具有特定的格式以及优先级，详情请看后面的函数;
2. 消息队列可以实现消息的随机查询，消息可以按照消息的类型进行读取，原则上还是按照先进先出的原则;
3. 消息队列独立于进程，等进程结束后，消息队列依然存在，以及其中没有被读取的结点也存在，除非手动删除或者系统重启;

3）消息队列的查看

```
ipcs
ipcs -q

删除消息队列:
    ipcrm -q 消息队列的id
```

2. 消息队列的函数

1）ftok

功能：通过给定文件路径对应的id 和 proj_id 计算键值;只要文件路径和proj_id不变，
则计算出来的key值就是不变的，则通过key值获取到的IPC对象就是同一个
头文件：

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

key_t ftok(const char *pathname, int proj_id);
```

参数：

char *pathname：文件的路径以及文件名，要求该文件存在且可以访问；
int proj_id：非0参数，用户自定义；

返回值：

成功，返回成功计算的key值：
失败，返回-1，更新errno；

```
key_t key = ftok("./", 2);
if(key < 0)
{
    perror("ftok");
    return -1;
}
printf("key = %#x\n", key);
```

2）msgget

功能：创建消息队列；
头文件：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

参数:

key_t key:;
int msgflg:
IPC_CREAT：创建消息队列，如果消息队列已经存在，则忽略该选项
IPC_CREAT | IPC_EXCL：如果消息队列不存在则创建，如果存在则该函数运行失败；
IPC_CREAT|0777

返回值：

成功，返回消息队列的id号，msqid；
失败，返回-1，更新errno；

3）msgsnd

功能：向指定消息队列中发送数据；
头文件：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
        int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```
参数：
    int msqid：消息队列的id号；
    void *msgp：消息包，消息结点；指定要发送的消息包的首地址；
```
            struct msgbuf {
                long mtype;       /* message type, must be > 0 */ 消息类型，必须大于0
                char mtext[1];    /* message data */ 消息内容，这个成员的类型结构可以改
```
变；但是它的大小由msgsz参数指定
                                                                    即该成员的大小，要与msgz参数填充的
一致，以字节为单位；
```
            };
```
    size_t msgsz：消息包（消息结点）中，消息内容的大小，以字节为单位；
    int msgflg：发送方式：
            0：阻塞方式发送，如果消息队列满了，则该函数阻塞；
            IPC_NOWAIT：非阻塞方式，如果消息队列满了，则该函数不阻塞，立即返回，errno被更新为EAGAIN；
返回值：
    成功，返回0；
    失败，返回-1，更新errno；

4）msgrcv

功能：从指定消息队列中接收数据； 接收完的数据会从队列中消除
头文件：
```
        #include <sys/types.h>
        #include <sys/ipc.h>
        #include <sys/msg.h>

        ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int
```
msgflg);
参数：
    int msqid：指定从哪个消息队列中读取数据；
    void *msgp：消息包，消息结点；指定存储读取到的消息内容的首地址，以什么形式发送，就以什么形式接收
```
            struct msgbuf {
                long mtype;       /* message type, must be > 0 */ 消息类型，必须大于0
                char mtext[1];    /* message data */ 消息内容，这个成员的类型结构可以改
```
变；但是它的大小由msgsz参数指定
                                                                    即该成员的大小，要与msgz参数填充的
一致，以字节为单位；
```
            };
```
    size_t msgsz：消息包（消息结点）中，消息内容的大小，以字节为单位；
    long msgtyp：指定要读取的消息类型；
        msgtyp == 0;   则读取消息队列中的第一条消息，按照先进先出的原则；
        msgtyp > 0;   读取消息队列中第一条消息类型 == msgtyp的消息，
                         如果msgflg中填充了MSG_EXCEPT，则会读取消息队列中第一条消息类型
```
!= msgtyp的消息；
        msgtyp < 0 ;     读取消息队列中第一条最小的消息包，且该消息包的类型<= msgtyp的绝对
```
值；

    int msgflg：
            0，阻塞方式，如果消息队列中没有要读取的消息包，则阻塞；

```
            IPC_NOWAIT：非阻塞方式，如果消息队列中没有要读取的消息包，不阻塞，设置 errno
set to ENOMSG;
返回值：
    成功，返回成功读取到的消息内容的大小，以字节为单位；
    失败，返回-1，更新errno;
```

代码示例：

```
若消息队列中有消息：
    mtype 100   101  99   100  101
    mtext aaa   bbb  ccc  ddd  eee
```

i. msgtyp == 0

```
    while(1)
    {
        //阻塞方式读取消息队列中第一条消息，先进先出的原则
        //res = msgrcv(msqid, &rcv, sizeof(rcv.mtext), 0, 0);

        //非阻塞方式读取消息队列中第一条消息，先进先出的原则
        res = msgrcv(msqid, &rcv, sizeof(rcv.mtext), 0, IPC_NOWAIT);

        if(res < 0)
        {
            perror("msgrcv");
            return -1;
        }
        printf("res=%ld : %ld %s\n", res, rcv.mtype, rcv.mtext);
    }

输出顺序：
    100 aaa   101 bbb   99 ccc   100 ddd    101 eee
```

ii. msgtyp > 0

```
    while(1)
    {
        //1.阻塞方式读取消息队列中第一条消息类型 == 101 的消息
        //res = msgrcv(msqid, &rcv, sizeof(rcv.mtext), 101, 0);

        //2.非阻塞方式读取消息队列中第一条消息 == 101 的消息
        //res = msgrcv(msqid, &rcv, sizeof(rcv.mtext), 101, IPC_NOWAIT);


        //3.非阻塞方式读取消息队列中第一条消息类型 != 101的消息
        res = msgrcv(msqid, &rcv, sizeof(rcv.mtext), 101, IPC_NOWAIT|020000);

        if(res < 0)
        {
            perror("msgrcv");
```

```
            return -1;
        }
        printf("res=%ld : %ld %s\n", res, rcv.mtype, rcv.mtext);
    }
```

注释1,2的现象：

```
    101 bbb     101 eee
```

第3个的现象：

```
    100 aaa    99 ccc   100 ddd
```

iii. msgtyp <0

```
    while(1)
    {
        //阻塞方式读取消息队列中<=msgtyp参数指定的消息类型中最小的那条消息;
        res = msgrcv(msqid, &rcv, sizeof(rcv.mtext), -100, 0);

        //非阻塞方式读取消息队列中<=msgtyp参数指定的消息类型中最小的那条消息;
        //res = msgrcv(msqid, &rcv, sizeof(rcv.mtext), -100, IPC_NOWAIT);

        if(res < 0)
        {
            perror("msgrcv");
            return -1;
        }
        printf("res=%ld : %ld %s\n", res, rcv.mtype, rcv.mtext);
    }
```

现象：

```
res=128 : 99 ccc
res=128 : 100 aaa
res=128 : 100 ddd
```

5）msgctl

功能：操作消息队列；

头文件：

```
        #include <sys/types.h>
        #include <sys/ipc.h>
        #include <sys/msg.h>

        int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

参数：

```
    int msqid：指定要控制的消息队列；
    int cmd：
        IPC_STAT：获取消息队列的属性；
        IPC_SET：设置消息队列的属性；
        IPC_RMID：删除消息队列；第三个参数填NULL；
```

返回值：

```
    成功，返回0；
    失败，返回-1，更新errno；
```

练习

> 1. 消息队列的大小计算：16384 16k
> 2. 要求用消息队列实现AB进程对话：
>
> 1）A进程发送一句话，B进程接收后打印；
>
> 2）B进程接着再发送一句话，A进程接收打印；
>
> 3）重复上述步骤，当A进程或者B进程接收到quit后退出AB进程。
>
> 在第二题的基础上实现AB进程能够随时收发。

第1题

A

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <string.h>

//接收发送的消息类型
#define SND 1
#define RCV 2

struct msgbuf
{
    long mtype;         //消息类型
    char mtext[128];    //消息内容
};


int main(int argc, const char *argv[])
{
    //创建key值
    key_t key = ftok("./", 9);
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("key = %#x\n", key);

    //创建消息队列
    int msqid = msgget(key, IPC_CREAT|0664);
    if(msqid < 0)
    {
        perror("msgget");
        return -1;
```

```
    }

    struct msgbuf snd = {SND};
    struct msgbuf rcv;
    while(1)
    {
        memset(snd.mtext, 0, sizeof(snd.mtext));
        //从终端获取数据
        printf("请输入>>>");
        fgets(snd.mtext, sizeof(snd.mtext), stdin);
        snd.mtext[strlen(snd.mtext)-1] = 0;

        //向消息队列中发送数据:mtype == 1
        if(msgsnd(msqid, &snd, sizeof(snd.mtext), 0) < 0)
        {
            perror("msgsnd");
            return -1;
        }
        if(strcasecmp(snd.mtext, "quit") == 0)
        {
            break;
        }

        memset(&rcv, 0, sizeof(rcv));
        //从消息队列中读取数据:mtypes == 2
        if(msgrcv(msqid, &rcv, sizeof(rcv.mtext), RCV, 0) < 0)
        {
            perror("msgrcv");
            return -1;
        }

        printf(":%ld  %s\n", rcv.mtype, rcv.mtext);
        if(strcasecmp(rcv.mtext, "quit") == 0)
        {
            //删除消息队列
            msgctl(msqid, IPC_RMID, NULL);

            break;
        }

    }

    return 0;
}
```

B

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <string.h>

//接收发送的消息类型
#define SND 2
#define RCV 1
```

```c
struct msgbuf
{
    long mtype;         //消息类型
    char mtext[128];    //消息内容
};


int main(int argc, const char *argv[])
{
    //创建key值
    key_t key = ftok("./", 9);
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    printf("key = %#x\n", key);

    //创建消息队列
    int msqid = msgget(key, IPC_CREAT|0664);
    if(msqid < 0)
    {
        perror("msgget");
        return -1;
    }

    struct msgbuf snd = {SND};
    struct msgbuf rcv;
    while(1)
    {
        memset(&rcv, 0, sizeof(rcv));
        //从消息队列中读取数据:mtypes == 2
        if(msgrcv(msqid, &rcv, sizeof(rcv.mtext), RCV, 0) < 0)
        {
            perror("msgrcv");
            return -1;
        }

        printf(":%ld  %s\n", rcv.mtype, rcv.mtext);
        if(strcasecmp(rcv.mtext, "quit") == 0)
        {
            //删除消息队列
            msgctl(msqid, IPC_RMID, NULL);
            break;
        }


        memset(snd.mtext, 0, sizeof(snd.mtext));
        //从终端获取数据
        printf("请输入>>>");
        fgets(snd.mtext, sizeof(snd.mtext), stdin);
        snd.mtext[strlen(snd.mtext)-1] = 0;

        //向消息队列中发送数据:mtype == 1
        if(msgsnd(msqid, &snd, sizeof(snd.mtext), 0) < 0)
        {
            perror("msgsnd");
            return -1;
```

```
        }
        if(strcasecmp(snd.mtext, "quit") == 0)
        {
            break;
        }

    }
    return 0;
}
```