

第三章：多进程编程

一、多进程理论基础

1.1 引入目的

- 1> 实现多任务并发执行的一种途径
- 2> 可以实现数据在多个进程之间进行通信，共同处理整个程序的相关数据，提供工作效率

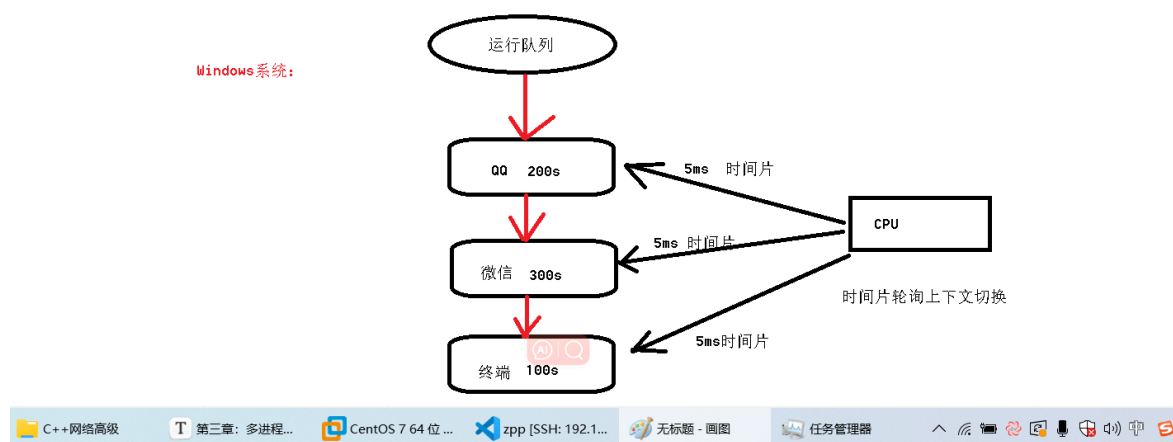
1.2 多进程相关概念

- 1> 进程是程序的一次执行过程，有一定的生命周期，包含了创建态、就绪态、执行态、挂起态、死亡态
- 2> 进程是计算机资源分配的基本单位，系统会给每个进程分配0--4G的虚拟内存，其中0--3G是用户空间，3--4G是内核空间

其中多个进程中0--3G的用户空间是相互独立的，但是，3--4G的内核空间是相互共享的

用户空间细分为：栈区、堆区、静态区

- 3> 进程的调度机制：时间片轮询上下文切换机制

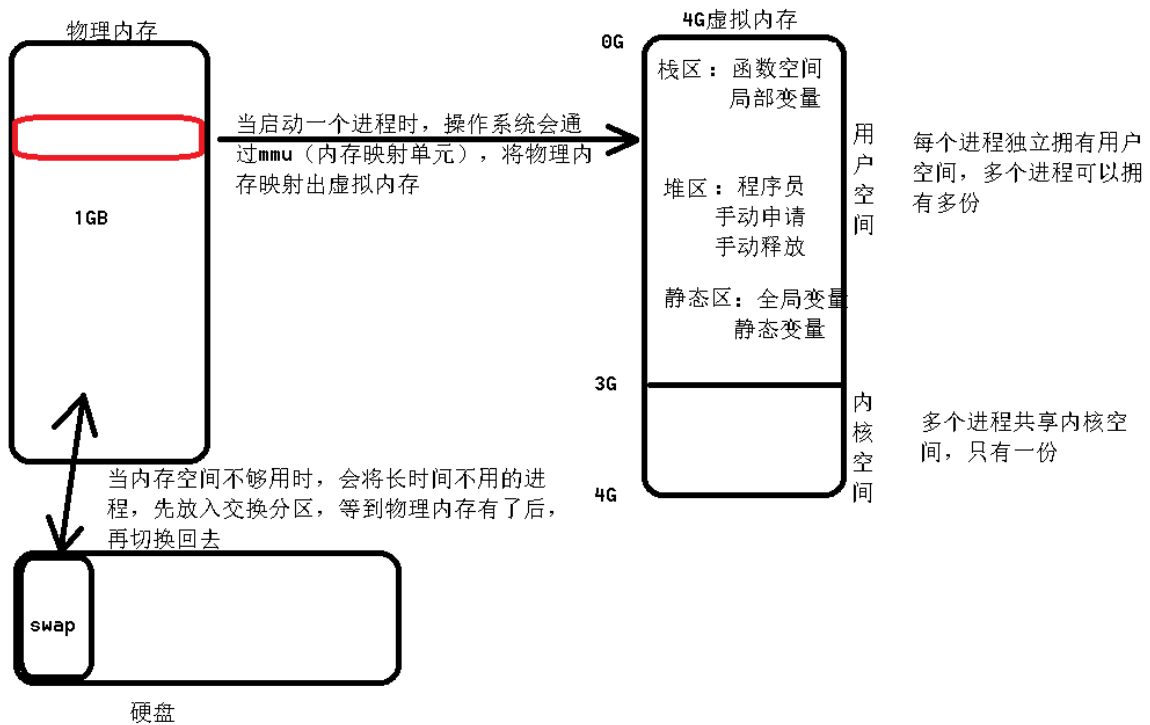


- 4> 并发和并行的区别：

并发：针对于单核CPU系统在处理多个任务时，使用相关的调度机制，实现多个任务进行细化时间片轮询时，在宏观上感觉是多个任务同时执行的操作，同一时刻，只有一个任务在被CPU处理

并行：是针对于多核CPU而言，处理多个任务时，同一时间，每个CPU处理的任务之间是并行的，实现的是真正意义上多个任务同时执行的

1.3 进程的内存管理（重点了解）



- 1> 物理内存：内存条上（硬件上）真正存在的存储空间
- 2> 虚拟内存：程序运行后，通过内存映射单元，将物理内存映射出4G的虚拟内存，共进程使用

1.4 进程和程序的区别

进程：是动态的，进程是程序的一次执行过程，是有生命周期的，进程会被分配0--3G的用户空间，进程是在内存上存着的

程序：是静态的，没有所谓的生命周期，程序存储在磁盘设备上的二进制文件

hello.cpp --> g++ ----> a.out

1.5 进程的种类

进程一共有三种：交互进程、批处理进程、守护进程

- 1> 交互进程：它是由shell控制，可以直接和用户进行交互的，例如文本编辑器
- 2> 批处理进程：内部维护了一个队列，被放入该队列中的进程，会被统一处理。例如 g++编译器的逐步到位的编译
- 3> 守护进程：脱离了终端而存在，随着系统的启动而运行，随着系统的退出而停止。例如：操作系统的服务进程

1.6 进程PID的概念

- 1> PID (Process ID) :进程号，进程号是一个大于等于0的整数值，是进程的唯一标识，不可能重复。
- 2> PPID(Parent Process ID):父进程号，系统中允许的每个进程，都是拷贝父进程资源得到的
- 3> 在linux系统中的 /proc目录下的数字命名的目录其实都是一个进程

```

● bash-4.2$ ls
1      13      1728    20      2606    287    32      396    404    508    596    665    8
10     1306    18      21      2607    288    33      397    41     520    598    666    858
1048   14      1830    22      2628    290    369    398    42     562    6      667    9
1050   15      1898    23      2639    292    370    399    43     564    60     668    95
1052   16      19      24      273     293    380     4      44     567    600    689    acpi
11     1677    1917    2495    275     296    381     400    45     568    602    694    asound
1185   1681    1941    2569    276     297    393     401    47     589    605    695    buddyinfo
1186   1682    1948    2599    278     30     394     402    482    591    639    7      bus
1187   17      2       2604    286     31     395     403    5      594    664    732    cgroups
● bash-4.2$ pwd
/proc
○ bash-4.2$ █

```

1.7 特殊的进程

1> 0号进程 (idle)：他是由linux操作系统启动后运行的第一个进程，也叫空闲进程，当没有其他进程运行时，会运行该进程。他也是1号进程和2号进程的父进程

2> 1号进程 (init)：他是由0号进程创建出来的，这个进程会完成一些硬件的必要初始化工作，除此之外，还会收养孤儿进程

3> 2号进程 (kthreadd)：也称调度进程，这个进程也是由0号进程创建出来的，主要完成任务调度问题

4> 孤儿进程：当前进程还正在运行，其父进程已经退出了。

说明：每个进程退出后，其分配的系统资源应该由其父进程进行回收，否则会造成资源的浪费

5> 僵尸进程：当前进程已经退出了，但是其父进程没有为其回收资源

1.8 有关进程操作的指令

1> **ps指令**：能够查看当前运行的进程相关属性

ps -ef :能够显示进程之间的关系

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	10:22	?	00:00:01	/usr/lib/systemd/systemd --switched-root --system --deserialize 22
root	2	0	0	10:22	?	00:00:00	[kthreadd]
root	4	2	0	10:22	?	00:00:00	[kworker/0:0H]
root	5	2	0	10:22	?	00:00:00	[kworker/u256:0]
root	6	2	0	10:22	?	00:00:00	[ksoftirqd/0]
root	7	2	0	10:22	?	00:00:00	[migration/0]
root	8	2	0	10:22	?	00:00:00	[rcu_bh]
root	9	2	0	10:22	?	00:00:01	[rcu_sched]
root	10	2	0	10:22	?	00:00:00	[lru-add-drain]
root	11	2	0	10:22	?	00:00:00	[watchdog/0]
root	13	2	0	10:22	?	00:00:00	[kdevtmpfs]
root	14	2	0	10:22	?	00:00:00	[netns]
root	15	2	0	10:22	?	00:00:00	[khungtaskd]

UID: 用户ID号

PID: 进程号
PPID: 父进程号
C: 用处不大
STIME: 开始运行的时间
TTY: 如果是问号表示这个进程不依赖于终端而存在
CDM: 名称

ps -ajx:能够显示当前进程的状态

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
0	1	1	1	?	-1	Ss	0	0:01	/usr/lib/systemd/systemd --switched-root --system --deserialize 22
0	2	0	0	?	-1	S	0	0:00	[kthreadd]
2	4	0	0	?	-1	S<	0	0:00	[kworker/0:0H]
2	5	0	0	?	-1	S	0	0:00	[kworker/u256:0]
2	6	0	0	?	-1	S	0	0:00	[ksoftirqd/0]
2	7	0	0	?	-1	S	0	0:00	[migration/0]
2	8	0	0	?	-1	S	0	0:00	[rcu_bh]
2	9	0	0	?	-1	R	0	0:01	[rcu_sched]

PGID: 进程组ID
SID: 会话组ID
STAT: 进程的状态

ps -aux:可以查看当前进程对CPU和内存的占用率

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.6	128020	6032	?	Ss	10:22	0:01	/usr/lib/systemd/systemd --switched-root --system --deserialize 22
root	2	0.0	0.0	0	0	?	S	10:22	0:00	[kthreadd]
root	4	0.0	0.0	0	0	?	S<	10:22	0:00	[kworker/0:0H]
root	5	0.0	0.0	0	0	?	S	10:22	0:00	[kworker/u256:0]
root	6	0.0	0.0	0	0	?	S	10:22	0:00	[ksoftirqd/0]
root	7	0.0	0.0	0	0	?	S	10:22	0:00	[migration/0]
root	8	0.0	0.0	0	0	?	S	10:22	0:00	[rcu_bh]

%CPU: CPU占用率
%MEM : 内存占用率

2> top: 动态查看进程的相关属性

```

top - 11:43:17 up 1:20, 1 user, load average: 0.04, 0.03, 0.05
Tasks: 111 total, 1 running, 110 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.7 us, 0.7 sy, 0.0 ni, 98.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 995676 total, 98992 free, 422940 used, 473744 buff/cache
KiB Swap: 1953788 total, 1953524 free, 264 used. 414260 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR S  %CPU  %MEM     TIME+ COMMAND
 1681 zpp        20   0 159040  2464  1108 S   0.3   0.2   0:02.55 sshd
  
```

```

1898 zpp      20    0  895384  87616  24368 s  0.3  8.8   0:12.63 node

1941 zpp      20    0   10.9g 114436  24172 s  0.3 11.5   0:07.83 node

1948 zpp      20    0  765236  46700  20368 s  0.3  4.7   0:05.33 node

   1 root      20    0  128020   6032   3548 s  0.0  0.6   0:01.10 systemd

   2 root      20    0         0         0         0 s  0.0  0.0   0:00.00 kthreadd

   4 root       0 -20         0         0         0 s  0.0  0.0   0:00.00 kworker/0:0H

   5 root      20    0         0         0         0 s  0.0  0.0   0:00.61
kworker/u256:0

```

3> kill指令：发送信号的指令

使用方式：kill -信号号 进程号

可以通过指令：kill -l查看能够发送的信号有哪些

```

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、一共可以发射62个信号，前32个是稳定信号，后面是不稳定信号

2、常用的信号

SIGHUP：当进程所在的终端被关闭后，终端会给运行在当前终端的每个进程发送该信号，默认结束进程

SIGINT：中断信号，当用户键入`ctrl + c`时发射出来

SIGQUIT：退出信号，当用户键入`ctrl + /`是发送，退出进程

SIGKILL：杀死指定的进程

SIGSEGV：当指针出现越界访问时，会发射，表示段错误

SIGPIPE：当管道破裂时会发送该信号

SIGALRM：当定时器超时后，会发送该信号

SIGSTOP：暂停进程，当用户键入`ctrl+z`时发射

SIGTSTP：也是暂停进程

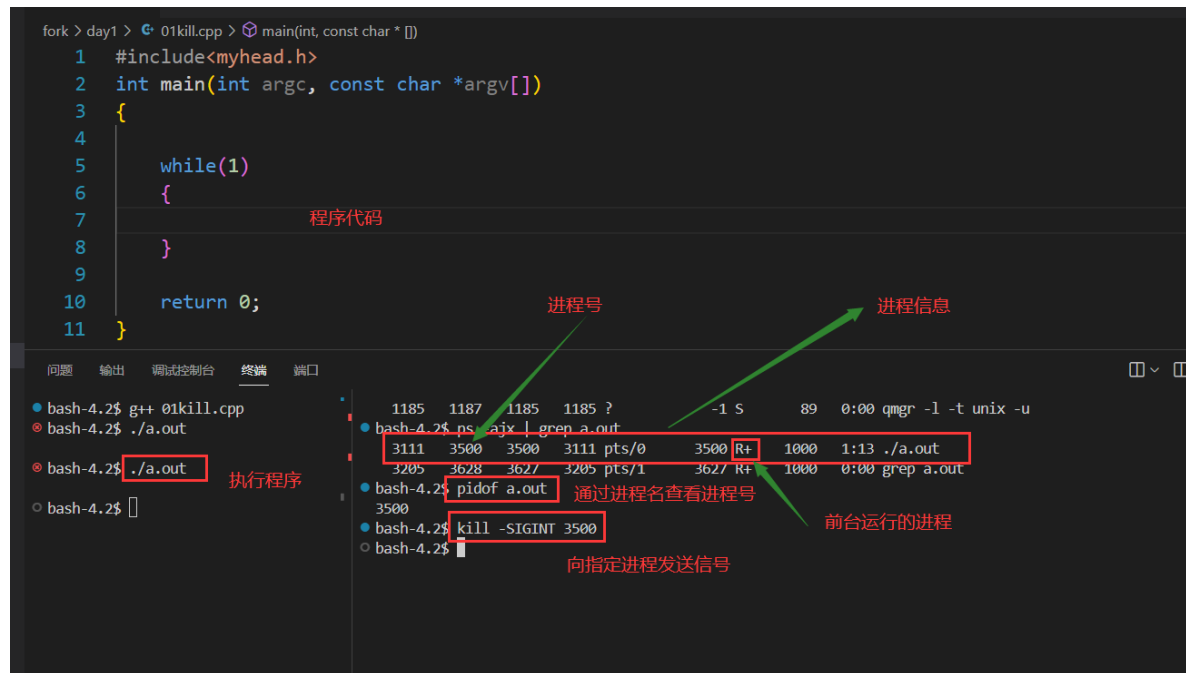
SIGTSTP、**SIGUSR2**：留给用户自定义的信号，没有默认操作

SIGCHLD：当子进程退出后，会向父进程发送该信号

3、有两个特殊信号：**SIGKILL**和**SIGSTOP**，这两个信号既不能被捕获，也不能被忽略

4> pidof: 查看进程的进程号

使用方式: pidof 进程名



1.9 进程状态的切换

1> 可以通过 man ps进行查看进程的状态

进程主状态:

D uninterruptible sleep (usually IO) 不可中断的休眠态, 通常是IO

操作

R running or runnable (on run queue) 运行态

S interruptible sleep (waiting for an event to complete) 可中

断的休眠态

T stopped by job control signal 停止态

t stopped by debugger during the tracing 调试时的停止态

W paging (not valid since the 2.6.xx kernel) 已经弃用的状态

X dead (should never be seen) 死亡态

Z defunct ("zombie") process, terminated but not reaped by its

parent 僵尸态

附加态:

< high-priority (not nice to other users) 高优先级进程

N low-priority (nice to other users) 低优先级进程

L has pages locked into memory (for real-time and custom IO)

锁在内存中的进程

s is a session leader 会话组组长

l is multi-threaded (using CLONE_THREAD, like NPTL pthreads

do) 包含多线程的进程

+ is in the foreground process group 前台运行的进程

2> 状态切换的实例

- 1、如果有停止的进程，可以在终端输入指令：`jobs -l`查看停止进程的作业号
- 2、通过使用指令：`bg 作业号` 实现将停止的进程进入后台运行状态，如果只有一个停止的进程，输入`bg`不加作业号也可以
- 3、对后台运行的进程，输入 `fg 作业号` 实现将后台运行的进程切换到前台运行
- 4、直接将可执行程序后台运行：`./可执行程序 &`

```

fork > day1 > 02stat.cpp > main(int, const char * [])
1  #include<myhead.h>
2  int main(int argc, const char *argv[])
3  {
4
5      while(1)                                程序代码
6      {
7          printf("hello world\n");
8          sleep(1);
9      }

```

初始状态是前台休眠态

作业号 进程号

查看当前终端所有停止进程的作业号

已停止 ./.out

bash-4.2\$ jobs -l

bash-4.2\$ kill -19 4337 向进程发送暂停信号

暂停态

bash-4.2\$ ps -ajxlgrep a.out

后台休眠态

bash-4.2\$ bg 1

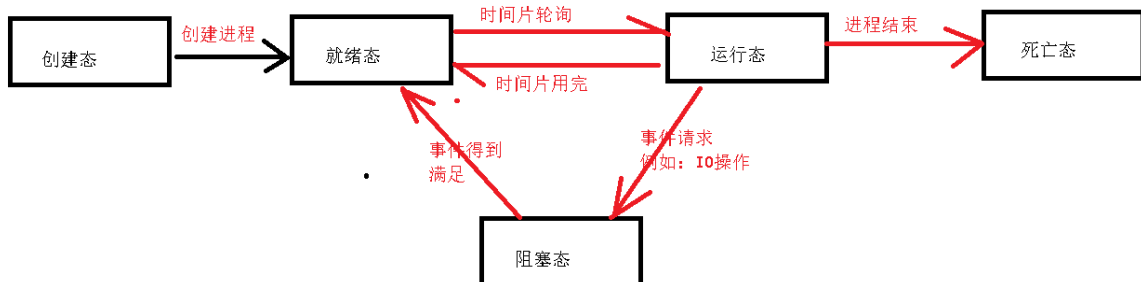
将停止的进程后台运行

表示后台运行

fg 作业号 将后台运行的进程设置成前台运行

前台休眠态

3> 进程主要状态的转换（五态图）



二、多进程实现

进程的创建过程，是子进程通过拷贝父进程得到的，新进程的创建直接拷贝父进程的资源，只需改变很少部分的数据即可，保留了父进程的大部分的数据信息（遗传基因），所以这个拷贝过程，系统通过一个函数fork来自动完成

2.1 进程的创建：fork

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

```
pid_t fork(void);
```

功能：通过拷贝父进程得到一个子进程

参数：无

返回值：成功在父进程中得到子进程的pid，在子进程中的到0，失败返回-1并置位错误码

1> 不关注返回值的案例

```
#include<myhead.h>
int main(int argc, const char *argv[])
{
    printf("ni hao xingqiu\n");

    fork();           //创建一个子进程

    printf("hello world\n");

    while(1);         //防止进程结束

    return 0;
}
```

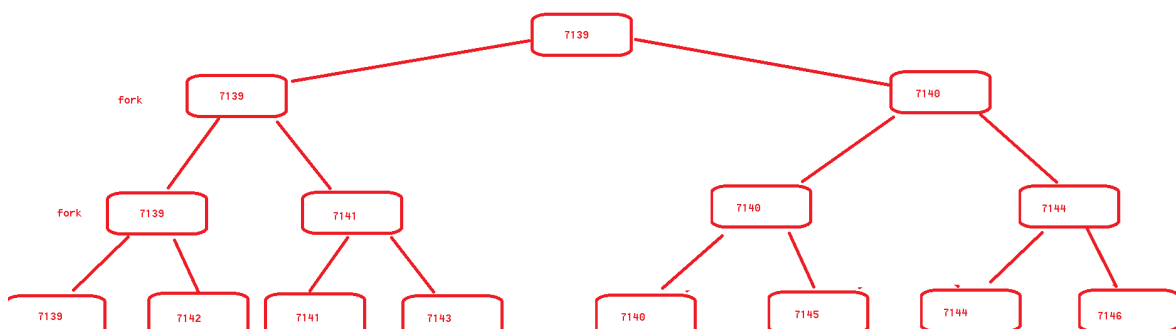
```
fork > day1 > 03fork.cpp > main(int, const char * [])
1  #include<myhead.h>
2  int main(int argc, const char *argv[])
3  {
4      printf("ni hao xingqiu\n"); 没有创建进程之前，只输出一次
5
6      fork();           //创建一个子进程
7
8      printf("hello world\n");    父子进程共同执行
9
10     while(1);         //防止进程结束
11
12     return 0;
13 }
```

问题 输出 调试控制台 终端 端口

```
● bash-4.2$ g++ 03fork.cpp
○ bash-4.2$ ./a.out
ni hao xingqiu
hello world
hello world
```

○ bash-4.2\$

2> 多个fork创建进程



如果不关注返回值的话，有n个fork，会产生 2^n 个进程

3> 关注返回值的情况

```
#include<myhead.h>
int main(int argc, const char *argv[])
{
    pid_t pid = -1;

    pid = fork();           //创建一个子进程，父进程会将返回值赋值给父进程中的pid变量
                           //子进程会将返回值赋值给子进程中的pid变量

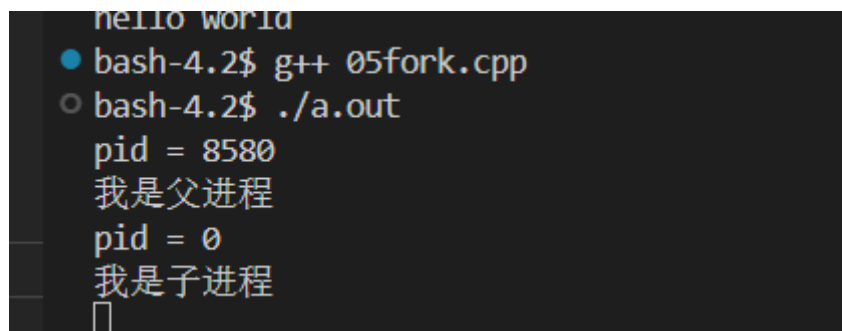
    printf("pid = %d\n", pid);    //对于父进程而言会得到大于0的数字，对于子进程而言会得到0

    //对pid进程判断
    if(pid > 0)
    {
        //父进程要做执行的代码
        printf("我是父进程\n");
    }else if(pid == 0)
    {
        //子进程要执行的代码
        printf("我是子进程\n");
    }else
    {
        perror("fork error");
        return -1;
    }

    while(1);              //防止进程结束

    return 0;
}
```

效果图



```
hello world
● bash-4.2$ g++ 05fork.cpp
○ bash-4.2$ ./a.out
pid = 8580
我是父进程
pid = 0
我是子进程
```

4> 父子进程并发执行的案例

```
#include<myhead.h>
int main(int argc, const char *argv[])
{
    pid_t pid = -1;
```

```

pid = fork();           //创建一个子进程，父进程会将返回值赋值给父进程中的pid变量
                        //子进程会将返回值赋值给子进程中的pid变量

//对pid进程判断
if(pid > 0)
{
    while(1)
    {
        //父进程要做执行的代码
        printf("我是父进程1111\n");
        sleep(1);
    }

}else if(pid == 0)
{
    while(1)
    {
        //子进程要执行的代码
        printf("我是子进程\n");
        sleep(1);
    }

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

return 0;
}

```

2.2 父子进程号的获取：getpid、getppid

```

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

pid_t getpid(void);
功能：获取当前进程的进程号
参数：无
返回值：当前进程的进程号

pid_t getppid(void);
功能：获取当前进程的父进程pid号
参数：无
返回值：当前进程的父进程pid

```

2.3 进程退出：exit/_exit

上述两个函数都可以完成进程的退出，区别是在退出进程时，是否刷新标准IO的缓冲区

exit属于库函数，使用该函数退出进程时，会刷新标准IO的缓冲区后退出

_exit属于系统调用（内核提供的函数），使用该函数退出进程时，不会刷新标准IO的缓冲区

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

```
void exit(int status);
```

功能：退出当前进程，并刷新当前进程打开的标准IO的缓冲区

参数：进程退出时的状态，会将改制 与 0377进行位与运算后，返回给回收资源的进程

返回值：无

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

```
void _exit(int status);
```

功能：退出当前进程，不刷新当前进程打开的标准IO的缓冲区

参数：进程退出时的状态，会将改制 与 0377进行位与运算后，返回给回收资源的进程

返回值：无

2.4 进程资源回收：wait、waitpid

有两个函数可以完成对进程资源的回收

wait是阻塞回收任意一个子进程的资源函数

waitpid：可以阻塞，也可以非阻塞完成对指定的进程号进程资源回收

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

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

```
pid_t wait(int *status);
```

功能：阻塞回收子进程的资源

参数：接收子进程退出时的状态，获取子进程退出时的状态与0377进行位与后的结果，如果不愿意接收，可以填NULL

返回值：成功返回回收资源的那个进程的pid号，失败返回-1并置位错误码

```
pid_t waitpid(pid_t pid, int *status, int options);
```

功能：可以阻塞也可以非阻塞回收指定进程的资源

参数1：进程号

>0：表示回收指定的进程，进程号位pid（常用）

=0：表示回收当前进程所在进程组中的任意一个子进程

=-1：表示回收任意一个子进程（常用）

<-1：表示回收指定进程组中的任意一个子进程，进程组id为给定的pid的绝对值

参数2：接收子进程退出时的状态，获取子进程退出时的状态与0377进行位与后的结果，如果不愿意接收，可以填NULL

参数3：是否阻塞

0：表示阻塞等待

: waitpid(-1, &status, 0);等价于 wait(&status)

WNOHANG：表示非阻塞

返回值：

>0：返回的是成功回收的子进程pid号

=0：表示本次没有回收到子进程

=-1：出错并置位错误码

```
#include<myhead.h>
```

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

```
{
```

```
    pid_t pid = -1;           //定义用于存储进程号的变量
```

```
    //创建进程
```



```

int get_file_size(const char *srcfile, const char *destfile)
{
    //以只读的形式打开源文件，以创建写的形式打开目标文件
    int srcfd, destfd;          //记录源文件和目标文件的文件描述符
    if((srcfd = open(srcfile, O_RDONLY)) == -1)
    {
        perror("open secfile error");
        return -1;
    }
    if((destfd = open(destfile, O_WRONLY|O_CREAT|O_TRUNC, 0664)) == -1)
    {
        perror("open destfile error");
        return -1;
    }

    //求源文件的大小
    int len = lseek(srcfd, 0, SEEK_END);

    //关闭文件
    close(srcfd);
    close(destfd);

    return len;                //将源文件大小返回
}

//定义拷贝文件的函数
int copy_file(const char *srcfile, const char *destfile, int start, int len)
{
    //以只读的形式打开源文件，以只写的形式打开目标文件
    int srcfd, destfd;          //记录源文件和目标文件的文件描述符
    if((srcfd = open(srcfile, O_RDONLY)) == -1)
    {
        perror("open secfile error");
        return -1;
    }
    if((destfd = open(destfile, O_WRONLY)) == -1)
    {
        perror("open destfile error");
        return -1;
    }

    //将两个文件的光标位置统一
    lseek(srcfd, start, SEEK_SET);
    lseek(destfd, start, SEEK_SET);

    //开始拷贝
    char buf[128] = "";          //数据的搬运工
    int sum = 0;                 //记录拷贝的总字节数
    while(1)
    {
        int res = read(srcfd, buf, sizeof(buf));          //从源文件中读取数据
        sum += res;          //将拷贝的字节数累加
        if(res==0 || sum>=len)
        {
            write(destfd, buf, res-(sum-len));          //将最后一次内容拷贝
            break;          //文件结束
        }
    }
}

```

```

        write(destfd, buf, res);                                //从源文件读多少，写入目标文件多
少
    }

    return 0;
}

/*****主程序*****/
int main(int argc, const char *argv[])
{
    //使用外部传参，将要拷贝的文件以及存储文件传进来
    if(argc != 3)
    {
        printf("input file error\n");
        printf("usage:./a.out srcfile destfile\n");
        return -1;
    }

    //获取源文件的长度，并且创建目标文件
    int len = get_file_size(argv[1], argv[2]);

    //创建父子进程，分别执行拷贝函数
    pid_t pid = fork();
    if(pid > 0)
    {
        //父进程拷贝前半部分内容
        copy_file(argv[1], argv[2], 0, len/2);                //从开头位置拷贝，拷贝len/2内容

        //回收子进程资源
        wait(NULL);

    }else if(pid == 0)
    {
        //子进程拷贝后半部分内容
        copy_file(argv[1], argv[2], len/2, len-len/2);        //从中间开始拷贝，拷贝剩余的
内容

        //退出进程
        exit(EXIT_SUCCESS);
    }else
    {
        perror("fork error");
        return -1;
    }

    printf("拷贝成功\n");
    return 0;
}

```

2.5 父进程创建两个进程并为其收尸

```

#include<myhead.h>
int main(int argc, const char *argv[])

```

```

{
    pid_t pid = fork();           //创建大儿子
    if(pid > 0)
    {
        //父进程
        pid_t pid_2 = fork();    //创建二儿子
        if(pid_2 > 0)
        {
            //父进程
            printf("我是父进程\n");
            //回收两个子进程的资源
            wait(NULL);
            wait(NULL);

        }else if(pid_2 == 0)
        {
            sleep(3);
            //二儿子进程内容
            printf("我是进程2\n");
            exit(EXIT_SUCCESS);
        }
    }else if(pid == 0)
    {
        sleep(3);
        //大儿子进程
        printf("我是进程1\n");
        exit(EXIT_SUCCESS);

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

    return 0;
}

```

```

bash-4.2$ ./a.out
我是父进程
我是进程1
我是进程2
bash-4.2$ ps -ajx|grep a.out
17139 18510 18510 17139 pts/3 18510 S+ 1000 0:00 ./a.out
18510 18511 18510 17139 pts/3 18510 S+ 1000 0:00 ./a.out
18510 18512 18510 17139 pts/3 18510 S+ 1000 0:00 ./a.out
18234 18524 18523 18234 pts/4 18523 R+ 1000 0:00 grep a.out
18234 18598 18597 18234 pts/4 18597 R+ 1000 0:00 grep a.out
bash-4.2$

```

2.6 僵尸进程和孤儿进程

1> 孤儿进程：当前进程还正在运行，其父进程已经退出了。

说明：每个进程退出后，其分配的系统资源应该由其父进程进行回收，否则会造成资源的浪费

```

#include<myhead.h>
int main(int argc, const char *argv[])
{
    pid_t pid = -1;           //定义用于存储进程号的变量

```



```

}else if(pid == 0)
{
    //子进程程序代码

    sleep(5);

    exit(EXIT_SUCCESS);    //子进程退出

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

while(1);                //防止进程退出

return 0;
}

```

我是父进程
我是父进程
我是父进程
我是父进程
我是父进程
我是父进程
我是父进程
我是父进程
我是父进程

5秒后，子进程退出，
父进程没有为其收尸，
子进程称为僵尸进程

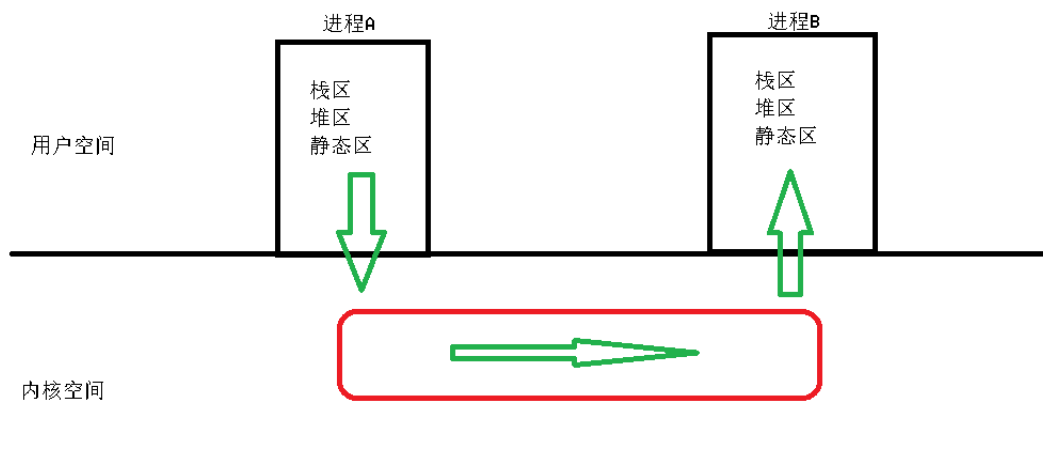
```

bash-4.2$ ps -ajx|grep a.out 前5秒父子进程共存
17139 20101 20101 17139 pts/3 20101 S+ 1000 0:00 ./a.out
20101 20102 20101 17139 pts/3 20101 S+ 1000 0:00 ./a.out
18234 20105 20104 18234 pts/4 20104 R+ 1000 0:00 grep a.out
bash-4.2$ ps -ajx|grep a.out
17139 20101 20101 17139 pts/3 20101 S+ 1000 0:00 ./a.out
20101 20102 20101 17139 pts/3 20101 Z+ 1000 0:00 [a.out] <defunct>
18234 20141 20140 18234 pts/4 20140 R+ 1000 0:00 grep a.out
bash-4.2$

```

三、 进程间通信 IPC

- 1> 由于多个进程的用户空间是相互独立的，其栈区、堆区、静态区的数据都是彼此私有的，所以不可能通过用户空间中的区域完成多个进程之间数据的通信
- 2> 可以使用外部文件来完成多个进程之间数据的传递，一个进程向文件中写入数据，另一个进程从文件中读取数据。该方式要必须保证写进程先执行，然后再执行读进程，要保证进程执行的同步性
- 3> 我们可以利用内核空间来完成对数据的通信工作，本质上，在内核空间创建一个特殊的区域，一个进程向该区域中存放数据，另一个进程可以从该区域中读取数据



4> 引入原因：用户空间中的数据，不能作为多个进程之间数据交换的容器

```
#include<myhead.h>

int num = 520;           //定义一个全局变量

int main(int argc, const char *argv[])
{
    //int num = 520;       //定义一个变量

    pid_t pid = fork();   //创建一个子进程

    if(pid > 0)
    {
        //父进程
        num = 1314;
        printf("父进程中: num = %d\n", num);    //1314

        wait(NULL);      //等待回收子进程

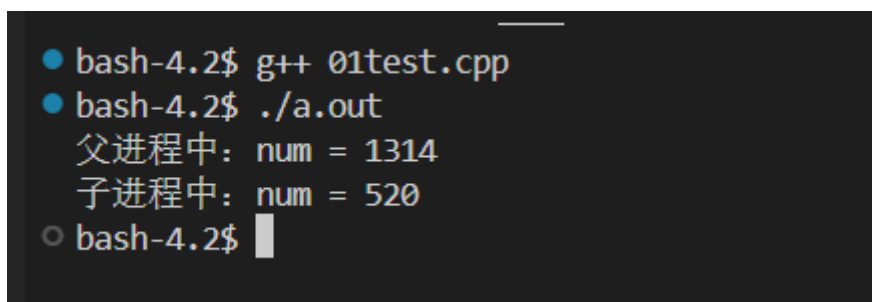
    }else if(pid == 0)
    {
        //子进程
        sleep(3);

        printf("子进程中: num = %d\n", num);    //?1314 520?

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

    return 0;
}
```

效果图：



```
● bash-4.2$ g++ 01test.cpp
● bash-4.2$ ./a.out
父进程中: num = 1314
子进程中: num = 520
○ bash-4.2$
```

3.1 进程间通信的基础概念

1> IPC :interprocess communication 进程间通信

2> 使用内核空间来完成多个进程间相互通信，根据使用的容器或方式不同，分为三类通信机制

3> 进程间通信方式分类

- 1、内核提供的通信方式（传统的通信方式）
 - 无名管道
 - 有名管道
 - 信号
- 2、system v提供的通信方式
 - 消息队列
 - 共享内存
 - 信号量（信号灯集）
- 3、套接字通信：socket 网络通信（跨主机通信）

3.2 无名管道

1> 管道的原理：管道是一种特殊的文件，该文件不用于存储数据，只用于进程间通信。管道分为有名管道和无名管道。

文件类型：bcd-lsp
b: 块设备文件
c: 字符设备文件
d: 目录文件，文件夹
-: 普通文件
l: 链接文件
s: 套接字文件（网络编程）
p: 管道文件

2> 在内核空间创建一个管道通信，一个进程可以将数据写入管道，经由管道缓冲到另一个进程中读取

3> 无名管道：顾名思义就是没有名字的管道，会在内存中创建出该管道，不存在于文件系统，随着进程结束而消失

4> 无名管道仅适用于亲缘进程间通信，不适用于非亲缘进程间通信

5> 无名管道的API

```
#include <unistd.h>

int pipe(int fildes[2]);
```

功能：创建一个无名管道，并返回该管道的两个文件描述符
参数：是一个整型数组，用于返回打开的管道的两端的文件描述符， fildes[0]表示读端
fildes[1]表示写端
返回值：成功返回0，失败返回-1并置位错误码

```
#include<myhead.h>
int main(int argc, const char *argv[])
{
    //可以在此创建管道文件，并返回该管道文件的两端，那么父子进程都会拥有该管道两端的文件描述符
    int fildes[2];           //存放管道文件的两端文件描述符

    //创建无名管道，并返回该管道的两端文件描述符
    if(pipe(fildes) == -1)
    {
        perror("pipe error");
        return -1;
    }
}
```

```

printf("filides[0] = %d, filides[1] = %d\n", filides[0], filides[1]);    //3
4

pid_t pid = fork();                //创建一个子进程

//也不可以在此创建管道文件，因为如果在此创建，那么父进程中和子进程中会分别创建一个无名管道

if(pid > 0)
{
    //父进程
    //不用读端，就关闭
    close(filides[0]);

    char wbuf[128] = "hello world";    //想要将该数据发送给子进程使用
    //如果在此创建管道，那么只能父进程使用，子进程用不了，因为子进程中没有管道文件的读端和
    写端文件描述符

    //将上述数据发送给子进程，只需将数据通过写端写入管道文件中
    write(filides[1], wbuf, strlen(wbuf));

    //关闭写端
    close(filides[1]);

    wait(NULL);                //等待回收子进程

}else if(pid == 0)
{
    //子进程
    //关闭写端
    close(filides[1]);

    //通过读端从管道文件中读取数据
    char rbuf[128] = "";

    read(filides[0], rbuf, sizeof(rbuf));

    printf("收到父进程的数据为: %s\n", rbuf);    //将数据输出到终端

    //关闭读端
    close(filides[0]);
    //退出子进程
    exit(EXIT_SUCCESS);
}else
{
    perror("fork error");
    return -1;
}

return 0;
}

```

```

● bash-4.2$ g++ 02pipe.cpp
● bash-4.2$ ./a.out
  fildes[0] = 3, fildes[1] = 4
  收到父进程的数据为: hello world
○ bash-4.2$ 

```

6> 管道通信特点

- 1、管道可以实现自己给自己发消息
- 2、对管道中数据的操作是一次性的，当管道中的数据被读取后，就从管道中消失了，再读取时会被阻塞
- 3、管道文件的大小：64K
- 4、由于返回的是管道文件的文件描述符，所以对管道的操作只能是文件IO相关函数，但是，不可以使用lseek对光标进行偏移，必须做到先进先出
- 5、管道的读写特点：
 - 当读端存在时：写端有多少写多少，直到写满64k后，在write处阻塞
 - 当读端不存在时：写端再向管道中写入数据时，会发生管道破裂，内核空间会向用户空间发射一个SIGPIPE信号，进程收到该信号后，退出
 - 当写端存在时：读端有多少读多少，没有数据，会在read处阻塞
 - 当写端不存在时：读端有多少读多少，没有数据，不会在read处阻塞了
- 6、管道通信是半双工通信方式
 - 单工：只能进程A向B发送消息
 - 半双工：同一时刻只能A向B发消息
 - 全双工：任意时刻，AB可以互相通信

验证自己跟自己通信

```

#include<myhead.h>
int main(int argc, const char *argv[])
{
    int fildes[2];           //存放管道文件的两端文件描述符

    //创建无名管道，并返回该管道的两端文件描述符
    if(pipe(fildes) == -1)
    {
        perror("pipe error");
        return -1;
    }
    printf("fildes[0] = %d, fildes[1] = %d\n", fildes[0], fildes[1]);    //3
    4

    //定义两个容器
    char wbuf[128] = "ni hao xingqiu";
    char rbuf[128] = "";

    //将wbuf中的数据写入管道文件中，从管道中读取数据放入rbuf中
    write(fildes[1], wbuf, strlen(wbuf));    //将数据写入管道文件中

    read(fildes[0], rbuf, sizeof(rbuf));    //从管道文件中读取数据

    printf("rbuf = %s\n", rbuf);    //输出到终端

    //关闭文件描述符

```

```

close(fildes[0]);
close(fildes[1]);

return 0;
}

```

对管道文件大小的验证

```

#include<myhead.h>
int main(int argc, const char *argv[])
{
    int fildes[2];           //存放管道文件的两端文件描述符

    //创建无名管道，并返回该管道的两端文件描述符
    if(pipe(fildes) == -1)
    {
        perror("pipe error");
        return -1;
    }
    printf("fildes[0] = %d, fildes[1] = %d\n", fildes[0], fildes[1]);    //3
4

    char buf = 'A';         //定义一个字符变量
    int count = 0;          //记录向管道中写入数据的个数

    while(1)
    {
        write(fildes[1], &buf, 1);    //向管道文件中每次写入1字节的数据
        count++;
        printf("count = %d\n", count);
    }

    //关闭文件描述符
    close(fildes[0]);
    close(fildes[1]);

    return 0;
}

```

3.3 有名管道

- 1> 顾名思义就是有名字的管道文件，会在文件系统中创建一个真实存在的管道文件
- 2> 既可以完成亲缘进程间通信，也可以完成非亲缘进程间通信
- 3> 有名管道的API

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

int mkfifo(const char *pathname, mode_t mode);
功能：创建一个管道文件，并存在与文件系统中
参数1：管道文件的名称
参数2：管道文件的权限，内容详见open函数的mode参数
返回值：成功返回0，失败返回-1并置位错误码
```

注意：管道文件被创建后，其他进程就可以进行打开读写操作了，但是，必须要保证当前管道文件的两端都打开后，才能进行读写操作，否则函数会在open处阻塞

4> 案例

create.cpp

```
#include<myhead.h>
int main(int argc, const char *argv[])
{
    //该文件主要负责创建管道文件，注意：如果管道文件已经存在，则mkfifo函数会报错
    if(mkfifo("./myfifo", 0664) == -1)
    {
        perror("mkfifo error");
        return -1;
    }
    printf("管道创建成功\n");

    return 0;
}
```

send.cpp

```
#include<myhead.h>
int main(int argc, const char *argv[])
{
    //打开管道文件
    int sfd = -1;
    if((sfd = open("./myfifo", O_WRONLY)) == -1)
    {
        perror("open error");
        return -1;
    }

    //准备要写入的数据
    char wbuf[128] = "";
    while(1)
    {
        printf("请输入>>>");
        fgets(wbuf, sizeof(wbuf), stdin); //从终端输入数据
        wbuf[strlen(wbuf)-1] = 0; //将换行换成'\0'

        //将数据写入管道
        write(sfd, wbuf, strlen(wbuf));

        if(strcmp(wbuf, "quit") == 0)
    }
```

```

        {
            break;
        }
    }

    //关闭文件
    close(sfd);

    return 0;
}

```

recv.cpp

```

#include<myhead.h>
int main(int argc, const char *argv[])
{
    //打开管道文件
    int rfd = -1;
    if((rfd = open("./myfifo", O_RDONLY)) == -1)
    {
        perror("open error");
        return -1;
    }

    //准备要写入的数据
    char rbuf[128] = "";
    while(1)
    {
        //将容器清空
        bzero(rbuf, sizeof(rbuf));

        //从管道文件中读取数据
        read(rfd, rbuf, sizeof(rbuf));
        printf("收到数据为: %s\n", rbuf);

        if(strcmp(rbuf, "quit") == 0)
        {
            break;
        }
    }

    //关闭文件
    close(rfd);

    return 0;
}

```



```

// create.cpp
#include<myhead.h>
int main(int argc, const char *argv[])
{
    //该文件主要负责创建管道文件，注意：如果管道文件已经存在，则mkfifo函数会报错
    if(mkfifo("./myfifo1", 0664) == -1)
    {
        perror("mkfifo error");
        return -1;
    }
    if(mkfifo("./myfifo2", 0664) == -1)
    {
        perror("mkfifo error");
        return -1;
    }
    printf("管道创建成功\n");
}

// send.cpp
#include<myhead.h>
int main(int argc, const char *argv[])
{
    //打开管道文件
    int sfd = open("./myfifo1", O_WRONLY);
    if(sfd == -1)
    {
        perror("open error");
        return -1;
    }
    //准备要发送的数据
    char wb[1024];
    while(1)
    {
        //从管道文件中读取数据
        read(rfd, rbuf, sizeof(rbuf));
        printf("收到数据为: %s\n", rbuf);
        if(strcmp(rbuf, "quit") == 0)
        {
            break;
        }
        //关闭文件
        close(rfd);
    }
}

// recv.cpp
#include<myhead.h>
int main(int argc, const char *argv[])
{
    //打开管道文件
    int rfd = open("./myfifo1", O_RDONLY);
    if(rfd == -1)
    {
        perror("open error");
        return -1;
    }
    //准备接收数据的缓冲区
    char rb[1024];
    while(1)
    {
        //从管道文件中读取数据
        read(rfd, rbuf, sizeof(rbuf));
        printf("收到数据为: %s\n", rbuf);
        if(strcmp(rbuf, "quit") == 0)
        {
            break;
        }
        //关闭文件
        close(rfd);
    }
}

```

两个进程非亲缘

```

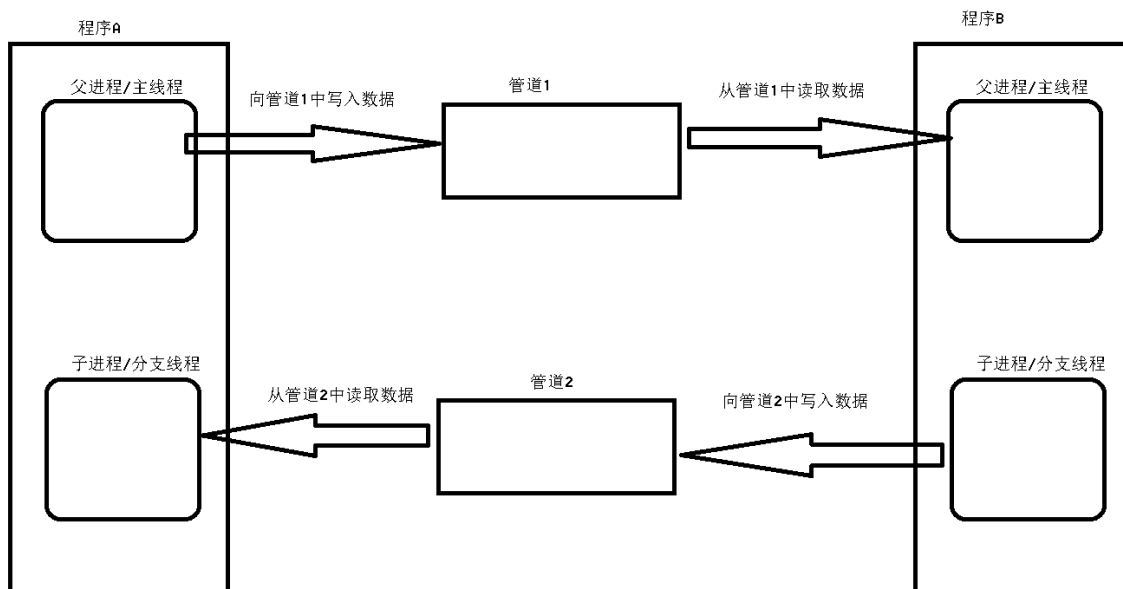
bash-4.2$ ./s.out
请输入>>>hello a
请输入>>>nihao
请输入>>>xingqiu
请输入>>>dudududu
请输入>>>dididi
请输入>>>quit
bash-4.2$

bash-4.2$ ./r.out
收到数据为: hello a
收到数据为: nihao
收到数据为: xingqiu
收到数据为: dudududu
收到数据为: dididi
收到数据为: quit
bash-4.2$

bash-4.2$ pidof s.out
31818
bash-4.2$ pidof r.out
32100
bash-4.2$ ps -ajx | grep s.out
  St  1000  0:00 ./s.out

```

练习：使用有名管道实现，两个进程之间相互通信，提示：可以使用多进程或多线程



1> create.cpp

```

#include<myhead.h>
int main(int argc, const char *argv[])
{
    //该文件主要负责创建管道文件，注意：如果管道文件已经存在，则mkfifo函数会报错
    if(mkfifo("./myfifo1", 0664) == -1)
    {
        perror("mkfifo error");
        return -1;
    }
    if(mkfifo("./myfifo2", 0664) == -1)
    {
        perror("mkfifo error");
        return -1;
    }
    printf("管道创建成功\n");
}

```

```
    return 0;
}
```

2> send.cpp

```
#include<myhead.h>
int main(int argc, const char *argv[])
{
    //创建子进程
    pid_t pid = fork();
    if(pid > 0)
    {
        //父进程，完成向管道1中写入数据
        //打开管道文件
        int sfd = -1;
        if((sfd = open("./myfifo1", O_WRONLY)) == -1)
        {
            perror("open error");
            return -1;
        }

        //准备要写入的数据
        char wbuf[128] = "";
        while(1)
        {
            fgets(wbuf, sizeof(wbuf), stdin); //从终端输入数据
            wbuf[strlen(wbuf)-1] = 0; //将换行换成'\0'

            //将数据写入管道
            write(sfd, wbuf, strlen(wbuf));

            if(strcmp(wbuf, "quit") == 0)
            {
                break;
            }
        }

        //关闭文件
        close(sfd);

        //回收子进程资源
        wait(NULL);
    }else if(pid == 0)
    {
        //子进程，完成从管道2中读取数据
        //打开管道文件
        int rfd = -1;
        if((rfd = open("./myfifo2", O_RDONLY)) == -1)
        {
            perror("open error");
            return -1;
        }
    }
```

```

//准备要写入的数据
char rbuf[128] = "";
while(1)
{
    //将容器清空
    bzero(rbuf, sizeof(rbuf));

    //从管道文件中读取数据
    read(rfd, rbuf, sizeof(rbuf));
    printf("收到数据为: %s\n", rbuf);

    if(strcmp(rbuf, "quit") == 0)
    {
        break;
    }
}

//关闭文件
close(rfd);

//退出进程
exit(EXIT_SUCCESS);
}else
{
    perror("fork error");
    return -1;
}

return 0;
}

```

3> recv.cpp

```

#include<myhead.h>
int main(int argc, const char *argv[])
{
    //创建子进程
    pid_t pid = fork();
    if(pid > 0)
    {
        //父进程，完成向管道1中写入数据
        //打开管道文件
        int rfd = -1;
        if((rfd = open("./myfifo1", O_RDONLY)) == -1)
        {
            perror("open error");
            return -1;
        }

        //准备要写入的数据
        char rbuf[128] = "";
        while(1)
        {
            //将容器清空

```

```

        bzero(rbuf, sizeof(rbuf));

        //从管道文件中读取数据
        read(rfd, rbuf, sizeof(rbuf));
        printf("收到数据为: %s\n", rbuf);

        if(strcmp(rbuf, "quit") == 0)
        {
            break;
        }
    }

    //关闭文件
    close(rfd);

    //回收子进程资源
    wait(NULL);

} else if(pid == 0)
{
    //子进程，完成从管道2中读取数据
    //打开管道文件
    int sfd = -1;
    if((sfd = open("./myfifo2", O_WRONLY)) == -1)
    {
        perror("open error");
        return -1;
    }

    //准备要写入的数据
    char wbuf[128] = "";
    while(1)
    {
        fgets(wbuf, sizeof(wbuf), stdin); //从终端输入数据
        wbuf[strlen(wbuf)-1] = 0; //将换行换成'\0'

        //将数据写入管道
        write(sfd, wbuf, strlen(wbuf));

        if(strcmp(wbuf, "quit") == 0)
        {
            break;
        }
    }

    //关闭文件
    close(sfd);

    //退出进程
    exit(EXIT_SUCCESS);
} else
{
    perror("fork error");
    return -1;
}

```

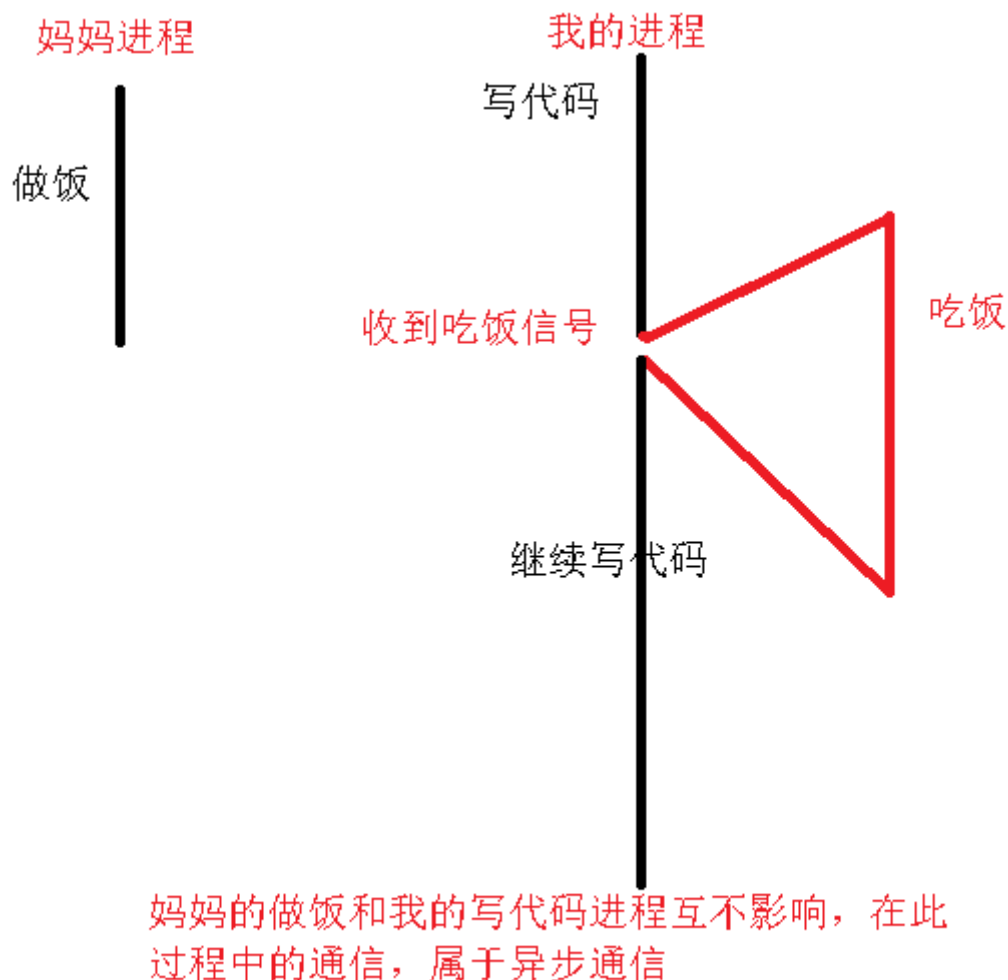
```
return 0;  
}
```

3.4 信号

1> 信号相关概念

- 1、信号是软件模拟硬件的中断功能，信号是软件实现的，中断是硬件实现的
中断：停止当前正在执行的事情，去做另一件事情
- 2、信号是linux内核实现的，没有内核就没有信号的概念
- 3、用户可以给进程发信号：例如键入ctrl+c
内核可以向进程发送信号：例如SIGPIPE
一个进程可以给另一个进程发送信号，需要通过相关函数来完成
- 4、信号通信是属于异步通信工作
同步：表示多个任务有先后顺序的执行，例如去银行办理业务
异步：表示多个任务没有先后顺序执行，例如你在敲代码，你妈妈在做饭

2> 通信原理图



3> 信号的种类及功能

- | | | | | |
|-------------|-------------|-------------|-------------|-------------|
| 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、一共可以发射62个信号，前32个是稳定信号，后面是不稳定信号

2、常用的信号

SIGHUP: 当进程所在的终端被关闭后，终端会给运行在当前终端的每个进程发送该信号，默认结束进程

SIGINT: 中断信号，当用户键入`ctrl + c`时发射出来

SIGQUIT: 退出信号，当用户键入`ctrl + /`是发送，退出进程

SIGKILL: 杀死指定的进程

SIGSEGV: 当指针出现越界访问时，会发射，表示段错误

SIGPIPE: 当管道破裂时会发送该信号

SIGALRM: 当定时器超时后，会发送该信号

SIGSTOP: 暂停进程，当用户键入`ctrl+z`时发射

SIGTSTP: 也是暂停进程

SIGTSTP、**SIGUSR2** : 留给用户自定义的信号，没有默认操作

SIGCHLD: 当子进程退出后，会向父进程发送该信号

3、有两个特殊信号: **SIGKILL**和**SIGSTOP**，这两个信号既不能被捕获，也不能被忽略

4> 对应信号的处理方式有三种: 捕获、忽略、默认

5> 对信号的处理函数: `signal`

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

功能: 将信号与信号处理方式绑定到一起

参数1: 要处理的信号

参数2: 处理方式

SIG_IGN: 忽略

SIG_DFL: 默认，一般信号的默认操作都是杀死进程

typedef void (*sighandler_t)(int): 用户自定义的函数

返回值: 成功返回处理方式的起始地址，失败返回**SIG_ERR**并置位错误码

注意: 只要程序与信号绑定一次，后续但凡程序收到该信号，对应的处理方式就会立即响应

```
#include<myhead.h>

//定义信号处理函数
void handler(int signo)
{
    if(signo == SIGINT)
    {
        printf("用户键入了ctrl + c\n");
    }
}
```

```

/*****主程序*****/
int main(int argc, const char *argv[])
{
    /*1、尝试忽略SIGINT信号,SIG_IGN
    if(signal(SIGINT, SIG_IGN) == SIG_ERR)
    {
        perror("signal error");
        return -1;
    }
    */
    /*2、尝试捕获SIGINT信号
    if(signal(SIGINT, handler) == SIG_ERR)
    {
        perror("signal error");
        return -1;
    }
    */

    //3、尝试默认相关信号的操作
    if(signal(SIGINT, SIG_DFL) == SIG_ERR)
    {
        perror("signal error");
        return -1;
    }

    while(1)
    {
        printf("我真的还想再活五百年\n");
        sleep(1);
    }

    return 0;
}

```

尝试捕获和忽略SIGKILL信号

```

#include<myhead.h>

//定义信号处理函数
void handler(int signo)
{
    if(signo == SIGINT)
    {
        printf("用户键入了ctrl + c\n");
    }
}

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

```

```

/*1、尝试忽略SIGKILL信号,SIG_IGN,函数报错,错误原因参数不合法
if(signal(SIGKILL, SIG_IGN) == SIG_ERR)
{
    perror("signal error");
    return -1;
}*/

/*2、尝试捕获SIGINT信号,执行报错,错误原因参数不合法
if(signal(SIGKILL, handler) == SIG_ERR)
{
    perror("signal error");
    return -1;
}*/

/*3、尝试默认相关信号的操作,执行报错,错误原因参数不合法
if(signal(SIGKILL, SIG_DFL) == SIG_ERR)
{
    perror("signal error");
    return -1;
}*/

while(1)
{
    printf("我真的还想再活五百年\n");
    sleep(1);
}

return 0;
}

```

```

● bash-4.2$ g++ 02signal.cpp
⊗ bash-4.2$ ./a.out
signal error: Invalid argument
● bash-4.2$ g++ 02signal.cpp
⊗ bash-4.2$ ./a.out
signal error: Invalid argument
● bash-4.2$ g++ 02signal.cpp
⊗ bash-4.2$ ./a.out
signal error: Invalid argument
○ bash-4.2$ 

```

6> 使用信号的方式完成对僵尸进程的回收

当子进程退出后, 会向父进程发送一个SIGCHLD的信号, 当父进程收到该信号后, 可以将其进行捕获, 在信号处理函数中, 可以以非阻塞的方式回收僵尸进程

```

#include<myhead.h>
//定义信号处理函数
void handler(int signo)
{

```



```

    if(signo == SIGCHLD)
    {
        //回收僵尸进程
        while(waitpid(-1, NULL, WNOHANG) > 0);
    }
}

int main(int argc, const char *argv[])
{
    //当子进程退出后，会向父进程发送一个SIGCHLD的信号，我们可以将其捕获，在信号处理函数中将子
    进程资源回收
    if(signal(SIGCHLD, handler) == SIG_ERR)
    {
        perror("signal error");
        return -1;
    }

    //创建10个僵尸进程
    for(int i=0; i<10; i++)
    {
        if(fork() == 0)                //当子进程创建出来后，立马扼杀在摇篮中
        {
            exit(EXIT_SUCCESS);
        }
    }

    while(1);

    return 0;
}

```

7> 信号发送函数：kill、raise

```

#include <signal.h>

int kill(pid_t pid, int sig);
功能：向指定进程或进程组发送信号
参数1：进程号或进程组号
    >0:表示向执行进程发送信号
    =0: 向当前进程所在的进程组中的所有进程发送信号
    =-1: 向所有进程发送信号
    <-1:向指定进程组发送信号，进程组的ID号为给定pid的绝对值
参数2：要发送的信号
返回值：成功返回0，失败返回-1并置位错误码

#include <signal.h>

int raise(int sig);

```

功能：向自己发送信号 等价于：kill(getpid(), sig);

参数：要发送的信号

返回值：成功返回0，失败返回非0数组

```
#include<myhead.h>

//定义信号处理函数
void handler(int signo)
{
    if(signo == SIGUSR1)
    {
        printf("逆子，何至于此!!!\n");
        raise(SIGKILL);           //向自己发送一个自杀信号 kill(getpid(),
SIGKILL)
    }
}

int main(int argc, const char *argv[])
{
    //将子进程发送的信号绑定到指定功能中
    if(signal(SIGUSR1, handler) == SIG_ERR)
    {
        perror("signal error");
        return -1;
    }

    //创建父子进程
    pid_t pid = fork();
    if(pid > 0)
    {
        //父进程
        while(1)
        {
            printf("我真的还想再活五百年\n");
            sleep(1);
        }
    }
    else if(pid == 0)
    {
        //子进程
        sleep(5);
        printf("红尘已经看破，叫上父亲一起死吧\n");

        kill(getppid(), SIGUSR1);    //向自己的父进程发送了一个自定义的信号

        exit(EXIT_SUCCESS);         //退出进程
    }

    return 0;
}
```

```

● bash-4.2$ g++ 04kill.cpp
⊗ bash-4.2$ ./a.out
我真的还想再活五百年
我真的还想再活五百年
我真的还想再活五百年
我真的还想再活五百年
我真的还想再活五百年
红尘已经看破，叫上父亲一起死吧
逆子，何至于此!!!
已杀死
○ bash-4.2$ 

```

8> 总结：信号可以完成多个进程间通知作用，但是，不能进行数据传输功能

3.5 system V提供的进程间通信概述

1> 对于内核提供的三种通信方式，对于管道而言，只能实现单向的数据通信，对于信号通信而言，只能完成多进程之间消息的通知，不能起到数据传输的效果。为了解决上述问题，引入的系统 V 进程间通信

2> system V提供的进程间通信方式分别是：消息队列、共享内存、信号量（信号灯集）

3> 有关system V进程间通信对象相关的指令

```

ipcs    可以查看所有的信息（消息队列、共享内存、信号量）
ipcs -q: 可以查看消息队列的信息
ipcs -m: 可以查看共享内存的信息
ipcs -s: 可以查看信号量的信息

ipcrm -q/m/s ID : 可以删除指定ID的IPC对象

```

```

ubuntu@ubuntu:~/IO/day3$ ipcs
----- 消息队列 -----
键          msqid      拥有者  权限      已用字节数  消息

----- 共享内存段 -----
键          shmid      拥有者  权限      字节      连接数  状态
0x00000000  32777      ubuntu  600      524288    2       目标
0x00000000  11         ubuntu  600      67108864  2       目标
0x00000000  16         ubuntu  600      524288    2       目标
0x00000000  65555      ubuntu  600      524288    2       目标
0x00000000  65557      ubuntu  600      524288    2       目标
0x00000000  22         ubuntu  600      524288    2       目标
0x00000000  32791      ubuntu  600      524288    2       目标
0x00000000  24         ubuntu  700      18908     2       目标
0x00000000  32804      ubuntu  600      524288    2       目标
0x00000000  32813      ubuntu  700      408800    2       目标
0x00000000  49         ubuntu  600      16777216  2       目标
0x00000000  56         ubuntu  600      524288    2       目标

----- 信号量数组 -----
键          semid      拥有者  权限      nsems

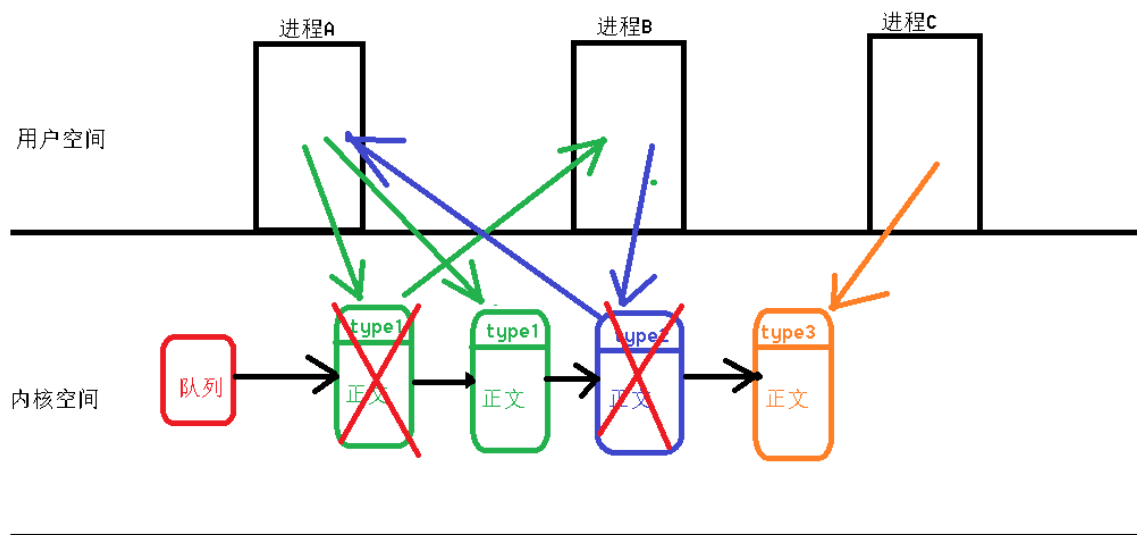
```

4> 上述的三种通信方式，也是借助内核空间完成的相关通信，原理是在内核空间创建出相关的对象容器，在进行进程间通信时，可以将信息放入对象中，另一个进程就可以从该容器中取数据了。

5> 与内核提供的管道、信号通信不同：system V的ipc对象实现了数据传递的容器与程序相分离，也就是说，即使程序以已经结束，但是放入到容器中的数据依然存在，除非将容器手动删除

3.6 消息队列

1> 实现原理



2> 消息队列实现的API

1、创建key值

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

```
key_t ftok(const char *pathname, int proj_id); //ftok("/", 'k');
```

功能：通过给定的文件以及给定的一个随机值，创建出一个4字节整数的key值，用于system V

IPC对象的创建

参数1：一个文件路径，要求是已经存在的文件路径，提供了key值3字节的内容，其中，文件的设备号占1字节，文件的inode号占2字节

参数2：一个随机整数，取后8位（1字节）跟前面的文件共同组成key值，必须是非0的数字

返回值：成功返回key值，失败返回-1并置位错误码

2、通过key值，创建消息队列

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

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

功能：通过给定的key值，创建出一个消息队列的对象，并返回消息队列的句柄ID，后期可以通过该ID操作整个消息队列

参数1：key值，该值可以是IPC_PRIVATE，也可以是ftok创建出来的，前者只用于亲缘进程间的通信

参数2：创建标识

IPC_CREAT：创建并打开一个消息队列，如果消息队列已经存在，则直接打开

IPC_EXCL：确保本次创建处理的是一个新的消息队列，如果消息队列已经存在，则报错，错误码位EEXIST

0664：该消息队列的操作权限

eg：IPC_CREAT|0664 或者 IPC_CREAT|IPC_EXCL|0664

返回值：成功返回消息队列的ID号，失败返回-1并置位错误码

3、向消息队列中存放数据

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

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

功能：向消息队列中存放一个指定格式的消息

参数1：打开的消息队列的id号

参数2：要发送的消息的起始地址，消息一般定义为一个结构体类型，由用户手动定义

```
struct msgbuf {
    long mtype;          /* message type, must be > 0 */    消息的类型
    char mtext[1];       /* message data */    消息正文
    . . .
};
```

参数3：消息正文的大小

参数4：是否阻塞的标识

0：标识阻塞形式向消息队列中存放消息，如果消息队列满了，就在该函数处阻塞

IPC_NOWAIT：标识非阻塞的形式向消息队列中存放消息，如果消息队列满了，直接返回

返回值：成功返回0，失败返回-1并置位错误码

4、从消息队列中取消息

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

功能：从消息队列中取数指定类型的消息放入给定的容器中

参数1：打开的消息队列的id号

参数2：要接收的消息的起始地址，消息一般定义为一个结构体类型，由用户手动定义

```
struct msgbuf {
    long mtype;          /* message type, must be > 0 */    消息的类型
    char mtext[1];       /* message data */    消息正文
    . . .
};
```

参数3：消息正文的大小

参数4：要接收的消息类型

0：表示每次都取消息队列中的第一个消息，无论类型

>0：读取队列中第一个类型为msgtyp的消息

<0：读取队列中的一个消息，消息为绝对值小于msgtyp的第一个消息

eg: 10-->8-->3-->6-->5-->20-->2

-5：会从队列中绝对值小于5的类型的消息中选取第一个消息，就是3

参数5：是否阻塞的标识

0：标识阻塞形式向消息队列中读取消息，如果消息队列空了，就在该函数处阻塞

IPC_NOWAIT：标识非阻塞的形式向消息队列中读取消息，如果消息队列空了，直接返回

返回值：成功返回实际读取的正文大小，失败返回-1并置位错误码

5、销毁消息队列

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

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

功能：对给定的消息队列执行相关的操作，该操作由cmd参数而定

参数1：消息队列的ID号

参数2：要执行的操作

IPC_RMID：删除一个消息队列，当cmd为该值时，第三个参数可以省略填NULL即可

IPC_STAT：表示获取当前消息队列的属性，此时第三个参数就是存放获取的消息队列属性的容器起始地址

IPC_SET：设置当前消息队列的属性，此时第三个参数就是要设置消息队列的属性数据的起始地址

参数3: 消息队列数据容器结构体, 如果第二个参数为IPC_RMID, 则该参数忽略填NULL即可, 如果是 IPC_STAT、IPC_SET填如下结构体:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* Ownership and permissions */
    time_t msg_stime; /* Time of last msgsnd(2) */ 最后
    time_t msg_rtime; /* Time of last msgrcv(2) */ 最后
    time_t msg_ctime; /* Time of last change */ 最后
    unsigned long __msg_cbytes; /* Current number of bytes in queue
    (nonstandard) */ 已用字节数
    msgqnum_t msg_qnum; /* Current number of messages in
    queue */ 消息队列中消息个数
    msglen_t msg_qbytes; /* Maximum number of bytes allowed
    in queue */ 最大消息个数
    pid_t msg_lspid; /* PID of last msgsnd(2) */ 最后
    pid_t msg_lrpid; /* PID of last msgrcv(2) */ 最后
};
```

该结构体的第一个成员类型如下:

```
struct ipc_perm {
    key_t __key; /* Key supplied to msgget(2) */ key
    uid_t uid; /* Effective UID of owner */ 当前进
    gid_t gid; /* Effective GID of owner */ 当前进
    uid_t cuid; /* Effective UID of creator */ 消息队
    gid_t cgid; /* Effective GID of creator */ 消息队
    unsigned short mode; /* Permissions */ 消息队
    unsigned short __seq; /* Sequence number */ 队列号
};
```

返回值: 成功返回0, 失败返回-1并置位错误码

3> 发送端实现

```
#include<myhead.h>

//消息类型的定义
struct msgBuf{
    long mtype; /* 消息的类型 */
    char mtext[1024]; /*消息正文 */
};
#define MSGSZ (sizeof(struct msgBuf)-sizeof(long)) /*正文的大小 */

int main(int argc, const char *argv[])
{
    //1、创建key值, 用于创建一个消息队列
    key_t key = ftok("/", 'k');
```

```

//参数1: 已经存在的路径
//参数2: 是一个随机值
if(key == -1)
{
    perror("ftok error");
    return -1;
}
printf("key = %#x\n", key);          //输出键值

//2、通过key值创建一个消息队列，并返回该消息队列的id
int msqid = -1;
if((msqid = msgget(key, IPC_CREAT|0664)) == -1)
{
    perror("msgget error");
    return -1;
}
printf("msgqid = %d\n", msqid);      //输出id号

//3、向消息队列中存放消息
//组建一个消息
struct msgBuf buf;
while(1)
{
    printf("请输入消息的类型: ");
    scanf("%ld", &buf.mtype);
    getchar();                      //吸收回车
    printf("请输入消息正文:");
    fgets(buf.mtext, MSGSZ, stdin);  //从终端输入数据
    buf.mtext[strlen(buf.mtext)-1] = '\0';    //将换行更换成'\0'

    //将上述组装的消息放入消息队列中，以阻塞的方式将其放入消息队列
    msgsnd(msqid, &buf, MSGSZ, 0);
    printf("消息存入成功\n");

    //判断退出条件
    if(strcmp(buf.mtext, "quit") == 0)
    {
        break;
    }
}
return 0;
}

```

4> 接收端实现

```

#include<myhead.h>

//消息类型的定义
struct msgBuf{
    long mtype;          // 消息的类型
    char mtext[1024];    //消息正文
};
#define MSGSZ (sizeof(struct msgBuf)-sizeof(long))    //正文的大小

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

```

```

//1、创建key值，用于创建一个消息队列
key_t key = ftok("/", 'k');
//参数1: 已经存在的路径
//参数2: 是一个随机值
if(key == -1)
{
    perror("ftok error");
    return -1;
}
printf("key = %#x\n", key);      //输出键值

//2、通过key值创建一个消息队列，并返回该消息队列的id
int msqid = -1;
if((msqid = msgget(key, IPC_CREAT|0664)) == -1)
{
    perror("msgget error");
    return -1;
}
printf("msgqid = %d\n", msqid);  //输出id号

//3、从消息队列中取消息
//组建一个消息
struct msgBuf buf;
while(1)
{
    //清空容器
    bzero(&buf, sizeof(buf));

    //读取消息
    msgrcv(msqid, &buf, MSGSZ, 1, 0);
    //参数4: 表示读取的消息类型
    //参数5: 表示是否阻塞读取

    printf("读取到的消息为:%s\n", buf.mtext);

    if(strcmp(buf.mtext, "quit") == 0)
    {
        break;
    }
}

//4、删除消息队列
if(msgctl(msqid, IPC_RMID, NULL) == -1)
{
    perror("msgctl error");
    return -1;
}

return 0;
}

```

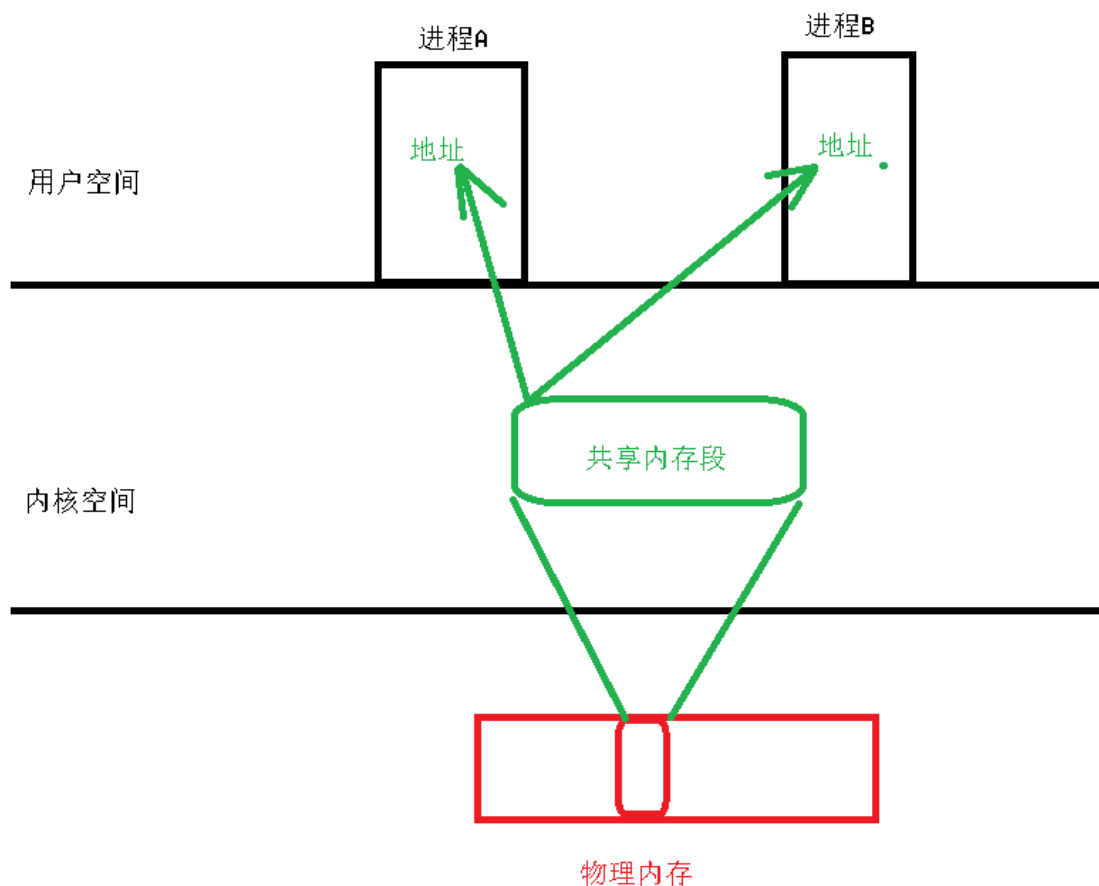
5> 注意事项:

- 1、对于消息而言，由两部分组成：消息的类型和消息正文，消息结构体由用户自定义
- 2、对于消息队列而言，任意一个进程都可以向消息队列中发送消息，也可以从消息队列中取消息

- 3、多个进程，使用相同的key值打开的是同一个消息队列
- 4、对消息队列中的消息读取操作是一次性的，被读取后，消息队列中不存在该消息了
- 5、消息队列的大小：16K7

3.7 共享内存

1> 原理图



2> 共享内存的API

1、创建key值

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

```
key_t ftok(const char *pathname, int proj_id); //ftok("/", 'k');
```

功能：通过给定的文件以及给定的一个随机值，创建一个4字节整数的key值，用于system V IPC对象的创建

参数1：一个文件路径，要求是已经存在的文件路径，提供了key值3字节的内容，其中，文件的设备号占1字节，文件的inode号占2字节

参数2：一个随机整数，取后8位（1字节）跟前面的文件共同组成key值，必须是非0的数字

返回值：成功返回key值，失败返回-1并置位错误码

2、通过key值创建共享内存段

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

```
int shmget(key_t key, size_t size, int shmflg);
```

功能：申请指定大小的物理内存，映射到内核空间，创建出共享内存段

参数1: key值，可以是IPC_PRIVATE,也可以是ftok创建出来的key值

参数2: 申请的大小，是一页（4096字节）的整数倍，并且向上取整

参数3: 创建标识

IPC_CREAT: 创建并打开一个共享内存，如果共享内存已经存在，则直接打开

IPC_EXCL: 确保本次创建处理的是一个新的共享内存，如果共享内存已经存在，则报错，错误码位EEXIST

0664: 该共享内存的操作权限

eg: IPC_CREAT|0664 或者 IPC_CREAT|IPC_EXCL|0664

返回值：成功返回共享内存段的id，失败返回-1并置位错误码

3、将共享内存段的地址映射到用户空间

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

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

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

功能：将共享内存段映射到用户空间

参数1: 共享内存的id号

参数2: 物理内存的起始地址，一般填NULL，由系统自动选择一个合适的对齐页

参数3: 对共享内存段的操作

0: 表示读写操作

SHM_RDONLY: 只读

返回值：成功返回用于操作共享内存的指针，失败返回(void*)-1并置位错误码

4、释放共享内存的映射关系

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

功能：将进程与共享内存的映射取消

参数：共享内存的指针

返回值：成功返回0，失败返回-1并置位错误码

5、共享内存的控制函数

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

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

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

功能：根据给定的不同的cmd执行不同的操作

参数1: 共享内存的ID

参数2: 要操作的指令

IPC_RMID: 删除共享内存段，第三个参数可以省略

IPC_STAT: 获取当前共享内存的属性

IPC_SET: 设置当前共享内存的属性

参数3: 如果参数2为IPC_RMID，则参数3可以省略填NULL，如果参数2为另外两个，参数3填如下

结构体变量

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Last change time */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};
```

该结构体的第一个成员结构体:

```
struct ipc_perm {
    key_t    __key;    /* Key supplied to shmget(2) */
    uid_t    uid;      /* Effective UID of owner */
    gid_t    gid;      /* Effective GID of owner */
};
```

```

        uid_t      cuid;      /* Effective UID of creator */
        gid_t      cgid;      /* Effective GID of creator */
        unsigned short mode;   /* Permissions + SHM_DEST and
                                SHM_LOCKED flags */
        unsigned short __seq;  /* Sequence number */
    };
    返回值：成功返回0，失败返回hi-1并置位错误码

```

3> 发送端流程

```

#include<myhead.h>

#define PAGE_SIZE 4096          //一页的大小

int main(int argc, const char *argv[])
{
    //1、创建key值
    key_t key = ftok("/", 'k');
    if(key == -1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key = %x\n", key);    //输出key值

    //2、通过key值创建共享内存段
    int shmid = -1;
    if((shmid= shmget(key, PAGE_SIZE, IPC_CREAT|0664)) == -1)
    {
        perror("shmget error");
        return -1;
    }
    printf("shmid = %d\n", shmid);

    //3、将共享内存段映射到用户空间
    char *addr = (char *)shmat(shmid, NULL, 0);
    //NULL表示让系统自动寻找对齐页
    //0表示对该共享内存段的操作是读写操作打开
    if(addr == (void*)-1)
    {
        perror("shmat error");
        return -1;
    }
    printf("addr = %p\n", addr);    //输出共享内存段映射的地址

    //4、对共享内存进行操作
    while(1)
    {
        printf("请输入>>>");
        fgets(addr, PAGE_SIZE, stdin);    //从终端输入数据放入共享内存中
        addr[strlen(addr)-1] = 0;

        if(strcmp(addr, "quit") == 0)
        {
            break;
        }
    }
}

```

```

}

sleep(5);          //休眠5秒
printf("结束吧\n");

//5、取消映射
if(shmdt(addr) == -1)
{
    perror("取消映射\n");
    return -1;
}

return 0;
}

```

4> 接收端流程

```

#include<myhead.h>

#define PAGE_SIZE 4096          //一页的大小

int main(int argc, const char *argv[])
{
    //1、创建key值
    key_t key = ftok("/", 'k');
    if(key == -1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key = %#x\n", key);    //输出key值

    //2、通过key值创建共享内存段
    int shmid = -1;
    if((shmid= shmget(key, PAGE_SIZE, IPC_CREAT|0664)) == -1)
    {
        perror("shmget error");
        return -1;
    }
    printf("shmid = %d\n", shmid);

    //3、将共享内存段映射到用户空间
    char *addr = (char *)shmat(shmid, NULL, 0);
    //NULL表示让系统自动寻找对齐页
    //0表示对该共享内存段的操作是读写操作打开
    if(addr == (void*)-1)
    {
        perror("shmat error");
        return -1;
    }
    printf("addr = %p\n", addr);    //输出共享内存段映射的地址

    //4、对共享内存进行操作

```

```

while(1)
{
    sleep(2);
    printf("读取到消息为: %s\n", addr);    //通过地址访问共享内存中的数据

    if(strcmp(addr, "quit") == 0)
    {
        break;
    }
}

//5、取消映射
if(shmdt(addr) == -1)
{
    perror("取消映射\n");
    return -1;
}

//6、删除共享内存段
if(shmctl(shmid, IPC_RMID, NULL) == -1)
{
    perror("shmctl error");
    return -1;
}

return 0;
}

```

5> 注意:

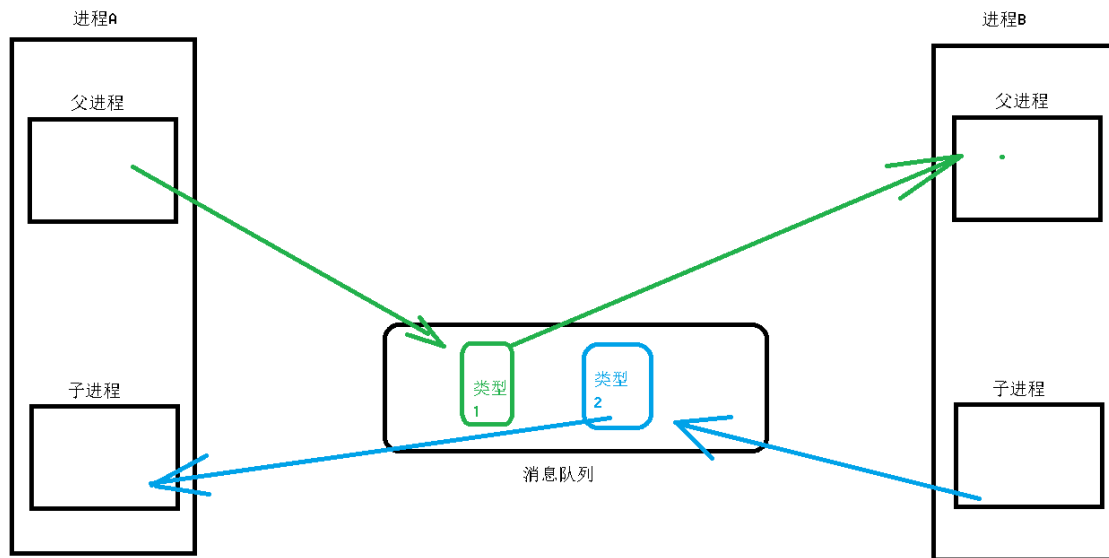
1、共享内存是多个进程共享同一个内存空间，使用时可能会产生竞态，为了解决这个问题，共享内存一般会跟**信号量**一起使用，完成**进程的同步**功能

2、共享内存VS消息队列：**消息队列能够保证数据的不丢失性，而共享内存能够保证数据的时效性**

3、对共享内存的读取操作不是一次性的，当读取后，数据依然存放在共享内存中

4、使用共享内存，跟正常使用指针是一样的，使用时，无需再进行用户空间与内核空间的切换了，所以说，共享内存是所有进程间通信方式中**效率最高**的一种通信方式。

练习：使用消息队列完成两个进程间相互通信



test1.cpp

```
#include<myhead.h>

//消息类型的定义
struct msgBuf{
    long mtype;        // 消息的类型
    char mtext[1024];   //消息正文
};
#define MSGSZ (sizeof(struct msgBuf)-sizeof(long))    //正文的大小

int main(int argc, const char *argv[])
{
    //1、创建key值，用于创建一个消息队列
    key_t key = ftok("/", 'k');
    //参数1: 已经存在的路径
    //参数2: 是一个随机值
    if(key == -1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key = %#x\n", key);    //输出键值

    //2、通过key值创建一个消息队列，并返回该消息队列的id
    int msqid = -1;
    if((msqid = msgget(key, IPC_CREAT|0664)) == -1)
    {
        perror("msgget error");
        return -1;
    }
    printf("msgqid = %d\n", msqid);    //输出id号

    //创建子进程
    pid_t pid = fork();
    if(pid > 0)
    {
        //父进程
        //3、向消息队列中存放消息
    }
}
```

```

//组建一个消息
struct msgBuf buf = {.mtype = 1};          //向消息队列中放入类型为1的消息
while(1)
{
    fgets(buf.mtext, MSGSZ, stdin);        //从终端输入数据
    buf.mtext[strlen(buf.mtext)-1] = '\0';    //将换行更换成'\0'

    //将上述组装的消息放入消息队列中，以阻塞的方式将其放入消息队列
    msgsnd(msqid, &buf, MSGSZ, 0);

    //判断退出条件
    if(strcmp(buf.mtext, "quit") == 0)
    {
        break;
    }
}

}else if(pid == 0)
{
    //子进程
    struct msgBuf buf;
    while(1)
    {
        //清空容器
        bzero(&buf, sizeof(buf));

        //读取消息
        msgrcv(msqid, &buf, MSGSZ, 2, 0);    //从消息队列中读取类型2的消息
        //参数4: 表示读取的消息类型
        //参数5: 表示是否阻塞读取

        printf("读取到的消息为:%s\n", buf.mtext);

        if(strcmp(buf.mtext, "quit") == 0)
        {
            break;
        }
    }

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

return 0;
}

```

test2.cpp

```

#include<myhead.h>

//消息类型的定义
struct msgBuf{
    long mtype;        // 消息的类型
    char mtext[1024];    //消息正文

```

```

    };
#define MSGSZ (sizeof(struct msgBuf)-sizeof(long)) //正文的大小

int main(int argc, const char *argv[])
{
    //1、创建key值，用于创建一个消息队列
    key_t key = ftok("/", 'k');
    //参数1: 已经存在的路径
    //参数2: 是一个随机值
    if(key == -1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key = %#x\n", key); //输出键值

    //2、通过key值创建一个消息队列，并返回该消息队列的id
    int msqid = -1;
    if((msqid = msgget(key, IPC_CREAT|0664)) == -1)
    {
        perror("msgget error");
        return -1;
    }
    printf("msqid = %d\n", msqid); //输出id号

    //创建子进程
    pid_t pid = fork();
    if(pid > 0)
    {
        //父进程
        struct msgBuf buf;
        while(1)
        {
            //清空容器
            bzero(&buf, sizeof(buf));

            //读取消息
            msgrcv(msqid, &buf, MSGSZ, 1, 0); //从消息队列中读取类型2的消息
            //参数4: 表示读取的消息类型
            //参数5: 表示是否阻塞读取

            printf("读取到的消息为:%s\n", buf.mtext);

            if(strcmp(buf.mtext, "quit") == 0)
            {
                break;
            }
        }
    }

    }else if(pid == 0)
    {
        //子进程
        //3、向消息队列中存放消息
        //组建一个消息
        struct msgBuf buf = {.mtype = 2}; //向消息队列中放入类型为1的消息
    }
}

```



```

while(1)
{
    fgets(buf.mtext, MSGSZ, stdin);          //从终端输入数据
    buf.mtext[strlen(buf.mtext)-1] = '\0';    //将换行更换成'\0'

    //将上述组装的消息放入消息队列中，以阻塞的方式将其放入消息队列
    msgsnd(msqid, &buf, MSGSZ, 0);

    //判断退出条件
    if(strcmp(buf.mtext, "quit") == 0)
    {
        break;
    }
}

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

return 0;
}

```

效果图

```

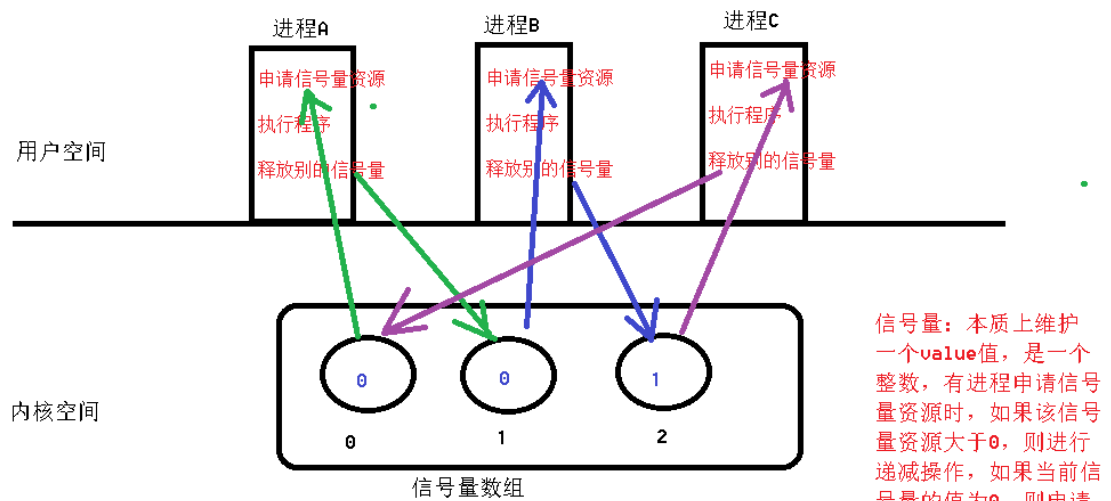
ubuntu@ubuntu:~/IO/day1/test$ g++ 05test1.cpp -o t1.out
ubuntu@ubuntu:~/IO/day1/test$ g++ 06test2.cpp -o t2.out
ubuntu@ubuntu:~/IO/day1/test$ ./t1.out
key = 0x6b010002
msqid = 2
读取到的消息为:hello a
nihao
dudududu
didididi
读取到的消息为:xingqiu
读取到的消息为:dadadad
[]

ubuntu@ubuntu:~/IO/day1/test$ ./t2.out
key = 0x6b010002
msqid = 2
hello a
读取到的消息为:nihao
读取到的消息为:dudududu
读取到的消息为:didididi
xingqiu
dadadad
[]

```

3.8 信号量（信号灯集）

1> 原理图



信号量：本质上维护一个value值，是一个整数，有进程申请信号量资源时，如果该信号量资源大于0，则进行递减操作，如果当前信号量的值为0，则申请资源的进程处于阻塞状态，直到另一个进程将该value值增加到大于0当有进程对该信号量进行释放操作时，该value值会递增

2> 信号量相关API

1、创建key值

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

```
key_t ftok(const char *pathname, int proj_id); //ftok("/", 'k');
```

功能：通过给定的文件以及给定的一个随机值，创建出一个4字节整数的key值，用于system v IPC对象的创建

参数1：一个文件路径，要求是已经存在的文件路径，提供了key值3字节的内容，其中，文件的设备号占1字节，文件的inode号占2字节

参数2：一个随机整数，取后8位（1字节）跟前面的文件共同组成key值，必须是非0的数字

返回值：成功返回key值，失败返回-1并置位错误码

2、通过key值创建信号量集

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

```
int semget(key_t key, int nsems, int semflg);
```

功能：通过给定的key值创建一个信号量集

参数1：key值，该值可以是IPC_PRIVATE,也可以是ftok创建出来的，前者只用于亲缘进程间的通信

参数2：信号量数组中信号量的个数

参数3：创建标识

IPC_CREAT：创建并打开一个信号量集，如果信号量集已经存在，则直接打开

IPC_EXCL：确保本次创建处理的是一个新的信号量集，如果信号量集已经存在，则报错，错误码位EEXIST

0664：该信号量集的操作权限

eg：IPC_CREAT|0664 或者 IPC_CREAT|IPC_EXCL|0664

返回值：成功返回信号量集的id，失败返回-1并置位错误码

3、关于信号量集的操作：P（申请资源）V（释放资源）

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

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

功能：完成对信号量数组的操作

参数1：信号量数据ID号

参数2：有关信号量操作的结构体变量起始地址，该结构体中包含了操作的信号量编号和申请还是释放的操作

```
struct sembuf
{
    unsigned short sem_num; /* semaphore number */    要操作的信号量的编号
    short          sem_op;  /* semaphore operation */  要进行的操作，大于0
表示释放资源，小于0表示申请资源
    short          sem_flg; /* operation flags */      操作标识位，0标识阻塞方
式，IPC_NOWAIT表示非阻塞
}
```

参数3：本次操作的信号量的个数

返回值：成功返回0，失败返回-1并置位错误码

4、关于信号量集的控制函数

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

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

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

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

功能：执行有关信号量集的控制函数，具体控制内容取决于cmd

参数1：信号量集的ID

参数2：要操作的信号量的编号，编号是从0开始

参数3：要执行的操作

IPC_RMID：表示删除信号量集，cmd为该值时，参数2可以忽略，参数4可以不填

SETVAL：表示对参数2对应的信号量进行设置操作（初始值）

GETVAL：表示对参数2对应的信号量进行获取值操作

SETALL：设置信号量集中所有信号量的值

GETALL：获取信号量集中的所有信号量的值

IPC_STAT：表示获取当前信号量集的属性

IPC_SET：表示设置当前信号量集的属性、

参数4：根据不同的cmd值，填写不同的参数值，所以该处是一个共用体变量

```
union semun {
    int          val; /* Value for SETVAL */    设置信号量的值
    struct semid_ds *buf; /* Buffer for IPC_STAT, IPC_SET */ 关于
信号量集属性的操作
    unsigned short *array; /* Array for GETALL, SETALL */    对于信
号量集中所有信号量的操作
    struct seminfo *__buf; /* Buffer for IPC_INFO
(Linux-specific) */
};
```

返回值：成功时：SETVAL、IPC_RMID返回0，GETVAL返回当前信号量的值，失败返回-1并置位错误码

例如：

1) 给0号信号量设置初始值为1

```
union semun us; /*定义一个共用体变量
us.val = 1; /*对该共用体变量赋值
semctl(semid, 0, SETVAL, us); //该函数就完成了对0号信号量设置初始值为1的操作
```

2) 删除信号量集

```
semctl(semid, 0, IPC_RMID);
```

3> 将上述函数进行二次封装，封装成为只有信号量集的创建、申请资源、释放资源、销毁信号量集

sem.h

```
#ifndef _SEM_H_
#define _SEM_H_

//创建信号量集并初始化:semcount表示本次创建的信号量集中信号灯的个数
int create_sem(int semcount);

//申请资源操作, semno表示要被申请资源的信号量编号
int P(int semid, int semno);

//释放资源操作, semno表示要被释放资源的信号量编号
int V(int semid, int semno);

//删除信号量集
int delete_sem(int semid);

#endif
```

sem.cpp

```
#include<myhead.h>
union semun {
    int val; // 设置信号量的值
    struct semid_ds *buf; //关于信号量集属性的操作
    unsigned short *array; //对于信号量集中所有信号量的操作
    struct seminfo *__buf; /* Buffer for IPC_INFO(Linux-specific) */
};

//定义一个关于对信号量初始化函数
int init_sem(int semid, int semno)
{
    int val = -1;
    printf("请输入第%d个信号量的初始值: ", semno+1); //让用户输入信号量的初始值
    scanf("%d", &val);
    getchar(); //吸收回车，以免影响其他程序

    //调用semctl完成设置
    union semun us;
    us.val = val;

    if(semctl(semid, semno, SETVAL, us) == -1)
    {
        perror("semctl error");
        return -1;
    }
    return 0;
}

//创建信号量集并初始化:semcount表示本次创建的信号量集中信号灯的个数
int create_sem(int semcount)
```

```

{
    //1、创建key值
    key_t key = ftok("/", 'k');
    if(key == -1)
    {
        perror("ftok error");
        return -1;
    }

    //2、通过key值创建信号量集
    int semid = -1;
    if((semid = semget(key, semcount, IPC_CREAT|IPC_EXCL|0664)) == -1)
    {
        if(errno == EEXIST)           //表示信号量集已经存在，直接打开即可
        {
            semid = semget(key, semcount, IPC_CREAT|0664);    //将信号量集直接打开
            return semid;
        }
        perror("semget error");
        return -1;
    }

    //3、循环将信号量集中的所有信号量进行初始化
    //该操作，只有在第一次创建信号量集时需要进行操作，后面再打开该信号量集时，就无需进行初始化
    操作了
    for(int i=0; i<semcount; i++)
    {
        init_sem(semid, i);           //调用自定义函数将每个信号量进行初始化
    }

    //将信号量集的id返回
    return semid;
}

//申请资源操作，semno表示要被申请资源的信号量编号
int P(int semid, int semno)
{
    //定义一个结构体变量
    struct sembuf buf;
    buf.sem_num = semno;           //要操作的信号编号
    buf.sem_op = -1;               //-1表示要申请该信号量的资源
    buf.sem_flg = 0;               //表示阻塞形式进行申请

    //调用semop函数完成资源的申请
    if(semop(semid, &buf, 1) == -1)
    {
        perror("P error");
        return -1;
    }

    return 0;
}

//释放资源操作，semno表示要被释放资源的信号量编号
int V(int semid, int semno)
{
    //定义一个结构体变量
    struct sembuf buf;

```

```

buf.sem_num = semno;      //要操作的信号编号
buf.sem_op = 1;           //1表示要释放该信号量的资源
buf.sem_flg = 0;          //表示阻塞形式进行释放

//调用semop函数完成资源的释放
if(semop(semid, &buf, 1) == -1)
{
    perror("V error");
    return -1;
}

return 0;
}

//删除信号量集
int delete_sem(int semid)
{
    //调用semctl函数完成对该信号量集的删除
    if(semctl(semid, 0, IPC_RMID) == -1)
    {
        perror("delete error");
        return -1;
    }

    return 0;
}

```

4> 使用信号量集完成共享内存中两个进程对共享内存使用的同步问题

shmsnd.cpp

```

#include<myhead.h>
#include"sem.h"      //将自定义的头文件加入

#define PAGE_SIZE 4096      //一页的大小

int main(int argc, const char *argv[])
{
    //11、创建并打开信号量集
    int semid = create_sem(2);      //调用自定义函数，完成对信号量集的创建

    //1、创建key值
    key_t key = ftok("/", 'k');
    if(key == -1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key = %#x\n", key);      //输出key值

    //2、通过key值创建共享内存段
    int shmid = -1;
    if((shmid= shmget(key, PAGE_SIZE, IPC_CREAT|0664)) == -1)
    {
        perror("shmget error");
    }
}

```

```

        return -1;
    }
    printf("shmid = %d\n", shmid);

    //3、将共享内存段映射到用户空间
    char *addr = (char *)shmat(shmid, NULL, 0);
    //NULL表示让系统自动寻找对齐页
    //0表示对该共享内存段的操作是读写操作打开
    if(addr == (void*)-1)
    {
        perror("shmat error");
        return -1;
    }
    printf("addr = %p\n", addr);           //输出共享内存段映射的地址

    //4、对共享内存进行操作
    while(1)
    {
        //22、调用自定义函数：申请0号信号量的资源
        P(semid, 0);

        printf("请输入>>>");
        fgets(addr, PAGE_SIZE, stdin);     //从终端输入数据放入共享内存中
        addr[strlen(addr)-1] = 0;

        //33、调用自定义函数：释放1号信号量的资源
        V(semid, 1);

        if(strcmp(addr, "quit") == 0)
        {
            break;
        }
    }

    //5、取消映射
    if(shmdt(addr) == -1)
    {
        perror("取消映射\n");
        return -1;
    }

    //44、调用自定义函数：删除信号量集
    delete_sem(semid);

    return 0;
}

```

shmrcv.cpp

```

#include<myhead.h>
#include"sem.h"

#define PAGE_SIZE 4096           //一页的大小

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

```

```

//11、调用自定义函数：创建并打开信号量集
int semid = create_sem(2);

//1、创建key值
key_t key = ftok("/", 'k');
if(key == -1)
{
    perror("ftok error");
    return -1;
}
printf("key = %#x\n", key);    //输出key值

//2、通过key值创建共享内存段
int shmid = -1;
if((shmid= shmget(key, PAGE_SIZE, IPC_CREAT|0664)) == -1)
{
    perror("shmget error");
    return -1;
}
printf("shmid = %d\n", shmid);

//3、将共享内存段映射到用户空间
char *addr = (char *)shmat(shmid, NULL, 0);
//NULL表示让系统自动寻找对齐页
//0表示对该共享内存段的操作是读写操作打开
if(addr == (void*)-1)
{
    perror("shmat error");
    return -1;
}
printf("addr = %p\n", addr);    //输出共享内存段映射的地址

//4、对共享内存进行操作
while(1)
{
    //22、调用自定义函数完成对1号信号量资源的申请
    P(semid, 1);

    printf("读取到消息为: %s\n", addr);    //通过地址访问共享内存中的数据

    if(strcmp(addr, "quit") == 0)
    {
        break;
    }

    //33、释放0号信号量的资源
    V(semid, 0);
}

//5、取消映射
if(shmdt(addr) == -1)
{
    perror("取消映射\n");
    return -1;
}

//6、删除共享内存段
if(shmctl(shmid, IPC_RMID, NULL) == -1)

```



```

    {
        perror("shmctl error");
        return -1;
    }

    return 0;
}

```

5> 注意:

- 1、信号量集是完成多个进程间同步问题的，一般不进行信息的通信
- 2、信号量集的使用，本质上是对多个value值进行管控，每个信号量控制一个进程，在进程执行前，申请一个信号量的资源，执行后，释放另一个信号量的资源
- 3、如果当前进程申请的信号量值为0，则当前进程在申请处阻塞，直到其他进程将该信号量中的资源增加到大于0

6> 练习：进程1输出字符A，进程2输出字符B，进程3输出啊字符C，使用信号量集完成，最终输出的结果为ABCABCABCABC...

A.cpp

```

#include<myhead.h>
#include"sem.h"          //引入自定义的头文件

int main(int argc, const char *argv[])
{
    //1、创建并打开信号量集
    int semid = create_sem(3);

    while(1)
    {
        //2、申请0号信号量的资源
        P(semid, 0);

        printf("A");
        fflush(stdout);      //刷新标准输出缓冲区

        sleep(1);

        //3、释放1号信号量的资源
        V(semid, 1);
    }

    return 0;
}

```

B.cpp

```

#include<myhead.h>
#include"sem.h"          //引入自定义的头文件

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

```

```

//1、创建并打开信号量集
int semid = create_sem(3);

while(1)
{
    //2、申请0号信号量的资源
    P(semid, 1);

    printf("B");
    fflush(stdout);          //刷新标准输出缓冲区

    sleep(1);

    //3、释放1号信号量的资源
    V(semid, 2);
}

return 0;
}

```

C.cpp

```

#include<myhead.h>
#include"sem.h"          //引入自定义的头文件

int main(int argc, const char *argv[])
{
    //1、创建并打开信号量集
    int semid = create_sem(3);

    while(1)
    {
        //2、申请0号信号量的资源
        P(semid, 2);

        printf("C");
        fflush(stdout);    //刷新标准输出缓冲区

        sleep(1);

        //3、释放1号信号量的资源
        V(semid, 0);
    }

    return 0;
}

```

