

进程

一. 相关概念

1. 进程和程序

进程：正在运行的程序（一个程序的一次执行的过程）；进程是程序的载体

进程和程序的区别：

1. 程序是静态的。包含数据段/正文段。它是保存在磁盘上的指令的有序集合，没有执行的概念。
2. 进程是动态的。是程序执行的过程。包含数据段/正文段/堆栈段
执行过程：创建，调度，消亡；
是一个抽象实体，系统执行程序时，分配和释放的各种资源；
进程是程序执行和资源管理的最小单位。

2. 进程状态

1. 初始态
2. 就绪态

进程已经获得除CPU以外的所有必要资源，只等待CPU时的状态。

系统会将多个处于就绪状态的进程排成一个就绪队列。

3. 执行态

进程已经获取CPU，正在执行。单片机中，处于执行状态的进程只有一个；多处理系统中，有多个处于执行状态的进程。

4. 阻塞态

正在执行的进程由于某种原因而暂时无法继续执行，便放弃处理机而处于暂停状态，即进程执行受阻。（这种状态又称等待状态或封锁状态）

5. 结束态

进程控制块：PCB

`vim /usr/src/linux-headers-3.2.0-29/include/linux/sched.h //1227行`

进程的内存结构：linux 采用虚拟内存管理技术，使得每个进程都有独立的地址空间。

3. 进程的模式（用户&内核）

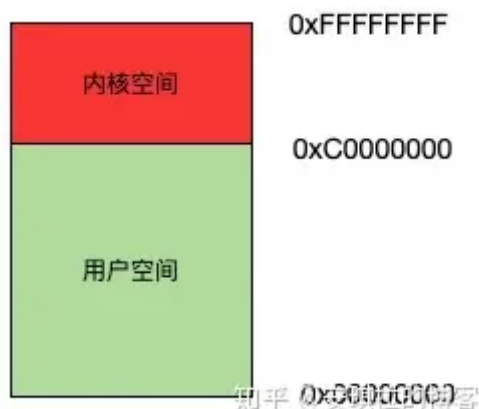
用户模式&内核模式

[深入User space\(用户空间\)与 Kernel space\(内核空间\) - 知乎 \(zhihu.com\)](#)

1) 虚拟内存空间

虚拟地址空间：32位操作系统内存一共只有4G，但操作系统为每一个进程都分配了4G的内存空间，这个内存空间实际是虚拟的，虚拟内存到真实内存有个映射关系。

操作系统将这4G可访问的内存空间分为二部分，一部分是内核空间，一部分是用户空间。



内核空间1G，用户空间3G

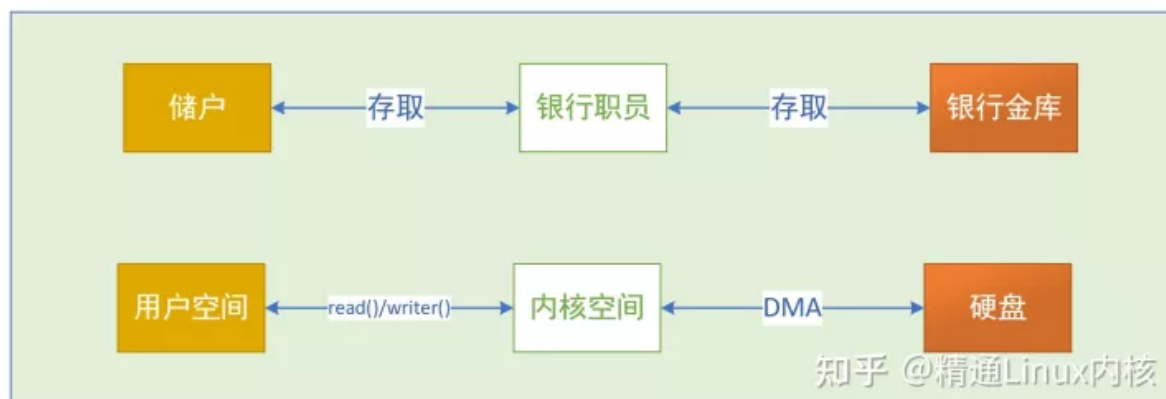
区分内核和用户空间的原因：

用户态的程序不能随意操作内核地址空间，这样对操作系统具有一定的安全保护作用。

如果应用程序能访问任意内存空间，如果程序不稳定常常把系统搞崩溃，比如清除操作系统的内存数据。

后来觉得让应用程序随便访问内存太危险了，就按照CPU 指令的重要程度对指令进行了分级，指令分为四个级别：Ring0~Ring3 (和电影分级有点像)。

linux 只使用了 Ring0 和 Ring3 两个运行级别，进程运行在 Ring3 级别时运行在用户态，指令只访问用户空间，而运行在 Ring0 级别时被称为运行在内核态，可以访问任意内存空间。

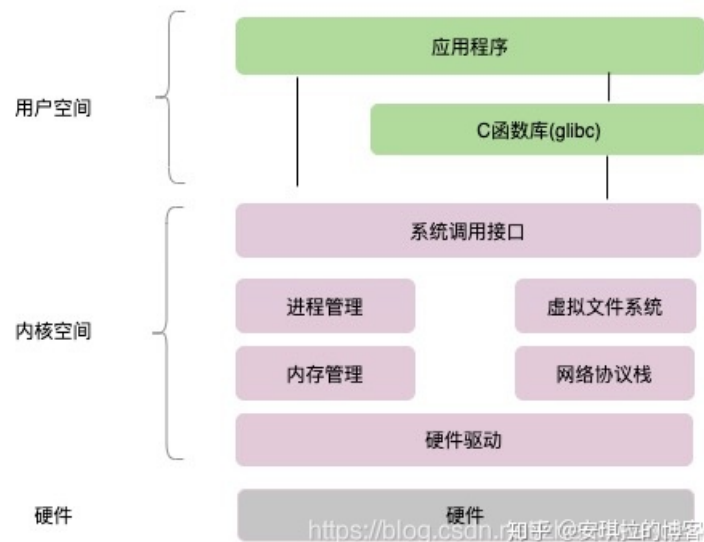


2) 内核态&用户态

当进程/线程运行在内核空间时就处于内核态，而进程/线程运行在用户空间时则处于用户态。

内核态下，CPU可以执行任何指令。运行的代码也不受任何的限制，可以自由地访问任何有效地址，也可以直接进行端口的访问。

在用户态下，进程运行在用户地址空间中，被执行的代码要受到 CPU 的很多检查，比如：进程只能访问映射其地址空间的页表项中规定的在用户态下可访问页面的虚拟地址。



4. 进程的一生

1. 出生fork
2. 任务execl
3. 结束exit()

僵尸进程和孤儿进程

僵尸进程：父进程没消亡 但不为子进程收尸 子进程是僵尸进程 避免僵尸进程

孤儿进程：父进程消亡 子进程还在运行 这时 子进程是孤儿进程 会被init进程领养并收尸

5. 进程的ID (PID)

init进程是进程祖先 PID 是1

ps aux 查看所有用户所有进程详细信息

pstree 查看进程树

top 动态查看进程信息

二. 进程系统调用

1. 进程查看以及创建

1.1 获取进程ID: getpid

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

- `pid_t getpid(void);` // `pid_t` 是 `int` 别名
- 功能: 获取当前调用进程 **PID**
- `pid_t getppid(void);`
- 功能: 获取当前调用进程的父进程的 **PID**

```
int main()
{
    printf("%d, %d\n",getpid(),getppid());
}
```

1.2 创建新进程fork()

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

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

- 功能: 创建新进程
- 返回值: 成功调用返回两次 在父进程中返回子进程 **PID** 在子进程中返回 **0**
- 出错返回 **-1**

1.3 如何区分两个进程

根据fork的返回值

```
int main()
{
    printf("main pid: %d\n",getpid());
    pid_t id = fork();
    if(id == 0)
    {
        printf("我是子进程: %d, 我的父进程是%d\n",getpid(),getppid());
    }
    else if(id > 0)
    {
        sleep(1);    //目的让子进程先走完
        printf("我是父亲:%d\n",getpid());
    }
}
```

sleep(1); //目的让子进程先走完，如果不加sleep，父进程可能先走完，走完后子进程的父进程变成init (pid = 1)。

而且发现 linux@ubuntu: ~\$ 不再出现，可是可以在CLI输入其他任意指令（说明不是死循环）。

原因： linux@ubuntu: ~\$ 是由shell脚本控制的，在一个死循环中，终端控制权是main进程，我们输入Ctrl+c，会将终端控制权回归给shell

```
linux@ubuntu:~/23031/io$ ./a.out
mainPid:10290
我是父亲:10290
linux@ubuntu:~/23031/io$ I 我是宝宝:10291 父亲:1
```

1.4 父子进程的执行顺序

先后顺序不确定，取决于系统调度策略

1.5 总结：

- 1.子进程复制父进程的数据空间 堆栈 代码段(共享正文段)
- 2.复制后 两个进程空间完全独立 子进程变量改变 父进程不会影响

实例：sleep控制进程执行先后顺序

2. 进程消亡

2.1 进程终止方式

有8种方式使进程终止，其中前5种为正常终止，它们是

- 1: 从 main 返回
- 2: 调用 exit
- 3: 调用 _exit 或 _Exit
- 4: 最后一个线程从其启动例程返回
- 5: 最后一个线程调用 pthread_exit

异常终止有3种，它们是

- 6: 调用 abort()
- 7: 接到一个信号并终止
- 8: 最后一个线程对取消请求做出响应

函数名：abort

功 能：异常终止一个进程

用 法：void abort(void);

头文件：#include <stdlib.h>

说明: **abort**函数是一个比较严重的函数, 当调用它时, 会导致程序异常终止,

2.2 exit和_exit和return

两者都是结束进程(整个程序), **_exit**是立即结束(不刷新IO缓存), **exit**会刷新缓存

```
void fun()
{
    printf("hello");    //注意没有\n
    exit(0);
}
int main()
{
    fun();
    puts("end");
}
```

//输出结果hello

```
void fun()
{
    printf("hello");    //注意没有\n
    _exit(0);
}
int main()
{
    fun();
    puts("end");
}
```

//输出结果: 什么都没有, 说明没有刷新缓冲区

```
void fun()
{
    printf("hello\n");    //注意没有\n
    _exit(0);
}
int main()
{
    fun();
    puts("end");
}
```

//输出结果:hello

```

int fun()
{
    printf("hello\n"); //注意没有\n
    return 0;
}
int main()
{
    fun();
    puts("end");
}

//输出结果:hello\n end

```

3. 进程收尸

3.1 阻塞wait

```

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

//收尸+遗产
pid_t wait(int *status); //scanf()

//功能:
等待一个子进程结束 取得子进程结束的状态 将其保存在status中

//返回值: 返回被收尸的子进程PID 如果没有子进程返回-1

//注意:


- a、父进程调用wait时 父进程会阻塞等待子进程结束
- b、如果父进程不关心子进程退出的状态 wait参数可以传NULL 就丢弃结束信息
- c、wait后 子进程占用的资源会被释放



//参数: status 如果为空 就丢弃结束信息
WIFEXITED(status) 判断是否正常退出 真--正常退出 假 是异常退出
WEXITSTATUS(status) 取得进程结束时的返回值
WIFSIGNALED(status) 判断进程是否被信号终止 如果是 则返回真
WTERMSIG(status) 判断是被哪个信号终止的

```

例子1: 读取收尸的子进程id

```

int main()
{
    pid_t id = fork();
    if(0 == id)
    {
        printf("child: %d\n", getpid());
    }
    else if(id > 0)
    {
        puts("father");
    }
}

```

```

        pid_t wpid = wait(NULL);
        printf("wpid is %d\n",wpid);
    }
    else
    {
        perror("fork");
        exit(-1);
    }
}

```

例子2: 读取退出状态

```

int main()
{
    pid_t id = fork();
    if(0 == id)
    {
        printf("child: %d\n",getpid());
        exit(666);
    }
    else if(id > 0)
    {
        puts("father");
        int s;
        pid_t wpid = wait(&s);
        printf("%d\n",WIFEXITED(s));    //正常结束, 返回1
        printf("%d\n",WEXITSTATUS(s)); //666
    }
    else
    {
        perror("fork");
        exit(-1);
    }
}

```

例子3: 判断是被哪个信号终止的

```

int main()
{
    pid_t id = fork();
    if(0 == id)
    {
        printf("child: %d\n",getpid());
        int a = 10, b = 0;
        int c = a/b;
    }
    else if(id > 0)
    {
        puts("father");
        int s;
        pid_t wpid = wait(&s);
        printf("%d\n",WIFEXITED(s));    //正常结束, 返回1
        printf("%d\n",WEXITSTATUS(s)); //1
        printf("%d\n",WTERMSIG(status));    8
    }
}

```



```

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

```

因为除法不能除以0，所以信号8会终止进程

所有的终止信号可以用 `kill -l` 查看

3.2 指定给哪个子进程收尸：waitpid

一个父进程可以有好多个子进程，waitpid可以给指定的子进程收尸

```

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

//功能：为指定pid的进程收尸

//参数
pid:
    pid<-1  等待其进程组组长ID（GID）等于pid的绝对值的任意子进程
    pid== -1  该进程任意子进程

status:
    如果为空 就丢弃结束信息
    WIFEXITED(status) 判断是否正常退出 真--正常退出 假 是异常退出
    WEXITSTATUS(status) 取得进程结束时的返回值
    WIFSIGNALED(status) 判断进程是否被信号终止 如果是 则返回真
    WTERMSIG(status) 判断是被哪个信号终止的

options: 可以决定是否等待子进程结束
    0: 等待子进程结束
    WNOHANG: 不等待

//返回值
>0 被收尸的子进程的PID
0 参数3用WNOHANG 且没有子进程退出
-1 出错

```

例子1: `waitpid(-1,&sta,0)` 等价于 `wait(&sta)`

```

#include "apue.h"

int main()
{
    pid_t id = fork();
    if(0 == id)
    {
        printf("child\n");
    }
}

```

```

    else if(id > 0)
    {
        waitpid(-1, NULL, 0);
        printf("father is %d\n", id);
    }
    else
    {
        perror("fork");
        exit(-1);
    }
    return 0;
}

```

输出:

child

father is 20503

例子2: 不阻塞, 为任意子进程收尸

```

#include "apue.h"

int main()
{
    pid_t id = fork();
    if(0 == id)
    {
        printf("child\n");
    }
    else if(id > 0)
    {
        waitpid(-1, NULL, WNOHANG);
        printf("father is %d\n", id);
    }
    else
    {
        perror("fork");
        exit(-1);
    }
    return 0;
}

```

输出:

father is 20503

child

二. 进程执行任务exec

exec族函数: 在本进程中加载另一个程序, 并且从头开始执行。本进程会完全被新进程替换

注意：在执行完毕后，除了**进程号没变**以外，其他内容都被替换掉了

应用：

1)本进程重生

2)通过fork创建新进程 让新进程执行其他任务—主要应用

1. execl

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

```
int execl(const char *path, const char *arg, ...);
```

格式：execl(可执行文件的路径,可执行文件名称,参数1, 参数2, 参数3, ...,NULL)

注意：可执行文件的路径必须是which后的结果：e.g. which ls

例子1

```
which ls
```

```
int main()
{
    puts("come...");
    execl("/bin/lis", "ls", "-l", NULL);
    puts("go...");
}
```

输出：

come

which ls的结果

不输出go...

例子2：执行已有任务

```
which touch
```

```
//touch.c
#include <unistd.h>

int main()
{
    execl("/usr/bin/touch", "touch", "a.txt", NULL);
}
```

例子3：创建自定义任务

```
//task.c
#include "apue.h"

int main()
{
    printf("pid is %d\n",getpid());
}
```

```
gcc -o task.c task
```

```
//exec11.c
#include "apue.h"

int main()
{
    pid_t id = fork();
    if(id > 0)
    {
        puts("father");
        wait(NULL);
    }
    else if(id == 0)
    {
        puts("child");
        execl("./task","task", NULL);
        puts("hah");
    }
    else
    {
        perror("fork");
        exit(-1);
    }
}
```

```
gcc exec11.c -o exec11
```

输出：

father

child

pid is 20615

不输出hah

2. execvp

```
int execvp(const char *file, char *const argv[]);
```

格式: `execvp`(可执行文件名, 指针数组)

```
int main()
{
    char*cmd[] = {"ls", "-l", NULL};
    execvp("ls", cmd);
}
```

三. 守护进程

1. 相关概念

前台进程: 依附终端, 终端结束, 进程结束

后台进程: 不依附终端, 自己独立存在

进程组: 多个进程

会话: 多个进程组

回顾:

父进程控制着终端, 父进程结束, 终端打印出开头的:

```
linux@ubuntu: ~$
```

2. 创建守护进程

步骤:

1. 创建子进程 父进程结束(init领养子进程)

```
pid_t id = fork();
if(id > 0)
{
    exit(0);
}
```

2. 创建新会话, 让子进程彻底脱离终端

```
setsid();
```

3. 改变当前工作目录

```
chdir("/tmp");
```

等价于`cd /tmp, touch a.log`

4. 修改文件掩码

```
umask(0)
```

5. 关闭所有可能打开的文件描述符

```
int n = getdtablesize();
for(int i = 0; i<n; i++)
{
    close(i);
}
```

例子：获取当前系统时间并写入

```
void myDaemon()
{
    //1. 创建子进程，父进程结束
    pid_t id = fork();
    if(id > 0)
    {
        exit(0);
    }
    //此时只有子进程

    //2. 创建新会话，子进程摆脱终端
    setsid();

    //3. 改变当前工作目录
    chdir("/tmp")

    umask(0);
    int n = getdtablesize();
    for(int i = 0; i<n; i++)
    {
        close(i);
    }
}

int main()
{
    myDaemon();
    FILE *fp = NULL;
    fp = fopen("time.log", "a");
    time_t t;
    if(fp != NULL)
    {
        while(1)
        {
```

```
        t = time(NULL);
        fprintf(fp, "%s", ctime(&t));
        fflush(fp);
        sleep(1);
    }
}
```

```
cat /tmp/time.log
```

要关闭整个进程需要：

```
ps aux
```

查看到./a.out的id后

```
kill id
```

四. 进程间通信

1. 概念

1.1 为什么进程间需要通信

Linux环境下，进程地址空间相互独立，每个进程各自有不同的用户地址空间

所以**进程和进程之间不能相互访问，要交换数据必须通过内核**，在内核中开辟一块缓冲区，进程1把数据从用户空间拷到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信（IPC，InterProcess Communication）。

1.2 传输方式

可以分为单工、半双工和全双工数据传输

单工：单工（Simplex Communication）模式的数据传输是单向的。通信双方中，一方固定为发送端，一方则固定为接收端。

信息只能沿一个方向传输，

半双工：半双工通信使用同一根传输线，既可以发送数据又可以接收数据，但不能同时进行发送和接收。数据传输允许数据在两个方向上传输，

但是，在任何时刻只能由其中的一方发送数据，另一方接收数据。

全双工：全双工数据通信允许数据同时在两个方向上传输，因此，全双工通信是两个单工通信方式的结合，它要求发送设备和接收设备都有独立的接收和发送能力，

1.3 异步同步

数据通信的**同步**：取数据的瞬间需要考虑发送端发送的频度 也就是当时发没发数据 如果没有发数据就取不到数据

数据通信的**异步**：取数据的时候 不用考虑发送端发送数据的频度 时间间隔 自动就接收了

1.4 进程间通信方式

传统的有：管道，信号

还有

IPC 通信：消息队列、共享内存、信号量

BSD（加州大学伯克利分校的伯克利软件发布中心开发）：套接字

2. 传统进程间通信方式

2.1 管道

管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道。

2.1.1 匿名管道

特点：只能用于亲属进程（父子进程）间通信，先pipe，再fork

原理：实为内核使用环形队列机制，借助内核缓冲区(4k)实现

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

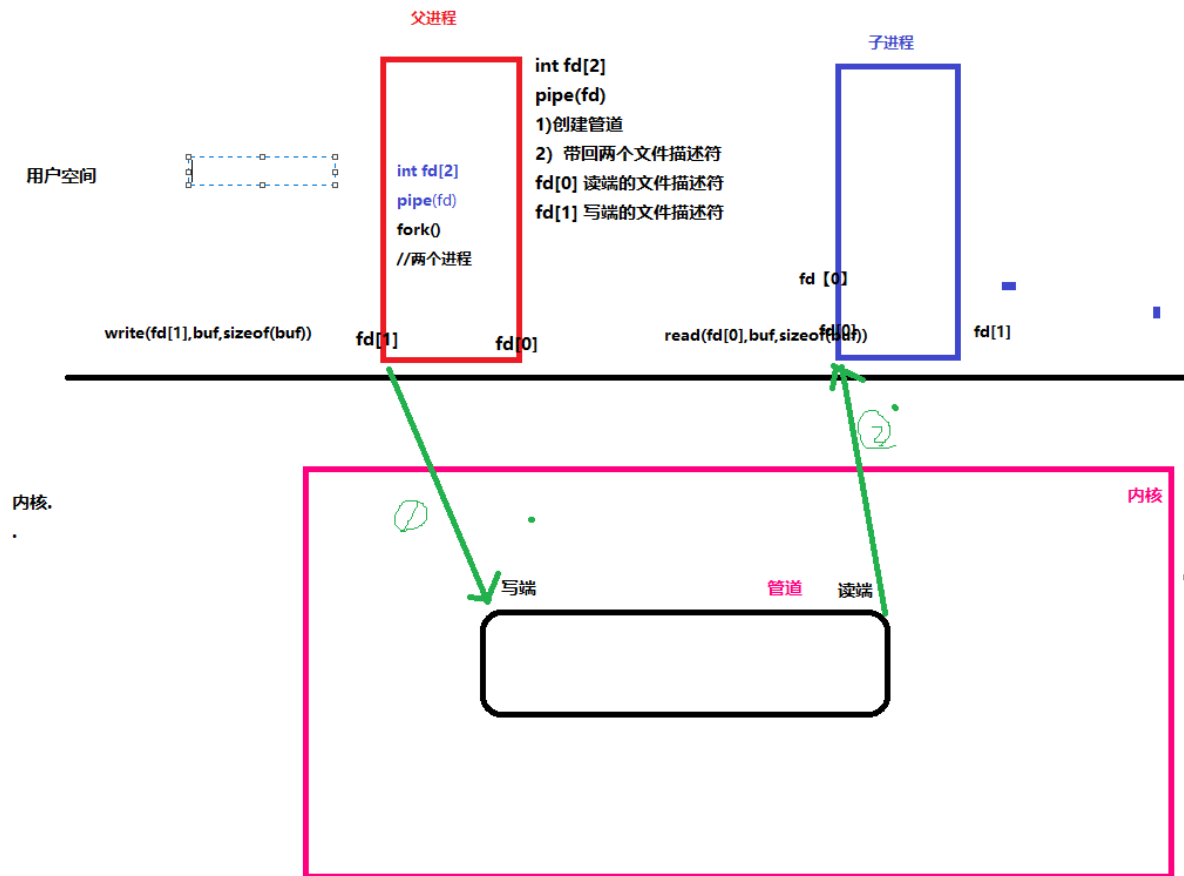
```
int pipe(int pipefd[2]);
```

功能：创建管道

将读端的文件描述符返回到pipefd[0]

将写端文件描述符返回到pipefd[1]

返回值：成功0 失败-1



```
#include "apue.h"
#include <string.h>
int main()
{
    int fd[2];
    int ret = pipe(fd);
    if(ret < 0)
    {
        perror("pipe");
        exit(-1);
    }

    int n = 666;
    write(fd[1], &n, sizeof(n));
    int x;
    read(fd[0], &x, sizeof(x));

    //close(fd[1]);

    puts("again.....");
    ssize_t retnum = read(fd[0], &x, sizeof(x));
    printf("read: %d, retnum is %ld\n", x, retnum);
}
```

匿名管道总结:

(1)关于读:

管道中有数据，数据读完就删除，read返回读到的字节数

管道中无数据，

如果管道写段全部关闭，read返回0

如果没有完全被关闭，read阻塞，一直等待着写

(2)关于写：

如果没有读端（读文件都关闭：管道破裂），write返回-1，设置errno=EPIPE，并产生一个SIGPIPE信号

如果有读端：

管道写满，写会阻塞，(如果读进程不读走管道缓冲区中的数据，那么写操作将一直阻塞)

管道未满，write将数据写入，并返回实际写入的字节数

进程结束后，管道文件不存在

匿名管道优缺点

优点：简单，相比信号，套接字实现的进程间通信要简单

缺点：

只能单向通信，双向需要建立两个管道

只能用于父子，有共同祖先的兄弟进程间通信

2.1.2 有名管道FIFO

FIFO常被称为命名管道 **能用于亲属或非亲属间通信**

mkfifo创建了一个FIFO，就可以使用open打开它，常见的文件I/O函数都可用于fifo。如：close、read、write、unlink等。

mkfifo

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

```
int mkfifo(const char *pathname, mode_t mode);
```

功能：按指定的权限创建管道文件

参数1：要创建的管道

参数2：同open 参数3 指定要创建的管道的操作权限

返回值：成功0 失败-1

unlink

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

- `int unlink(const char *pathname);`
- 功能：删除文件

创建管道文件

```
mkfifo hello  
ls -l hello
```

发现结果中，hello的文件属性是p，说明是管道文件

```
cat hello
```

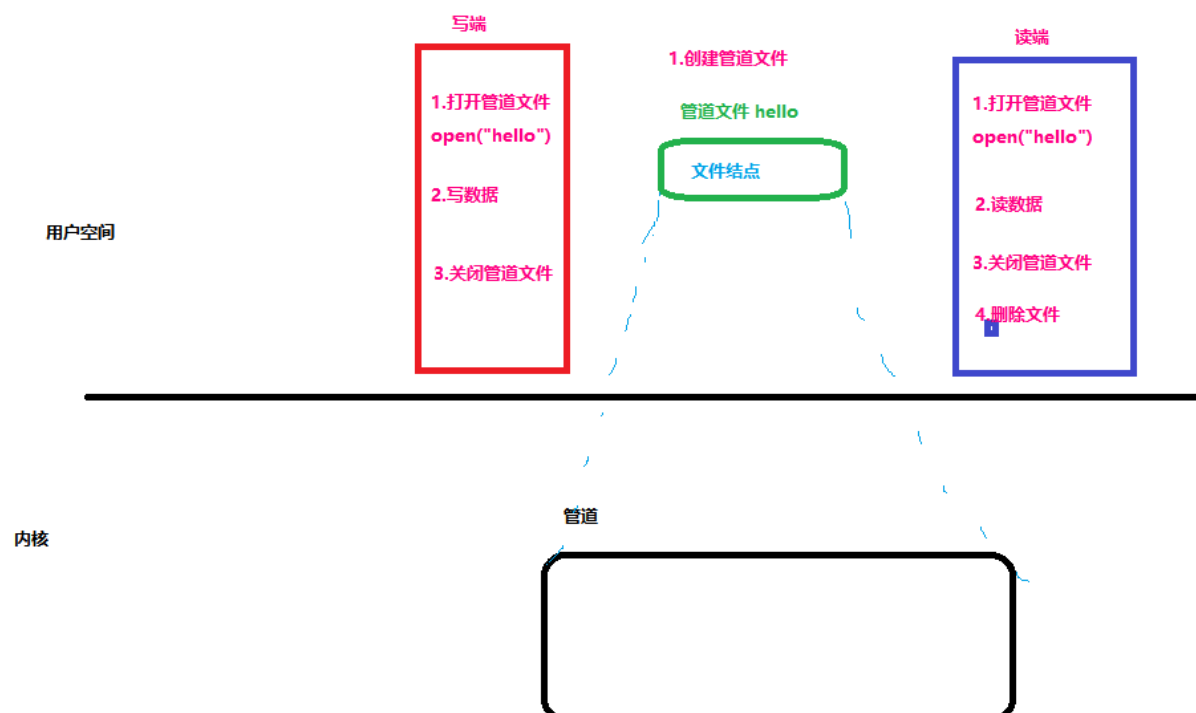
发现阻塞，于是可以新开一个终端，进入当前目录，输入：

```
echo input > hello
```

发现cat hello中显示input，说明实现了非亲属关系进程间的通信

操作步骤：

1. 创建管道fifo
2. 读写端操作
 1. open
 2. write
 3. close
 4. unlink（可以删除所有文件，包括link文件）



注意，我们创建的hello其实只是一个**文件结点**，它是对内核管道的映射

例子：

create.c (创建叫做pipe的管道文件)

```
#include "apue.h"

int main()
{
    int ret = mkfifo("pipe",0666);
    if(ret < 0)
    {
        perror("mkfifo");
        exit(-1);
    }
}
```

write.c

```
#include "apue.h"

int main()
{
    int fd = open("pipe",O_WRONLY);
    if(fd < 0)
    {
        perror("open");
        exit(-1);
    }
    int n = 666;
    write(fd, &n, sizeof(n));
    close(fd);
}
```

read.c

```
#include "apue.h"

int main()
{
    int fd = open("pipe",O_RDONLY);
    if(fd < 0)
    {
        perror("open");
        exit(-1);
    }
    int x;
    read(fd, &x, sizeof(x));
    printf("read: %d\n",x);
}
```

```
close(fd);

unlink("world");
}
```

编译过程:

```
gcc create.c -o create
ls //看到有一个叫做pipe的管道文件生成
gcc read.c -o read
gcc write.c -o write
```

```
./create
./read
```

另开一个terminal

```
./write
```

发现第一个terminal (执行read) 中出现write进来的数字

因为我们write一次后就删除了管道, 所以下一次需要重新运行:

```
./create
./read
```

例子2:

编写程序实现如下功能

read.c 从argv[1]所指定的文件中读取内容, 依次写到当前目录下的管道叫做pipe中

write.c 从当前目录下的管道叫做pipe中读取内容, 写到argv[1]所指定的文件中并保存

```
//create.c
#include "apue.h"

int main()
{
    int ret = mkfifo("pipe", 0666);
    if(ret < 0)
    {
        perror("mkfifo");
        exit(-1);
    }
}
```

```
//read.c
```

```

#include "apue.h"

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

    if(argc != 2)
    {
        printf("%s\n",argv[0]);
        exit(-1);
    }
    //open pipeline
    int fd = open("pipe",O_WRONLY);
    if(fd < 0)
    {
        perror("open");
        exit(-1);
    }
    //read from file
    int file_id = open(argv[1],O_RDONLY);
    if(file_id <0)
    {
        printf("open file");
        exit(-1);
    }

    char buf[50] = "\0";
    while(1)
    {
        ssize_t ret = read(file_id, buf, sizeof(buf));
        if(ret<=0)
        {

            break;
        }
        //write to the pipeline
        write(fd, buf, sizeof(buf));
        bzero(buf,sizeof(buf));
    }
    close(fd);
    close(file_id);
}

```

```

//write.c

#include "apue.h"

int main(int argc, char*argv[])
{
    //open pipeline
    int fd = open("pipe",O_RDONLY);
    if(fd < 0)
    {
        perror("open");
        exit(-1);
    }
}

```

```

//open file
if(argc != 2)
{
    printf("%s\n",argv[0]);
    exit(-1);
}
int file_id = open(argv[1],O_WRONLY);
if(file_id <0)
{
    printf("open file");
    exit(-1);
}

//read from pipeline
char ch[50] = "\0";
while(1)
{
    ssize_t ret = read(fd, ch, sizeof(ch));
    if(ret<=0)
    {
        break;
    }
    //write to the file
    write(file_id, ch, sizeof(ch));
    bzero(ch,sizeof(ch));
}

close(fd);
close(file_id);

//unlink("pipe");
}

```

编译过程:

- 1) 创建file.txt, 用来存储读出的内容
- 2) 编译三个.c文件
- 3) ./c
- 4) ./r apue.h //想把apue.h文件里的东西读出来
- 5) 另开一个终端, 执行./w file.txt //写到file.txt中
- 6) cat file.txt

有名管道总结:

没有读端, 写端也会阻塞

可以用于亲属/非亲属进程

不能定位

2.2 信号

可以传数据（但是只能long int一个一个传，费劲，所以基本不用它传数据）

同步问题：刚才我们写的管道，写端只有写完，读端才能读。信号可以帮助实现这种同步。

2.2.1 什么是信号

是在软件层面上对中断机制的一种模拟 **是一种异步通信方式**。信号可以在**用户空间进程**和**内核**之间直接交互，内核可以利用信号来通知用户空间的进程发生了哪些系统事件

2.2.2 同步和异步

数据通信的同步：取数据的瞬间需要考虑发送端发送的频率。也就是当时发没发数据。如果没有发数据，就取不到数据

数据通信的异步：取数据的时候，不用考虑发送端发送数据的频率。时间间隔，自动就接收了

2.2.3 信号来源

信号事件主要有两个来源：

- 硬件来源：用户按键输入Ctrl+C退出、硬件异常如无效的存储访问等。
- 软件终止：终止进程信号、其他进程调用kill函数、软件异常产生信号。

2.2.4 信号生命周期和处理流程

(1) 信号被某个进程产生，并设置此信号传递的对象（一般为对应进程的pid），然后传递给操作系统；

(2) 操作系统根据接收进程的设置（是否阻塞）而选择性的发送给接收者，如果接收者阻塞该信号（且该信号是可以阻塞的），操作系统将暂时保留该信号，而不传递，直到该进程解除了对此信号的阻塞（如果对应进程已经退出，则丢弃此信号），如果对应进程没有阻塞，操作系统将传递此信号。

(3) 目的进程接收到此信号后，将根据当前进程对此信号设置的预处理方式，暂时终止当前代码的执行，保护上下文（主要包括临时寄存器数据，当前程序位置以及当前CPU的状态）、转而执行中断服务程序，执行完成后在回复到中断的位置。当然，对于抢占式内核，在中断返回时还将引发新的调度。

每个进程收到的所有信号，都是由内核负责发送的，内核处理。

2.2.5 查看信号列表 kill -l

- SIGINT ctrl+c发出的信号
- SIGQUIT ctrl+\ 发出的信号 终止程序
- SIGILL 非法指令—cpu指令集
- SIGABRT 通过函数abort()发出的信号 实现程序终止
- SIGFPE 浮点异常 除数为0
- **SIGKILL** 必杀信号
- SIGPIPE 管道破裂
- SIGSEGV 段错误
- SIGALRM 通过alarm()发出的信号

- SIGTERM 终止信号 kill命令发出的信号 kill pid
- **SIGCHLD** 子进程停止或终止
- SIGCONT 使一个暂停(停止)的进程继续
- SIGSTOP 使进程暂停
- SIGTSTP ctrl+z发出的信号
- SIGUSR1 用户自定义信号
- SIGUSR2 用户自定义信号
- 9) SIGKILL和19) SIGSTOP信号, 不允许忽略和捕捉, 只能执行默认动作

用户自定义信号表示可以自己发送让他有所谓的含义

2.2.6 相关知识点

信号发送: kill, alarm, raise

信号接收: while(1), sleep(100), pause()

信号处理: 忽略, 捕捉, 默认

2.3 信号处理

2.3.1 注册信号signal()

功能: 告诉内核 当信号到来时 如何处理信号 ---注册信号

```
#include<signal.h>
```

signal(信号, 信号处理函数)

返回值: 失败-1

信号处理方式(三种)中:

默认: 传入SIG_DFL

忽略: 传入SIG_IGN

捕捉: 传入一个信号处理函数。处理函数要求: 返回值必须为void 形参必须为int (形参为信号编号)

例子: 忽略 (用户输入Ctrl+C无反应)

```
int main()
{
    signal(SIGINT, SIG_DFL);
    while(1)
    {
        printf("kill me\n");
        sleep(1);
    }
}
```

例子: 默认 (用户输入Ctrl+C, 杀死进程)

```
int main()
{
    signal(SIGINT, SIG_IGN);
    while(1)
    {
        printf("kill me\n");
        sleep(1);
    }
}
```

例子：捕捉

```
#include "apue.h"

void mysignal(int n)
{
    printf("hahahha");
    printf("killer is %d\n", n);
}

int main()
{
    signal(SIGQUIT, mysignal);
    while(1)
    {
        printf("waitiing...\n");
        sleep(1);
    }
}
```

信号处理函数中，形参n是信号列表的第n个。上个例子中，n=2

```
linux@ubuntu:~/23031/io/process$ kill -l
 1) SIGHUP          2) SIGINT          3) SIGQUIT          4) SIGILL
 5) SIGTRAP
 6) SIGABRT         7) SIGBUS          8) SIGFPE           9) SIGKILL  1
10) SIGUSR1
11) SIGSEGV        12) SIGUSR2         13) SIGPIPE         14) SIGALRM  1
15) SIGTERM
16) SIGSTKFLT      17) SIGCHLD         18) SIGCONT         19) SIGSTOP  2
20) SIGTSTP
21) SIGTTIN        22) SIGTTOU         23) SIGURG          24) SIGXCPU  2
25) SIGXFSZ
26) SIGVTALRM      27) SIGPROF         28) SIGWINCH        29) SIGIO     3
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+1
2      47) SIGRTMIN+13
48) SIGRTMIN+14    49) SIGRTMIN+15     50) SIGRTMAX-14     51) SIGRTMAX-1
3      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
```

例子：信号统一处理

```
#include "apue.h"

void mysignal(int n)
{
    switch(n)
    {
        case SIGINT:
            puts("kill me !!");
            break;
        case SIGQUIT:
            puts("kill me ??");
            break;
        case SIGTSTP:
            puts("kill me ~~");
            break;
    }
}

int main()
{
    signal(SIGQUIT, mysignal);
    signal(SIGINT, mysignal);
    signal(SIGTSTP, mysignal);
    while(1)
    {
```

```
        printf("waitting...\n");
        sleep(1);
    }
}
```

这时需要用 `ps aux`, `kill` 对应的进程id 来杀死进程

2.3.2 SIGCHLD信号

产生条件：子进程终止时/子进程收到SIGSTOP信号停止时

用途：借助SIGCHLD回收子进程

子进程结束运行，父进程收到SIGCHLD信号（虽然该信号默认处理动作时忽略），但我们可以捕捉该信号，并在捕捉函数中完成子进程状态的回收

```
void sig_fun(int s)
{
    wait(NULL);
    puts("收尸成功");
}

int main()
{
    pid_t id = fork();
    if(0 == id)
    {
        sleep(1);
        puts("child");
    }
    else if(id > 0)
    {
        signal(SIGCHLD, sig_fun);
        while(1);
    }
    else
    {
        perror("fork");
        exit(-1);
    }
}
```

2.4 信号接收

2.4.1 信号唤醒 pause()

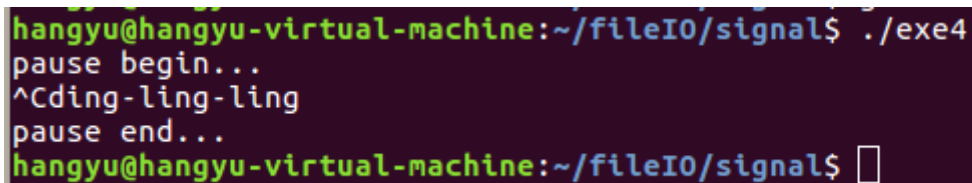
```
#include <unistd.h>
```

```
int pause(void);
```

功能：调用该函数可以造成进程主动挂起，等待信号唤醒（直到被信号中断）

返回值：失败返回-1

```
void sig_fun(int s)
{
    puts("ding-ling-ling");
}
int main()
{
    signal(SIGINT, sig_fun);
    puts("pause begin...");
    pause();
    puts("pause end...");
}
```



```
hangyu@hangyu-virtual-machine:~/fileI0/signal$ ./exe4
pause begin...
^Cding-ling-ling
pause end...
hangyu@hangyu-virtual-machine:~/fileI0/signal$
```

2.5 信号发送

2.5.1 kill()

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

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

功能：向pid进程发送信号sig

sig是信号的编号（kill -l查看）

返回值：成功0 失败-1

例子：实现用kill杀死一个进程：e.g. kill -9 pid ./a.out 9 pid

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

```

    if(argc != 3)
    {
        printf("%s\n",argv[0]);
        exit(-1);
    }
    pid_t pid = atoi(argv[2]);
    int sig = atoi(argv[1]);
    int ret = kill(pid, sig);
    if(ret < 0)
    {
        perror("kill");
        exit(-1);
    }
}

```

2.5.2 延时发送alarm()

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

```
unsigned int alarm(unsigned int seconds);
```

功能：定时 经过指定秒数 系统会自动向本进程发送SIGALRM--延时发送

参数：秒 为0时 取消之前设置的闹钟

返回值：alarm返回值特别 unsigned int类型 返回上次闹钟剩余时间

注意：经过指定秒数 系统会自动向本进程发送SIGALRM信号

每个进程都有且只有唯一一个定时器 如果alarm调用前已设置过闹钟 则任何以前的闹钟时间都被新值代替

```

void fun(int s)
{
    puts("ding-ling-ling");
    printf("%d\n",s);
}

int main()
{
    signal(SIGALRM, fun);
    alarm(5);
    while(1)
    {
        puts("waiting...");
        sleep(1);
    }
}

```

等待5s (waiting输出5次后) 输出ding-ling-ling, 之后一直输出waiting

例子：打字游戏

```
#include "apue.h"

int count = 0;

void fun(int s)
{
    printf("The speed of your typing is: %d\n",count);
    exit(0);
}

int main()
{
    signal(SIGALRM, fun);
    alarm(15);        //15 seconds
    char rand_char;
    srand(time(NULL));
    char input_char;
    while(1)
    {
        rand_char = rand()%26 + 'a';
        printf("%c\n",rand_char);

        scanf("%c",&input_char);
        getchar();
        if(input_char == rand_char)
        {
            count++;
        }
    }
}
```

2.5.3 raise

```
#include <signal.h>

int raise(int sig);
```

功能：发信号给本进程

参数：信号

返回值：成功0 失败-1

```

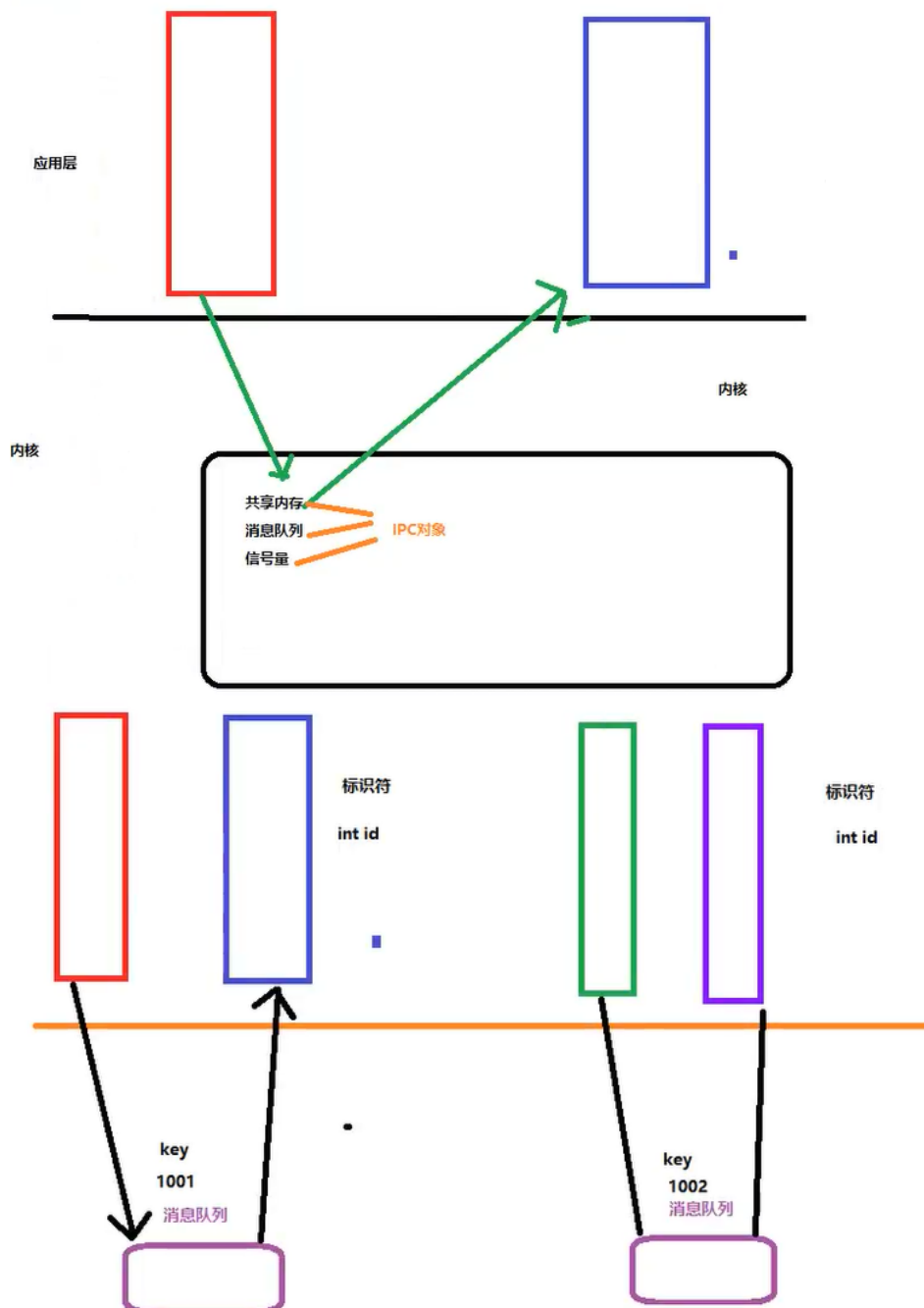
void fun(int s)
{
    printf("sig: %d\n",s);
}
int main()
{
    signal(SIGINT, fun);
    raise(2);
    while(1);
}

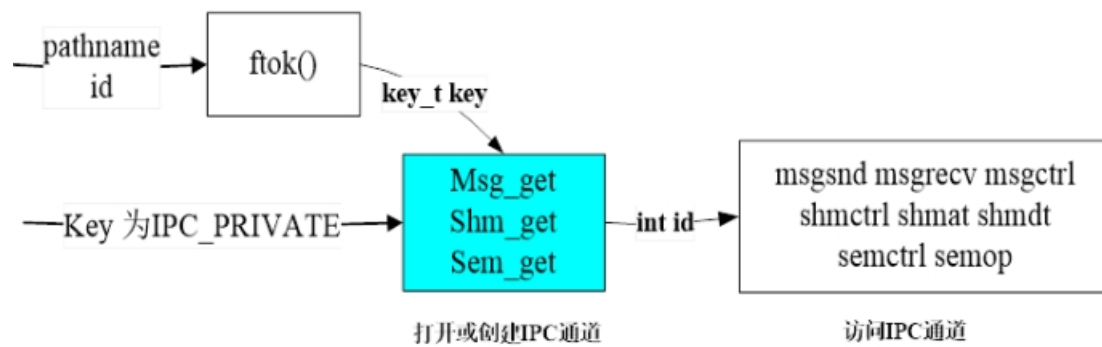
```

3. IPC通信

三种对象（三种进程间通信机制）：

消息队列、共享内存、信号灯集

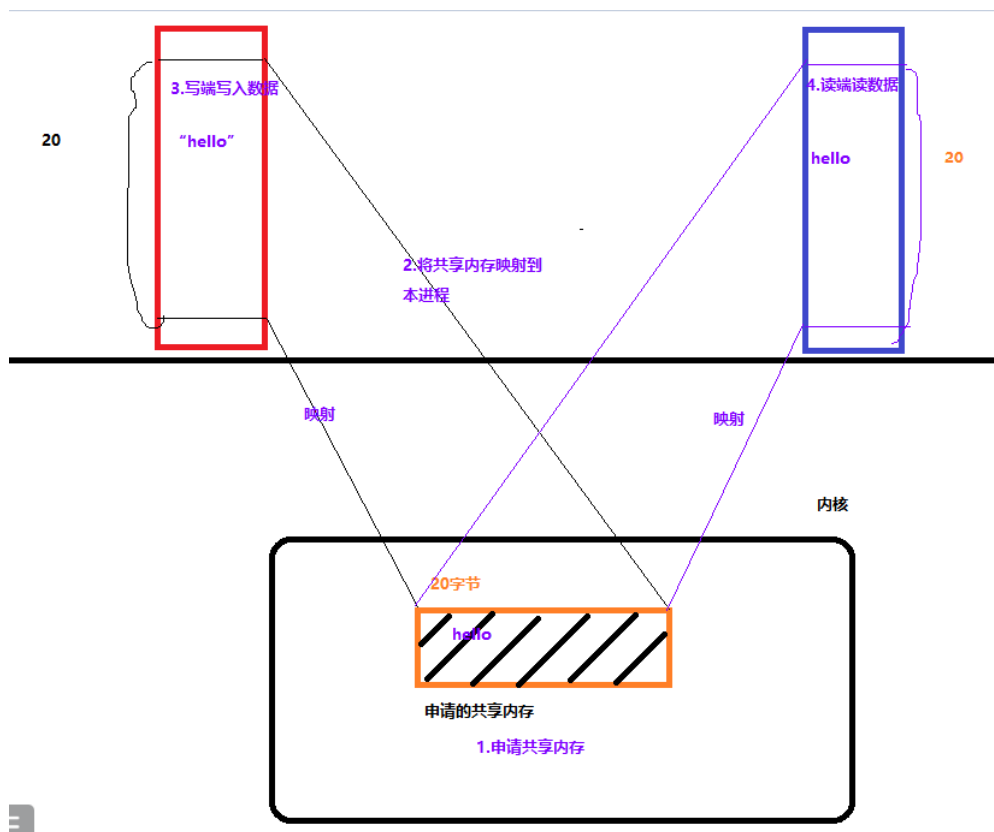




3.1 共享内存

3.1.1 映射

直接向当前进程内存写数据，数据映射到共享内存中，共享内存中的数据也会映射到另一个进程中，从而实现通信



3.1.2 实现共享内存的四步：

- 申请共享内存
- 将共享内存映射到本进程
- 写端写入数据
- 读端读出数据

3.1.3 特点

- (1) 共享内存是进程间最为高效的进程间通信方式
- (2) 共享内存是内核空间区域。
- (3) 如果多个进程使用同一块共享内存区，则需要同步和互斥机制。

3.1.4 操作命令

```
ipcs -m 查看共享内存对象
```

```
//其中nattch表示映射个数
```

```
ipcrm -m shmid 删除对象
```

```
system("ipcs -m");
```

3.1.5 创建/打开共享内存对象shmget()

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

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

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

//参数

key:

如果用于父子进程，表示此共享私有，填入IPC_PRIVATE

其他情况可以放入ftok()函数的返回值，表示指定ID的共享内存

size:

新创建的共享内存的大小，0表示取得已有的共享内存对象

shmflg: 权限（和open函数一样），可以用八进制表示法

* 0表示取得共享内存对象

* key不为IPC_PRIVATE时（IPC_PRIVATE key值相同 这个宏没有意义）

* IPC_CREAT 表示不存在就创建 存在就打开 例如：IPC_CREAT|0777

* IPC_CREAT和IPC_EXCL位时 对象不存在创建 已经存在 就报错

例子：

```
#include "apue.h"

int main()
{
    int id = shmget(IPC_PRIVATE, 10*sizeof(int), IPC_CREAT|0777);
    if(id < 0)
    {
        perror("shmget");
        exit(-1);
    }

    printf("id is: %d\n", id);
}
```

查看所有共享内存对象：

```
id is: 5865489
hangyu@hangyu-virtual-machine:~/IPC/IPC1$ ipcs -m

----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x00000000   65536      hangyu     600        524288     2          dest
0x00000000   917505     hangyu     600        67108864   2          dest
0x00000000   557058     hangyu     600        524288     2          dest
0x00000000   294915     hangyu     600        524288     2          dest
0x00000000   655364     hangyu     600        524288     2          dest
0x00000000   753669     hangyu     600        524288     2          dest
0x00000000   851974     hangyu     600        524288     2          dest
0x00000000   1015815    hangyu     600        524288     2          dest
0x00000000   4587528    hangyu     600        524288     2          dest
0x00000000   1212425    hangyu     600        524288     2          dest
0x00000000   1310730    hangyu     600        524288     2          dest
0x00000000   1409035    hangyu     600        524288     2          dest
0x00000000   1441804    hangyu     600        4194304    2          dest
0x00000000   2457613    hangyu     600        524288     2          dest
0x00000000   2162702    hangyu     600        524288     2          dest
0x00000000   5832719    hangyu     600        2097152    2          dest
0x00000000   2490384    hangyu     777        40         0          dest
0x00000000   5865489    hangyu     777        40         0          dest
```

3.1.6 将共享内存对象映射到用户空间地址中 shmat()

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

功能：将共享内存对象映射到用户空间地址中

参数1：映射的是哪块内存(内存id)

参数2：映射到用户空间的哪块地址；传NULL表示系统自动选地址

参数3：

0 表示共享内存可读写；

SHM_RDONLY 共享内存只读

返回值：成功返回映射后的地址 失败 (void*) -1

注意：使用时要cast成申请空间对应的类型（如果申请整型空间，则cast成int*类型）

```

#include "apue.h"

int main()
{
    int shmid = shmget(IPC_PRIVATE, 10*sizeof(int), IPC_CREAT|0777);
    if(shmid<0)
    {
        perror("shmget");
        exit(-1);
    }
    printf("shmid: %d\n",shmid);

    pid_t id = fork();
    if(id > 0)
    {
        sleep(1);
        int *p = shmat(shmid, NULL, SHM_RDONLY);
        if(p == (void*)-1)
        {
            perror("shmat");
            exit(-1);
        }
        //读
        int i;
        for(i = 0; i<10; i++)
        {
            printf("%d\n",p[i]);
        }
    }
    else if(0 == id)
    {
        int *p = shmat(shmid, NULL, 0);
        if(p == (void*)-1)
        {
            perror("shmat");
            exit(-1);
        }
        //写
        int i;
        for(i = 0; i<10; i++)
        {
            p[i] = i;
        }
    }
    else
    {
        perror("fork");
        exit(-1);
    }
}

```

3.1.7 解除映射 shmdt()

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

功能：解除映射

返回值：成功0 失败-1

读或写完成后解除映射

3.1.8 删除共享内存对象 shmctl()

```
int shmctl(shmid, IPC_RMID, NULL)
```

返回值：成功0 失败-1

读后删除共享内存对象

例子：

```
#include "apue.h"

int main()
{
    int shmid = shmget(IPC_PRIVATE, 10*sizeof(int), IPC_CREAT|0777);
    if(shmid<0)
    {
        perror("shmget");
        exit(-1);
    }
    printf("shmid: %d\n",shmid);

    pid_t id = fork();
    if(id > 0)
    {
        sleep(1);
        int *p = shmat(shmid, NULL, SHM_RDONLY);
        if(p == (void*)-1)
        {
            perror("shmat");
            exit(-1);
        }
        //读
        int i;
        for(i = 0; i<10; i++)
        {
            printf("%d\n",p[i]);
        }
        shmdt(p);
        shmctl(shmid, IPC_RMID, NULL);
    }
}
```

```

else if(0 == id)
{
    int *p = shmat(shmid, NULL, 0);
    if(p == (void*)-1)
    {
        perror("shmat");
        exit(-1);
    }
    //写
    int i;
    for(i = 0; i<10; i++)
    {
        p[i] = i;
    }
    shmdt(p);
}
else
{
    perror("fork");
    exit(-1);
}
}

```

3.1.9 父子间通信流程总结：

- 1) 创建共享内存对象
- 2) fork创建子进程
- 3) 写端：
 - 映射共享内存
 - 写入数据
 - 关闭共享内存
- 4) 读端：
 - 阻塞等待写端写完（sleep）
 - 映射共享内存
 - 读出数据
 - 关闭共享内存
 - 删除共享内存空间

3.1.10 不相关进程使用共享内存 ftok()

创建一个共享内存的key值

根据任意一个已存在的pathname和任意一个整数得到一个key值

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

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

参数1: 文件名包含路径 **注意: 必须已经存在**

参数2: 一个整数

返回值: 成功返回唯一key值 失败-1

例子: 写入一个整型数字并读出

```
//create.c

#include "apue.h"

int main()
{
    key_t key = ftok("/home",6);
    if(key < 0)
    {
        perror("ftok");
        exit(-1);
    }

    int shmid = shmget(key, sizeof(int), IPC_CREAT|0777);
    if(shmid < 0)
    {
        perror("shmget");
        exit(-1);
    }

    printf("shmid: %d\n",shmid);
    return 0;
}
```

```
#include "apue.h"

int main()
{

    key_t key = ftok("/home",6);
```

```

    if(key < 0)
    {
        perror("ftok");
        exit(-1);
    }

    int shmid = shmget(key, 0, 0);
    if(shmid < 0)
    {
        perror("shmget");
        exit(-1);
    }

    int *p = shmat(shmid, NULL, 0);
    if(p == (void*)-1)
    {
        perror("shmat");
        exit(-1);
    }
    *p = 888;
    shmdt(p);
}

```

```

//read.c

#include "apue.h"

int main()
{

    key_t key = ftok("/home",6);
    if(key < 0)
    {
        perror("ftok");
        exit(-1);
    }

    int shmid = shmget(key, 0, 0);
    if(shmid < 0)
    {
        perror("shmget");
        exit(-1);
    }

    int *p = shmat(shmid, NULL, 0);
    if(p == (void*)-1)
    {
        perror("shmat");
        exit(-1);
    }
    printf("%d\n", *p);
    shmdt(p);
    shmctl(shmid, IPC_RMID, NULL);

}

```


流程：

- * 创建三个文件：create.c, read.c, write.c

create.c中：

- * 调用一个共享内存文件，path = /home, proj_id = 6 （都任意）
- * 创建一个共享内存：shmget

write.c中：

- * 调用一个共享内存文件，path = /home, proj_id = 6
- * 创建一个共享内存：shmget
- * 映射：shmat
- * 写入数据
- * 解除映射

read.c 中

- * 调用一个共享内存文件，path = /home, proj_id = 6
- * 创建一个共享内存：shmget
- * 映射：shmat
- * 读出数据
- * 解除映射
- * 删除共享内存

3.1.11 为什么共享内存高效 (与管道对比)

```
1 #include "apue.h"
2
3 void change(char *p)
4 {
5     while(*p)
6     {
7         if(*p>='a' && *p<='z')
8         {
9             *p -= 32;
10        }
11        p++;
12    }
13 }
14
15 int main()
16 {
17     //1.打开管道文件
18     int fd = open("world", O_RDONLY);
19     if(fd < 0)
20     {
21         perror("open");
22         exit(-1);
23     }
24
25     //2.从管道中读取数据
26     char buf[100] = "\0";
27     read(fd, buf, sizeof(buf)); 3.从管道读取数据 存入缓冲区
28
29     //小写转为大写
30     change(buf);
31
32     //3.处理数据
33     printf("read: %s\n", buf); 4.从缓冲区获取数据 写到标准输出文件
34
35     //4.关闭管道文件
36     close(fd);
37
38     //5.删除管道
39     unlink("world");
40
41     return 0;
42 }
```

```
1 #include "apue.h"
2
3 int main()
4 {
5     //1.打开管道文件
6     int fd = open("world", O_WRONLY);
7     if(fd < 0)
8     {
9         perror("open");
10        exit(-1);
11    }
12
13    char buf[100] = "\0";
14    //2.向管道文件中写入数据
15    puts("input str:");
16    gets(buf); 1.从标准输入文件获取数据 存入缓冲区
17    write(fd, buf, strlen(buf)+1); 2.从缓冲区获取数据 写入管道
18
19    //3.关闭管道文件
20    close(fd);
21
22    return 0;
23 }
```

```
1 #include "apue.h"
2
3 int main()
4 {
5     int ret = mkfifo("world", 0666);
6     if(ret < 0)
7     {
8         perror("mkfifo");
9         exit(-1);
10    }
11    return 0;
12 }
```

使用管道(FIFO/消息队列)从一个文件传输信息到另外一个文件需要复制**四次**。

一是，服务器端将信息从相应的文件复制到server临时缓冲区中；

二是，从临时缓冲区中复制到管道（FIFO/消息队列）；

三是，客户端将信息从管道（FIFO/消息队列）复制到client端的缓冲区中；

四是，从client临时缓冲区将信息复制到输出文件中。

共享内存是最快的IPC形式。一旦这样的内存映射到共享它的进程的地址空间，这些进程间数据传递不再涉及到内核，

换句话说就是进程不再通过执行进入内核的系统调用来传递彼此的数据。

共享内存的消息复制只有**两次**。

一是，从输入文件到共享内存；

二是，从共享内存到输出文件。这样就很大程度上提高了数据存取的效率。

它将同一块内存区域映射到共享它的不同进程的地址空间中，使得这些进程间的通信就不需要再经过内核，只需对该共享的内存区域进程操作就可以了，但是它需要用户自己进行同步操作

3.2 消息队列

3.2.1 概念

消息队列是消息的链表，与使用共享内存类似，消息队列是IPC中的一种进程间通信方式

3.2.2 操作命令

查看创建的消息队列

```
ipcs -q
```

删除消息队列的命令：

```
ipcrm -q msqid
```

3.2.3 创建IPC消息队列：msgget()

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

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

功能：创建或打开一个消息队列

参数1: key:

如果用于父子进程，表示此共享私有，填入IPC_PRIVATE
其他情况可以放入ftok()函数的返回值，表示指定ID的共享内存

参数2：消息队列的访问权限及打开方式 例如：IPC_CREAT|O_RDONLY

- IPC_CREAT: 创建新的消息队列。
- IPC_EXCL: 与IPC_CREAT一同使用，表示如果要创建的消息队列已经存在，则返回错误。

返回值：成功返回消息队列标识符 失败-1

注意：创建消息队列 所有者必须有执行权限 否则无法获取，也就是说必须写成 IPC_CREAT|0777

例子：创建消息队列

```
//create.c

#include "apue.h"

int main()
{
    key_t key = ftok('/home/linux',6);
    if(key < 0)
    {
        perror("ftok");
        exit(-1);
    }
    int msgid = msgget(key, IPC_CREAT|0777);
    if(msgid < 0)
    {
        perror("msgget");
        exit(-1);
    }
    printf("msgid: %d\n",msgid);
}
```

3.2.4 将消息写入消息队列 msgsnd()

```
#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：将msgp里消息写入msqid的消息队列

参数1: msgget创建的消息队列的id

参数2:
struct msgbuf {
    long mtype;    /* message type, must be > 0 */消息类型 必须>0
    char mtext[1]; /* message data */消息正文
};
```

注意:

- (1) 结构体中, 类型必须放数据上边
- (2) 消息类型是用户自定义的, 可以是任意long型
- (3) 消息正文是用户自定义的, 可以是任意格式(int, char...)

参数3: 消息正文大小 注意!!!!!!: 不包括消息类型的大小

必须写成: `sizeof(msgbuf) - sizeof(mtype)`, 考虑到结构体内存对齐的问题

参数4: 如果是0, 表示消息队列满时, `msgsnd`会阻塞; 如果`IPC_NOWAIT`, 不阻塞

//返回值: 成功0 失败-1

```
//write.c

#include "apue.h"

typedef struct
{
    long type;
    int data;          //只想写入一个整型
}msg_t;

int main()
{
    key_t key = ftok('/home/linux',6);
    if(key < 0)
    {
        perror("ftok");
        exit(-1);
    }

    //打开消息队列
    int msgid = msgget(key, O_WRONLY);
    if(msgid < 0)
    {
        perror("msgget");
        exit(-1);
    }

    msg_t message;
    puts("Please enter the type and the data of the message");
    scanf("%ld %d",&message.type, &message.data);

    int ret = msgsnd(msgid, &message, sizeof(message)-sizeof(message.type),0);
    //0用来阻塞
    if(ret < 0)
    {
        perror("msgsnd");
        exit(-1);
    }
}
```

3.2.5 按类型接收消息: msgrcv

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

功能: 按类型接收消息

参数1: msgget的返回值

参数2: 存放消息缓存

参数3: 收到的消息正文的大小 注意: 不包括类型大小

参数4: 按msgtyp的类型接收消息 如果传0 表示接收任意类型消息

参数5: 如果是0 表示没有指定类型的消息 msgrcv会阻塞 如果设置成IPC_NOWAIT 表示不阻塞

返回值: 成功返回收到消息正文的字节数 失败-1

3.2.6 删除消息队列: msgctl

删除消息队列对象

格式: msgctl(msgqid, IPC_RMID, NULL)

```
//read.c

#include "apue.h"

typedef struct
{
    long type;
    int data;          //只想写入一个整型
}msg_t;

int main()
{
    key_t key = ftok('/home/linux',6);
    if(key < 0)
    {
        perror("ftok");
        exit(-1);
    }

    //打开消息队列
    int msgid = msgget(key, O_RDONLY);
    if(msgid < 0)
    {
        perror("msgget");
        exit(-1);
    }
}
```

```

    }

    msg_t message;

    ssize_t ret = msgrcv(msgid, &message, sizeof(message)-
sizeof(message.type),message.type,0);
    //0用来阻塞
    if(ret < 0)
    {
        perror("msgsnd");
        exit(-1);
    }
    printf("read: %d\n",message.data);

    msgctl(msgid, IPC_RMID, NULL)

}

```

3.2.7 消息队列trick

1) 同类型先进先出

```

./w
1 111
./w
1 222

```

此时

```

./r
1

```

会输出：111

(i.e. 一次只能输出一条，且是先进的)

2) 接收任意类型的数据

接收方接收的类型是0，表明接收任意类型的一条消息(先进先出顺序)

```

./w
1 111
./w
1 222

```

此时

```

./r
0

```

会输出：111

3.3 信号量

3.3.1 信号灯

协调进程同步的手段

信号灯值不能是负数!!! 只能进行+1或-1操作

3.3.2 进程同步

这是进程间的一种**运行关系**。“同”是协同，按照一定的顺序协同进行（**有序进行**），而不是同时。

即**一组**进程为了协调其推进速度，在某些地方需要相互等待或者唤醒，这种***进程间的相互制约***就被称作是进程同步。

这种合作现象在操作系统和并发式编程中属于经常性事件。具有**同步关系的一组并发进程**称为合作进程

3.3.3 信号量

它是不同进程间或一个给定进程内部不同线程间同步的机制

信号量的工作原理

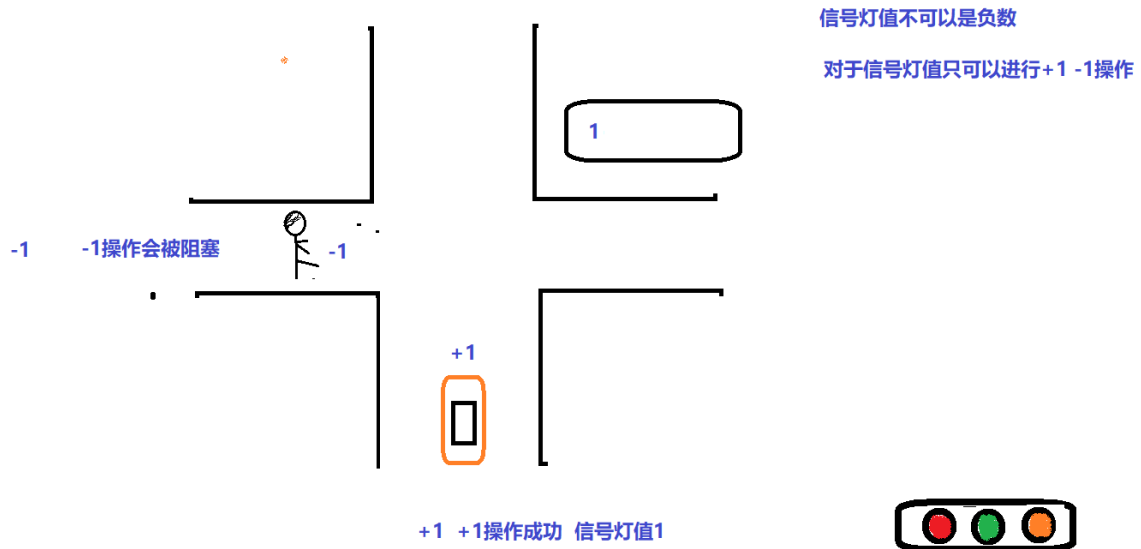
由于信号量只能进行两种操作等待和发送信号，即P(sv)和V(sv),他们的行为是这样的：

P(sv)：如果sv的值大于零，就给它减1；如果它的值为零，就挂起该进程的执行

V(sv)：如果有其他进程因等待sv而被挂起，就让它恢复运行，如果没有进程因等待sv而被挂起，就给它加1。

原子操作：不能进行分割的操作。一次性完成的操作。

PV都是原子操作：执行期间不允许有中断的发生。



3.3.4 查看命令

```
ipcs -s 查看信号量对象
ipcrm -s semid 删除信号量对象
```

3.3.5 创建信号灯集: semget()

什么是信号灯集：多个信号灯的集合（一个数组），适用于更复杂的“路口”

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

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

功能：创建或取得信号量集对象

参数1: key:

如果用于父子进程，表示此共享私有，填入IPC_PRIVATE

其他情况可以放入ftok()函数的返回值，表示指定ID的共享内存

参数2: 信号量的个数（一般是1）

参数3: 可以是0 也可以是IPC_CREAT|0777

返回值：成功返回信号量集对象 失败-1

3.3.6 对信号量集进行指定操作: semop()

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

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

功能：对信号量集semid进行sops指定的操作

参数2:

```
struct sembuf{
    unsigned short sem_num; /* semaphore number */信号量的编号(下标) 一般是0 表示
    第一个信号量
    short sem_op; /* semaphore operation */信号量操作 +1 1 P操作 -1 V操作
    short sem_flg; /* operation flags */SEM_UNDO
    //进程没有释放信号量而退出时 系统自动释放该进程中未释放的信号量
}
```

参数3: 操作的信号灯的个数 一般是1

3.3.7 设置信号量的值/删除信号量集: semctl()

设置信号量的值:

```
semctl(semid,0,SETVAL,3)//设置semid集中 编号为0的信号量值为3
```

删除信号量集:

```
semctl(semid,0,IPC_RMID,NULL);//删除信号量集semid中 编号为0的信号量
```

3.3.8 例子

父进程运行, 堵塞后子进程再运行, 5s后打印child, 之后父进程打印father

```
#include "apue.h"

int main()
{
    int semid = semget(IPC_PRIVATE, 1, IPC_CREAT|0777);
    if(semid < 0)
    {
        perror("semget");
    }
    printf("%d\n",semid);

    pid_t id;
    id = fork();
    if(id > 0)
    {
        printf("This is father\n");
        struct sembuf s;
        s.sem_num = 0;
        s.sem_op = -1;
        s.sem_flg = SEM_UNDO;

        semop(semid, &s, 1);
        puts("father...");
    }
    else if(id == 0)
    {
        sleep(5);
        puts("child...");
        struct sembuf s;
        s.sem_num = 0;
        s.sem_op = 1;
        s.sem_flg = SEM_UNDO;

        semop(semid, &s, 1);
    }
    else
    {
        perror("fork");
        exit(-1);
    }
}
```

输出:

32769

This is father

child...

father...

3.4 进程间通信方式总结

1. 管道

分为有名管道和无名管道

半双工 数据单向流动 缓冲区有限

无名管道: 亲属进程 优点:简单

有名管道:可以用于任意关系的两个进程

2. 信号量

信号灯 本质是计数器 用来控制多个进程或线程对共享资源的访问

优点: 可以同步进程; 缺点: 信号量有限

3. 信号

用于通知接收的进程哪个时间发生

4. 消息队列

是消息的链表 存在内核中 由消息队列标识符标识

优点:可以实现任意进程间通信 不需要考虑同步 消息有类型

缺点:信息的复制 需要额外消耗CPU时间 不适用于消息量大或操作频繁的场所

5. 共享内存

进程间通信最快、效率最高

优点:无需复制 快捷 适用信息量大

缺点:

进程间的读写操作的同步问题

只能同一个计算机系统内的诸多进程共享, 不方便网络通信 (单机)

6. 套接字