进程间通信

管道

子进程结束以后，发送给父进程：SIGCHLD

进程间通信：管道 信号量 共享内存 消息队列 套接字socket

进程与进程之间会被隔离，我们可以通过某种方式打破这种隔离，来进行进程之间的通信

如果不提上面的这种进程之间通讯机制，我们可以通过这种方式实现这种机制：

比如两个进程a和b，我们通过a读取hello，然后将其传递给b进程，然后由b进行输出，怎么实现呢？

我们可以将hello写到文件里面，然后b进程进行对该文件的读取，然后将其内容输出

但是这种方式很慢，效率低，所以我们一般采用上面几种常见的几种进程间通讯方式

将数据读取到内存，然后另一个命令进行对内存的读取

管道分类

有名管道 mkfifo创建，用在任意两个进程之间

无名管道 主要用在父子进程之间

利用管道实现两个进程之间的通讯

有名管道

```
mkfifo fifo
ll //会出现一个管道文件fifo
//mkfifo既是命令又是系统调用
//上面就是用命令创建了一个管道
```

利用代码来进行创建管道，将数据写入到管道里面

a.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include<fcntl.h>
int main(){
    //管道只能够以只读或者只写方式打开
    int fd = open("fifo",O_WRONLY);
    assert( fd != -1);

    char buff[128] = {0};
    printf("input:\n");
    fgets(buff,128,stdin);

    //数据写到管道里面
    write(fd,buff,strlen(buff));
```

```
    close(fd);

    exit(0);

}
```

将数据从管道里面读出来：

b.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include<fcntl.h>
int main(){
    //管道只能够以只读或者只写方式打开
    int fd = open("fifo",O_RDONLY);
    assert( fd != -1);
    char buff[128] = {0};
    int n = read(fd,buff,128);
    //这个时候管道没有读取到数据
    //就会在此阻塞住
    printf("read:%s\n",buff);


    close(fd);

    exit(0);

}
```

查询管道的大小一直是0，因为管道中的数据存储在内存之中，并不是存储在磁盘当中，一旦关机或者结束管道中的数据都被消除了

对上面两个代码进行编译，先执行b，因为管道中未存数据，所以其会在open的地方阻塞住，那么可以通过下面的效果来进行观察是否阻塞，可以在open之后对fd 的值进行输出

管道想要打开，需要两个进程，一个读一个写才能成功打开管道，否则就会阻塞住

所以对于有名管道来说，open有同步，需要：一读一写才能打开

如果管道为空，那么读操作会被阻塞

管道的写端关闭，读端会返回0 即read的返回值为0

写端

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include<fcntl.h>
int main(){
    //管道只能够以只读或者只写方式打开
    int fd = open("fifo",O_WRONLY);
    assert( fd != -1);
    printf("fd=%d\n",fd);
    while(1){
        char buff[128] = {0};
        printf("input:\n");
        fgets(buff,128,stdin);
        if( strncmp(buff,"end",3) == 0 ){
            break;
        }
        //数据写到管道里面
        write(fd,buff,strlen(buff));
    }
    close(fd);

    exit(0);

}
```

读端

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include<fcntl.h>
int main(){
    //管道只能够以只读或者只写方式打开
    int fd = open("fifo",O_RDONLY);
    assert( fd != -1);
    while(1){
        char buff[128] = {0};
        int n = read(fd,buff,128);
        //这个时候管道没有读取到数据
        //就会在此阻塞住
        if( n == 0 ){
            break;
        }
        printf("read(%d):%s\n",n,buff);
    }
    close(fd);

    exit(0);

}
```

无名管道

需要通过pipe(int file_description[2])创建无名管道

其中第一个参数是fdr，文件读描述符

第二个参数是fdw，文件写描述符

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include<fcntl.h>
int main(){
    int fd[2];
    assert( pipe(fd) != -1 );
    //上面如果通过说明管道被创建

    //fd[0] r , fd[1] w
    //r端只能进行读，w同理
    write(fd[1],"hello",5);

    char buff[128]={0};
    read(fd[0],buff,127);
    printf("buff=%s\n",buff);
    close(fd[0]);
    close(fd[1]);

    exit(0);

}
```

所以无名管道是应用于父子进程之间

父子进程可以进行读和写，但是某一时刻只能够有一个来进行读或者写，所以是一种半双工

所以我们可以通过关闭父进程中的读、删除子进程中的写，或者反面来实现父进程写数据，子进程读数据的效果

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include<fcntl.h>
int main(){
    int fd[2];
    assert( pipe(fd) != -1 );
    //上面如果通过说明管道被创建

    //fd[0] r , fd[1] w
    //r端只能进行读，w同理
    pid_t pid = fork();
    assert( pid != -1 );
    if( pid == 0 ){
```

```
        close(fd[1]);
        while(1){
            char buff[128] = {0};
            int n = read(fd[0],buff,127);
            if( n == 0 ){
                break;
            }
        }
        close(fd[0]);
        exit(0);
    }else{
        close(fd[0]);
        while(1){
            char buff[128] = {0};
            fgets(buff,128,stdin);
            if( strncmp(buff,"end",3) == 0 ){
                break;
            }
            write(fd[1],buff,strlen(buff));
        }

        close(fd[1]);
        exit(0);
    }


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

    exit(0);

}
```

注意要先创建管道后fork()

管道的面试问题

1. 有名和无名的区别?
    1. 有名是在任意两个进程间通信，无名只能在父子进程之间通信
2. 写入管道的数据在内存中存放，有名管道文件的大小一直为0，mkfifo -- > fifo ，关机之后fifo打开还存在
3. 半双工的方式
4. 实现

管道是如何实现的

两个指针，一个头一个尾，头指针指向待写入的位置，尾指针指向待读取的位置

当写超过其管道区域的时候，头指针指向最开始的位置，继续向后面执行，直到遇到尾指针停下来，此时写满

空：read阻塞

满：write阻塞

SIGPIPE 13 信号

管道写端关闭，读端的read返回值为0

管道读端关闭，写端会出现异常，会接收到读端传来的信号量 SIGPIPE (即13)，会将挂不到关闭

管道文件描述符：读是3 写是4 因为012都被别的占用了

作业： 临界资源 临界区 原子操作 信号量 pv操作和作用为哪些?

信号量

信号量 ---- > 同步进程

没有同步

有同步

同步就是对临界资源有一定的控制，使得在同一时刻对于某个区域只能够有一个进程能够进行访问，这就叫同步，实现进程同步就需要用到信号量

1. 临界资源：同一时刻只允许一个进程访问的资源叫做临界资源
2. 临界区：访问临界资源的代码段
3. 原子操作：原子操作是不可分割的最小操作

信号量是特殊的变量，对于值的改变是原子操作

p减1，获取资源 阻塞

v加1，释放资源

信号量函数的定义如下所示：

senctl

semget

semop

a.c

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

```
#include<assert.h>

int main(){
    int i = 0;
    for(;i<5;i++){
        write(1,"A",1);
        int n = rand()%3;
        sleep(n);
        write(1,"A",1);
        int n = rand()%3;
        sleep(n);
    }

    exit(0);


}
```

b.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>

int main(){
    int i = 0;
    for(;i<5;i++){
        write(1,"B",1);
        int n = rand()%3;
        sleep(n);
        write(1,"B",1);
        int n = rand()%3;
        sleep(n);
    }

    exit(0);


}
```

将两个进程后台执行

会发现a执行的时候，b也访问stdout了，所以使用信号量来控制这种情况

sem.h

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include<sys/sem.h>


union semun
```

```
{
    int val;
};

void sem_init();

void sem_p();

void sem_v();

void sem_destroy();
```

sem.c

```
#include "sem.h"

static int semid = -1;


//创建信号量
void sem_init(){
    //key值、创建个数、
    semid = semget((key_t)1234,1,IPC_CREAT|IPC_EXCL|0600);
    if(semid == -1){
        semid = semget((key_t)1234,1,0600);
        if(semid == -1){
            perror("semget error");
            return;
        }
    }
    else
    {
        union semun a;
        a.val = 1;
        if(semctl(semid,0,SETVAL,a) == -1)
        {
            perror("semctl error");
        }
    }

    return;
}

void sem_p()
{
    struct sembuf buf;
    buf.sem_num = 0;
    buf.sem_op = -1;
    buf.sem_flg = SEM_UNDO;

    if(semop(semid,&buf,1) == -1 )
    {
        perror("semop p error");
    }
}
```

```
void sem_v(){
    struct sembuf buf;
    buf.sem_num = 0;
    buf.sem_op = 1;
    buf.sem_flg = SEM_UNDO;

    if(semop(semid,&buf,1) == -1 )
    {
        perror("semop v error");
    }
}

void sem_destroy(){
    if( semctl(semid,0,IPC_RMID) == -1 ){
        perror("semctl error");
    }
}
```

a.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include "sem.h"

int main(){
    sem_init();
    int i = 0;
    for(;i<5;i++){
        sem_p();
        write(1,"A",1);
        int n = rand()%3;
        sleep(n);
        write(1,"A",1);
        sem_v();
        in = rand()%3;
        sleep(n);
    }
    sleep(10);
    sem_destroy();
    exit(0);


}
```

b.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#include "sem.h"
```

```
int main(){
    sem_init();
    int i = 0;
    for(;i<5;i++){
        sem_p();
        write(1,"B",1);
        int n = rand()%3;
        sleep(n);
        write(1,"B",1);
        sem_v();
        n = rand()%3;
        sleep(n);
    }

    exit(0);


}
```

共享内存

查看进程之间一些信息的命令

```
ips -s //信号量
ips -m //共享内存
ips -q //消息队列
ips //不加表示默认三个都显示
```

删除进程之间一些关系的命令

```
ipcrm -s 信号量的id //删除信号量
ipcrm -m            //删除共享内存
ipcrm -q            //删除消息队列
```

代码示例

共享内存的创建映射和断开

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#incude<sys/shm.h>
//int shmget(key_t key, size_t size, int shmflg);
//key表示设置的标识位，可以自己给数值，用于给共享内存空间起个id
//size表示共享内存的大小
//shmflg表示标志位，表示在创建的同时需要做哪些事情
//包括IPC_CREAT、IPC_EXCL等
int main(){
    int shmid = shmget((key_t)1234,256,IPC_CREAT|0600);
```

```
    //如果别人创建好了就直接获取该块的共享内存
    assert( shmid != -1 );


    exit(0);
}
```

编译通过，运行程序

调用`ipcs -m`可以看出多了一个共享内存id为1234的共享内存

但是`nnattch`标志为0，因为这块区域还没有被使用

原因是这块空间还没有被映射到某一块区域，可以使用`shmat`函数来进行映射

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
//shmid表示要被映射的共享内存id
//shmaddr表示要映射到这个空间的哪个位置
//表示是指定对这个空间可读还是可写
//SHM_RND
```

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#incude<sys/shm.h>
//int shmget(key_t key, size_t size, int shmflg);
//key表示设置的标识位，可以自己给数值，用于给共享内存空间起个id
//size表示共享内存的大小
//shmflg表示标志位，表示在创建的同时需要做哪些事情
//包括IPC_CREAT、IPC_EXCL等
int main(){
    int shmid = shmget((key_t)1234,256,IPC_CREAT|0600);
    //如果别人创建好了就直接获取该块的共享内存
    assert( shmid != -1 );
    char * s = (char*)shmat(shimid,NULL,0);
    //这里面的0表示既可读也可写
    assert( s != NULL );


    shmdt(s);
    //断开映射


    exit(0);
}
```

共享内存在多个进程的使用

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#incude<sys/shm.h>
```

```
//int shmget(key_t key, size_t size, int shmflg);
//key表示设置的标识位，可以自己给数值，用于给共享内存空间起个id
//size表示共享内存的大小
//shmflg表示标志位，表示在创建的同时需要做哪些事情
//包括IPC_CREAT、IPC_EXCL等
int main(){
    int shmid = shmget((key_t)1234,256,IPC_CREAT|0600);
    //如果别人创建好了就直接获取该块的共享内存
    assert( shmid != -1 );
    char * s = (char*)shmat(shimid,NULL,0);
    //这里面的0表示既可读也可写
    assert( s != NULL );
    strcpy(s,"Hello");


    shmdt(s);
    //断开映射


    exit(0);
}
```

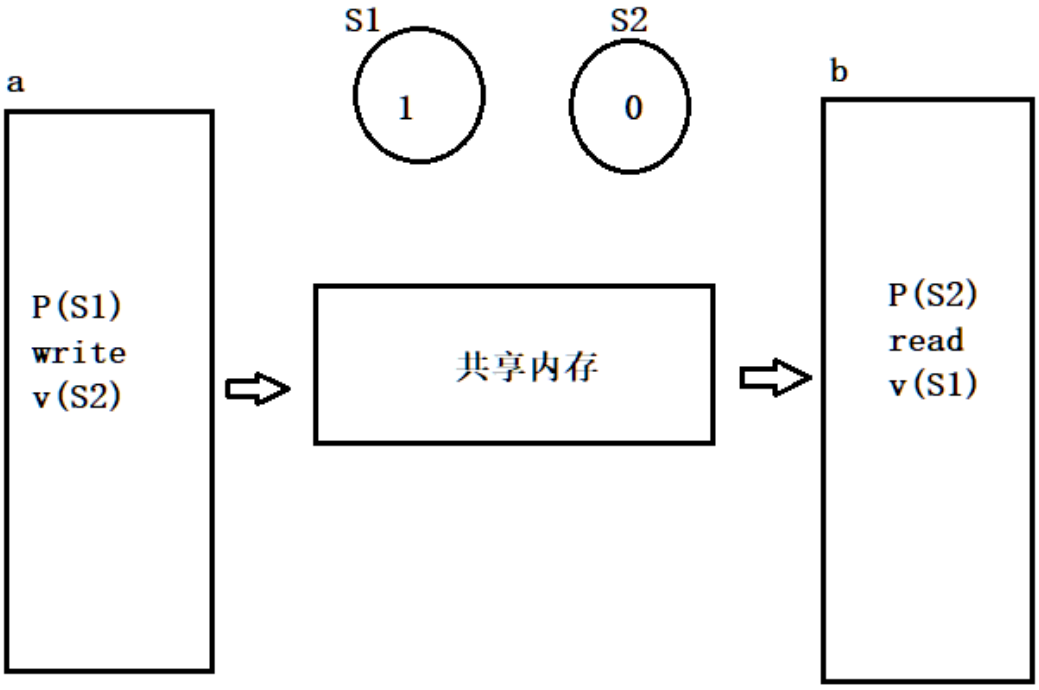
```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#incude<sys/shm.h>
//int shmget(key_t key, size_t size, int shmflg);
//key表示设置的标识位，可以自己给数值，用于给共享内存空间起个id
//size表示共享内存的大小
//shmflg表示标志位，表示在创建的同时需要做哪些事情
//包括IPC_CREAT、IPC_EXCL等
int main(){
    int shmid = shmget((key_t)1234,256,IPC_CREAT|0600);
    //如果别人创建好了就直接获取该块的共享内存
    assert( shmid != -1 );
    char * s = (char*)shmat(shimid,NULL,0);
    //这里面的0表示既可读也可写
    assert( s != NULL );
    printf("%s\n",s);

    shmdt(s);
    //断开映射


    exit(0);
}
```

先运行上面的程序，再运行下面的程序，然后输出为`Hello`

⚓其中一个程序，从键盘上接收输入的字符串，存放在共享内存空间，另外一个程序将该共享内存空间中的值打印出来

需要其中一个进程输入另一个进程才输出，没输入就不输出

设置两个信号量`S1`，`S2`，如下图所示：



sem.h

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#incude<sys/shm.h>
#include<sys/sem.h>
union semun{
    int val;
};

int sem_init();


int sem_p(int index);//第几个信号量

int sem_v(int index);

void sem_destroy();
```

sem.c

```
#include "sem.h"
#define SEM_NUM  2

static int semid = -1;


int sem_init(){
    semid = semget((key_t)1234,SEM_NUM,IPC_CREAT|IPC_EXCL|0600);
```

```
    if( semid == -1 ){
        semid = semget((key_t)1234,SEM_NUM,0600);
        if(semid == -1){
            perror("semget error");
            return -1;
        }
    }
    else{
        int a[SEM_NUM]={1,0};
        union semun semval;
        int i = 0;
        for(;i<SEM_NUM;i++){
            semval.val = a[i];
            if( semctl(semid,i,SETVAL,semval) == -1 ){
                perror("semctl setval error");
                return -1;
            }
        }
    }
    return 0;
}


int sem_p(int index)//第几个信号量
{
    if( index < 0 || index >= SEM_NUM )
    {
        printf("sem_p : index arg error\n");
        return -1;
    }
    struct sembuf buf;
    buf.sem_num = index; //第几个信号量
    buf.sem_op = -1; //操作
    buf.sem_flg = SEM_UNDO;

    if( semop(semid, &buf, 1 ) == -1 ){
        perror("semop p error");
        return -1;
    }

    return 0;
}
int sem_v(int index){
    if( index < 0 || index >= SEM_NUM )
    {
        printf("sem_v : index arg error\n");
        return -1;
    }
    struct sembuf buf;
    buf.sem_num = index; //第几个信号量
    buf.sem_op = 1; //操作
    buf.sem_flg = SEM_UNDO;

    if( semop(semid, &buf, 1 ) == -1 ){
        perror("semop v error");
        return -1;
    }
```

```
    return 0;
}

void sem_destroy(){
    if( semctl(semid,0,IPC_RMID) == -1 ){
        //0表示占位置的作用
        perror("semctl rm error");
    }
}
```

a.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#incude<sys/shm.h>
#include "sem.h"
int main(){
    int shmid = shmget((key_t)1234,256,IPC_CREAT|0600);
    //如果别人创建好了就直接获取该块的共享内存
    assert( shmid != -1 );
    char * s = (char*)shmat(shimid,NULL,0);
    //这里面的0表示既可读也可写
    assert( s != NULL );
    assert( sem_init() != -1 );
    while(1){
        printf("input:\n");
        char buff[128] = {0};
        fgets(buff,128,stdin);

        sem_p(0);
        strcpy(s,buff);
        sem_v(1);

        if( strncmp(buff,"end",3) == 0 ){
            break;
        }
        //由此可以看出是b先退出

    }
    shmdt(s);
    //断开映射


    exit(0);
}
```

b.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
```

```c
#incude<sys/shm.h>
#include "sem.h"
int main(){
    int shmid = shmget((key_t)1234,256,IPC_CREAT|0600);
    //如果别人创建好了就直接获取该块的共享内存
    assert( shmid != -1 );
    char * s = (char*)shmat(shimid,NULL,0);
    //这里面的0表示既可读也可写
    assert( s != NULL );

    assert( sem_init() != -1 );


    while(1){
        sem_p(1);
        if( strncmp(s,"end",3) == 0 ){
            break;
        }
        printf("s=%s\n",s);
        sem_v(0);
    }


    shmdt(s);
    //断开映射
    sem_destroy();
    exit(0);
```

```
gcc -o a a.c sem.c
gcc -o b b.c sem.c
```

先运行b程序，再运行a程序，然后再在a中输入数据，如果这个时候b程序没有输出的话，说明存在的共享内存有一个重名的共享内存，所以ipcrm -q id删除该共享内存，再次执行前面的运行操作

消息队列

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#incude<sys/msg.h>

struct mess
{
    long int type;
    char buff[32];
}

int main(){
    int msgid = msgget((key_t)1234,IPC_CREAT|0600);
    assert( msgid != -1 );

    struct mess dt;
    dt.type = 1;
```

```
    strcpy(dt.buff,"hello1");

    msgsnd(msgid,&dt,32,0);
    //往消息队列中写入一个数据

    exit(0);

}
```

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<assert.h>
#incude<sys/msg.h>

struct mess
{
    long int type;
    char buff[32];
}

int main(){
    int msgid = msgget((key_t)1234,IPC_CREAT|0600);
    assert( msgid != -1 );

    struct mess dt;
    msgrcv(msgid,&dt,32,1,0);
    //msgrcv(msgid,&dt,32,1,0);
    //这里面的参数中的第四个如果为0 则表示不管是什么类型都可以接受
    //消息队列id 消息结构体 大小 类型 标志
    //读取消息队列中的消息
    printf("read msg:%s\n",dt.buff);

    exit(0);

}
```

如果对消息队列中的消息类型获取的时候设置错误，则在消息队列中如果没有该类型就读取不出相应的数据