Linux进程间通信
wuzhh

进程通信概述

进程间通信：在用户空间实现进程通信是不可能的，通过**Linux内核**实现（两个进程操作同一个内核空间的“对象”，“对象”对应通信方式）
线程间通信：可以在用户空间内实现，可以通过全局变量通信

通信方式

单机模式下的进程通信（在同一个Linux内核空间下的进程通信）

管道通信：**无名**管道、**有名**管道（文件系统中有名），（对像为队列）

信号管道：信号（通知）通信包括：信号的**发送**、信号的**接收**和信号的**处理**（对象为信号）

IPC(Inter-Process Communication)通信：**共享内存**、**消息队列**和**信号灯**（基于文件IO的思想open、close、read、write）
　　在IPC的通信模式下，不管是使用消息队列还是共享内存，甚至是信号量，每个IPC的对象(object)都有唯一的名字，称为“键”(key)。通过“键”，进程能够识别所用的对象。“键”与IPC对象的关系就如同文件名称之于文件，通过文件名，进程能够读写文件内的数据，甚至多个进程能够共用一个文件。而在IPC的通讯模式下，通过“键”的使用也使得一个IPC对象能为多个进程所共用

Socket通信：存在于网络中两个进程之间的通信（两个Linux内核）

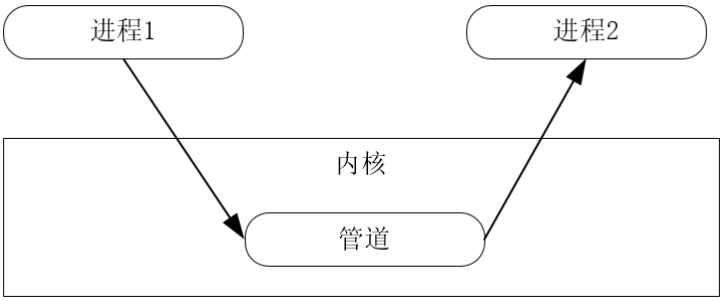
每一种通信方式都是基于文件IO的思想

一、管道

1.1无名管道

文件系统中无文件节点/文件名

通信过程：



管道文件是一个特殊的文件，是由**队列来实现**的
在文件IO中创建一个文件或打开一个文件是由open函数来实现的，他不能创建管道文件，只能用**pipe**函数来创建管道

函数形式：`int pipe(int fd[2])` 功能，创建管道，为系统调用（在`unistd.h`文件中声明）
　参数：传入得到的文件描述符，包含两个文件描述符，`fd[0](读端) fd[1](写端)`， 返回值：成功返回0，出错返回-1

管道特性

- 管道是创建在内存中，进程结束，空间释放，管道就不存在了；
- 管道中的内容，读完后自动删除，类似队列
- 如果管道中无内容读，则会造成读阻塞

无名管道缺点：不能实现不是父子进程（亲缘关系）之间的通信 -> **衍生出有名管道**

```c
//example pipe function 管道创建
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int fd[2];
    int ret;
    ret = pipe(fd);
    if(ret < 0)
    {
        printf("creat pipe failure\n");
        return -1;
    }
    printf("creat pipe sucess fd[0] = %d, fd[1] = %d\n", fd[0], fd[1]);
    return 0;
}
/* 一个进程打开时，内核会自动打开三个文件描述符：0 1 2，所以以上程序输出的fd[0] = 3；fd[1] = 4
*/

//example pipe function 单进程管道读写
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h" //调用函数memset函数
int main()
{
    int fd[2];
    int ret;
    char writebuf[] = "hello linux";    //写缓存
    char readbuf[128] = {0};            //读缓存
    ret = pipe(fd);
    if(ret < 0)
    {
        printf("creat pipe failure\n");
        return -1;
    }
    printf("creat pipe sucess fd[0] = %d, fd[1] = %d\n", fd[0], fd[1]);
    //写入管道
    write(fd[1], writebuf, sizeof(writebuf));//写哪里去 写什么 写多少个
    //管道读出
    read(fd[0], readbuf, 128);//从什么地方读，读到哪里去，读多少个
    printf("readbuf = %s\n", readbuf);

    /*验证管道读阻塞
    memset(readbuf, 0, 128);        //使用该函数需要包含头文件#include "string.h"
    read(fd[0], readbuf, 128);      //即清除第一次读到的内容，进行第二次读
    printf("second read after\n");  //测试是否能打印该语句 不能打印

```

```
    */

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

```

写阻塞

```
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h" //调用函数memset函数
int main()
{
    int fd[2];
    int ret;
    int i = 0;
    char writebuf[] = "hello linux";    //写缓存
    char readbuf[128] = {0};            //读缓存
    ret = pipe(fd);
    if(ret < 0)
    {
        printf("creat pipe failure\n");
        return -1;
    }
    printf("creat pipe sucess fd[0] = %d, fd[1] = %d\n", fd[0], fd[1]);
    //写入管道 写5500次 测试是否写满管道
    while(i < 5500)
    {
        write(fd[1], writebuf, sizeof(writebuf));//写哪里去 写什么 写多少个
        i++;
    }
    printf("write pipe end\n"); //写满无法打印该语句

    close(fd[0]);
    close(fd[1]);
    return 0;C
}
```

```
//无名管道实现父子进程之间通信 父进程先运行，打印5行 停留5s 子进程运行打印5行
//子进程将父进程所有内容都进行了拷贝，包括管道的文件描述符都进行了拷贝，所有可以在内核空间中的同一个管道进行通信
#include "unistd.h"
#include "stdio.h"
#include "sys/types.h"
#include "stdlib.h"
int main()
{
    pid_t pid;
    int fd[2];
    int ret;
    char process_inter = 0;
    ret = pipe(fd);     //需要先建立管道再创建进程 保证父子进程使用同一管道
    if(ret < 0)
```

```
    {
        printf("creat pipe failure\n");
        return -1;
    }
    printf("creat pipe sucess\n");

    pid = fork();
    if(pid == 0)    //子进程
    {
        int i = 0;
        read(fd[0], &process_inter, 1);//子进程读 从哪读 读到哪里去 读多少个 如果管道为空 则阻塞
        while(process_inter == 0);
        for(i = 0; i < 5; i++)
        {
            printf("this is child process i = %d\n", i);
            usleep(100);
        }
    }
    if(pid > 0)         //父进程
    {
        int i = 0;
        for(i = 0; i < 5; i++)
        {
            printf("this is parent process i = %d\n", i);
            usleep(100);
        }
        process_inter = 1;  //process_inter置一
        sleep(5);           //睡眠5s
        wirte(fd[1], &process_inter, 1);  //往管道写（写端fd[1]），写什么内容，写多少个
    }
    while(1);
    return 0;
}
```

内核代码的跟踪

内核实现的过程：

1. 初始化管道文件：`init_pipe_fs`
2. 注册和加载文件系统：`pipefs_mount pipe_fcntl`
3. 建立`write，read`管道，返回是文件及文件描述符：`create_read_pipe create_write_pipe`
4. 写管道和读管道：`pipe_write_open pipe_read_open pipe_write pipe_read`
5. close，关闭

> 实际上类似文件系统的加载实现过程，像文件一样，包括操作文件描述符一样操作管道，管道实际上是内存，将内存映射到文件系统上，在虚拟的`pipe_fs`中对管道进行处理

1.2有名管道

文件系统中有文件节点/文件名

所谓有名，即文件系统中存在一个文件节点，每个文件节点都有一个inode号，且为特殊的文件类型：p管道类型文件

1. 创建这个文件节点，不可通过open函数，open函数只能创建普通文件，不能创建特殊文件（管道-`mkdifo`， 套接字-`socket`，字符设备文件 - `mknod`，块设备文件 - `mknod`，符号链接文件 -

`ln -S`，目录文件`mkdit`）

2. 管道文件只有inode号，不占磁盘空间，和套接字，字符设备文件、块设备文件一样。普通文件和符号链接文件及目录文件，不仅有inode号，还占用磁盘块空间

实现原理概述：实现一个有名管道实际上就是实现一个FIFO文件，有名管道建立后，它的读写以及关闭操作都与普通管道相同，有名管道的文件inode节点在**磁盘**上，其余文件数据存在于内存缓冲页面中和普通管道相同

`mkfifo`

`mkfifo`用来创建管道文件的节点，没有在内核中创建管道，只有通过`open`函数打开这个文件时才会在内核空间创建管道

函数形式：`int mkfifo(const char *filename, mode_t mode);` 功能：创建管道文件
参数`*filename`：管道文件文件名； 参数`mode`：权限，创建的文件权限仍然和`umask`有关； 返回值：创建成功返回0，失败返回-1

```c
//创建管道
#include "stdio.h"
#include "unistd.h"
#include "stdlib.h"
int main()
{
    int ret;
    ret = mkfifo("./myfifo", 0777);
    if(ret < 0)
    {
        printf("creat myfifo failure\n");
        return -1;
    }
    printf("creat myfifo sucess\n");
    return 0;
}
/*********************************************************************************************************/
//mkfifo 用法，通过管道实现无亲缘关系进程间通信 ////注意管道文件，进程文件要在同一个目录下////
//程序实现：存在两个文件 first.c 和 second.c 文件内容都是打印语句，现通过有名管道通信实现first.c先打印second.c后打印

/* first.c*/
#include "unistd.h"
#include "stdio.h"
#include "sys/type.h"
#include "stdlib.h"
#incldue "fcntl.h"
int main()
{
    int fd;
    int i;
    char processs_inter = 0;    //通信标志变量
    fd = open("./myfifo", O_WRONLY);//打开管道文件，打开方式为写操作，打开管道文件时，内核空间自动生成一个管道
    if(fd < 0)
    {
        printf("open myfifo failure\n");
    }
```

```
    printf("open myfifo sucess");
    for(i = 0; i < 5; i++)
    {
        printf("this is first process i = %d\n", i);
        usleep(100);
    }
    process_inter = 1;
    sleep(5);   //睡眠5s,
    write(fd, &process_inter, 1);
    while(1);
    return 0;
}

/* second.c */
#include "unistd.h"
#include "stdio.h"
#include "sys/type.h"
#include "stdlib.h"
#include "fcntl.h"
int main()
{
    int fd;
    int i;
    char processs_inter = 0;    //通信标志变量
    fd = open("./myfifo", O_RDONLY);//打开管道文件，打开方式为写操作
    if(fd < 0)
    {
        printf("open myfifo failure\n");
    }
    printf("open myfifo sucess");
    read(fd, &process_inter, 1);        //先读通信标志位
    while(process_inter == 0);          //如果是0，说明进程first 还未运行，如果为1，说明first进程运行完
    for(i = 0; i < 5; i++)
    {
        printf("this is second process i = %d\n", i);
        usleep(100);
    }
    while(1);
    return 0;
}
/* 命令一：gcc -o first first.c
 * 命令二：gcc -o second second.c
 * 打开第二个终端：./first      //运行第一个进程 这是管道生成有写端 但读端还不存在
 * 运行第二个进程：./sencond    //运行第二个进程，管道读端链接 管道打开成功，第一个进程先运行5s后第二个进程后运行
*/
```

二、信号通信

2.1概述

软中断信号，signal，又称信号，用来通知进程发生了**异步**事件，

进程之间可以互相通过系统调用**kill**发送软中断信号，内核也可以因为内部事件给进程发送信号，**通知**进程发生了某个事件

注意：信号只是用来通知某进程发生了某事件，并不给该进程传递任何数据

在内核中，存在一个通信对象-->信号，用户空间中存在需要相互通信的进程，进程需要先向内核请求，然后内核进行发信号到要通信的进程，信号与管道不一样，**内核中本身就存在信号**，且内核中存在多种信号(linux命令：`kill -l`查看信号种类，一共有64种)

- 告诉内核发什么信号（信号值）
- 告诉内核发给谁（pid号）

kill 用于向任何进程组或进程发送信号

信息	标注
所需头文件	`#include <signal.h> #include <sys/types.h>`
函数原型	`int kill(pid_t pid, int sig)`
函数传入值	pid：正数，要接受信号的进程的进程号；0，信号被发送到所有和pid进程在哦同一个进程组的进程； -1，信号发送给所有进程表中的进程（除了进程号最大的进程外） sig：信号
函数返回值	成功：0；出错：-1

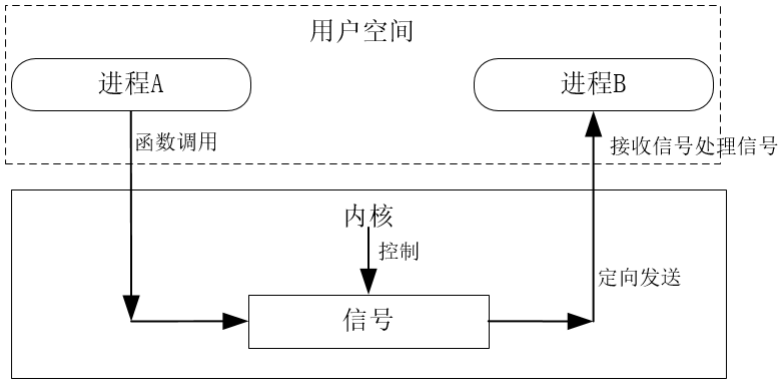
```c
//kill 命令的实现
#include "sys/type.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main(int argc, char *argv[])
{
    int sig;    //信号值
    int pid;    //进程pid号
    if(argc < 3)
    {
        printf("please input param\n");
        return -1;
    }
    sig = atoi(argv[1]);    //获取输入的信号值
    pid = atoi(argc[2]);    //获取输入的进程pid
    printf("sig = %d, pid = %d", sig, pid);
    kill(pid, sig);         //调用kill函数杀死指定进程
    return 0;
}
/* 查看当前系统进程命令：ps -axj*/
```

2.2信号通信通信框架

2.2.1信号发送

信号通信，内核向用户空间进程发送信号，只有内核才能发送信号，用户空间进程不能发送信号

信号通信的框架



- 信号的发送：发送信号进程，kill(向指定进程发送信号)、raise、alarm
- 信号的接收：接收信号进程，pause()、sleep、while(1)
- 信号的处理：接收信号进程，signal

raise：功能：只能发信号给自己（告诉内核发什么信号；） == `kill(getpid(), sig)`

所需头文件	#include <signal.h> #include <sys/type.h>
函数原型	int raise(int sig);
函数传入值	sig：信号
函数返回值	成功0，出错-1

```c
//raise 示例 进程无输出，直接被杀死
#include "sys/type.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    printf("raise before"); //语句不会打印，printf是库函数由于存在库缓存，内容先写到库缓存，行缓存函数，且无\n，则无打印
    raise(9);       //杀死进程自己
    printf("raise after\n");
    return 0;
}
//进程状态变化示例1
#include "sys/type.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    pid_t pid;
    pid = fork();
    if(pid > 0)     //父进程
    {
        sleep(5);
```

```
        while(1);
    }
    if(pid == 0)    //子进程
    {
        printf("raise function before\n");
        raise(SIGTSTP);     //发送SIGSTP信号 进程状态变为暂停 T
        printf("raise function after\n");
        exit(0);
    }
    return 0
}
//进程状态变化示例2 父进程：S -> R 子进程：T -> Z
//初始时父进程为睡眠状态S，子进程为暂停状态T，当调用kill函数后，父进程进入运行状态R，杀死进程，子进程退出，但是父进程没有回收子进程资源（父进程此刻在死循环状态while(1)），子进程僵死（Z）
#include "sys/type.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    pid_t pid;
    pid = fork();
    if(pid > 0)     //父进程
    {
        sleep(5);
        if(waitpid(pid, NULL, WNOHANG) == 0)//等待子进程退出 设置为非阻塞 当子进程不退出时返回值为0
        {
            kill(pid, 9);   //杀死进程
        }
        //wait(NULL); //调用该函数先回收资源，再进入死循环，子进程不会僵死
        while(1);   //死循环
    }
    if(pid == 0)    //子进程
    {
        printf("raise function before\n");
        raise(SIGTSTP);     //发送SIGSTP信号 进程状态变为暂停 T
        printf("raise function after\n");
        exit(0);
    }
    return 0
}
/* SIGSTP：该信号用于暂停交互进程，用户可键入SUSP字符（通常是Ctrl-Z）发出这个信号 暂停进程
 *
*/
```

alarm

alarm函数定时一段时间再发送信号，alarm只能让内核向自己（当前进程）发送信号

alarm只会发送SIGALARM信号

alarm会让内核定时一段时间之后发送信号，raise会让内核立即发信号

所需头文件	#include <unistd.h>
函数原型	unsigned int alarm(unsigned int seconds);
函数传入值	seconds：指定秒数
函数返回值	成功：如果调用此alarm()前，进程已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回0，出错-1

```c
//示例 当输出i = 8时，当前进程终止
#include "sys/type.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int i;
    printf("alarm before\n");
    alarm(9);   //定时9s,
    printf("alarm after\n");
    //保证当前进程在收到alarm信号前还未终止
    while(i < 20)
    {
        i++;
        sleep(1);   //每次循环停1s
        printf("process things, i = %d\n", i);
    }
    return 0
}
```

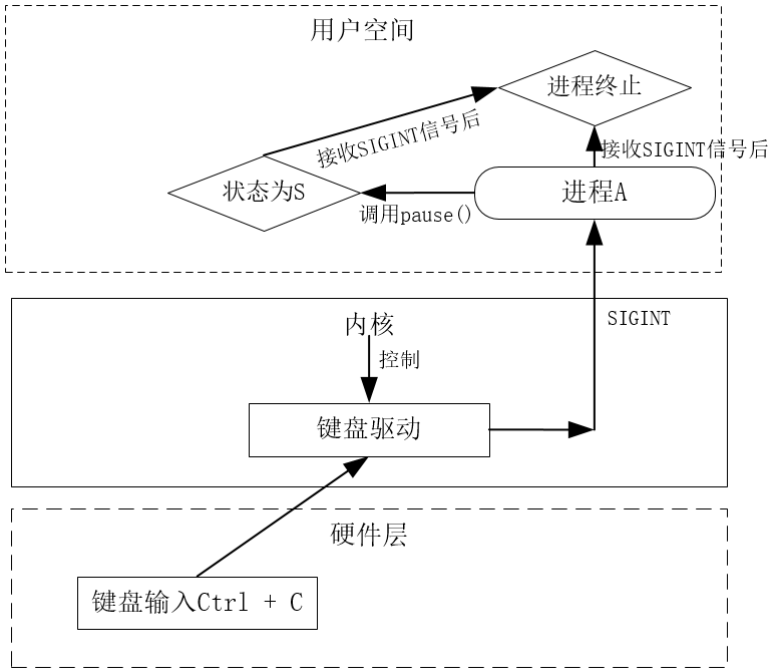
2.2.2信号接收

接收信号的进程，要有睡眠条件：要想使接收的进程能收到信号，这个进程不能先结束
常用手段：sleep（S）、pause（将进程状态保持为S，睡眠状态）、while(1)

pause：所需头文件`#include <unistd.h>` 函数原型：`int pause(void)` 函数返回值：成功0，出错-1

```c
//
#include "sys/type.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int i;
    i = 0;
    printf("pause befor\n");
    pause();    //进程状态变为睡眠状态 当键盘输入Ctrl + C后进程终止 以下代码不会执行
    printf("pause after\n");
    while(i < 20)
    {
        i++;
        sleep(1);
        printf("process things, i = %d\n", i);
    }
```

```
    return 0;
}
/* SIGINT：该信号在用户键入INTR字符（通常是Ctrl + C）时发出，终端驱动程序发送此信号并送到前台进程中的每一个进程
 *
*/
```



2.2.3信号处理

默认信号处理方式：`SIGALRM(终止)、SIGINT(终止)、SIGKILL(终止)、SIGSTOP(暂停)`

自己处理信号的方法告诉内核，进程收到这个信号就会采用自己定义的处理方式

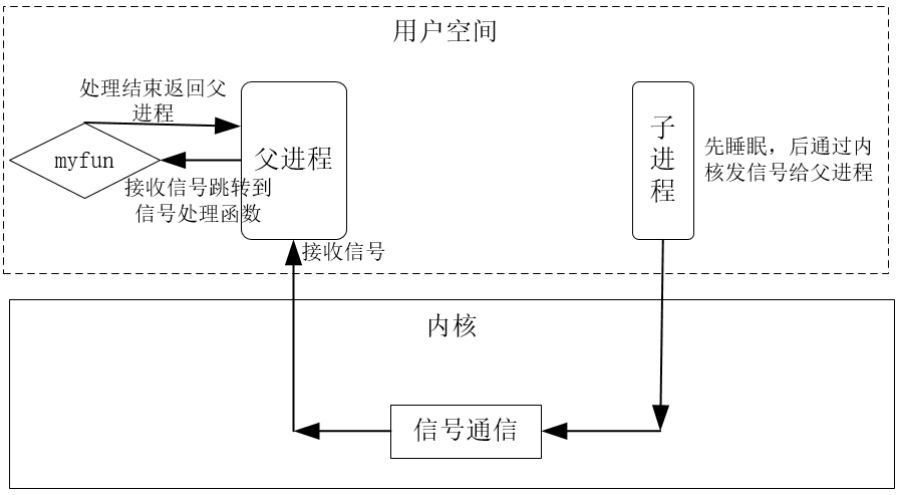
所需头文件	#include <signal.h>
函数原型	void(*signal(int signum, void(*handler)(int)))(int);
函数传入值	signum：指定信号 handler：SIG_IGN，忽略该信号； SIG_DFL，采用系统默认方式处理信号。 自定义的信号处理函数指针
函数返回值	成功：设置之前的信号处理方式；出错-1

```
//函数分析signal
/* void(*signal(int signum, void(*handler)(int)))(int);
 * 参数signum：整型变量，信号值
 * 参数handler：函数指针，指向自定义的信号值处理函数
 * 返回值：返回一个函数指针
*/
void(*signal(int signum, void(*handler)(int)))(int);
A = void(*handler)(int) //---->//函数指针变量，函数的形式：含有一个整型参数，无返回值

void(*signal(int signum,A))(int);
/* signal：含有两个参数，
 * 参数1：信号值；  参数2：函数指针
```

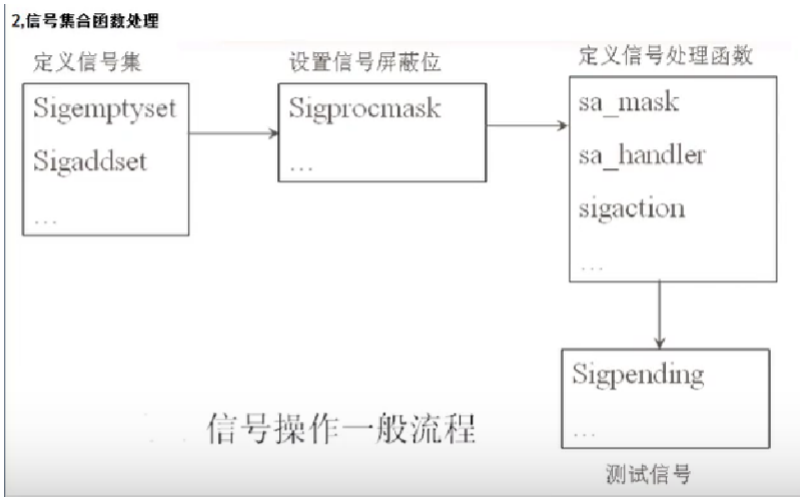
```
 * 返回值：函数指针
 * 功能：第一个参数处理哪个信号，第二个参数告诉内核怎样处理这个信号
*/

/*********************************************************************************************************/
//signal function example 示例 自定义信号处理函数
/* 程序执行流程：进程中通过alarm让内核经过9s后发送14号信号给当前进程（自身），当前进程接收到信号值时，主函数先打印8条语句，
 * 然后跳到myfun中运行，打印10条语句，打印完后，跳到主函数中继续运行打印语句，打印到i = 20时结束进程
 *
*/
#include "sys/type.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
void myfun(int signum)
{
    int i;
    i = 0;
    while(i < 10)
    {
        printf("process signal signum = %d\n", signum);//打印10条语句
        sleep(1);
        i++;
    }
    return; //返回main函数
}
int main()
{
    int i;
    i = 0;
    signal(14, myfun);  //处理的信号值（14 对应SIGALRM），怎样处理（用myfun函数进行处理）
    printf("alarm before\n");
    alarm(9);
    printf("alarm after\n");
    while(i < 20)
    {
        i++;
        sleep(1);//睡眠1s
        printf("process things, i = %d\n", i);//先打印i = 1~8，跳到myfun中打印，然后继续打印i = 9~20,结束
    }
    return 0;
}

/*********************************************************************************************************/
//signal function example 示例 SIG_IGN、SIG_DFL
/* 程序执行流程1（SIG_IGN）：进程中通过alarm让内核经过9s后发送14号信号给当前进程（自身），当前进程接收到信号值时，主函数已打
 * 印8条语句，由于信号处理函数忽略收到的信号值，不会跳到myfun中打印10条语句，仍然在主函数中继续运行打印语句，直到打印到i =
 * 20时结束进程
 * 程序执行流程2（SIG_DFL）：进程中通过alarm让内核经过9s后发送14号信号给当前进程（自身），终止当前进程，程序打印到i = 8位置
```

```c
*/
#include "sys/type.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
void myfun(int signum)
{
    int i;
    i = 0;
    while(i < 10)
    {
        printf("process signal signum = %d\n", signum);//打印10条语句
        sleep(1);
        i++;
    }
    return; //返回main函数
}
int main()
{
    int i;
    i = 0;
    signal(14, myfun);  //处理的信号值（14 对应SIGALRM），怎样处理（用myfun函数进行处理）
    printf("alarm before\n");
    alarm(9);
    printf("alarm after\n");
    signal(14, SIG_IGN);    //告诉内核忽略信号值为14的信号，这条语句刷新line82的代码，进程按照这次的信号处理方式执行
    signal(14, SIG_DFL);    //告诉内核默认执行信号值为14的信号处理方式，这条语句刷新line82和line86的代码，进程立即zhon'z
    while(i < 20)
    {
        i++;
        sleep(1);//睡眠1s
        printf("process things, i = %d\n", i);//先打印i = 1~8，跳到myfun中打印，然后继续打印i = 9~20,结束
    }
    return 0;
}
```

综合实例：程序执行框图如下，父子进程间信号通信



```c
//函数执行输出流程：
//父进程打印语句直到i = 9；然后跳转到myfun中打印直到i = 4；然后重新跳转到父进程持续打印直到i = 14（子进程一共睡眠了20s），跳转到myfun1中打印receive signum = 17，接收到信号值为17的信号，且此时myfun1中wait函数将子进程资源回收，子进程不会出现僵死进程，然后重新跳转到父进程中持续打印语句；
//通过命令：ps -axj 查看得父进程状态为S+，子进程不存在
#include "sys/type.h"
#include "signal.h"
#include "stdio.h"
#include "stdlib.h"
void myfun(int signum)
{
    int i;
    i = 0;
    while(i < 5)
    {
        printf("receive signum = %d, i = %d\n", signum, i);
        sleep(1);   //睡眠1s
        i++;
    }
    return; //退出
}
void myfun1(int signum)
{
    printf("receive signum = %d\n", signum);
    wait(NULL); //利用wait函数回收子进程资源，防止子进程僵死
    return; //退出返回
}
int main()
{
    pid_t pid;
    pid = fork();
    if(pid > 0)
    {
        int i;
        i = 0;
        signal(10, myfun);  //对10号信号值进行处理，跳入到myfun函数中处理
        signal(17, myfun1); //回收子进程资源，防止僵死
        while(1)
        {
            printf("parent process things, i = %d\n", i);
            sleep(1);
            i++;
        }
    }
    if(pid == 0)
    {
        sleep(10);  //子进程先睡眠10s
        kill(getppid(), 10); //向父进程发10号信号值（SIGUSRI）
        sleep(10);
        exit(0);    //退出 该函数内部实际上通过kill函数发送了SIGCHLD信号（信号值为17）；即内部为kill(getppid,17);
    }
    return 0;
}
//kill函数 wait函数
```



内核代码跟踪

- 进程结构中的信号结构
- 发送信号：`do_kill do_send_specific do_send_sig_info send_signal __send_signal/list_add_tail __sigqueue_alloc`
- 注册信号：用户空间中进行注册，进程调度时，用户空间切换到内核空间，然后内核对用户空间中注册的信号进行读取
- 处理信号：涉及用户态和内核态的切换过程

大致流程：调用kill，内存中开辟一个空间，该空间为一个信号队列，信号队列存储在中断向量表中，然后再由另外的控制程序去去信号，再分配给进程

三、共享内存

IPC和文件I/O函数的比较

文件I/O	IPC
`open`	`Msg_get、Shm_get、Sem_get`
`read/write`	`msgsnd msgrecv、shmat shmdt、semop`
`close`	`msgctrl、shmctrl、semctrl`

共享内存的创建shmget

内核空间中生成一块缓存，类似用户空间的数组或malloc函数分配的空间一样

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

这里的key标识共享内存的键值，相当于进程的pid；key参数赋值0/`IPC_PRIVATE`表示创建新共享内存 对应key值都为0

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmget(key_t key, int size, int shmflg);
函数传入值	key:　IPC_PRIVATE或ftok的返回值 size:　共享内存的大小 shmflg: 同open函数的权限位，也可用8进制表示法
函数返回值	成功：共享内存段标识符ID（类似文件描述符）；出错-1

共享内存特点：

- 共享内存创建之后，一直存在与内核中，直到被删除或系统关闭；
- 共享内存和管道不一样，读取后，内容仍在其共享内存中

相关命令

查看IPC对象：`ipcs -m` 查看系统共享内存 `ipcs -q` 查看系统消息队列信息 `ipcs -s` 查看系统信号量消息

删除IPC对象：`ipcrm -m id` 移出id标识的共享内存段（同样使用参数`-q -s`时移除对应内容）

```
//shmget简单示例
#include "sys/type.h"
#include "sys/shm.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int shmid;
    shmid = shmget(IPC_PRIVATE, 128, 0777);
    if(shmid < 0)
    {
        printf("creat share memory failure\n");
        return -1;
    }
    printf("creat share memory sucess shmid = %d\n", shmid);
    system("ipcs -m");          //查看共享内存
    system("ipcrm -m shmid");   //删除共共享内存 如果只创建，不删除（即注释该行代码）,则系统共享内存一直存在
    return 0;
}

```

ftok：创建key值

`char ftok(const char *path, char key)` 参数`*path`：文件路径和文件名；参数`key`：一个字符；返回值：正确返回一个key值，错-1

用`IPC_PRIVATE`**操作时，共享内存的key值都一样，都是0**，所以使用`ftok`来创建key值，只要key值一样，用户空间的进程通过这个函数打开，则会对内核同一个IPC对象操作

```
//shmget简单示例 利用ftok创建key值
#include "sys/type.h"
#include "sys/shm.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int shmid;
    int key;
    key = ftok("./a.c", 'a');   //当前目录下的a.c文件 传入的参数为a 传入的参数不同，key值不同
    if(key < 0)
```

```
    {
        printf("creat key failure\n");
        return -2;
    }
    printf("creat key sucess key = %X\n", key);
    shmid = shmget(key, 128, IPC_CREAT | 0777);
    if(shmid < 0)
    {
        printf("creat share memory failure\n");
        return -1;
    }
    printf("creat share memory sucess shmid = %d\n", shmid);
    system("ipcs -m");          //查看共享内存
    system("ipcrm -m shmid");    //删除共共享内存 如果只创建，不删除（即注释该行代码）,则系统共享内存一直存在
    return 0;
}
```

shmat

`shmat` 将共享内存映射到用户空间地址中 为方便用户空间对共享内存的操作方式，使用地址映射的方式

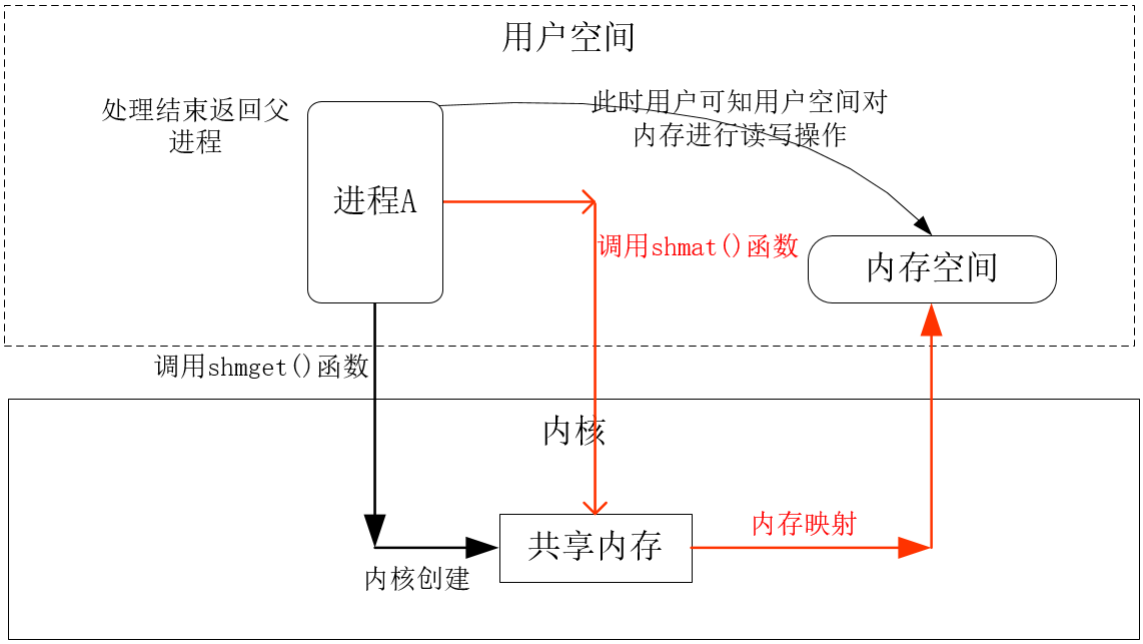
`void *shmat(int shmid, const void *shmaddr, int shmflg);`

参数`shmid`：ID号； 参数`*shmaddr`：映射到的地址，NULL为系统自动完成的映射；

参数`shmflg`：`SHM_RDONLY`**共享内存只读**，默认0，表示共享内存**可读写**

返回值：成功；**映射后的地址**；失败：NULL

程序执行框架




```
//实例
#include "sys/type.h"
#include "sys/shm.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main()
```

```c
{
    int shmid;
    int key;
    int *p;
    key = ftok("./a.c", 'a');   //当前目录下的a.c文件 传入的参数为a 传入的参数不同，key值不同
    if(key < 0)
    {
        printf("creat key failure\n");
        return -2;
    }
    printf("creat key sucess key = %X\n", key);
    shmid = shmget(key, 128, IPC_CREAT | 0777);
    if(shmid < 0)
    {
        printf("creat share memory failure\n");
        return -1;
    }
    printf("creat share memory sucess shmid = %d\n", shmid);
    system("ipcs -m");          //查看共享内存
    p = (char *)shmat(shmid, NULL, 0);//映射哪块共享内存 内核自动分配自动 访问权限是读写的
    if(p == NULL)
    {
        printf("shmat function failure\n");
        return -3;
    }
    //写共享内存
    fgets(p, 128, stdin);//写到哪里去， 写多少个 从什么地方写
    //读共享内存
    printf("share memory data:%s", p);
    printf("second read share memory data:%s", p);  //第二次读仍然可以读到
    //system("ipcrm -m shmid"); //删除共共享内存 如果只创建，不删除（即注释该行代码）,则系统共享内存一直存在
    return 0;
}
```

shmdt/shmctl

`int shmdt(const void *shmaddr);` 功能：将进程里的地址映射删除 参数 `*shmaddr`：共享内存映射后的地址；返回值：成功0，出错-1

`int shmctl(int shmid, int cmd, struct shmid_ds *buf);` 功能删除共享内存对象（内核层面）返回值：成功0，出错-1

参数 `shmid`：要操作的共享内存标识符

参数 `cmd`： `IPC_STAT`(获取对象属性，相当于`ipce -m`)、`IPC_SET`（设置对象属性）、`IPC_RMID`（删除对象，相当于`ipcrm -m`）

参数 `buf`： 指定`IPC_STAT/IPC_SET`时用一保存/设置属性

```c
//shmdt实例
#include "sys/type.h"
#include "sys/shm.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
```

```c
int main()
{
    int shmid;
    int key;
    int *p;
    key = ftok("./a.c", 'a');   //当前目录下的a.c文件 传入的参数为a 传入的参数不同，key值不同
    if(key < 0)
    {
        printf("creat key failure\n");
        return -2;
    }
    printf("creat key sucess key = %X\n", key);
    shmid = shmget(key, 128, IPC_CREAT | 0777);
    if(shmid < 0)
    {
        printf("creat share memory failure\n");
        return -1;
    }
    printf("creat share memory sucess shmid = %d\n", shmid);
    system("ipcs -m");          //查看共享内存
    p = (char *)shmat(shmid, NULL, 0);//映射哪块共享内存 内核自动分配自动 访问权限是读写的
    if(p == NULL)
    {
        printf("shmat function failure\n");
        return -3;
    }
    //写共享内存
    fgets(p, 128, stdin);//写到哪里去， 写多少个 从什么地方写
    //读共享内存
    printf("share memory data:%s", p);
    printf("second read share memory data:%s", p);  //第二次读仍然可以读到

    shmdt(p);   //释放（删除）内核共享内存映射到用户空间的内存
    memcpy(p, "abce", 4);//向用户空间中的映射内存写内容，无法执行，因为此时该地址空间已被释放
    //system("ipcrm -m shmid"); //删除共共享内存 如果只创建，不删除（即注释该行代码）,则系统共享内存一直存在
    return 0;
}


/*********************************************************************************************************/
//shmctl实例 shmctl可实现ipcrm -m [共享内存ID] 即命令移除共享内存功能，同样可实现ipcs -m查看共享内存功能
#include "sys/type.h"
#include "sys/shm.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int shmid;
    int key;
    int *p;
```

```
    key = ftok("./a.c", 'a');   //当前目录下的a.c文件 传入的参数为a 传入的参数不同，key值不同
    if(key < 0)
    {
        printf("creat key failure\n");
        return -2;
    }
    printf("creat key sucess key = %X\n", key);
    shmid = shmget(key, 128, IPC_CREAT | 0777);
    if(shmid < 0)
    {
        printf("creat share memory failure\n");
        return -1;
    }
    printf("creat share memory sucess shmid = %d\n", shmid);
    system("ipcs -m");          //查看共享内存
    p = (char *)shmat(shmid, NULL, 0);//映射哪块共享内存 内核自动分配自动 访问权限是读写的
    if(p == NULL)
    {
        printf("shmat function failure\n");
        return -3;
    }
    //写共享内存
    fgets(p, 128, stdin);//写到哪里去， 写多少个 从什么地方写（键盘输入）
    //读共享内存
    printf("share memory data:%s", p);
    printf("second read share memory data:%s", p);  //第二次读仍然可以读到

    shmdt(p);   //释放（删除）内核共享内存映射到用户空间的内存
    shmctl(shmid, IPC_RMID, NULL);//删除哪一个共享内存，选择命令为删除，删除指令不需要配置结构体
    system("ipce -m");  //查看系统共享内存是否还存在
    //system("ipcrm -m shmid"); //删除共共享内存 如果只创建，不删除（即注释该行代码）,则系统共享内存一直存在
    return 0;
}
```

综合实例

父子进程间共享内存单向通信实例

父进程写内容进共享内存，子进程读共享内存，形成单向通信实例

类似无名管道的创建时机，需要先创建共享内存在创建父进程

```
//
/* getppid()：获取当前父进程的pid值
 *
*/
#include "sys/type.h"
#include "sys/shm.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
void myfun(int signum)
```

```c
{
    return;     //信号处理函数中不需要进行任何处理直接返回
}
int main()
{
    int shmid;
    int key;
    int *p;
    int pid;

    shmid = shmget(IPC_PRIVATE, 128, IPC_CREAT | 0777); //内核空间生成共享内存key值默认0，大小为128字节，
    if(shmid < 0)
    {
        printf("creat share memory failure\n");
        return -1;
    }
    printf("creat share memory sucess shmid = %d\n", shmid);
    pid = fork();
    if(pid > 0)     //父进程
    {
        signal(SIGUAR2, myfun);     //信号处理函数 处理子进程发过来的信号
        p = (char *)shmat(shmid, NULL, 0);//映射哪块共享内存 内核自动分配自动 访问权限是读写的
        if(p == NULL)
        {
            printf("parent process:shmat function failure\n");
            return -3;
        }
        while(1)
        {
            //写共享内存
            printf("parent process start write share memory:\n");
            fgets(p, 128, stdin);   //写到哪里去，写多少个，以什么方式写（这里键盘输入）
            kill(pid, SIGUSR1); //告诉子进程，发一个信号值 告诉子进程去读数据
            pause();            //进入等待
        }
    }
    if(pid == 0)        //子进程
    {
        signal(SIGUSR1, myfun);     //处理父进程发过来的信号
        //父进程需要地址映射，子进程同样需要
        p = (char *)shmat(shmid, NULL, 0);//映射哪块共享内存 内核自动分配自动 访问权限是读写的
        if(p == NULL)
        {
            printf("parent process:shmat function failure\n");
            return -3;
        }
        while(1)
        {
            pause();        //等待父进程写
            //开始去读
            printf("share memory data:%s\n", p);    //读父进程写进共享内存的数据
            kill(getppid(), SIGUSR2);   //告诉父进程读完
        }
    }
```

```
    shmdt(p);   //释放（删除）内核共享内存映射到用户空间的内存
    shmctl(shmid, IPC_RMID, NULL);//删除哪一个共享内存，选择命令为删除，删除指令不需要配置结构体
    system("ipce -m");  //查看系统共享内存是否还存在
    //system("ipcrm -m shmid"); //删除共共享内存 如果只创建，不删除（即注释该行代码）,则系统共享内存一直存在
    return 0;
}
```

无亲缘关系进程间共享内存单向通信

```
/* server -> client单向通信
 * 对于服务器端先打开共享内容，然后先把自身pid放进共享内存中，然后客户端读共享内存中信息获取服务器pid，然后写入自身pid，
 * 然后服务器端读共享内存获取客户端pid 至此，服务器和客户端pid交换完成
*/
/*********************************************************************************************************/
//server.c
#include "sys/type.h"
#include "sys/shm.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
struct mybuf{       //用于交换信息
    int pid;        //占4字节
    char buf[124];  //分配124字节 124 + 4刚好等于设定的共享内存的大小
};
void myfun(int signum)
{
    return;     //信号处理函数中不需要进行任何处理直接返回
}
int main()
{
    int shmid;
    int key;
    struct mybuf *p;
    int pid;
    key = ftok("./a.c", 'a');   //路径为当前目录下的a.c文件 字符为a
    if(key < 0)
    {
        printf("creat key failure\n");
        return -2;
    }
    printf("creat key sucess key = %X\n", key);

    shmid = shmget(key, 128, IPC_CREAT | 0777); //传入创建的key作为共享内存的标识，大小为128字节，
    if(shmid < 0)
    {
        printf("creat share memory failure\n");
        return -1;
    }
    printf("creat share memory sucess shmid = %d\n", shmid);

```

```
    signal(SIGUAR2, myfun);      //信号处理函数 处理客户端发过来的信号
    p = (struct mybuf *)shmat(shmid, NULL, 0);//映射哪块共享内存 内核自动分配自动 访问权限是读写的
    if(p == NULL)
    {
        printf("parent process:shmat function failure\n");
         return -3;
    }
    //读写前先交换pid 此为服务器，获取客户端pid
    p -> pid = getpid();     //写入自身（服务器）pid到共享内存中
    pause();                //停止，等待客户端读取共享内存的自身（服务器）pid
    pid = p -> pid;          //读客户端pid，客户端将自身的pid写入共享内存

    while(1)
    {
        //写共享内存
        printf("parent process start write share memory:\n");
        fgets(p -> buf, 128, stdin);    //写到哪里去，写多少个，以什么方式写（这里键盘输入）
        kill(pid, SIGUSR1); //告诉客户端进程，发一个信号值 告诉子进程去读数据
        pause();            //进入等待 等待客户端进程读
     }

    shmdt(p);   //释放（删除）内核共享内存映射到用户空间的内存
    shmctl(shmid, IPC_RMID, NULL);//删除哪一个共享内存，选择命令为删除，删除指令不需要配置结构体
    system("ipce -m");  //查看系统共享内存是否还存在
    //system("ipcrm -m shmid"); //删除共共享内存 如果只创建，不删除（即注释该行代码）,则系统共享内存一直存在
    return 0;
}

/*************************************************************************************************/
//client.c
#include "sys/type.h"
#include "sys/shm.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
struct mybuf{       //用于交换信息
    int pid;        //占4字节
    char buf[124];  //分配124字节 124 + 4刚好等于设定的共享内存的大小
};
void myfun(int signum)
{
    return;     //信号处理函数中不需要进行任何处理直接返回
}
int main()
{
    int shmid;
    int key;
    struct mybuf *p;
    int pid;
    key = ftok("./a.c", 'a');   //路径为当前目录下的a.c文件 字符为a
    if(key < 0)
```

```c
    {
        printf("creat key failure\n");
        return -2;
    }
    printf("creat key sucess key = %X\n", key);

    shmid = shmget(key, 128, IPC_CREAT | 0777); //传入创建的key作为共享内存的标识，大小为128字节，
    if(shmid < 0)
    {
        printf("creat share memory failure\n");
        return -1;
    }
    printf("creat share memory sucess shmid = %d\n", shmid);

    signal(SIGUAR1, myfun);     //信号处理函数 处理子进程发过来的信号
    p = (struct mybuf *)shmat(shmid, NULL, 0);//映射哪块共享内存 内核自动分配自动 访问权限是读写的
    if(p == NULL)
    {
        printf("parent process:shmat function failure\n");
         return -3;
    }
    //读写前先交换pid 此为客户端，获取服务器pid
    pid = p -> pid;          //读共享内存中 读服务器写入的自身pid
    p -> pid = getpid();     //写共享内存，写入自身（客户端）的pid
    kill(pid, SIGUSR2);      //发送信号通知服务器获取完了pid 并写了客户端pid
    pause();                //停止，等待客户端读取共享内存的自身（服务器）pid
    //开始在共享内存中持续读数据
    while(1)
    {
        pause();             //进入等待 等待服务器写
        printf("client process receive data from share memory:%s\n", p -> buf);
        fgets(p -> buf, 128, stdin);    //写到哪里去，写多少个，以什么方式写（这里键盘输入）
        kill(pid, SIGUSR2); //告诉服务器，发一个信号值 告诉服务器可以继续写

     }

    shmdt(p);   //释放（删除）内核共享内存映射到用户空间的内存
    shmctl(shmid, IPC_RMID, NULL);//删除哪一个共享内存，选择命令为删除，删除指令不需要配置结构体
    system("ipce -m");  //查看系统共享内存是否还存在
    //system("ipcrm -m shmid"); //删除共共享内存 如果只创建，不删除（即注释该行代码）,则系统共享内存一直存在
    return 0;
}
```

四、消息队列

与文件IO对比

文件I/O	消息队列
`open`	`msgget`
`read`	`msgrcv`
`write`	`msgsnd`
`close`	`msgctrl`

先进先出结构，同样在内核区开辟一个内存进行通信

通过`ipcs`命令，可同时查看共享内存、消息队列、信号量信息，三者可类比学习

megget、msgctl

megget：创建新的消息队列或获取已有的消息队列
msgctl：直接控制消息队列的行为

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>
函数原型	int msgget(key_t key, int flag);
函数传入值	key： 和消息队列关联的key值 flag：消息队列的访问权限
函数返回值	成功：消息队列ID（类似文件描述符）；出错-1

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>
函数原型	int msgctl(int msgqid, int cmd, struct msqid_ds *buf);
函数传入值	msqid：消息队列ID cmd：IPC_STAT：读取消息队列的属性，并将其保存在buf执行的缓存区中 IPC_SET：设置消息队列的属性，值取自buf参数 IPC_RMID：从系统中删除消息队列 buf：消息队列缓存区
函数返回值	成功：0；出错-1

```c
//msgget、msgctl
#include "sys/type.h"
#include "sys/msg.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int msgid;
    msgid = msgget(IPC_PRIVATE, 0777);  //默认宏 key为0
    if(msgid < 0)
    {
        printf("creat message queue failure\n");
```

```
        return -1;
    }
    printf("creat message queue sucess msgid = %d\n", msgid);
    system("ipcs -q");  //查看系统消息队列
    //删除消息队列
    msgctl(msgid, IPC_RMID, NULL);
    system("ipcs, -q");
    return 0;
}
```

msgsnd、msgrcv

可设置阻塞/非阻塞

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>
函数原型	int msgsnd(int msqid, const void *msgp, size_t size, int flag);
函数传入值	msqid：消息队列ID msgp：指向消息的指针 size：发送的消息正文的字节数 flag：IPC_NOWAIT 消息没有发送完函数也会立即返回 0，直到发送完成，函数才返回
函数返回值	成功：0；出错-1

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>
函数原型	int msgrcv(int msqid, void *msgp, size_t size, long msgtype, int flag);
函数传入值	msqid：消息队列ID *msgp：接收消息的缓冲区 size：要接收的消息的字节数 msgtype：0，接收消息队列中的第一个消息； 大于0，接收消息队列中第一个类型为msgtyp的消息 小于0，接收消息队列中类型值不大于msgtyp的绝对值且类型之又最小的消息 flag：IPC_NOWAIT，若没有消息，进程立即返回ENOMSG 0，若无消息，函数一直阻塞
函数返回值	成功：接收到消息的长度；出错-1

```
//常用消息的结构体
struct msgbuf{
    long mtype;      //消息类型
    char mtext[N];   //消息正文
};
```

实例：

```
//msgsnd 写
#include "sys/type.h"
#include "sys/msg.h"
#include "signal.h"
```

```c
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"     //调用strlen
struct msgbuf{
    long type;          //消息类型
    char voltage[124];   //自定义
    char ID[4];         //自定义
}
int main()
{
    int msgid;
    struct msgbuf sendbuf;
    msgid = msgget(IPC_PRIVATE, 0777);  //默认宏 key为0
    if(msgid < 0)
    {
        printf("creat message queue failure\n");
        return -1;
    }
    printf("creat message queue sucess msgid = %d\n", msgid);
    system("ipcs -q");  //查看系统消息队列
    //初始化消息对象
    sendbuf.type = 100; //设定消息类型为100
    printf("please input message:\n");
    fgets(sendbuf.voltage, 124, stdin);
    //开始写消息队列
    msgsnd(msgid, (void *)&sendbuf, strlen(sendbuf.voltage), 0);//告诉内核向哪个消息队列写，写什么消息

    while(1);
    //删除消息队列
    msgctl(msgid, IPC_RMID, NULL);/
    system("ipcs, -q");
    return 0;
}

//msgrcv 读
#include "sys/type.h"
#include "sys/msg.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"     //调用strlen
struct msgbuf{
    long type;          //消息类型
    char voltage[124];   //自定义
    char ID[4];         //自定义
}
int main()
{
    int msgid;
    int readret;    //接收读返回值
    struct msgbuf sendbuf, recvbuf;     //定义发送，接收缓存
    msgid = msgget(IPC_PRIVATE, 0777);  //默认宏 key为0
    if(msgid < 0)
    {
        printf("creat message queue failure\n");
```

```
        return -1;
    }
    printf("creat message queue sucess msgid = %d\n", msgid);
    system("ipcs -q");  //查看系统消息队列
    //初始化消息对象
    sendbuf.type = 100; //设定消息类型为100
    printf("please input message:\n");
    fgets(sendbuf.voltage, 124, stdin);
    //开始写消息队列
    msgsnd(msgid, (void *)&sendbuf, strlen(sendbuf.voltage), 0);//告诉内核向哪个消息队列写，写什么消息

    //开始从消息队列中读消息
    memset(recv.voltage, 0, 124);   //清空接收缓存区
    readret = msgcrv(msgid, (void *)&recvbuf, 124, 100, 0); //读
    printf("recv:%s\n", recvbuf.voltage);
    printf("readret = %d\n", readret);
    /*
    //第二次读消息队列测试
    readret = msgcrv(msgid, (void *)&recvbuf, 124, 100, 0); //且阻塞
    printf("send read after\n");        //无法打印
    */
    //删除消息队列
    msgctl(msgid, IPC_RMID, NULL);/
    system("ipcs, -q");
    return 0;
}
```

无亲缘进程间消息队列间通信

由于是无亲缘关系的进程，为保证两个进程对同一个消息队列对象进行通信，需要通过ftok函数创建同一个key

实例1 单向通信

```
//写消息进消息队列进程
#include "sys/type.h"
#include "sys/msg.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"     //调用strlen
struct msgbuf{
    long type;          //消息类型
    char voltage[124];   //自定义
    char ID[4];         //自定义
}
int main()
{
    int msgid;
    int readret;    //接收读返回值
    int key;
    struct msgbuf sendbuf, recvbuf;     //定义发送，接收缓存
    key = ftok("./a.c", 'a');           //创建key
    if(key < 0)
```

```
    {
        printf("creat key failure\n");
        return -2;
    }
    msgid = msgget(key, IPC_CREAT | 0777);  //默认宏 key为0
    if(msgid < 0)
    {
        printf("creat message queue failure\n");
        return -1;
    }
    printf("creat message queue sucess msgid = %d\n", msgid);
    system("ipcs -q");  //查看系统消息队列
    sendbuf.type = 100; //设定消息类型为100
    //写消息队列
    while(1)
    {
        memset(sendbuf.voltage, 0, 124);    //清除读缓存区
        printf("please input message:\n");
        fgets(sendbuf.voltage, 124, stdin);
         msgsnd(msgid, (void *)&sendbuf, strlen(sendbuf.voltage), 0);//告诉内核向哪个消息队列写，写什么消息
    }
    //删除消息队列
    msgctl(msgid, IPC_RMID, NULL);
    system("ipcs, -q");
    return 0;
}

/*************************************************************************************************/
//从消息队列读消息进程
#include "sys/type.h"
#include "sys/msg.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"     //调用strlen
struct msgbuf{
    long type;          //消息类型
    char voltage[124];   //自定义
    char ID[4];         //自定义
}
int main()
{
    int msgid;
    int readret;    //接收读返回值
    int key;
    struct msgbuf sendbuf, recvbuf;     //定义发送，接收缓存
    key = ftok("./a.c", 'a');           //创建key
    if(key < 0)
    {
        printf("creat key failure\n");
        return -2;
    }
    msgid = msgget(key, IPC_CREAT | 0777);  //默认宏 key为0
    if(msgid < 0)
    {
```

```
        printf("creat message queue failure\n");
        return -1;
    }
    printf("creat message queue sucess msgid = %d\n", msgid);
    system("ipcs -q");  //查看系统消息队列

    //读消息队列
    while(1)
    {
        memset(recvbuf.voltage, 0, 124);    //清除接收缓存区
        //告诉内核向哪个消息队列读，读消息到哪去，读多少个，读哪种消息 以阻塞方式读
         msgrcv(msgid, (void *)&recvbuf, 124, 100, 0);
        printf("receive data from message queue:%s", recebuf.voltage);
    }
    //删除消息队列
    msgctl(msgid, IPC_RMID, NULL);
    system("ipcs, -q");
    return 0;
}
```

实例2 双向通信

同一个消息队列，无亲缘关系进程间读写

```
//server.c 服务器端
#include "sys/type.h"
#include "sys/msg.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"     //调用strlen
struct msgbuf{
    long type;          //消息类型
    char voltage[124];   //自定义
    char ID[4];         //自定义
}
int main()
{
    int msgid;
    int readret;    //接收读返回值
    int key;
    int pid;        //为实现同时收发功能，通过父子进程实现

    struct msgbuf sendbuf, recvbuf;     //定义发送，接收缓存
    key = ftok("./a.c", 'a');           //创建key
    if(key < 0)
    {
        printf("creat key failure\n");
        return -2;
    }
    msgid = msgget(key, IPC_CREAT | 0777);  //默认宏 key为0
    if(msgid < 0)
    {
        printf("creat message queue failure\n");
        return -1;
    }
```

```
    printf("creat message queue sucess msgid = %d\n", msgid);
    system("ipcs -q");  //查看系统消息队列
    pid = fork();//保证父子进程都对同一个消息队列进行操作，fork函数需要在消息队列创建后才创建
    if(pid > 0)     //父进程 负责写 写的消息类型是100
    {
        sendbuf.type = 100; //设定消息类型为100
        //写消息队列
        while(1)
        {
            memset(sendbuf.voltage, 0, 124);    //清除发送缓存区
            printf("please input message:\n");
            fgets(sendbuf.voltage, 124, stdin);
            msgsnd(msgid, (void *)&sendbuf, strlen(sendbuf.voltage), 0);//告诉内核向哪个消息队列写，写什么消息
        }
    }
    if(pid == 0)        //子进程 负责读 读的消息类型为200
    {
        while(1)
        {
            memset(recvbuf.voltage, 0, 124);    //清除读缓存区
            msgcrv(msgid, (void *)&recvbuf, 124, 200, 0);
            printf("receive message from message queue:%s\n", recvbuf.voltage);
        }
    }

    //删除消息队列
    msgctl(msgid, IPC_RMID, NULL);
    system("ipcs, -q");
    return 0;
}

/*************************************************************************************/
//client.c
#include "sys/type.h"
#include "sys/msg.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"     //调用strlen
struct msgbuf{
    long type;          //消息类型
    char voltage[124];   //自定义
    char ID[4];         //自定义
}
int main()
{
    int msgid;
    int readret;    //接收读返回值
    int key;
    int pid;        //为实现同时收发功能，通过父子进程实现

    struct msgbuf sendbuf, recvbuf;     //定义发送，接收缓存
    key = ftok("./a.c", 'a');           //创建key
```

```
    if(key < 0)
    {
        printf("creat key failure\n");
        return -2;
    }
    msgid = msgget(key, IPC_CREAT | 0777);  //默认宏 key为0
    if(msgid < 0)
    {
        printf("creat message queue failure\n");
        return -1;
    }
    printf("creat message queue sucess msgid = %d\n", msgid);
    system("ipcs -q");  //查看系统消息队列
    pid = fork();//保证父子进程都对同一个消息队列进行操作，fork函数需要在消息队列创建后才创建
    if(pid == 0)        //子进程 负责写 写的消息类型是200
    {
        sendbuf.type = 200; //设定消息类型为200
        //写消息队列
        while(1)
        {
            memset(sendbuf.voltage, 0, 124);    //清除发送缓存区
            printf("please input message:\n");
            fgets(sendbuf.voltage, 124, stdin);
            msgsnd(msgid, (void *)&sendbuf, strlen(sendbuf.voltage), 0);//告诉内核向哪个消息队列写，写什么消息
        }
    }
    if(pid > 0)     //父进程 负责读 读的消息类型为100
    {
        while(1)
        {
            memset(recvbuf.voltage, 0, 124);    //清除读缓存区
            msgcrv(msgid, (void *)&recvbuf, 124, 100, 0);
            printf("receive message from message queue:%s\n", recvbuf.voltage);
        }
    }

    //删除消息队列
    msgctl(msgid, IPC_RMID, NULL);
    system("ipcs, -q");
    return 0;
}
```

五、信号灯

和消息队列以及共享内存一样，信号灯存在于内核空间
信号灯：信号量的集合，利用函数对多个信号量集合的控制 IPC对象是一个信号灯集（多个信号量）

semget、semctl

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semget(key_t key, int nsems, int semflg);
函数传入值	key：和信号灯集关联的key值 nsems：信号灯集中包含的信号灯数目 semflg：信号灯集的访问权限
函数返回值	成功：信号灯集ID；出错-1

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semctl(int semid, int semnum, int cmd, union semun arg(***));
函数传入值	semid：信号灯集ID semnum：要修改的信号灯编号 cmd：GETVAL，获取信号灯的值 SETVAL，设置信号灯的值 IPC_RMID，从系统中删除信号灯集合
函数返回值	成功：0；出错-1

```c
//设置信号灯值的共用体union
union semun{
    int val;    //设置信号灯的值
    struct semid_ds *buf;    //获取/设置对象属性
    unsigned short *array;
    struct seminfo *__buf;
}
```

```c
//semget、semctl
#include "sys/type.h"
#include "sys/sem.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int semgid;
    semgid = semget(IPC_PRIVATE, 3, 0777);  //默认宏 key为0
    if(semgid < 0)
    {
        printf("creat semaphore failure\n");
        return -1;
    }
    printf("creat semaphore sucess msgid = %d\n", semgid);
    system("ipcs -s");  //查看系统信号灯
    while(1);
    return 0;
}
```

```c

//semctl  实现删除一个信号灯
#include "sys/type.h"
#include "sys/sem.h"
#include "signal.h"
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
int main()
{
    int semgid;
    semgid = semget(IPC_PRIVATE, 3, 0777);  //默认宏 key为0
    if(semgid < 0)
    {
        printf("creat semaphore failure\n");
        return -1;
    }
    printf("creat semaphore sucess msgid = %d\n", semgid);
    system("ipcs -s");  //查看系统信号灯
    semctl(semid, 0, IPC_RMID, NULL);       //删除信号灯集合 删除时，第四个参数可有可无
    system("ipcs -s");
    return 0;
}

```

semop

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semop(int semid, struct sembuf *opsptr, size_t nops);

函数传入值	semid：信号灯集ID
	struct sembuf{
	short sem_num;//要操作的信号灯的编口
	short sem_op;//0，等待，直到信号灯的值变为0
	//1，释放资源 V操作 -1，分配资源 P操作
	short sem_flg; //0，IPC_NOWAIT，SEM_UNDO
	};
	nops：要操作的信号灯数

函数返回值	成功：0；出错-1

```c
//通过信号量确定每次主线程先运行，然后到子线程运行
#include "stdio.h"
#include "stdlib.h"
#include "pthread.h"
#include "semaphore.h"
sem_t sem;  //定义信号量
void *fun(void *var)    //子线程
{
    int j;
    //p操作，等待
    sem_wait(sem);  //进行等待
    for(j = 0; j < 10; j++)
    {
```

```c
        usleep(100);
        printf("this is fun j = %d\n", j);
    }
}
int main()  //主线程
{
    int i;
    char str[] = "hello linux\n";
    pthread_t tid;
    int ret;
    sem_init(&sem, 0, 0);   //对哪一个信号量进行初始化，选择用于线程间通信，初始值为0
    ret = pthread_create(&tid, NULL, fun, (void *)str);
    if(ret < 0)
    {
        printf("creat thread failure\n");
        return -1;
    }
    for(i = 0; i < 10; i++)
    {
        usleep(100);
        printf("this is main fun i = %d\n", i);
    }
    //v操作
    sem_post(&sem);
    while(1);
    return 0;
}
/*************************************************************************************/
//信号灯实现控制 主线程先运行后子线程后运行
/*************************************************************************************/
#include "stdio.h"
#include "stdlib.h"
#include "pthread.h"
#include "sys/ipc.h"
#include "sys/sem.h"
union semun{
    int val;    //设置信号灯的值
    struct semid_ds *buf;   //获取/设置对象属性
    unsigned short *array;
    struct seminfo *__buf;
}
int semid;  //定义信号灯集
union semun mysemun;
struct sembuf mysembuf;
void *fun(void *var)    //子线程
{
    int j;
    //p操作，等待 信号灯的p操作
    mysembuf.sem_op = -1;
    semop(semid, &mysembuf, 1);
    for(j = 0; j < 10; j++)
    {
        usleep(100);
        printf("this is fun j = %d\n", j);
    }
```

```c
}
int main()  //主线程
{
    int i;
    char str[] = "hello linux\n";
    pthread_t tid;
    int ret;
    semid = semget(IPC_PRIVATE, 3, 0777);   //创建信号灯对象
    if(semid < 0)
    {
        printf("creat semaphore failure\n");
        return -1;
    }
    printf("creat semaphore sucess, semid = %d\n", semid);
    system("ipcs -s");  //查看系统创建的信号灯
    //信号灯初始化
    mysembuf.sem_num = 0;   //信号灯编号为0
    mysembuf.sem_flg = 0
    ret = pthread_create(&tid, NULL, fun, (void *)str);
    if(ret < 0)
    {
        printf("creat thread failure\n");
        return -1;
    }
    for(i = 0; i < 10; i++)
    {
        usleep(100);
        printf("this is main fun i = %d\n", i);
    }
    //v操作
    mysembuf.sem_op = 1;
    semop(semid, &mysembuf, 1);
    while(1);
    return 0;
}
```

实例 无亲缘关系之间信号灯通信

```c
//server.c 输出10条语句
#include "stdio.h"
#include "stdlib.h"
#include "pthread.h"
#include "sys/ipc.h"
#include "sys/sem.h"
union semun{
    int val;    //设置信号灯的值
    struct semid_ds *buf;   //获取/设置对象属性
    unsigned short *array;
    struct seminfo *__buf;
}
int semid;  //定义信号灯集
union semun mysemun;
struct sembuf mysembuf;
int main()  //主线程
{
    int i;
    int key;
```

```
    key = ftok("./a.c", 'a');
    if(key < 0)
    {
        printf("creat key failure\n");
        return -1;
    }
    printf("creat key sucess\n");
    semid = semget(key, 3, IPC_CREAT | 0777);   //创建信号灯对象
    if(semid < 0)
    {
        printf("creat semaphore failure\n");
        return -2;
    }
    printf("creat semaphore sucess, semid = %d\n", semid);
    system("ipcs -s");  //查看系统创建的信号灯
    //信号灯初始化
   // mysembuf.sem_val = 0; //信号灯编号为0
   // semctl(semid, 0, SETVAL, mysemun);

    mysembuf.sem_num = 0;
    mysembuf.sem_flg = 0;
    for(i = 0; i < 10; i++)
    {
        usleep(100);
        printf("this is main fun i = %d\n", i);
    }
    //v操作
    mysembuf.sem_op = 1;
    semop(semid, &mysembuf, 1);
    while(1);
    return 0;
}
/*************************************************************************************
************************************/
//client.c 输出10条语句
#include "stdio.h"
#include "stdlib.h"
#include "pthread.h"
#include "sys/ipc.h"
#include "sys/sem.h"
union semun{
    int val;    //设置信号灯的值
    struct semid_ds *buf;   //获取/设置对象属性
    unsigned short *array;
    struct seminfo *__buf;
}
int semid;  //定义信号灯集
union semun mysemun;
struct sembuf mysembuf;
int main()  //主线程
{
    int i;
    int key;
    key = ftok("./a.c", 'a');
    if(key < 0)
    {
        printf("creat key failure\n");
        return -1;
```

```
    }
    printf("creat key sucess\n");
    semid = semget(key, 3, IPC_CREAT | 0777);   //创建信号灯对象
    if(semid < 0)
    {
        printf("creat semaphore failure\n");
        return -2;
    }
    printf("creat semaphore sucess, semid = %d\n", semid);
    system("ipcs -s");  //查看系统创建的信号灯
    //信号灯初始化
    mysembuf.sem_val = 0;   //信号灯编号为0
    semctl(semid, 0, SETVAL, mysemun);

    mysembuf.sem_num = 0;
    mysembuf.sem_flg = 0;
    //p操作
    mysembuf.sem_op = -1;
    semop(semid, &mysembuf, 1);
    for(i = 0; i < 10; i++) //后运行
    {
        usleep(100);
        printf("this is main fun i = %d\n", i);
    }
    //v操作
    /mysembuf.sem_op = 1;
    semop(semid, &mysembuf, 1);
    while(1);
    return 0;
}
```

随笔

进程调度、内存管理、文件系统、IO调度

数据处理：面向对象思想、数据库设计、多线程