

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），（对象为队列）

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）

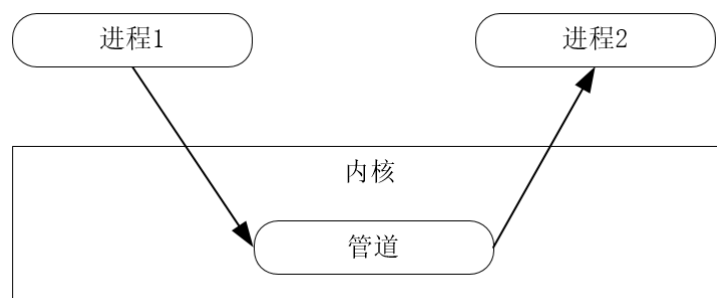
每一种通信方式都是基于文件IO的思想

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

```

1 //example pipe function 管道创建
2 #include "unistd.h"
3 #include "stdio.h"
4 #include "stdlib.h"
5 int main()
6 {
7     int fd[2];
8     int ret;
9     ret = pipe(fd);
10    if(ret < 0)
11    {
12        printf("creat pipe failure\n");
13        return -1;
14    }
15    printf("creat pipe sucess fd[0] = %d, fd[1] = %d\n", fd[0], fd[1]);
16    return 0;
17 }
18 /* 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;
19    fd[1] = 4
20    */
21 //example pipe function 单进程管道读写
22 #include "unistd.h"
23 #include "stdio.h"
24 #include "stdlib.h"
25 #include "string.h" //调用函数memset函数
26 int main()
27 {
28     int fd[2];
29     int ret;
30     char writebuf[] = "hello linux"; //写缓存
31     char readbuf[128] = {0}; //读缓存
32     ret = pipe(fd);
33     if(ret < 0)
34     {
35         printf("creat pipe failure\n");
36         return -1;
37     }
38     printf("creat pipe sucess fd[0] = %d, fd[1] = %d\n", fd[0], fd[1]);
39     //写入管道
40     write(fd[1], writebuf, sizeof(writebuf)); //写哪里去 写什么 写多少个
41     //管道读出
42     read(fd[0], readbuf, 128); //从什么地方读，读到哪里去，读多少个
43     printf("readbuf = %s\n", readbuf);
44
45     /*验证管道读阻塞
46     memset(readbuf, 0, 128); //使用该函数需要包含头文件#include
47     "string.h"
48     read(fd[0], readbuf, 128); //即清除第一次读到的内容，进行第二次读
49     printf("second read after\n"); //测试是否能打印该语句 不能打印

```

```

50     */
51
52     close(fd[0]);
53     close(fd[1]);
54     return 0;
55 }
56

```

写阻塞

```

1  #include "unistd.h"
2  #include "stdio.h"
3  #include "stdlib.h"
4  #include "string.h" //调用函数memset函数
5  int main()
6  {
7      int fd[2];
8      int ret;
9      int i = 0;
10     char writebuf[] = "hello linux"; //写缓存
11     char readbuf[128] = {0}; //读缓存
12     ret = pipe(fd);
13     if(ret < 0)
14     {
15         printf("creat pipe failure\n");
16         return -1;
17     }
18     printf("creat pipe sucess fd[0] = %d, fd[1] = %d\n", fd[0], fd[1]);
19     //写入管道 写5500次 测试是否写满管道
20     while(i < 5500)
21     {
22         write(fd[1], writebuf, sizeof(writebuf)); //写哪里去 写什么 写多少个
23         i++;
24     }
25     printf("write pipe end\n"); //写满无法打印该语句
26
27     close(fd[0]);
28     close(fd[1]);
29     return 0;
30 }

```

```

1  //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
2  //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
3  #include "unistd.h"
4  #include "stdio.h"
5  #include "sys/types.h"
6  #include "stdlib.h"
7  int main()
8  {
9      pid_t pid;
10     int fd[2];
11     int ret;
12     char process_inter = 0;
13     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
14     if(ret < 0)

```

```

15     {
16         printf("creat pipe failure\n");
17         return -1;
18     }
19     printf("creat pipe sucess\n");
20
21     pid = fork();
22     if(pid == 0)    //子进程
23     {
24         int i = 0;
25         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
26         while(process_inter == 0);
27         for(i = 0; i < 5; i++)
28         {
29             printf("this is child process i = %d\n", i);
30             usleep(100);
31         }
32     }
33     if(pid > 0)    //父进程
34     {
35         int i = 0;
36         for(i = 0; i < 5; i++)
37         {
38             printf("this is parent process i = %d\n", i);
39             usleep(100);
40         }
41         process_inter = 1; //process_inter置一
42         sleep(5);          //睡眠5s
43         write(fd[1], &process_inter, 1); //往管道写（写端fd[1]），写什么内容，写
多少个
44     }
45     while(1);
46     return 0;
47 }

```

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

文件系统中存在文件节点/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 - `mknod`，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

```
1 //创建管道
2 #include "stdio.h"
3 #include "unistd.h"
4 #include "stdlib.h"
5 int main()
6 {
7     int ret;
8     ret = mkfifo("./myfifo", 0777);
9     if(ret < 0)
10    {
11        printf("creat myfifo failure\n");
12        return -1;
13    }
14    printf("creat myfifo sucess\n");
15    return 0;
16 }
17 /*****
18 *****/
19 //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
20 //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
21 /* first.c */
22 #include "unistd.h"
23 #include "stdio.h"
24 #include "sys/types.h"
25 #include "stdlib.h"
26 #include "fcntl.h"
27 int main()
28 {
29     int fd;
30     int i;
31     char processs_inter = 0; //通信标志变量
32     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
33     //时，内核空间自动生成一个管道
34     if(fd < 0)
35     {
36         printf("open myfifo failure\n");
37     }
```

```

37     printf("open myfifo sucess");
38     for(i = 0; i < 5; i++)
39     {
40         printf("this is first process i = %d\n", i);
41         usleep(100);
42     }
43     process_inter = 1;
44     sleep(5);    //睡眠5s,
45     write(fd, &process_inter, 1);
46     while(1);
47     return 0;
48 }
49
50 /* second.c */
51 #include "unistd.h"
52 #include "stdio.h"
53 #include "sys/types.h"
54 #include "stdlib.h"
55 #include "fcntl.h"
56 int main()
57 {
58     int fd;
59     int i;
60     char processs_inter = 0;    //通信标志变量
61     fd = open("./myfifo", O_RDONLY); //打开管道文件，打开方式为写操作
62     if(fd < 0)
63     {
64         printf("open myfifo failure\n");
65     }
66     printf("open myfifo sucess");
67     read(fd, &process_inter, 1);    //先读通信标志位
68     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
69     for(i = 0; i < 5; i++)
70     {
71         printf("this is second process i = %d\n", i);
72         usleep(100);
73     }
74     while(1);
75     return 0;
76 }
77 /* 命令一: gcc -o first first.c
78 * 命令二: gcc -o second second.c
79 * 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
80 * 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
81 */

```

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 | <code>#include <signal.h> #include <sys/types.h></code> |
| 函数原型 | <code>int kill(pid_t pid, int sig)</code> |
| 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 |

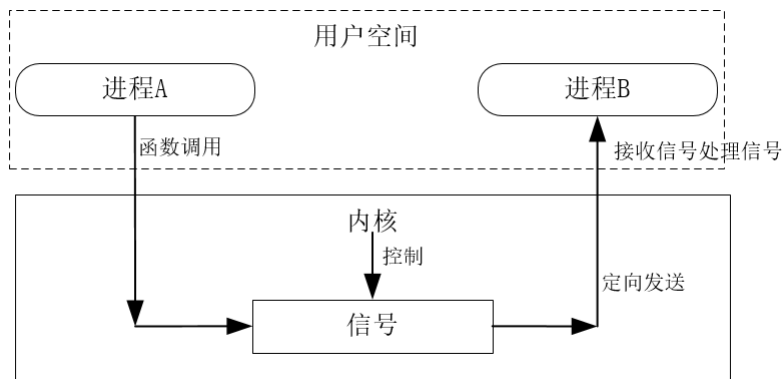
```
1 //kill 命令的实现
2 #include "sys/type.h"
3 #include "signal.h"
4 #include "unistd.h"
5 #include "stdio.h"
6 #include "stdlib.h"
7 int main(int argc, char *argv[])
8 {
9     int sig;    //信号值
10    int pid;    //进程pid号
11    if(argc < 3)
12    {
13        printf("please input param\n");
14        return -1;
15    }
16    sig = atoi(argv[1]);    //获取输入的信号值
17    pid = atoi(argv[2]);    //获取输入的进程pid
18    printf("sig = %d, pid = %d", sig, pid);
19    kill(pid, sig);        //调用kill函数杀死指定进程
20    return 0;
21 }
22 /* 查看当前系统进程命令: ps -axj*/
```

2.2信号通信框架

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

```
1 //raise 示例 进程无输出，直接被杀死
2 #include "sys/type.h"
3 #include "signal.h"
4 #include "stdio.h"
5 #include "stdlib.h"
6 int main()
7 {
8     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
9     raise(9);              //杀死进程自己
10    printf("raise after\n");
11    return 0;
12 }
13 //进程状态变化示例1
14 #include "sys/type.h"
15 #include "signal.h"
16 #include "stdio.h"
17 #include "stdlib.h"
18 int main()
19 {
20     pid_t pid;
21     pid = fork();
22     if(pid > 0) //父进程
23     {
24         sleep(5);
```



```

25     while(1);
26 }
27 if(pid == 0)    //子进程
28 {
29     printf("raise function before\n");
30     raise(SIGTSTP);    //发送SIGTSTP信号 进程状态变为暂停 T
31     printf("raise function after\n");
32     exit(0);
33 }
34 return 0
35 }
36 //进程状态变化示例2 父进程: S -> R 子进程: T -> Z
37 //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)
38 #include "sys/types.h"
39 #include "signal.h"
40 #include "stdio.h"
41 #include "stdlib.h"
42 int main()
43 {
44     pid_t pid;
45     pid = fork();
46     if(pid > 0)    //父进程
47     {
48         sleep(5);
49         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
50         {
51             kill(pid, 9);    //杀死进程
52         }
53         //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
54         while(1);    //死循环
55     }
56     if(pid == 0)    //子进程
57     {
58         printf("raise function before\n");
59         raise(SIGTSTP);    //发送SIGTSTP信号 进程状态变为暂停 T
60         printf("raise function after\n");
61         exit(0);
62     }
63     return 0
64 }
65 /* SIGTSTP: 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
66 *
67 */

```

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 |

```

1 //示例 当输出i = 8时, 当前进程终止
2 #include "sys/types.h"
3 #include "signal.h"
4 #include "stdio.h"
5 #include "stdlib.h"
6 int main()
7 {
8     int i;
9     printf("alarm before\n");
10    alarm(9); //定时9s,
11    printf("alarm after\n");
12    //保证当前进程在收到alarm信号前还未终止
13    while(i < 20)
14    {
15        i++;
16        sleep(1); //每次循环停1s
17        printf("process things, i = %d\n", i);
18    }
19    return 0
20 }
```

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

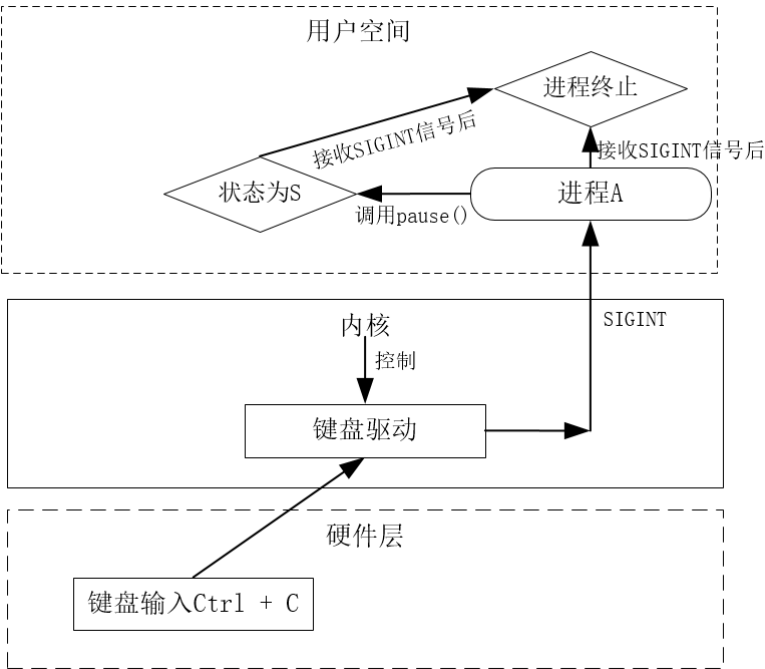
常用手段: sleep (S) 、 pause (将进程状态保持为S, 睡眠状态) 、 while(1)

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

```

1 //
2 #include "sys/types.h"
3 #include "signal.h"
4 #include "stdio.h"
5 #include "stdlib.h"
6 int main()
7 {
8     int i;
9     i = 0;
10    printf("pause befor\n");
11    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
12    printf("pause after\n");
13    while(i < 20)
14    {
15        i++;
16        sleep(1);
17        printf("process things, i = %d\n", i);
18    }
```

```
19     return 0;
20 }
21 /* 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
22  *
23  */
```



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 |

```
1 //函数分析signal
2 /* void(*signal(int signum, void(*handler)(int)))(int);
3  * 参数signum: 整型变量，信号值
4  * 参数handler: 函数指针，指向自定义的信号值处理函数
5  * 返回值: 返回一个函数指针
6  */
7 void(*signal(int signum, void(*handler)(int)))(int);
8 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
9
10 void(*signal(int signum,A))(int);
11 /* signal: 含有两个参数，
12  * 参数1: 信号值; 参数2: 函数指针
```

```

13  * 返回值：函数指针
14  * 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
15  */
16
17  /*****
18  *****/
19  //signal function example 示例 自定义信号处理函数
20  /* 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，
21  * 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
22  */
23  #include "sys/types.h"
24  #include "signal.h"
25  #include "stdio.h"
26  #include "stdlib.h"
27  void myfun(int signum)
28  {
29      int i;
30      i = 0;
31      while(i < 10)
32      {
33          printf("process signal signum = %d\n", signum); //打印10条语句
34          sleep(1);
35          i++;
36      }
37      return; //返回main函数
38  }
39  int main()
40  {
41      int i;
42      i = 0;
43      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）
44      printf("alarm before\n");
45      alarm(9);
46      printf("alarm after\n");
47      while(i < 20)
48      {
49          i++;
50          sleep(1); //睡眠1s
51          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
52      }
53      return 0;
54  }
55
56  /*****
57  *****/
58  //signal function example 示例 SIG_IGN、SIG_DFL
59  /* 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
60  * 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 =
61  * 20时结束进程
62  * 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

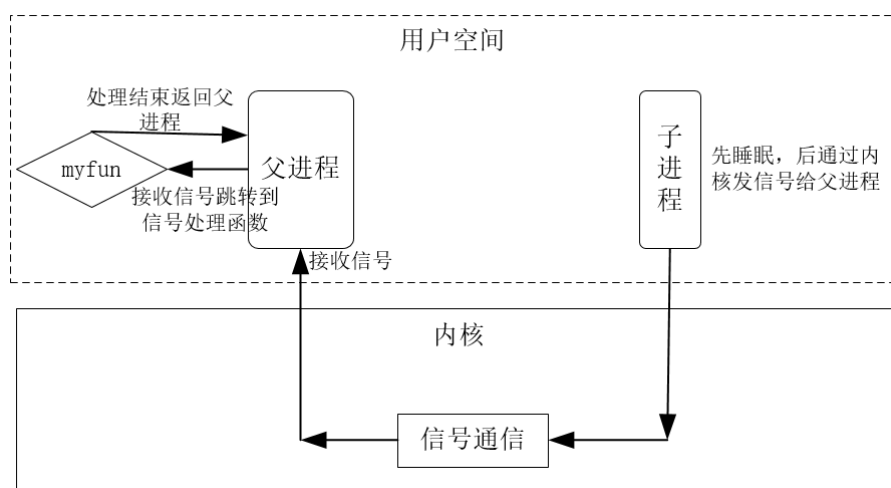
```

```

62  */
63  #include "sys/types.h"
64  #include "signal.h"
65  #include "stdio.h"
66  #include "stdlib.h"
67  void myfun(int signum)
68  {
69      int i;
70      i = 0;
71      while(i < 10)
72      {
73          printf("process signal signum = %d\n", signum); //打印10条语句
74          sleep(1);
75          i++;
76      }
77      return; //返回main函数
78  }
79  int main()
80  {
81      int i;
82      i = 0;
83      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）
84      printf("alarm before\n");
85      alarm(9);
86      printf("alarm after\n");
87      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
88      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zhong
89      while(i < 20)
90      {
91          i++;
92          sleep(1); //睡眠1s
93          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
94      }
95      return 0;
96  }

```

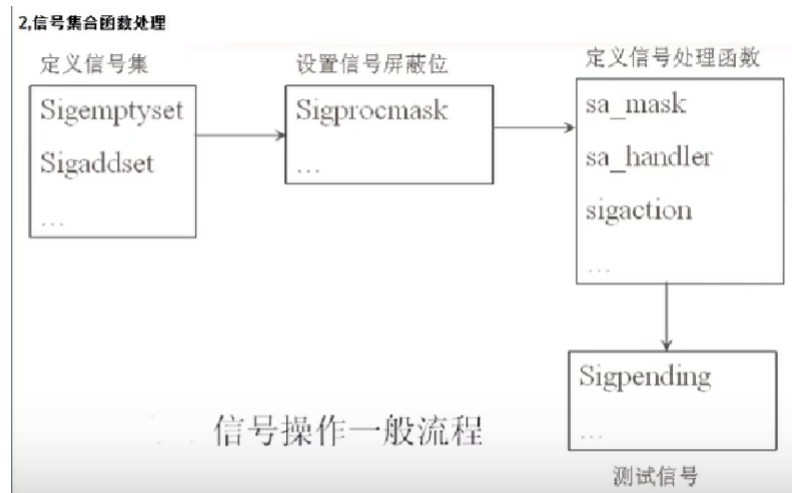
综合实例：程序执行框图如下，父子进程间信号通信



```

1 //函数执行输出流程:
2 //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;
3 //通过命令: ps -axj 查看得父进程状态为S+, 子进程不存在
4 #include "sys/types.h"
5 #include "signal.h"
6 #include "stdio.h"
7 #include "stdlib.h"
8 void myfun(int signum)
9 {
10     int i;
11     i = 0;
12     while(i < 5)
13     {
14         printf("receive signum = %d, i = %d\n", signum, i);
15         sleep(1); //睡眠1s
16         i++;
17     }
18     return; //退出
19 }
20 void myfun1(int signum)
21 {
22     printf("receive signum = %d\n", signum);
23     wait(NULL); //利用wait函数回收子进程资源, 防止子进程僵死
24     return; //退出返回
25 }
26 int main()
27 {
28     pid_t pid;
29     pid = fork();
30     if(pid > 0)
31     {
32         int i;
33         i = 0;
34         signal(10, myfun); //对10号信号值进行处理, 跳入到myfun函数中处理
35         signal(17, myfun1); //回收子进程资源, 防止僵死
36         while(1)
37         {
38             printf("parent process things, i = %d\n", i);
39             sleep(1);
40             i++;
41         }
42     }
43     if(pid == 0)
44     {
45         sleep(10); //子进程先睡眠10s
46         kill(getppid(), 10); //向父进程发10号信号值 (SIGUSR1)
47         sleep(10);
48         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);
49     }
50     return 0;
51 }
52 //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）

```
1 //shmget简单示例
2 #include "sys/types.h"
3 #include "sys/shm.h"
4 #include "signal.h"
5 #include "unistd.h"
6 #include "stdio.h"
7 #include "stdlib.h"
8 int main()
9 {
10     int shmid;
11     shmid = shmget(IPC_PRIVATE, 128, 0777);
12     if(shmid < 0)
13     {
14         printf("creat share memory failure\n");
15         return -1;
16     }
17     printf("creat share memory sucess shmid = %d\n", shmid);
18     system("ipcs -m"); //查看共享内存
19     system("ipcrm -m shmid"); //删除共享内存 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
20     return 0;
21 }
22
```

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

```
1 //shmget简单示例 利用ftok创建key值
2 #include "sys/types.h"
3 #include "sys/shm.h"
4 #include "signal.h"
5 #include "unistd.h"
6 #include "stdio.h"
7 #include "stdlib.h"
8 int main()
9 {
10     int shmid;
11     int key;
12     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
13     if(key < 0)
```



```

14 {
15     printf("creat key failure\n");
16     return -2;
17 }
18 printf("creat key sucess key = %X\n", key);
19 shmids = shmget(key, 128, IPC_CREAT | 0777);
20 if(shmids < 0)
21 {
22     printf("creat share memory failure\n");
23     return -1;
24 }
25 printf("creat share memory sucess shmids = %d\n", shmids);
26 system("ipcs -m"); //查看共享内存
27 system("ipcrm -m shmids"); //删除共享内存 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
28     return 0;
29 }

```

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

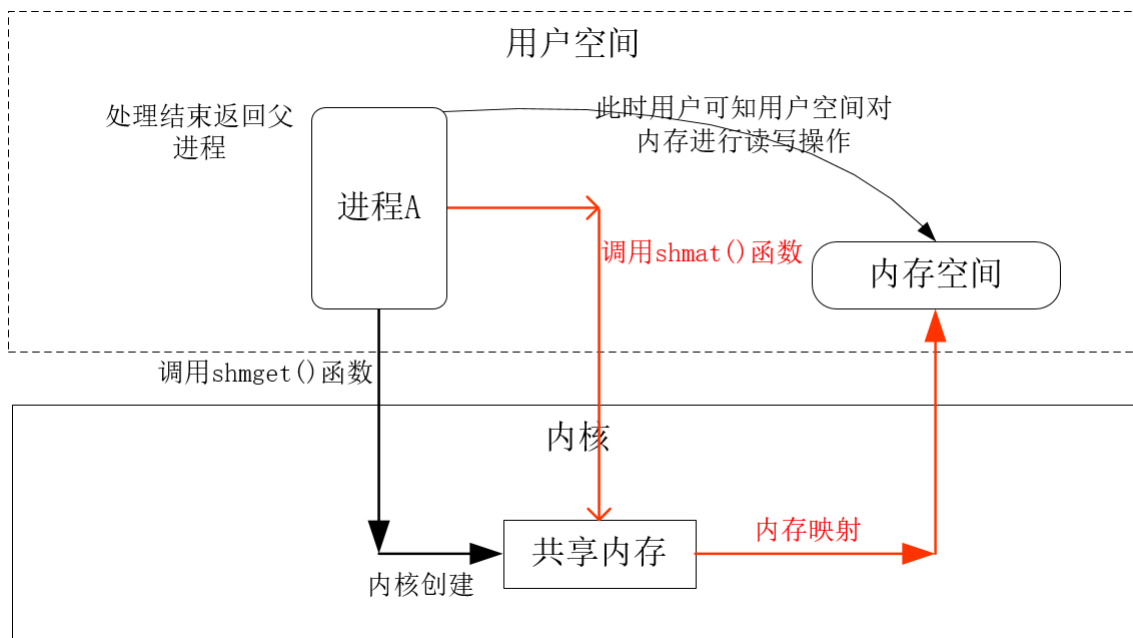
```
void *shmat(int shmids, const void *shmaddr, int shmflg);
```

参数 shmids: 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



```

1 //实例
2 #include "sys/type.h"
3 #include "sys/shm.h"
4 #include "signal.h"
5 #include "unistd.h"
6 #include "stdio.h"
7 #include "stdlib.h"
8 int main()

```

```

9  {
10     int shmid;
11     int key;
12     int *p;
13     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
14     if(key < 0)
15     {
16         printf("creat key failure\n");
17         return -2;
18     }
19     printf("creat key sucess key = %X\n", key);
20     shmid = shmget(key, 128, IPC_CREAT | 0777);
21     if(shmid < 0)
22     {
23         printf("creat share memory failure\n");
24         return -1;
25     }
26     printf("creat share memory sucess shmid = %d\n", shmid);
27     system("ipcs -m");           //查看共享内存
28     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
29     if(p == NULL)
30     {
31         printf("shmat function failure\n");
32         return -3;
33     }
34     //写共享内存
35     fgets(p, 128, stdin); //写到哪里去, 写多少个 从什么地方写
36     //读共享内存
37     printf("share memory data:%s", p);
38     printf("second read share memory data:%s", p); //第二次读仍然可以读到
39     //system("ipcrm -m shmid"); //删除共享内存 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
40     return 0;
41 }

```

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

```

1  //shmdt实例
2  #include "sys/type.h"
3  #include "sys/shm.h"
4  #include "signal.h"
5  #include "unistd.h"
6  #include "stdio.h"
7  #include "stdlib.h"

```

```

8  int main()
9  {
10     int shmid;
11     int key;
12     int *p;
13     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
14     if(key < 0)
15     {
16         printf("creat key failure\n");
17         return -2;
18     }
19     printf("creat key sucess key = %X\n", key);
20     shmid = shmget(key, 128, IPC_CREAT | 0777);
21     if(shmid < 0)
22     {
23         printf("creat share memory failure\n");
24         return -1;
25     }
26     printf("creat share memory sucess shmid = %d\n", shmid);
27     system("ipcs -m");           //查看共享内存
28     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
29     if(p == NULL)
30     {
31         printf("shmat function failure\n");
32         return -3;
33     }
34     //写共享内存
35     fgets(p, 128, stdin); //写到哪里去, 写多少个 从什么地方写
36     //读共享内存
37     printf("share memory data:%s", p);
38     printf("second read share memory data:%s", p); //第二次读仍然可以读到
39
40     shmdt(p); //释放(删除)内核共享内存映射到用户空间的内存
41     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
42     //system("ipcrm -m shmid"); //删除共享内存 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
43     return 0;
44 }
45
46
47 /*****
*****/
48 //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
49 #include "sys/types.h"
50 #include "sys/shm.h"
51 #include "signal.h"
52 #include "unistd.h"
53 #include "stdio.h"
54 #include "stdlib.h"
55 int main()
56 {
57     int shmid;
58     int key;
59     int *p;

```

```

60     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
61     if(key < 0)
62     {
63         printf("creat key failure\n");
64         return -2;
65     }
66     printf("creat key sucess key = %X\n", key);
67     shmid = shmget(key, 128, IPC_CREAT | 0777);
68     if(shmid < 0)
69     {
70         printf("creat share memory failure\n");
71         return -1;
72     }
73     printf("creat share memory sucess shmid = %d\n", shmid);
74     system("ipcs -m");           //查看共享内存
75     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
76     if(p == NULL)
77     {
78         printf("shmat function failure\n");
79         return -3;
80     }
81     //写共享内存
82     fgets(p, 128, stdin); //写到哪里去, 写多少个 从什么地方写 (键盘输入)
83     //读共享内存
84     printf("share memory data:%s", p);
85     printf("second read share memory data:%s", p); //第二次读仍然可以读到
86
87     shmdt(p); //释放 (删除) 内核共享内存映射到用户空间的内存
88     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
89     system("ipce -m"); //查看系统共享内存是否还存在
90     //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
91     return 0;
92 }

```

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存在创建父进程

```

1 //
2 /* getpid(): 获取当前父进程的pid值
3  *
4  */
5 #include "sys/type.h"
6 #include "sys/shm.h"
7 #include "signal.h"
8 #include "unistd.h"
9 #include "stdio.h"
10 #include "stdlib.h"
11 void myfun(int signum)

```

```

12 {
13     return;    //信号处理函数中不需要进行任何处理直接返回
14 }
15 int main()
16 {
17     int shmid;
18     int key;
19     int *p;
20     int pid;
21
22     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,
23     if(shmid < 0)
24     {
25         printf("creat share memory failure\n");
26         return -1;
27     }
28     printf("creat share memory sucess shmid = %d\n", shmid);
29     pid = fork();
30     if(pid > 0)    //父进程
31     {
32         signal(SIGUSR2, myfun);    //信号处理函数 处理子进程发过来的信号
33         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
34         if(p == NULL)
35         {
36             printf("parent process:shmat function failure\n");
37             return -3;
38         }
39         while(1)
40         {
41             //写共享内存
42             printf("parent process start write share memory:\n");
43             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)
44             kill(pid, SIGUSR1); //告诉子进程, 发一个信号值 告诉子进程去读数据
45             pause();    //进入等待
46         }
47     }
48     if(pid == 0)    //子进程
49     {
50         signal(SIGUSR1, myfun);    //处理父进程发过来的信号
51         //父进程需要地址映射, 子进程同样需要
52         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
53         if(p == NULL)
54         {
55             printf("parent process:shmat function failure\n");
56             return -3;
57         }
58         while(1)
59         {
60             pause();    //等待父进程写
61             //开始去读
62             printf("share memory data:%s\n", p);    //读父进程写进共享内存的数据
63             kill(getppid(), SIGUSR2);    //告诉父进程读完
64         }
65     }

```

```

66     shmdt(p);    //释放（删除）内核共享内存映射到用户空间的内存
67     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
68     system("ipce -m"); //查看系统共享内存是否还存在
69     //system("ipcrm -m shmid"); //删除共享内存 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
70     return 0;
71 }

```

无亲缘关系进程间共享内存单向通信

```

1  /* server -> client单向通信
2   * 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，
3   * 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
4   */
5   /*****
   *****/
6   //server.c
7   #include "sys/types.h"
8   #include "sys/shm.h"
9   #include "signal.h"
10  #include "unistd.h"
11  #include "stdio.h"
12  #include "stdlib.h"
13  #include "string.h"
14  struct mybuf{    //用于交换信息
15      int pid;      //占4字节
16      char buf[124]; //分配124字节 124 + 4刚好等于设定的共享内存的大小
17  };
18  void myfun(int signal)
19  {
20      return;    //信号处理函数中不需要进行任何处理直接返回
21  }
22  int main()
23  {
24      int shmid;
25      int key;
26      struct mybuf *p;
27      int pid;
28      key = ftok("./a.c", 'a'); //路径为当前目录下的a.c文件 字符为a
29      if(key < 0)
30      {
31          printf("creat key failure\n");
32          return -2;
33      }
34      printf("creat key success key = %X\n", key);
35
36      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，
37      if(shmid < 0)
38      {
39          printf("creat share memory failure\n");
40          return -1;
41      }
42      printf("creat share memory success shmid = %d\n", shmid);
43

```

```

44     signal(SIGUSR2, myfun);    //信号处理函数 处理客户端发过来的信号
45     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
46     if(p == NULL)
47     {
48         printf("parent process:shmat function failure\n");
49         return -3;
50     }
51     //读写前先交换pid 此为服务器，获取客户端pid
52     p -> pid = getpid();    //写入自身（服务器）pid到共享内存中
53     pause();                //停止，等待客户端读取共享内存的自身（服务器）pid
54     pid = p -> pid;         //读客户端pid，客户端将自身的pid写入共享内存
55
56     while(1)
57     {
58         //写共享内存
59         printf("parent process start write share memory:\n");
60         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）
61         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
62         pause();            //进入等待 等待客户端进程读
63     }
64
65     shmdt(p);    //释放（删除）内核共享内存映射到用户空间的内存
66     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
67     system("ipce -m"); //查看系统共享内存是否还存在
68     //system("ipcrm -m shmid"); //删除共享内存 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
69     return 0;
70 }
71
72 /*****
*****/
73 //client.c
74 #include "sys/types.h"
75 #include "sys/shm.h"
76 #include "signal.h"
77 #include "unistd.h"
78 #include "stdio.h"
79 #include "stdlib.h"
80 #include "string.h"
81 struct mybuf{    //用于交换信息
82     int pid;    //占4字节
83     char buf[124]; //分配124字节 124 + 4刚好等于设定的共享内存的大小
84 };
85 void myfun(int signum)
86 {
87     return;    //信号处理函数中不需要进行任何处理直接返回
88 }
89 int main()
90 {
91     int shmid;
92     int key;
93     struct mybuf *p;
94     int pid;
95     key = ftok("./a.c", 'a');    //路径为当前目录下的a.c文件 字符为a
96     if(key < 0)

```

```

97     {
98         printf("creat key failure\n");
99         return -2;
100    }
101    printf("creat key sucess key = %X\n", key);
102
103    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,
104    if(shmid < 0)
105    {
106        printf("creat share memory failure\n");
107        return -1;
108    }
109    printf("creat share memory sucess shmid = %d\n", shmid);
110
111    signal(SIGUARG, myfun); //信号处理函数 处理子进程发过来的信号
112    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
113    if(p == NULL)
114    {
115        printf("parent process:shmat function failure\n");
116        return -3;
117    }
118    //读写前先交换pid 此为客户端, 获取服务器pid
119    pid = p -> pid; //读共享内存中 读服务器写入的自身pid
120    p -> pid = getpid(); //写共享内存, 写入自身(客户端)的pid
121    kill(pid, SIGUSR2); //发送信号通知服务器获取完了pid 并写了客户端pid
122    pause(); //停止, 等待客户端读取共享内存的自身(服务器)pid
123    //开始在共享内存中持续读数据
124    while(1)
125    {
126        pause(); //进入等待 等待服务器写
127        printf("client process receive data from share memory:%s\n", p ->
    buf);
128        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)
129        kill(pid, SIGUSR2); //告诉服务器, 发一个信号值 告诉服务器可以继续写
130
131    }
132
133    shmdt(p); //释放(删除)内核共享内存映射到用户空间的内存
134    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
135    system("ipce -m"); //查看系统共享内存是否还存在
136    //system("ipcrm -m shmid"); //删除共享内存 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
137    return 0;
138 }

```

四、消息队列

与文件IO对比

| 文件I/O | 消息队列 |
|--------------------|---------------------|
| <code>open</code> | <code>msgget</code> |
| <code>read</code> | <code>msgrcv</code> |
| <code>write</code> | <code>msgsnd</code> |
| <code>close</code> | <code>msgctl</code> |

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 | <code>#include <sys/types.h></code> <code>#include <sys/ipc.h></code> <code>#include <sys/msg.h></code> |
| 函数原型 | <code>int msgget(key_t key, int flag);</code> |
| 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 | <code>#include <sys/types.h></code> <code>#include <sys/ipc.h></code> <code>#include <sys/msg.h></code> |
| 函数原型 | <code>int msgctl(int msgqid, int cmd, struct msqid_ds *buf);</code> |
| 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 |

```
1 //msgget、msgctl
2 #include "sys/type.h"
3 #include "sys/msg.h"
4 #include "signal.h"
5 #include "unistd.h"
6 #include "stdio.h"
7 #include "stdlib.h"
8 int main()
9 {
10     int msgid;
11     msgid = msgget(IPC_PRIVATE, 0777); //默认宏 key为0
12     if(msgid < 0)
13     {
14         printf("creat message queue failure\n");
```

```
15     return -1;
16 }
17 printf("creat message queue sucess msgid = %d\n", msgid);
18 system("ipcs -q"); //查看系统消息队列
19 //删除消息队列
20 msgctl(msgid, IPC_RMID, NULL);
21 system("ipcs, -q");
22 return 0;
23 }
```

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 |

```
1 //常用消息的结构体
2 struct msgbuf{
3     long mtype; //消息类型
4     char mtext[N]; //消息正文
5 };
```

实例:

```
1 //msgsnd 写
2 #include "sys/type.h"
3 #include "sys/msg.h"
4 #include "signal.h"
```

```

5  #include "unistd.h"
6  #include "stdio.h"
7  #include "stdlib.h"
8  #include "string.h"      //调用strlen
9  struct msgbuf{
10     long type;           //消息类型
11     char voltage[124];   //自定义
12     char ID[4];          //自定义
13 }
14 int main()
15 {
16     int msgid;
17     struct msgbuf sendbuf;
18     msgid = msgget(IPC_PRIVATE, 0777); //默认宏 key为0
19     if(msgid < 0)
20     {
21         printf("creat message queue failure\n");
22         return -1;
23     }
24     printf("creat message queue sucess msgid = %d\n", msgid);
25     system("ipcs -q"); //查看系统消息队列
26     //初始化消息对象
27     sendbuf.type = 100; //设定消息类型为100
28     printf("please input message:\n");
29     fgets(sendbuf.voltage, 124, stdin);
30     //开始写消息队列
31     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
32
33     while(1);
34     //删除消息队列
35     msgctl(msgid, IPC_RMID, NULL); //
36     system("ipcs, -q");
37     return 0;
38 }
39
40 //msgrcv 读
41 #include "sys/types.h"
42 #include "sys/msg.h"
43 #include "signal.h"
44 #include "unistd.h"
45 #include "stdio.h"
46 #include "stdlib.h"
47 #include "string.h"      //调用strlen
48 struct msgbuf{
49     long type;           //消息类型
50     char voltage[124];   //自定义
51     char ID[4];          //自定义
52 }
53 int main()
54 {
55     int msgid;
56     int readret;         //接收读返回值
57     struct msgbuf sendbuf, recvbuf; //定义发送，接收缓存
58     msgid = msgget(IPC_PRIVATE, 0777); //默认宏 key为0
59     if(msgid < 0)
60     {
61         printf("creat message queue failure\n");

```

```

62         return -1;
63     }
64     printf("creat message queue sucess msgid = %d\n", msgid);
65     system("ipcs -q"); //查看系统消息队列
66     //初始化消息对象
67     sendbuf.type = 100; //设定消息类型为100
68     printf("please input message:\n");
69     fgets(sendbuf.voltage, 124, stdin);
70     //开始写消息队列
71     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
72
73     //开始从消息队列中读消息
74     memset(recv.voltage, 0, 124); //清空接收缓存区
75     readret = msgrcv(msgid, (void *)&recvbuf, 124, 100, 0); //读
76     printf("recv:%s\n", recvbuf.voltage);
77     printf("readret = %d\n", readret);
78     /*
79     //第二次读消息队列测试
80     readret = msgrcv(msgid, (void *)&recvbuf, 124, 100, 0); //且阻塞
81     printf("send read after\n"); //无法打印
82     */
83     //删除消息队列
84     msgctl(msgid, IPC_RMID, NULL); //
85     system("ipcs, -q");
86     return 0;
87 }

```

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

```

1 //写消息进消息队列进程
2 #include "sys/types.h"
3 #include "sys/msg.h"
4 #include "signal.h"
5 #include "unistd.h"
6 #include "stdio.h"
7 #include "stdlib.h"
8 #include "string.h" //调用strlen
9 struct msgbuf{
10     long type; //消息类型
11     char voltage[124]; //自定义
12     char ID[4]; //自定义
13 }
14 int main()
15 {
16     int msgid;
17     int readret; //接收读返回值
18     int key;
19     struct msgbuf sendbuf, recvbuf; //定义发送，接收缓存
20     key = ftok("./a.c", 'a'); //创建key
21     if(key < 0)

```

```

22     {
23         printf("creat key failure\n");
24         return -2;
25     }
26     msgid = msgget(key, IPC_CREAT | 0777); //默认宏 key为0
27     if(msgid < 0)
28     {
29         printf("creat message queue failure\n");
30         return -1;
31     }
32     printf("creat message queue sucess msgid = %d\n", msgid);
33     system("ipcs -q"); //查看系统消息队列
34     sendbuf.type = 100; //设定消息类型为100
35     //写消息队列
36     while(1)
37     {
38         memset(sendbuf.voltage, 0, 124); //清除读缓存区
39         printf("please input message:\n");
40         fgets(sendbuf.voltage, 124, stdin);
41         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
42     }
43     //删除消息队列
44     msgctl(msgid, IPC_RMID, NULL);
45     system("ipcs, -q");
46     return 0;
47 }
48
49 /*****
*****/
50 //从消息队列读消息进程
51 #include "sys/types.h"
52 #include "sys/msg.h"
53 #include "signal.h"
54 #include "unistd.h"
55 #include "stdio.h"
56 #include "stdlib.h"
57 #include "string.h" //调用strlen
58 struct msgbuf{
59     long type; //消息类型
60     char voltage[124]; //自定义
61     char ID[4]; //自定义
62 }
63 int main()
64 {
65     int msgid;
66     int readret; //接收读返回值
67     int key;
68     struct msgbuf sendbuf, recvbuf; //定义发送，接收缓存
69     key = ftok("./a.c", 'a'); //创建key
70     if(key < 0)
71     {
72         printf("creat key failure\n");
73         return -2;
74     }
75     msgid = msgget(key, IPC_CREAT | 0777); //默认宏 key为0
76     if(msgid < 0)
77     {

```

```

78     printf("creat message queue failure\n");
79     return -1;
80 }
81 printf("creat message queue sucess msgid = %d\n", msgid);
82 system("ipcs -q"); //查看系统消息队列
83
84 //读消息队列
85 while(1)
86 {
87     memset(recvbuf.voltage, 0, 124); //清除接收缓存区
88     //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
89     msgrcv(msgid, (void *)&recvbuf, 124, 100, 0);
90     printf("receive data from message queue:%s", recebuf.voltage);
91 }
92 //删除消息队列
93 msgctl(msgid, IPC_RMID, NULL);
94 system("ipcs, -q");
95 return 0;
96 }

```

实例2 双向通信

同一个消息队列，无亲缘关系进程间读写

```

1 //server.c 服务器端
2 #include "sys/types.h"
3 #include "sys/msg.h"
4 #include "signal.h"
5 #include "unistd.h"
6 #include "stdio.h"
7 #include "stdlib.h"
8 #include "string.h" //调用strlen
9 struct msgbuf{
10     long type; //消息类型
11     char voltage[124]; //自定义
12     char ID[4]; //自定义
13 }
14 int main()
15 {
16     int msgid;
17     int readret; //接收读返回值
18     int key;
19     int pid; //为实现同时收发功能，通过父子进程实现
20
21     struct msgbuf sendbuf, recvbuf; //定义发送，接收缓存
22     key = ftok("./a.c", 'a'); //创建key
23     if(key < 0)
24     {
25         printf("creat key failure\n");
26         return -2;
27     }
28     msgid = msgget(key, IPC_CREAT | 0777); //默认宏 key为0
29     if(msgid < 0)
30     {
31         printf("creat message queue failure\n");
32         return -1;
33     }

```

```

34     printf("creat message queue sucess msgid = %d\n", msgid);
35     system("ipcs -q"); //查看系统消息队列
36     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
37     if(pid > 0) //父进程 负责写 写的消息类型是100
38     {
39         sendbuf.type = 100; //设定消息类型为100
40         //写消息队列
41         while(1)
42         {
43             memset(sendbuf.voltage, 0, 124); //清除发送缓存区
44             printf("please input message:\n");
45             fgets(sendbuf.voltage, 124, stdin);
46             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
47         }
48     }
49     if(pid == 0) //子进程 负责读 读的消息类型为200
50     {
51         while(1)
52         {
53             memset(recvbuf.voltage, 0, 124); //清除读缓存区
54             msgcrv(msgid, (void *)&recvbuf, 124, 200, 0);
55             printf("receive message from message queue:%s\n",
    recvbuf.voltage);
56         }
57     }
58
59     //删除消息队列
60     msgctl(msgid, IPC_RMID, NULL);
61     system("ipcs, -q");
62     return 0;
63 }
64
65 /*****
    *****/
66 //client.c
67 #include "sys/types.h"
68 #include "sys/msg.h"
69 #include "signal.h"
70 #include "unistd.h"
71 #include "stdio.h"
72 #include "stdlib.h"
73 #include "string.h" //调用strlen
74 struct msgbuf{
75     long type; //消息类型
76     char voltage[124]; //自定义
77     char ID[4]; //自定义
78 }
79 int main()
80 {
81     int msgid;
82     int readret; //接收读返回值
83     int key;
84     int pid; //为实现同时收发功能, 通过父子进程实现
85
86     struct msgbuf sendbuf, recvbuf; //定义发送, 接收缓存
87     key = ftok("./a.c", 'a'); //创建key

```

```

88     if(key < 0)
89     {
90         printf("creat key failure\n");
91         return -2;
92     }
93     msgid = msgget(key, IPC_CREAT | 0777); //默认宏 key为0
94     if(msgid < 0)
95     {
96         printf("creat message queue failure\n");
97         return -1;
98     }
99     printf("creat message queue sucess msgid = %d\n", msgid);
100    system("ipcs -q"); //查看系统消息队列
101    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
102    if(pid == 0) //子进程 负责写 写的消息类型是200
103    {
104        sendbuf.type = 200; //设定消息类型为200
105        //写消息队列
106        while(1)
107        {
108            memset(sendbuf.voltage, 0, 124); //清除发送缓存区
109            printf("please input message:\n");
110            fgets(sendbuf.voltage, 124, stdin);
111            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
112        }
113    }
114    if(pid > 0) //父进程 负责读 读的消息类型为100
115    {
116        while(1)
117        {
118            memset(recvbuf.voltage, 0, 124); //清除读缓存区
119            msgcrv(msgid, (void *)&recvbuf, 124, 100, 0);
120            printf("receive message from message queue:%s\n",
    recvbuf.voltage);
121        }
122    }
123
124    //删除消息队列
125    msgctl(msgid, IPC_RMID, NULL);
126    system("ipcs, -q");
127    return 0;
128 }

```

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 |

```

1 //设置信号灯值的共用体union
2 union semun{
3     int val;    //设置信号灯的值
4     struct semid_ds *buf;    //获取/设置对象属性
5     unsigned short *array;
6     struct seminfo *__buf;
7 }

```

```

1 //semget、semctl
2 #include "sys/type.h"
3 #include "sys/sem.h"
4 #include "signal.h"
5 #include "unistd.h"
6 #include "stdio.h"
7 #include "stdlib.h"
8 int main()
9 {
10     int semgid;
11     semgid = semget(IPC_PRIVATE, 3, 0777); //默认宏 key为0
12     if(semgid < 0)
13     {
14         printf("creat semaphore failure\n");
15         return -1;
16     }
17     printf("creat semaphore sucess msgid = %d\n", semgid);
18     system("ipcs -s"); //查看系统信号灯
19     while(1);
20     return 0;
21 }

```

```

22
23 //semctl 实现删除一个信号灯
24 #include "sys/types.h"
25 #include "sys/sem.h"
26 #include "signal.h"
27 #include "unistd.h"
28 #include "stdio.h"
29 #include "stdlib.h"
30 int main()
31 {
32     int semgid;
33     semgid = semget(IPC_PRIVATE, 3, 0777); //默认宏 key为0
34     if(semgid < 0)
35     {
36         printf("creat semaphore failure\n");
37         return -1;
38     }
39     printf("creat semaphore sucess msgid = %d\n", semgid);
40     system("ipcs -s"); //查看系统信号灯
41     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
42     system("ipcs -s");
43     return 0;
44 }
45

```

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 *ops, 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号 short sem_op; //0, 等待, 直到信号灯的值为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 |

```

1 //通过信号量确定每次主线程先运行，然后到子线程运行
2 #include "stdio.h"
3 #include "stdlib.h"
4 #include "pthread.h"
5 #include "semaphore.h"
6 sem_t sem; //定义信号量
7 void *fun(void *var) //子线程
8 {
9     int j;
10    //p操作，等待
11    sem_wait(sem); //进行等待
12    for(j = 0; j < 10; j++)
13    {

```

```

14     usleep(100);
15     printf("this is fun j = %d\n", j);
16 }
17 }
18 int main() //主线程
19 {
20     int i;
21     char str[] = "hello linux\n";
22     pthread_t tid;
23     int ret;
24     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
25     ret = pthread_create(&tid, NULL, fun, (void *)str);
26     if(ret < 0)
27     {
28         printf("creat thread failure\n");
29         return -1;
30     }
31     for(i = 0; i < 10; i++)
32     {
33         usleep(100);
34         printf("this is main fun i = %d\n", i);
35     }
36     //v操作
37     sem_post(&sem);
38     while(1);
39     return 0;
40 }
41 /*****
42 *****/
43 //信号灯实现控制 主线程先运行后子线程后运行
44 /*****
45 *****/
46 #include "stdio.h"
47 #include "stdlib.h"
48 #include "pthread.h"
49 #include "sys/ipc.h"
50 #include "sys/sem.h"
51 union semun{
52     int val; //设置信号灯的值
53     struct semid_ds *buf; //获取/设置对象属性
54     unsigned short *array;
55     struct seminfo *__buf;
56 }
57 int semid; //定义信号灯集
58 union semun mysemun;
59 struct sembuf mysembuf;
60 void *fun(void *var) //子线程
61 {
62     int j;
63     //p操作, 等待 信号灯的p操作
64     mysembuf.sem_op = -1;
65     semop(semid, &mysembuf, 1);
66     for(j = 0; j < 10; j++)
67     {
68         usleep(100);
69         printf("this is fun j = %d\n", j);
70     }

```

```

69 }
70 int main() //主线程
71 {
72     int i;
73     char str[] = "hello linux\n";
74     pthread_t tid;
75     int ret;
76     semid = semget(IPC_PRIVATE, 3, 0777); //创建信号灯对象
77     if(semid < 0)
78     {
79         printf("creat semaphore failure\n");
80         return -1;
81     }
82     printf("creat semaphore sucess, semid = %d\n", semid);
83     system("ipcs -s"); //查看系统创建的信号灯
84     //信号灯初始化
85     mysembuf.sem_num = 0; //信号灯编号为0
86     mysembuf.sem_flg = 0
87     ret = pthread_create(&tid, NULL, fun, (void *)str);
88     if(ret < 0)
89     {
90         printf("creat thread failure\n");
91         return -1;
92     }
93     for(i = 0; i < 10; i++)
94     {
95         usleep(100);
96         printf("this is main fun i = %d\n", i);
97     }
98     //v操作
99     mysembuf.sem_op = 1;
100     semop(semid, &mysembuf, 1);
101     while(1);
102     return 0;
103 }

```

实例 无亲缘关系之间信号灯通信

```

1 //server.c 输出10条语句
2 #include "stdio.h"
3 #include "stdlib.h"
4 #include "pthread.h"
5 #include "sys/ipc.h"
6 #include "sys/sem.h"
7 union semun{
8     int val; //设置信号灯的值
9     struct semid_ds *buf; //获取/设置对象属性
10    unsigned short *array;
11    struct seminfo *__buf;
12 }
13 int semid; //定义信号灯集
14 union semun mysemun;
15 struct sembuf mysembuf;
16 int main() //主线程
17 {
18     int i;
19     int key;

```

```

20     key = ftok("./a.c", 'a');
21     if(key < 0)
22     {
23         printf("creat key failure\n");
24         return -1;
25     }
26     printf("creat key sucess\n");
27     semid = semget(key, 3, IPC_CREAT | 0777);    //创建信号灯对象
28     if(semid < 0)
29     {
30         printf("creat semaphore failure\n");
31         return -2;
32     }
33     printf("creat semaphore sucess, semid = %d\n", semid);
34     system("ipcs -s");    //查看系统创建的信号灯
35     //信号灯初始化
36     // mysembuf.sem_val = 0; //信号灯编号为0
37     // semctl(semid, 0, SETVAL, mysemun);
38
39     mysembuf.sem_num = 0;
40     mysembuf.sem_flg = 0;
41     for(i = 0; i < 10; i++)
42     {
43         usleep(100);
44         printf("this is main fun i = %d\n", i);
45     }
46     //v操作
47     mysembuf.sem_op = 1;
48     semop(semid, &mysembuf, 1);
49     while(1);
50     return 0;
51 }
52 /*****
53 *****/
54 //client.c 输出10条语句
55 #include "stdio.h"
56 #include "stdlib.h"
57 #include "pthread.h"
58 #include "sys/ipc.h"
59 #include "sys/sem.h"
60 union semun{
61     int val;    //设置信号灯的值
62     struct semid_ds *buf;    //获取/设置对象属性
63     unsigned short *array;
64     struct seminfo *__buf;
65 }
66 int semid;    //定义信号灯集
67 union semun mysemun;
68 struct sembuf mysembuf;
69 int main()    //主线程
70 {
71     int i;
72     int key;
73     key = ftok("./a.c", 'a');
74     if(key < 0)
75     {
76         printf("creat key failure\n");
77         return -1;

```

```

77     }
78     printf("creat key sucess\n");
79     semid = semget(key, 3, IPC_CREAT | 0777);    //创建信号灯对象
80     if(semid < 0)
81     {
82         printf("creat semaphore failure\n");
83         return -2;
84     }
85     printf("creat semaphore sucess, semid = %d\n", semid);
86     system("ipcs -s");    //查看系统创建的信号灯
87     //信号灯初始化
88     mysembuf.sem_val = 0;    //信号灯编号为0
89     semctl(semid, 0, SETVAL, mysemun);
90
91     mysembuf.sem_num = 0;
92     mysembuf.sem_flg = 0;
93     //p操作
94     mysembuf.sem_op = -1;
95     semop(semid, &mysembuf, 1);
96     for(i = 0; i < 10; i++)    //后运行
97     {
98         usleep(100);
99         printf("this is main fun i = %d\n", i);
100    }
101    //v操作
102    /mysembuf.sem_op = 1;
103    semop(semid, &mysembuf, 1);
104    while(1);
105    return 0;
106 }

```

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