西南交通大學

嵌入式系统及应用

西南交通大学

电气工程学院

孙永奎博士

E-mail: yksun@swjtu.edu.cn

第7章 Linux系统进程与线程

- 7.1 Linux 的进程
- 7.2 进程间通信
- 7.3 Linux 的线程
- 7.4 线程的互斥与同步
- 3学时

7.1 Linux的进程

- 进程的概念
  - 进程是一个独立的可调度的活动
  - 进程是一个抽象实体，当它执行某个任务时，要分配和释放各种资源
  - 进程是可以并行执行的计算单位
  - 进程是一个程序的一次执行的过程，同时也是资源分配的最小单元
- 进程的特点
  - 并发性、动态性、交互性
  - 立性、异步性

7.1 Linux的进程

进程与程序

- 由于一个进程对应一个程序的执行，但进程不等同于程序。因为程序是静态的概念，进程是动态的概念。
- 进程是程序执行的过程，包括了动态创建、调度和消亡的整个过程。进程是程序执行和资源管理的最小单位。
- 对系统而言，当用户在各级系统中键入命令执行一个程序的时候，它将启动一个进程，因此，一个程序可以对应多个进程。

7.1 Linux的进程

- 进程的管理
  - Linux环境下的进程管理包括启动进程和调度进程。

- 1）启动进程
  - ✓ 手工启动
  - ✓ 调度启动

- 手工启动
  - 前台启动
  - 后台启动

7.1 Linux的进程

- 前台启动：
  - 手工启动一个进程的最常用方式。一般地，当用户输入一个命令时，就已经启动了一个进程，并且是一个前台的进程。
- 后台启动：
  - 往往是在该进程非常耗时，且用户也不急着需要结果的时候启动。一般地，当用户输入一个命令结尾加上一个“&”号，就是后台启动一个进程。

7.1 Linux的进程

- 2）调度进程
  - 进程的中断操作、改变优先级、查看进程状态等。
- Linux环境下常见的进程调用命令

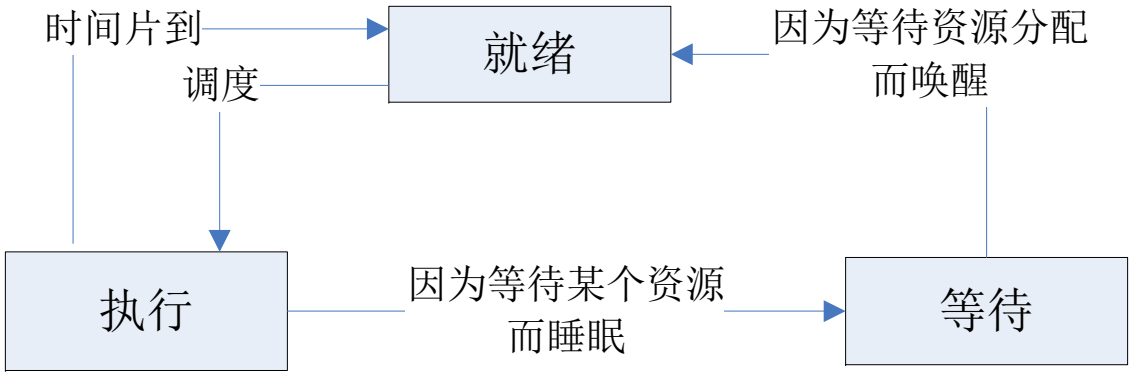
Linux环境下常见的进程调用命令

命 令	作 用
ps	查看系统中的进程
top	动态显示系统中的进程
nice	按用户指定的优先级运行
renice	改变正在运行进程的优先级
kill	终止进程（包括后台进程）
crontabl	用于安装、删除或者列出用于驱动 cron 后台进程的任务
bg	将挂起的进程放到后台执行

7.1 Linux的进程

❖进程的状态

- 根据它的生命周期可以划分成3种状态。
- 执行态：该进程正在运行，即进程正在占用CPU。
- 就绪态：进程已经具备执行的一切条件，正在等待分配CPU的处理时间片。
- 等待态：进程不能使用CPU，若等待事件发生（等待的资源分配到）则可将其唤醒



进程的状态转换

7.1 Linux的进程

- 进程的创建
  - 在Linux中创建一个新进程的方法是使用fork()函数
- 进程标识符(PID）
  - Linux环境下在进程启动时，系统会分配一个唯一的数值给每个进程
- 父子进程
  - fork()函数用于从已存在的进程中创建一个新进程。新进程称为子进程，而原进程称为父进程。
  - 子进程是父进程的一个复制品，
  - 进程标识有进程号（PID）和它的父进程号（PPID）

7.1 Linux的进程

- PID惟一地标识一个进程。
- PID和PPID都是非零的正整数。
- 在Linux中获得当前进程的PID和PPID的系统调用函数为getpid和getppid函数。

所需头文件	#include <sys/types.h> // 提供类型 pid_t 的定义 #include <unistd.h>
函数原型	pid_t fork(void)
函数返回值	0：子进程
	子进程 ID（大于 0 的整数）：父进程
	-1：出错

7.1 Linux的进程

- 例7-1：设计一个程序，要求显示Linux系统分配给此程序的进程号（PID）和它的父进程号（PPID）。
- 源程序代码:

```
/*6-1.c 程序：显示 Linux 系统分配的进程号（PID）和它的父进程号（PPID）*/
#include<stdio.h>                          /*文件预处理，包含标准输入输出库*/
#include<unistd.h>                         /*文件预处理，包含进程控制函数库*/
int main ()                                /*C 程序的主函数，开始入口*/
{
    printf("系统分配的进程号(PID)是：%d\n",getpid());      /*显示输出进程号*/
    printf("系统分配的父进程号(PPID)是：%d\n",getppid());  /*显示输出父进程号*/
    return 0;
}
```

多次运行例7.1的程序，每一次运行的结果PID值都是不一样的，所以说PID是惟一地标识一个进程。

❖getpid函数说明

所需头文件	#include<unistd.h>
函数功能	取得当前进程的进程号
函数原型	Pid_t getpid(void);
函数传入值	无
函数返回值	执行成功则返回当前进程的进程标识符。
相关函数	fork，kill，getppid
备注	

❖getppid函数说明

所需头文件	#include<unistd.h>
函数功能	取得当前进程的父进程号
函数原型	Pid_t getpid(void);
函数传入值	无
函数返回值	执行成功则返回当前进程的父进程标识符。
相关函数	fork，kill，getpid
备注	

7.1 Linux的进程

- Linux 与进程相关的主要函数

函数名	函数功能
getpid	取得当前进程的进程号
getppid	取得当前进程的父进程号
fork	从已存在进程中创建一个新进程
exec 函数族	在进程中启动另一个程序执行
system	在进程中开始另一个进程
sleep	让进程暂停执行一段时间
exit	用来终止进程
_exit	用来终止进程
wait	暂停父进程，等待子进程运行完成
waitpid	暂停父进程，等待子进程运行完成

7.1 Linux的进程

进程的创建- fork()函数

- 使用fork()函数，创建两个一模一样的进程，通过fork的返回值来区分父进程和子进程，错误返回-1，对于父进程返回子进程的ID，对于子进程返回0.

```
void main(){
        int i;
        if((child=fork())==-1)
        {           printf(“Fork error\n”);
                    exit(1);
}
else if ( fork() == 0 ) {
                     /* 子进程程序 */
                    for ( i = 1; i < 5; i ++ )
                    printf("This is child process\n");
                    exit(1)
         } else {
                    /* 父进程程序*/
                     for ( i = 1; i < 5; i ++ )
                              printf("This is process process\n");
        }
}
```

7.1 Linux的进程

进程的创建- exec函数族

- 进程如何来启动另一个程序的执行。在Linux中要使用exec类的函数，exec类的函数不止一个，但大致相同，在Linux中，它们分别是：execl，execlp，execle，execv，execve和execvp.

```
void main()
  {
  int rtn; /*子进程的返回数值*/
  while(1) {
        /* 从终端读取要执行的命令 */
        printf( ">" );
        fgets( command, 256, stdin );
        command[strlen(command)-1] = 0;
        if ( fork() == 0 ) {
              /* 子进程执行此命令 */
              execlp( command, command );
              /* 如果exec函数返回，表明没有正常执行命令，打印错误信息*/
              perror( command );
              exit( errorno );
        }
        else {
              /* 父进程， 等待子进程结束，并打印子进程的返回值 */
              wait ( &rtn );
              printf( " child process return %d\n",. rtn );
        }
  }}
```

7.1 Linux的进程

- 实际上，在Linux中并没有exec函数，而是有6个以exec开头的函数族。
- exec函数族的6个成员函数的语法

所需头文件	#include<unistd.h>
函数原型	int execl(con…
	int execv(cor…
	int execle(co…
	int execve(co…
	int execlp(const char *file,const char *arg,…)
	int execvp(const char *file,,char *const argv[ ])
函数返回值	-1：出错

事实上，这6个函数中真正的系统调用只有execve，其他5个都是库函数，它们最终都会调用execve这个系统调用。

7.1 Linux的进程

❖ 进程的创建- system函数

- system函数是一个和操作系统紧密相关的函数。用户可以使用它在自己的程序中调用系统提供的各种命令。
- 使用时不需要预处理头文件“unistd.h”。

```
#include<stdio.h>                    /*文件预处理，包含标准输入输出库*/
#include<stdlib.h>                   /*文件预处理，包含 system 函数库*/
int main ()                          /*C 程序的主函数，开始入口*/
{
    int newret;
    printf("系统分配的进程号(PID)是: %d\n",getpid());      /*显示输出进程号*/
    newret=system("ping www.lupaworld.com");        /*调用 ping 程序，创建新进程*/
    return 0;
}
```

7.1 Linux的进程

❖system函数说明

所需头文件	**#include<stdlib.h>**
函数功能	在进程中开始另一个进程
函数原型	int system(const char *string);
函数传入值	系统变量
函数返回值	执行成功则返回执行 shell 命令后的返回值，调用/bin/sh 失败则返回 127，其它失败原因则返回-1，参数 string 为空(NULL)，则返回非零值。
相关函数	fork，execve，waitpid，popen
备注	system()调用 fork()产生子进程，子进程调用/bin/sh -c string 来执行参数 string 字符串所代表的命令，此命令执行完后随即返回原调用的进程。如果调用成功，返回 shell 命令后的返回值也可能是 127，因此，最好能检查 errno 来确定。

7.1 Linux的进程

❖ 进程创建- sleep函数

◆ sleep函数说明

所需头文件	#include<unistd.h>
函数功能 函数原型 函数传入值 函数返回值 相关函数	让进程暂停执行一段时间 unsigned int sleep(unsigned int seconds); seconds 暂停时间，单位：秒。 执行成功则在返回 0，失败则返回剩余秒数。 signal，alarm
备注	sleep()会令目前的进程暂停(进入睡眠状态)，直到达到参数 seconds 所指定的时间，或是被信号所中断。

7.1 Linux的进程

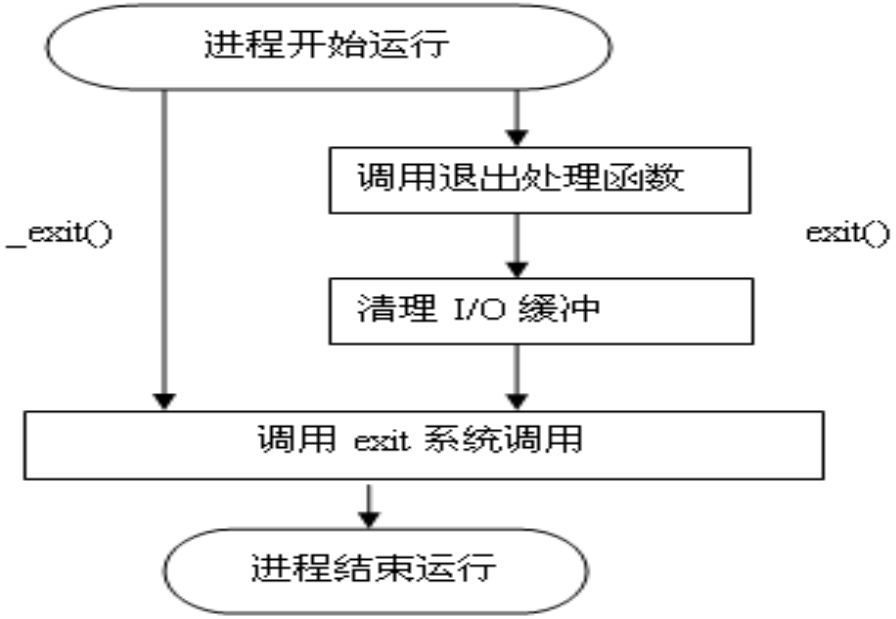
```
/*6-5.c 程序：显示当前目录下的文件信息，并测试网络连通状况*/
#include<stdio.h>                    /*文件预处理，包含标准输入输出库*/
#include<unistd.h>                   /*文件预处理，包含 fork 函数库*/
#include<sys/types.h>                /*文件预处理，包含 fork 函数库*/
int main ()                          /*C 程序的主函数，开始入口*/
{
    pid_t result;
    result=fork();                   /*调用 fork 函数，返回值存在变量 result 中*/
    if(result==-1) /*通过 result 的值来判断 fork 函数的返回情况，这儿先进行出错处理*/
    {
        perror("创建子进程失败");
        exit(0);
    }
    else if (result==0)              /*返回值为 0 代表子进程*/
    {
        printf("测试终止进程的_exit 函数!\n");
        printf("这一行我们用缓存!");
        _exit(0);
    }
    else                             /*返回值大于 0
    {
        printf("测试终止进程的 exit 函数!\n");
        printf("这一行我们用缓存!");
        exit(0);
    }
}
```

观察结果可以看出，调用exit函数时，缓冲区中的记录能正常输出；而调用_exit时，缓冲区中的记录无法输出。

7.1 Linux的进程

❖ 进程终止

- _exit()函数作用：直接使进程停止运行,清除其使用的内存空间,并清除其在内核中的各种数据结构；
- exit()函数则在执行退出之前加了若干道工序，exit函数在调用exit系统之前要查看文件的打开情况，把文件缓冲区中的内容写回文件。



exit 和_exit 函数的区别

7.1 Linux的进程

exit函数说明

所需头文件	#include<stdlib.h>
函数功能	正常终止进程
函数原型	void exit(int status);
函数传入值	整数 status
函数返回值	无
相关函数	_exit，atexit，on_exit
备注	exit()用来正常终止目前进程的执行，并把参数 status 返回给父进程，而进程所有的缓冲区数据会自动写回并关闭未关闭的文件。

_exit函数说明

所需头文件	#include<stdlib.h>
函数功能	终止进程执行
函数原型	void _exit(int status);
函数传入值	整数 status
函数返回值	无
相关函数	exit，wait，abort
备注	exit()用来立刻终止目前进程的执行，并把参数 status 返回给父进程，并关闭未关闭的文件。_exit()不会处理标准 I/O 缓冲区。

第7章 Linux系统进程与线程

- 7.1 Linux 的进程
- 7.2 进程间通信
- 7.3 Linux 的线程
- 7.4 线程的互斥与同步
- 3学时

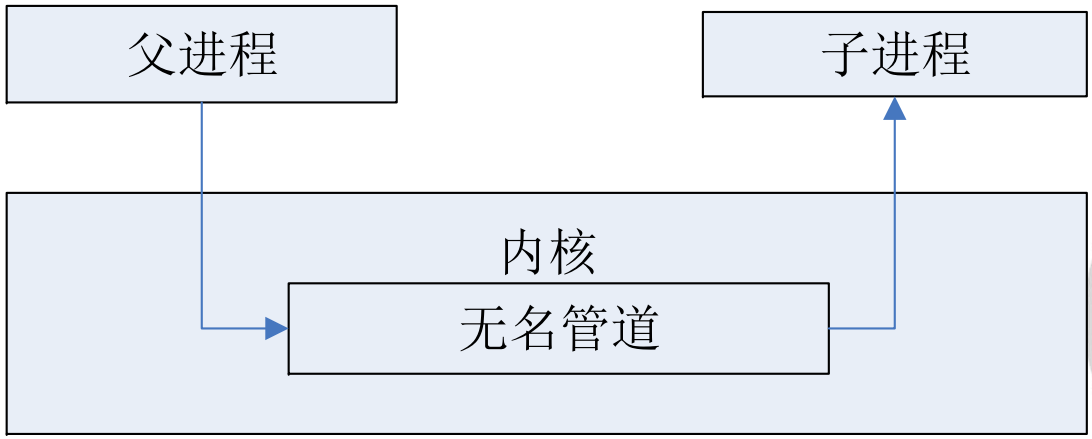
7.2 进程间通信

- 进程间通信方式的种类
  - 管道（Pipe）
  - 信号（Signal）
  - 消息队列（Messge Queue）
  - 共享内存（Shared memory）
  - 信号量（Semaphore）
  - 套接字（Socket）

7.2 进程间通信

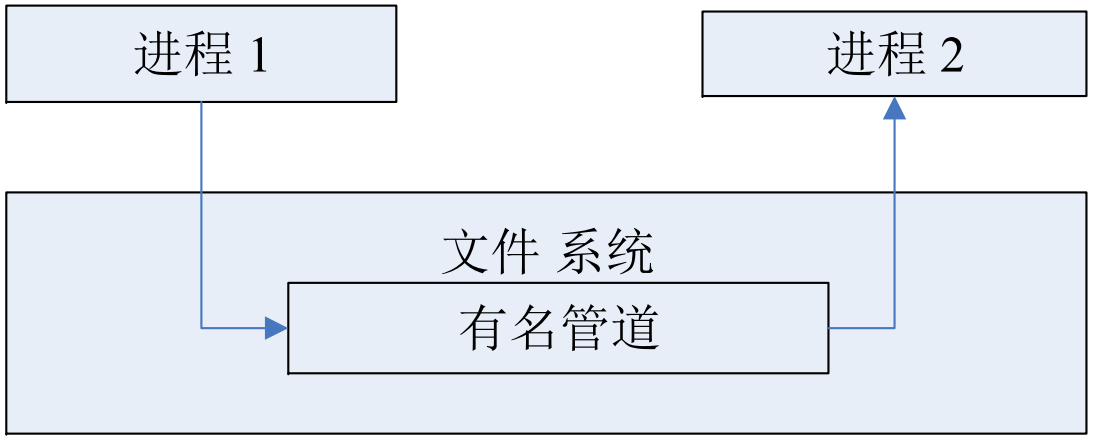
1)管道（Pipe）

- 无名管道（Pipe）：可用于具有亲缘关系进程间的通信（也就是父子进程或者兄弟进程之间）
- 它是一个半双工的通信模式，具有固定的读端和写端。
- 对于它的读写也可以使用普通的read()和write()等函数。



7.2 进程间通信

- 有名管道（named pipe）：除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信
  - 该管道可以通过路径名来指出，并且在文件系统中可见
  - 对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾，它们不支持如lseek()等文件定位操作



7.2 进程间通信

❖2)信号（Signal）：

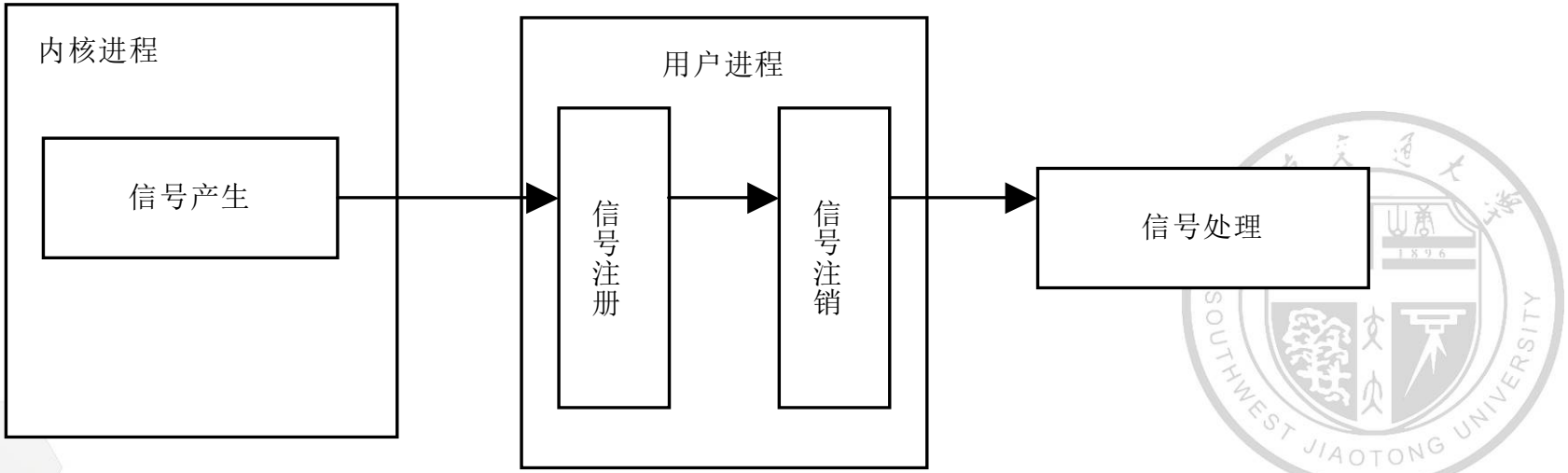
- 在软件层次上对中断机制的一种模拟，一种异步通信方式
- 用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一样的。
- 信号可以直接进行用户空间进程和内核进程之间的交互
- 它可以在任何时候发给某一进程，而无需知道该进程的状态。如果该进程当前并未处于执行态，则该信号就由内核保存起来，直到该进程恢复执行再传递给它为止。
- 如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程
- 不可靠信号和可靠信号

7.2 进程间通信

- 一个完整的信号生命周期
  - 信号产生
  - 信号在进程中注册
  - 信号在进程中注销
  - 执行信号处理函数

7.2 进程间通信

- 用户进程对信号的响应可以有3种方式
  - 忽略信号，即对信号不做任何处理，但是有两个信号不能忽略，即SIGKILL及SIGSTOP。
  - 捕捉信号，定义信号处理函数，当信号发生时，执行相应的自定义处理函数。
  - 执行缺省操作，Linux对每种信号都规定了默认操作。



7.2 进程间通信

❖常见信号

信 号 名	含 义	默 认 操 作
SIGHUP	该信号在用户终端连接（正常或非正常）结束时发出，通常是在终端的控制进程结束时，通知同一会话内的各个作业与控制终端不再关联	终止
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	终止
SIGQUIT	该信号和 SIGINT 类似，但由 QUIT 字符（通常是 Ctrl-\）来控制	终止
SIGILL	该信号在一个进程企图执行一条非法指令时（可执行文件本身出现错误，或者试图执行数据段、堆栈溢出时）发出	终止
SIGFPE	该信号在发生致命的算术运算错误时发出。这里不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术错误	终止
SIGKILL	该信号用来立即结束程序的运行，并且不能被阻塞、处理或忽略	终止
SIGALRM	该信号当一个定时器到时的时候发出	终止
SIGSTOP	该信号用于暂停一个进程，且不能被阻塞、处理或忽略	暂停进程
SIGTSTP	该信号用于交互停止进程，用户键入 SUSP 字符时（通常是 Ctrl+Z）发出这个信号	停止进程
SIGCHLD	子进程改变状态时，父进程会收到这个信号	忽略
SIGABORT	进程异常终止时发出	终止

7.2 进程间通信

❖信号发送与捕捉

- kill()函数可以发送信号给进程或进程组（实际上，kill系统命令只是kill()函数的一个用户接口）。这里需要注意的是，它不仅可以中止进程（实际上发出SIGKILL信号），也可以向进程发送其他信号

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：要发送信号的进程号
		0：信号被发送到所有和当前进程在同一个进程组的进程
		–1：信号发给所有的进程表中的进程（除了进程号最大的进程外）
		<-1：信号发送给进程组号为-pid 的每一个进程
	sig：信号	
函数返回值	成功：0	
	出错：–1	

7.2 进程间通信

❖信号处理函数signal()

- 信号处理的主要方法有两种，一种是使用简单的signal()函数，另一种是使用信号集函数组。

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：指定信号代码	
	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：以前的信号处理配置	
	出错：-1	

❖3）消息队列（Messge Queue）：

- 是消息的链接表，包括Posix消息队列SystemV消息队列。
- 克服了前两种通信方式中信息量有限的缺点，具有写权限的进程可以按照一定的规则向消息队列中添加新消息；
- 对消息队列有读权限的进程则可以从消息队列中读取消息
- 消息队列具有一定的FIFO特性，但是它可以实现消息的随机查询，比FIFO具有更大的优势
- 消息队列的实现包括创建或打开消息队列msgget()、添加消息msgsnd()、读取消息msgrcv()和控制消息队列msgctl()这四种操作

7.2 进程间通信

❖创建或打开消息队列msgget() 函数

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int msgget(key_t key, int msgflg)
函数传入值	key：消息队列的键值，多个进程可以通过它访问同一个消息队列，其中有个特殊值 IPC_PRIVATE。它用于创建当前进程的私有消息队列。
	msgflg：权限标志位
函数返回值	成功：消息队列 ID
	出错：–1

7.2 进程间通信

❖添加消息msgsnd()函数

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)	
函数传入值	msqid：消息队列的队列 ID	

	msgp：指向消息结构的指针。该消息结构 msgbuf 通常为：
	struct msgbuf
	{
	long mtype; /* 消息类型，该结构必须从这个域开始 */
	char mtext[1]; /* 消息正文 */
	}	

	msgsz：消息正文的字节数（不包括消息类型指针变量）	
	msgflg：	IPC_NOWAIT 若消息无法立即发送（比如：当前消息队列已满），函数会立即返回。
		0：msgsnd 调用阻塞直到发送成功为止。
函数返回值	成功：0	
	出错：−1	

7.2 进程间通信

❖读取消息msgrcv() 函数

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>		
函数原型	int msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg)		
函数传入值	msqid：消息队列的队列 ID		
	msgp：消息缓冲区，同于 msgsnd()函数的 msgp		
	msgsz：消息正文的字节数（不包括消息类型指针变量）		
	msgtyp：	0：接收消息队列中第一个消息	
		大于 0：接收消息队列中第一个类型为 msgtyp 的消息	
		小于 0：接收消息队列中第一个类型值不小于 msgtyp 绝对值且类型值又最小的消息	
	msgflg：	MSG_NOERROR：若返回的消息比 msgsz 字节多，则消息就会截短到 msgsz 字节，且不通知消息发送进程	
		IPC_NOWAIT 若在消息队列中并没有相应类型的消息可以接收，则函数立即返回	
		0：msgsnd()调用阻塞直到接收一条相应类型的消息为止	
函数返回值	成功：0		
	出错：-1		

7.2 进程间通信

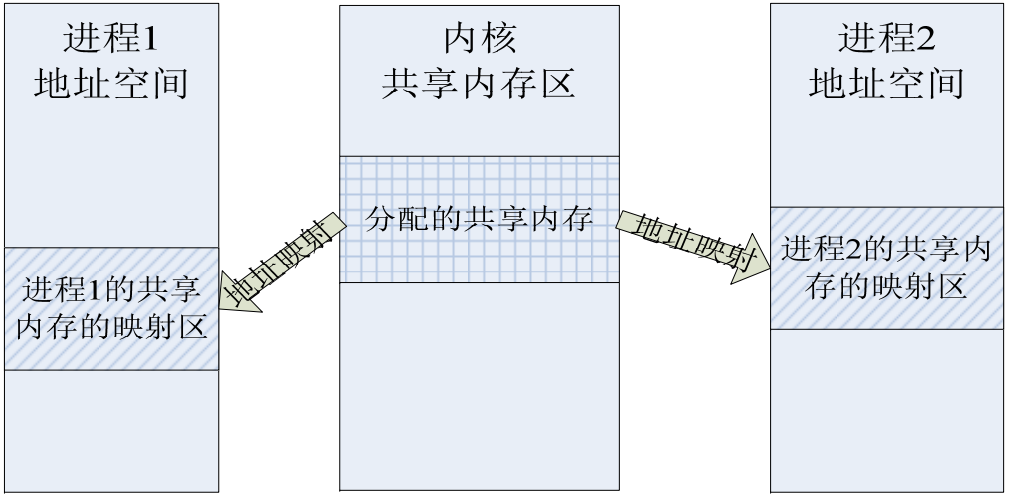
❖控制消息队列msgctl()函数

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgctl (int msgqid, int cmd, struct msqid_ds *buf)	
函数传入值	msqid：消息队列的队列 ID	
	cmd： 命令参数	IPC_STAT：读取消息队列的数据结构 msqid_ds，并将其存储在 buf 指定的地址中
		IPC_SET：设置消息队列的数据结构 msqid_ds 中的 ipc_perm 域（IPC 操作权限描述结构）值。这个值取自 buf 参数
		IPC_RMID：从系统内核中删除消息队列
	buf：描述消息队列的 msgqid_ds 结构类型变量	
函数返回值	成功：0	
	出错：−1	

7.2 进程间通信

❖4）共享内存（Shared memory）

- 最有用的进程间通信方式。
- 它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。
- 这种通信方式需要依靠某种同步机制，如互斥锁和信号量等



7.2 进程间通信

❖共享内存的实现

- 第一步是创建共享内存，函数是shmget()，也就是从内存中获得一段共享内存区域。
- 第二步映射共享内存，把共享内存映射到具体的进程空间中，这里使用的函数是shmat()。
- 可以使用不带缓冲的I/O读写命令对其进行操作。
- 当然还有撤销映射的操作，函数为shmdt()。

7.2 进程间通信

❖创建共享内存函数shmget()

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：共享内存的键值，多个进程可以通过它访问同一个共享内存，其中有个特殊值 IPC_PRIVATE。它用于创建当前进程的私有共享内存。
	size：共享内存区大小
	shmflg：同 open()函数的权限位，也可以用八进制表示法
函数返回值	成功：共享内存段标识符
	出错：-1

❖映射共享内存函数shmat()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	char *shmat(int shmid, const void *shmaddr, int shmflg)	
函数传入值	shmid：要映射的共享内存区标识符	
	shmaddr：将共享内存映射到指定地址（若为 0 则表示系统自动分配地址并把该段共享内存映射到调用进程的地址空间）	
	shmflg	SHM_RDONLY：共享内存只读
		默认 0：共享内存可读写
函数返回值	成功：被映射的段地址	
	出错：-1	

7.2 进程间通信

❖5）信号量（Semaphore）：

- 主要作为进程之间以及同一进程的不同线程之间的同步和互斥手段。

❖6）套接字（Socket）：

- 这是一种更为一般的进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

7.2 进程间通信

- 1、信号同步程序编程实例
- 2、消息队列程序实例
- 3、共享内存编程实例（P151）

第7章 Linux系统进程与线程

- 7.1 Linux 的进程
- 7.2 进程间通信
- 7.3 Linux 的线程
- 7.4 线程的互斥与同步
- 3学时

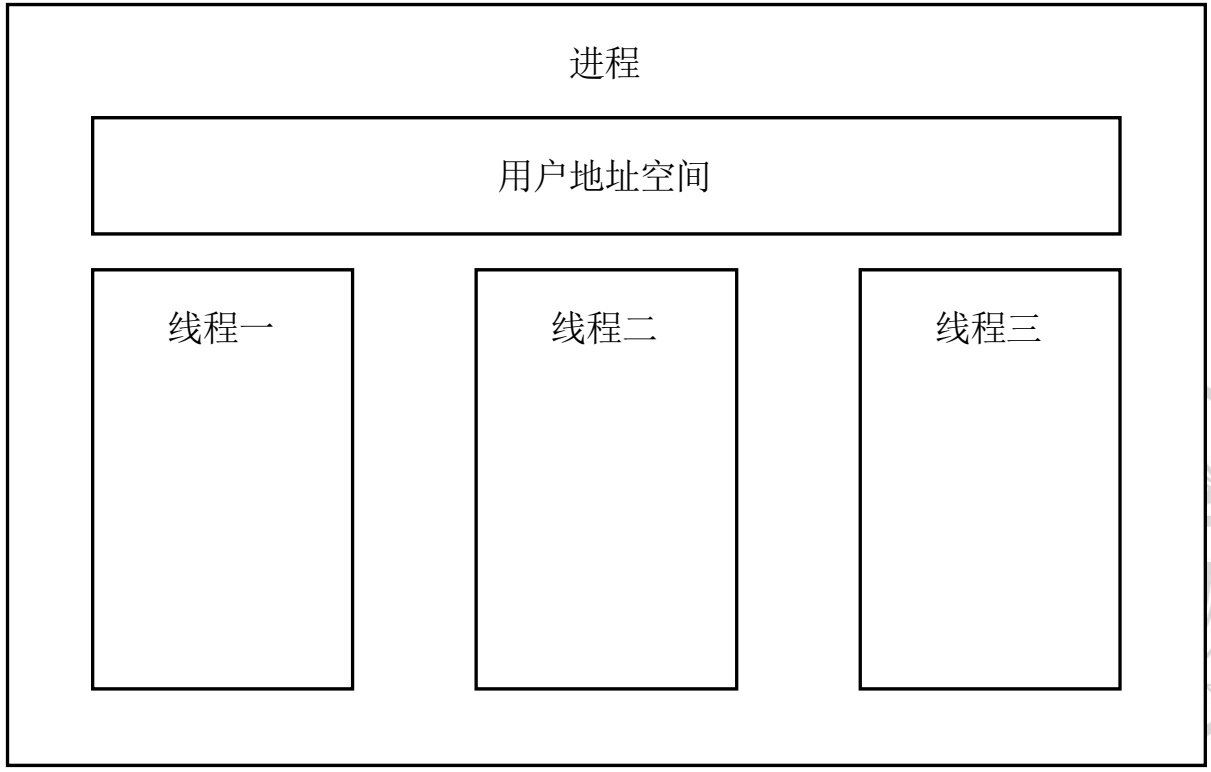
7.3 Linux 的线程

- 进程是系统中程序执行和资源分配的基本单位。
  - 每个进程都拥有自己的数据段、代码段和堆栈段，不同进程间要进行数据的传递只能通过通信的方式进行
  - 进程在进行切换等操作时都需要有比较复杂的上下文切换
- 线程是进程内独立的一条运行路线，处理器调度的最小单元，也可以称为轻量级进程。
  - 线程可以对进程的内存空间和资源进行访问，并与同一进程中的其他线程共享。因此，线程的上下文切换的开销比创建进程小很多。
  - 一个进程的开销大约是一个线程开销的30倍左右。
  - 一个进程可以有多个线程，也就是有多个线程控制表及堆栈寄存器，但却共享一个用户地址空间
  - 任何线程对系统资源的操作都会给其他线程带来影响，多线程中的同步是非常重要的问题。

7.3 Linux 的线程

❖进程与线程的关系图

- 由于同一进程下的线程之间共享数据空间，它们共享全局变量、共享进程指令、大多数数据和打开的文件(如描述字)、信号处理程序和信号处置、当前工作目录、用户ID和组ID，一个线程的数据可以直接为其他线程所用。



7.3 Linux 的线程

- 线程机制的分类和特性
- （1）用户级线程
  - 主要解决的是上下文切换的问题，它的调度算法和调度过程全部由用户自行选择决定，在运行时不需要特定的内核支持
  - 操作系统往往会提供一个用户空间的线程库，该线程库提供了线程的创建、调度和撤销等功能，而内核仍然仅对进程进行管理
  - 主要缺点是在一个进程中的多个线程的调度中无法发挥多处理器的优势

7.3 Linux 的线程

- （2）轻量级线程
  - 内核支持的用户线程，是内核线程的一种抽象对象。
  - 每个线程拥有一个或多个轻量级线程，而每个轻量级线程分别被绑定在一个内核线程上。
- （3）内核线程
  - 允许不同进程中的线程按照同一相对优先调度方法进行调度，可以发挥多处理器的并发优势

7.3 Linux 的线程

- 线程的基本操作函数
  - 创建线程函数 **pthread_create（）**
  - 等待线程的结束函数**pthread_join（）**
  - 终止线程函数 **pthread_exit（）**
  - 3个基本线程函数，在调用它们前均要包括pthread.h头文件

7.3 Linux 的线程

❖ （1）创建线程函数pthread_create（）

所需头文件	#include <pthread.h>
函数原型	int pthread_create ((pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg))
函数传入值	thread：线程标识符
	attr：线程属性设置（其具体设置参见 9.2.3 小节），通常取为 NULL
	start_routine：线程函数的起始地址，是一个以指向 void 的指针作为参数和返回值的函数指针
	arg：传递给 start_routine 的参数
函数返回值	成功：0
	出错：返回错误码

- 注意：第一个参数为指向线程标识符的指针
- 第二个参数用来设置线程属性
- 第三个参数是线程运行函数的起始地址
- 最后一个参数是运行函数的参数

7.3 Linux 的线程

❖ 等待线程的结束函数pthread_join（）

所需头文件	#include <pthread.h>
函数原型	int pthread_join ((pthread_t th, void **thread_return))
函数传入值	th：等待线程的标识符
	thread_return：用户定义的指针，用来存储被等待线程结束时的返回值（不为 NULL 时）
函数返回值	成功：0
	出错：返回错误码

这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

7.3 Linux 的线程

❖ 终止线程函数 pthread_exit（）

- 一个线程的结束有两种途径，一种是函数结束，调用它的线程也就结束，另一种方式是通过函数pthread_exit 来实现。
- 主动终止自身线程

所需头文件	#include <pthread.h>
函数原型	void pthread_exit(void *retval)
函数传入值	retval：线程结束时的返回值，可由其他函数如 pthread_join()来获取

- 一个线程中要终止另一个线程函数pthread_cancel()函数

所需头文件	#include <pthread.h>
函数原型	int pthread_cancel((pthread_t th)
函数传入值	th：要取消的线程的标识符
函数返回值	成功：0
	出错：返回错误码

7.3 Linux 的线程

❖ 简单的多线程编程举例

- Linux系统下的多线程遵循POSIX线程接口，称为pthread
- 编写Linux下的多线程程序，需要使用头文件pthread.h，连接时需要使用库libpthread.a
- Linux下pthread 的实现是通过系统调用clone()来实现的。

举例（教材P139）：主线程创建一个子进程，并等待子线程执行返回，多次执行，结果会不同

如：This is a pthread.
This is the main process.
This is a pthread.
This is the main process.
This is a pthread.
This is the main process

第7章 Linux系统进程与线程

- 7.1 Linux 的进程
- 7.2 进程间通信
- 7.3 Linux 的线程
- 7.4 线程的互斥与同步
- 3学时

7.4 线程的互斥与同步

❖1、互斥与同步的概念

- 在多任务操作系统环境下，多个进程会同时运行，并且一些进程之间可能存在一定的关联。多个进程可能为了完成同一个任务会相互协作，这样形成进程之间的同步关系。而且在不同进程之间，为了争夺有限的系统资源（硬件或软件资源）会进入竞争状态，这就是进程之间的互斥关系。
- 进程之间的互斥与同步关系存在的根源在于临界资源。临界资源是在同一个时刻只允许有限个（通常只有一个）进程可以访问（读）或修改（写）的资源，通常包括硬件资源（处理器、内存、存储器以及其他外围设备等）和软件资源（共享代码段，共享结构和变量等）。访问临界资源的代码叫做临界区，临界区本身也会成为临界资源。

7.4 线程的互斥与同步

- 线程共享进程的资源和地址空间，对这些资源操作时，必须考虑到线程间资源访问的同步和互斥
  - 互斥锁、信号量

2、信号量sem

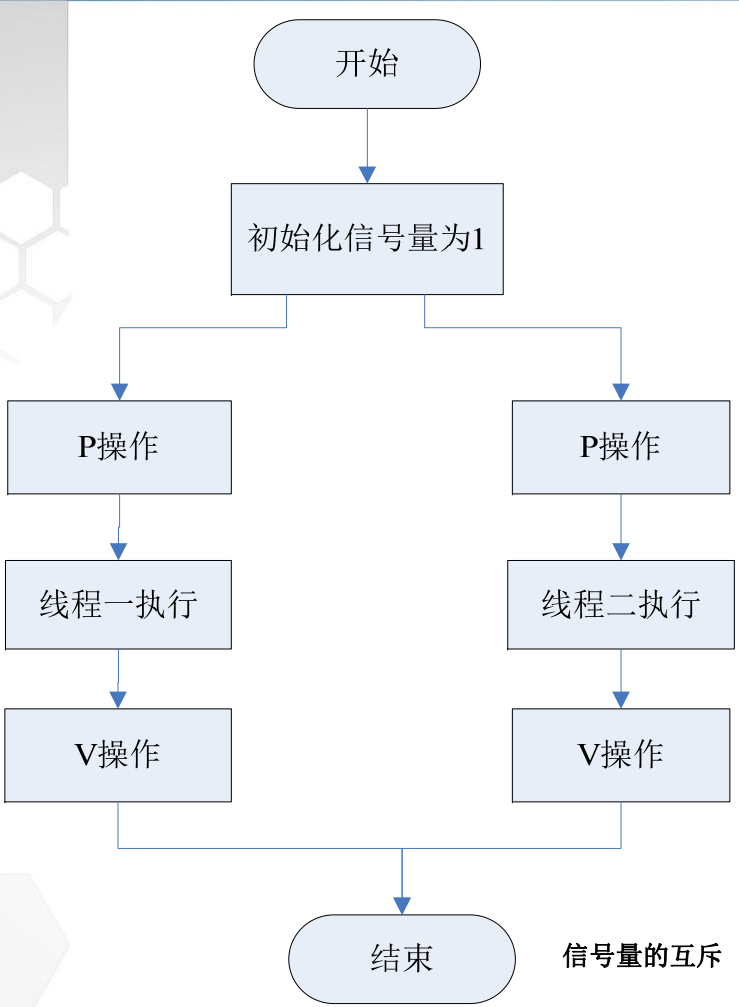
- 用来解决进程之间的同步与互斥问题的一种进程之间通信机制，
- 广泛用于进程或线程间的同步与互斥
- 包括一个称为信号量的变量和在该信号量下等待资源的进程等待队列，以及对信号量进行的两个原子操作（PV操作）。
- 其中信号量对应于某一种资源，取一个非负的整型值。
- 信号量值指的是当前可用的该资源的数量，若它等于0则意味着目前没有可用的资源。

7.4 线程的互斥与同步

- PV原子操作的具体定义为：
  - P操作：如果有可用的资源（信号量值>0），则占用一个资源（给信号量值减去一，进入临界区代码）;如果没有可用的资源（信号量值等于0），则被阻塞到，直到系统将资源分配给该进程（进入等待队列，一直等到资源轮到该进程）。
  - V操作：如果在该信号量的等待队列中有进程在等待资源，则唤醒一个阻塞进程。如果没有进程等待它，则释放一个资源（给信号量值加一）。
- 使用信号量访问临界区的伪代码所下所示：

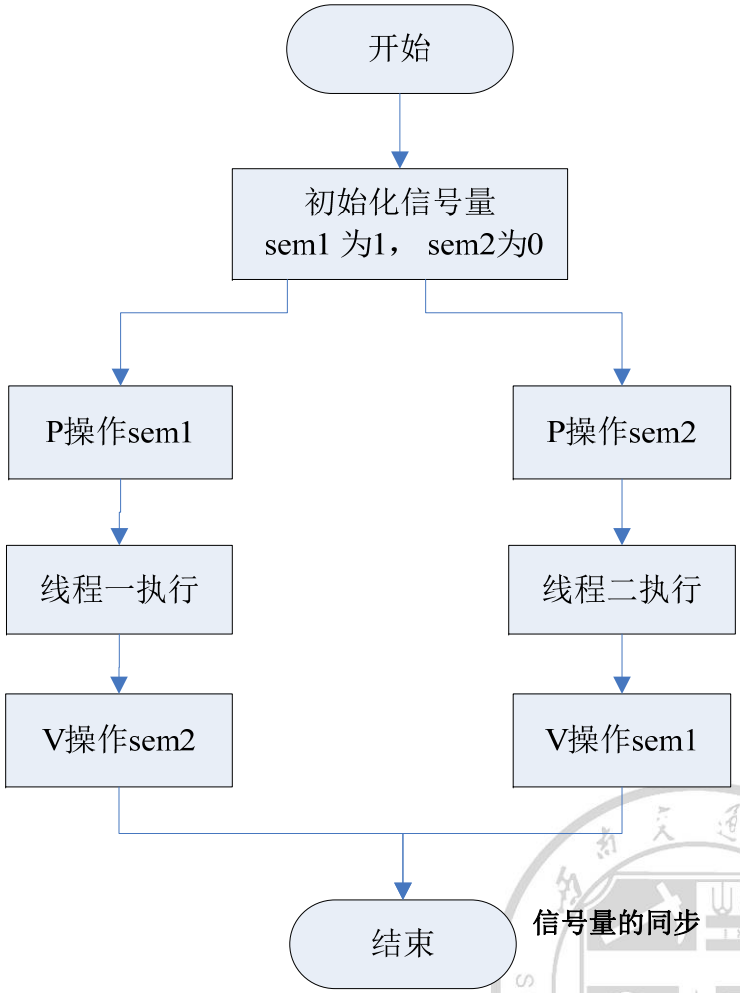
```
{
    /* 设R为某种资源，S为资源R的信号量*/
    INIT_VAL(S);                                    /* 对信号量S进行初始化 */
    非临界区;
    P(S);                                           /* 进行P操作 */
    临界区（使用资源R）;        /* 只有有限个（通常只有一个）进程被允许进入该区*/
    V(S);                                           /* 进行V操作 */
    非临界区;
}
```

7.4 线程的互斥与同步



信号量的互斥

互斥，几个进程（或线程）往往只设置一个信号量sem



信号量的同步

同步，几个进程（或线程）往往只设置多个信号量sem

7.4 线程的互斥与同步

❖信号量的使用

- 第一步：创建信号量或获得在系统已存在的信号量，调用semget()函数。
- 第二步：初始化信号量，此时使用semctl()函数的SETVAL操作。
- 第三步：进行信号量的PV操作，此时调用semop()函数。这一步是实现进程之间的同步和互斥的核心工作部分。
- 第四步：如果不需要信号量，则从系统中删除它，此时使用semclt()函数的IPC_RMID操作。需要注意，在程序中不应该出现对已经被删除的信号量的操作。

7.4 线程的互斥与同步

❖semget()函数

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：信号量的键值，多个进程可以通过它访问同一个信号量，其中有个特殊值IPC_PRIVATE。它用于创建当前进程的私有信号量。
	nsems：需要创建的信号量数目，通常取值为 1。
	semflg：同 open()函数的权限位，也可以用八进制表示法，其中使用 IPC_CREAT 标志创建新的信号量，即使该信号量已经存在（具有同一个键值的信号量已在系统中存在），也不会出错。如果同时使用 IPC_EXCL 标志可以创建一个新的唯一的信号量，此时如果该信号量已经存在，该函数会返回出错。
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值。
	出错：-1

7.4 线程的互斥与同步

❖semctl()函数

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：semget()函数返回的信号量标识符。
	semnum：信号量编号，当使用信号量集时才会被用到。通常取值为 0，就是使用单个信号量（也是第一个信号量）。
	cmd：指定对信号量的各种操作，当使用单个信号量（而不是信号量集）时，常用的有以下几种： IPC_STAT：获得该信号量（或者信号量集合）的 semid_ds 结构，并存放在由第四个参数 arg 的 buf 指向的 semid_ds 结构中。semid_ds 是在系统中描述信号量的数据结构。 IPC_SETVAL：将信号量值设置为 arg 的 val 值。 IPC_GETVAL：返回信号量的当前值。 IPC_RMID：从系统中，删除信号量（或者信号量集）

	arg：是 union semnn 结构，该结构可能在某些系统中并不给出定义，此时必须由程序员自己定义。
	union semun
	{
	int val;
	struct semid_ds *buf;
	unsigned short *array;
	}

函数返回值	成功：根据 cmd 值的不同而返回不同的值。 IPC_STAT、IPC_SETVAL、IPC_RMID：返回 0 IPC_GETVAL：返回信号量的当前值

7.4 线程的互斥与同步

❖ semop()函数

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 *sops, size_t nsops)
函数传入值	semid：semget()函数返回的信号量标识符。

	sops：指向信号量操作数组，一个数组包括以下成员：
	struct sembuf
	{
	short sem_num; /* 信号量编号，使用单个信号量时，通常取值为 0 */
	short sem_op;
	/* 信号量操作：取值为-1 则表示 P 操作，取值为+1 则表示 V 操作*/
	short sem_flg;
	/* 通常设置为 SEM_UNDO。这样在进程没释放信号量而退出时，系统自动释放该进程中未释放的信号量 */
	}

	nsops：操作数组 sops 中的操作个数（元素数目），通常取值为 1（一个操作）
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值。
	出错：-1

7.4 线程的互斥与同步

❖信号量线程控制相关函数

- sem_init()用于创建一个信号量，并初始化它的值。
- sem_wait()和sem_trywait()都相当于P操作，在信号量大于零时它们都能将信号量的值减一，两者的区别在于若信号量小于零时，sem_wait()将会阻塞进程，而sem_trywait()则会立即返回。
- sem_post()相当于V操作，它将信号量的值加一同时发出信号来唤醒等待的进程。
- sem_getvalue()用于得到信号量的值。
- sem_destroy()用于删除信号量。

7.4 线程的互斥与同步

❖sem_init()函数

所需头文件	#include <semaphore.h>
函数原型	int sem_init(sem_t *sem,int pshared,unsigned int value)
函数传入值	sem：信号量指针
	pshared：决定信号量能否在几个进程间共享。由于目前 Linux 还没有实现进程间共享信号量，所以这个值只能够取 0，就表示这个信号量是当前进程的局部信号量
	value：信号量初始化值
函数返回值	成功：0
	出错：−1

❖sem_wait()函数

所需头文件	#include <pthread.h>
函数原型	int sem_wait(sem_t *sem) int sem_trywait(sem_t *sem) int sem_post(sem_t *sem) int sem_getvalue(sem_t *sem) int sem_destroy(sem_t *sem)
函数传入值	sem：信号量指针
函数返回值	成功：0
	出错：−1

7.4 线程的互斥与同步

❖信号量程序示例（P141）

```
linux@ubuntu64-vm:~/threadtest/threadmux$ ls
threadmux.c
linux@ubuntu64-vm:~/threadtest/threadmux$ gcc -o threadmux threadmux.c -pthread
linux@ubuntu64-vm:~/threadtest/threadmux$ ./threadmux
I'm thread 2
thread2 : value = 0
I'm thread 1
thread1 : value = 1
thread1 : value = 2
thread2 : value = 3
thread1 : value = 4
thread2 : value = 5
thread1 : value = 6
thread1 : value = 7
thread2 : value = 8
thread1 : value = 9
thread2 : value = 10
end of thread 1
end of thread 2
```

作业

- 1．通过自定义信号完成进程间的通信。
- 2．编写一个简单的管道程序实现文件传输。