项目整体设计方案：

1. 完成线程的创建和撤消create()和destroy();调用下面
2. 调用上面的线程创建和删除函数和my\_swtch以及new\_int8来实现先来先服务方式和时间片轮转调度
3. 改变时间片的大小，观察结果的变化
4. 同样通过调用第一个创建和撤销线程的函数和P/V函数，来实现信号量的同步和互斥访问
5. 消息缓冲通信：通过send原语和receive原语来实现消息缓冲通信，send调用了getbuf和insert来获取空闲缓冲区和插入缓冲区。receive则调用了remov和insert来实现读取数据和将空闲缓冲区插入到缓冲区。
6. linux模拟shell主要通过fork()来创建一个子进程，被创建的子进程是父进程精确的副本。创建成功后，父子进程采用“写时复制”共享代码和数据，直到一方要写入时才对相应内容进行复制。然后通过进程调度来实现模拟shell的功能。
7. 创建管道，主要通过调用mkfifo来创建有名管道，然后通过open打开管道，通过read和write来进行管道操作，最后通过close来给关闭管道。
8. 通过消息队列传递信息：通过msgget创建消息队列，然后通过msgsnd添加一个消息，msgrcv从消息队列中读取消息，msgctl从消息队列中删除消息。
9. 通过共享内存传递信息：通过shmget创建一个共享内存。通过shmat映射地址空间

然后再映射的地址上谢谢如和读取信息。

任务分配：

**汪冠鸿**：第二章1，第四章2，

**林剑峰**：第二章4，第四章4，

**纪文韬**：第二章2，3，第四章3，

**王立敏**：第二章5，6，第四章5，

详细设计思路

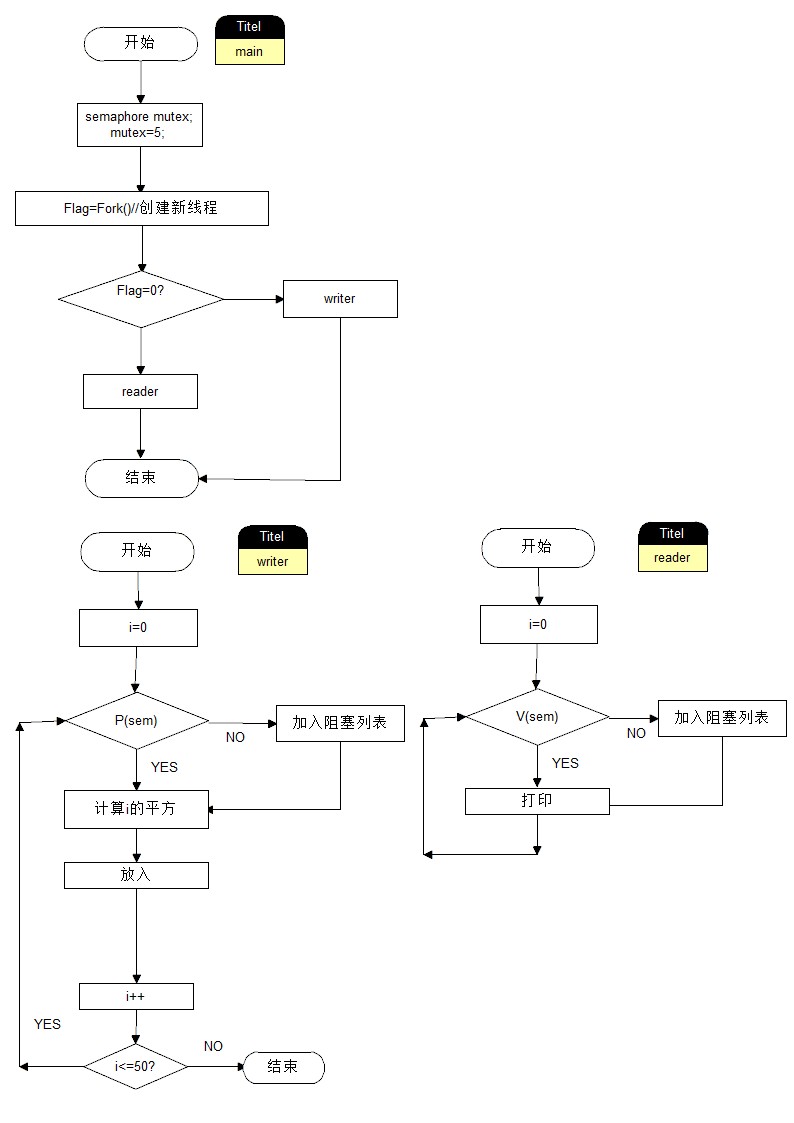
**王立敏:**

**第二章5 实现生产者和消费者的同步问题：**

Sem表示记录型信号量

void p(semaphore \*sem)//请求资源

void v(semaphore \*sem)//释放资源



**第二章6 实现消息缓冲通信**

获取空闲缓冲区函数struct buffer \*getbuf(void)

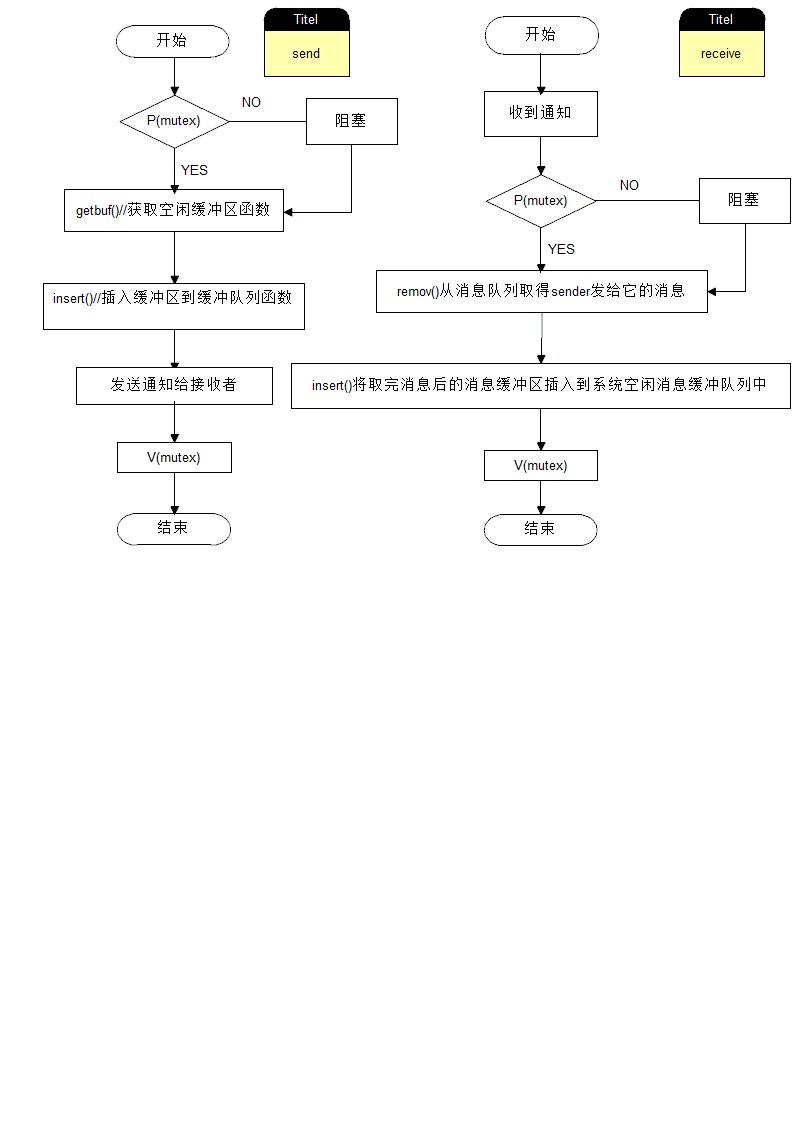
插入缓冲区到缓冲队列函数void insert(struct buffer \*\*mq,struct buffer \*buff)

发送原语void send(char \*receiver,char \*a,int size)//调用getbuf()和insert()实现。

获取消息缓冲区函数struct buffer \*remov(struct buffer \*\*mq,int sender)

接收原语int receive(char \*sender,char \*b)//调用remov()和insert()实现

流程图：

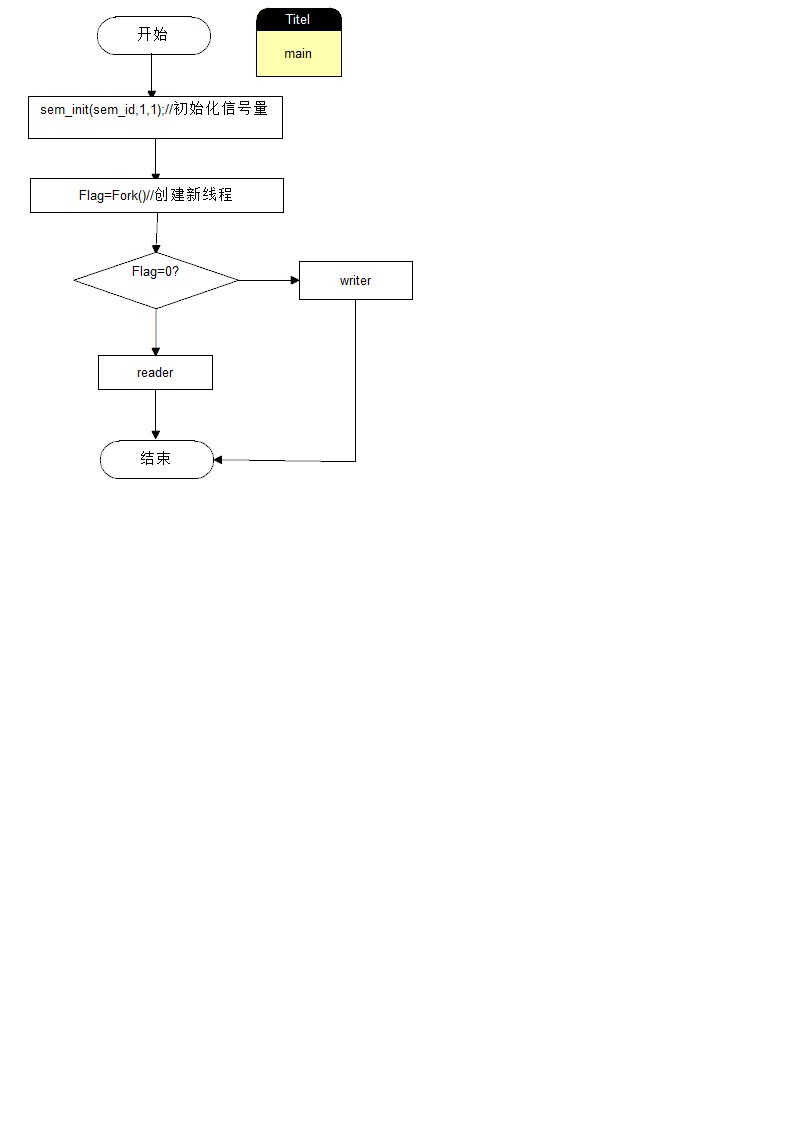


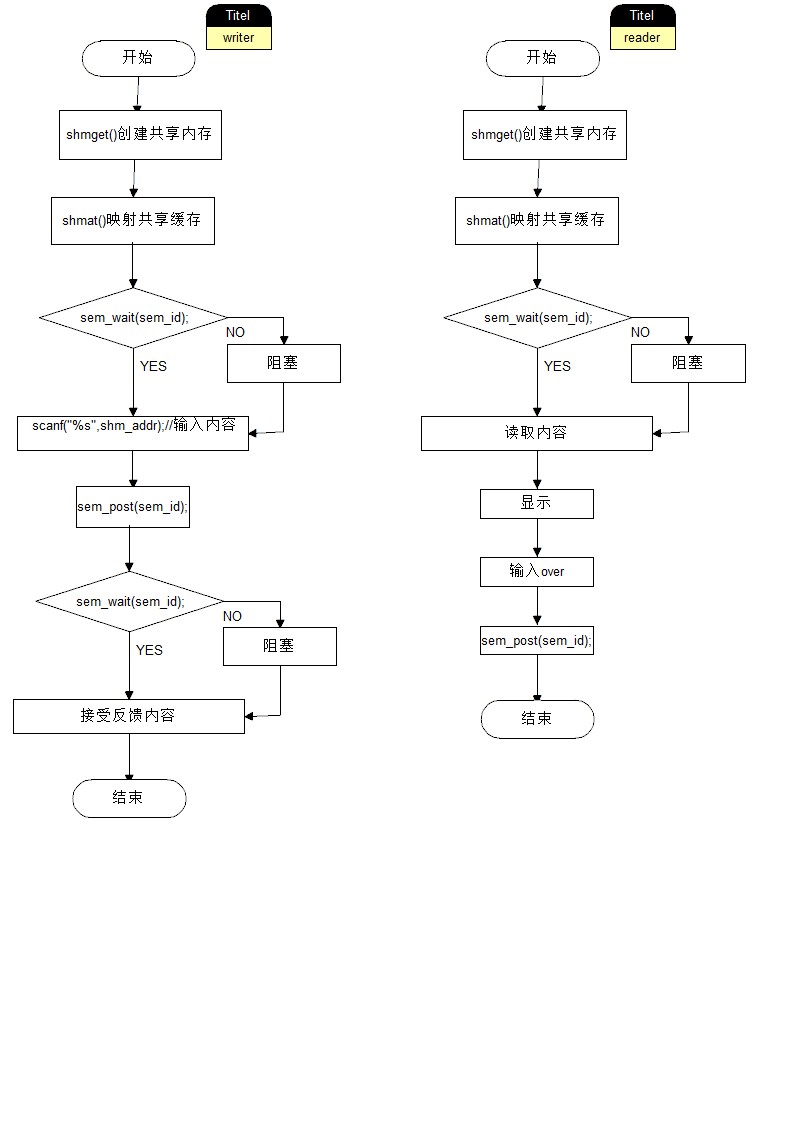
**第四章6 实现两个进程对共享内存的互斥使用**

函数：void writer()

void reader()

流程图：





**汪冠鸿**

线程的创建与撤销

1. **初始化TCB**
2. 函数原型申明：

void initTCB(void );

1. 函数功能描述：

初始化TCB中各个部分的值

3）输入(参数):

无

4）输出(返回值):

空

1. **线程创建函数格式说明：**
2. 函数申明原型：

typedef int (far \*codeptr)(void); /\*定义了一个函数指针类型 \*/

int create(char \*name,codeptr code,int stck) ；

1. 函数功能描述：

在main()函数中被调用，创建一个新进程，让其执行code开始的代码。

1. 输入(参数)：

name:新创建线程的外部标识符

code:新创建线程要执行的代码的入口地址，此处用函数名作为传入地址

stck:新创建线程的私有堆栈的长度

1. 输出(返回值):若创建成功，返回新创建进程的内部标识符(TCB的数组下标)，若创建失败，返回-1.
2. **线程撤销函数格式说明:**

1).函数申明原型：

void destroy(int id);

2).函数功能描述：

撤销内部标识符为id的指定线程

3).输入(参数):

id:将要被撤销的线程的内部标识符

4).输出(返回值):

返回为空

1. **撤销进程并重新调度函数格式说明:**

1).函数申明原型：

void over(void)

2).函数功能描述:

首先调用destory()撤销当前进程，然后重新进行CPU调度

3).输入(参数)：

空

4).输出(返回值):

空

Linux进程管理

1. **创建进程：**

1).函数原型：

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

2).参数：无

3).功能：

创建子进程。fork()用来创建一个子进程，被创建的子进程是父进程精确的副本。创建成功后，父子进程采用“写时复制”共享代码和数据，直到一方要写入时才对相应内容进行复制。

4).返回值：

创建成功，父进程返回子进程的进程，子进程中返回0；创建失败，则不会产生子进程，函数的返回值为-1。

1. **更改进程运行的程序**

1).函数原型：

#include <unistd.h>

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

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

2).参数：

path: 是新的被执行程序的路径名；

arg: 是若干个字符串，作为执行程序的参数，其中第一个字符串为可执行程序的程序名，最后一个字符串为NULL。

file:是新的被执行程序的文件名

argv:是字符串数组，作为执行程序的参数，必须为NULL作为结束字符

3).功能：exec系列函数，用来将进程执行的程序更换成路径名所指定的新程序。在exec调用之前存在的用户级上下文的内容，在exec调用成功之后，将不能再被访问。

4).返回值：

执行成功时无返回值，失败时返回-1

1. **终止进程：**

1).函数原型：

#include <stdlib.h>

void exit(int status);

2).参数：

status是返回给父进程的状态值

3).功能：

终止调用进程

4).返回值:

无

1. **等待一个进程终止:**

1).函数原型：

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status)；

pid\_t waitpid(pid\_t pid, int \*status, int options);

2).参数：

\*status 如果不为NULL,则保存结束的子进程的状态相关的信息

＞0,只等进程标识数为pid的子进程终止；

pid =0，等任一组标识数等于当前进程标识数的子进程终止；

-1，等任一子进程终止；

＜-1，等任一组标识数等于|pid|的子进程终止；

options: 可以为0，也可以是下列常量或它们的组合

WNOHANG:表示如果没有已经终止的子进程，调用者立即返回

WUTRACED:跟进程跟踪调试有关

3).功能：

wait函数挂起当前进程，直到它的一个子进程终止，或者该进程收到一个终止自己的信号为止，如果调用时子进程已经处于僵尸状态，则当前进程无需挂起，子进程所占用的系统资源将被释放。waitpid类似于wait，但它可指定所等待的子进程

4).返回值：

通常返回终止子进程的PID。如果使用了WNOHANG而且没有子进程终止，则返回0；如果出错则返回-1。

1. **获取进程的标识符、父进程的标识数**

1).函数原型：

#include <sys/types.h>

#include <unistd.h>

pid\_t get\_pid(void)

pid\_t get\_ppid(void)

2).参数：无

3).功能：

get\_pid函数用来获取调用进程的进程标识数，get\_ppid则用来获取调用进程父进程的标识数。

4).返回值：调用进程的标识数，或调用进程父进程的标识数。

**林剑锋**

第二章 基于DOS的多任务系统的实现

**——线程的同步与互斥**

**1.创建记录型信号量的数据结构**

typedef struct{

int value;

struct TCB \*wq;

} semaphore;

**2. 对信号量的P操作和V操作**

（1）void p(semaphore \*sem)

void p(semaphore \*sem)

{

struct TCB \*\*qp;

disable();

sem->value=sem->value-1;

if(sem->value<0){

qp=&(sem->wq);

block(qp);

}

enable();

}

// 通过

（2）void v(semaphore \*sem)

void v(semaphore \*sem)

{

struct TCB \*\*qp;

disable();

qp=&(sem->wq);

sem->value=sem->value+1;

if(sem->value<=0)

wakeup\_first(qp);

enable();

}

//释放

（3）

为了实现互斥，可设置一互斥信号量mutex：

semaphore mutex; mutex=1;

在相应的需要互斥的线程中，将临界区CS放在P(&mutex)和V(&mutex)之间。

**3.实验思路**

当两个或多个线程共用统一资源(如某一变量，或某一数据结构）, 线程之间存在着同步与互斥的问题。这里我们选择记录型信号量机制来作为线程的同步机制,并设置互斥信号量来实现互斥。

**第四章 LINUX进程管理**

**——线程的创建与LINUX消息队列通信机制**

**1.线程的创建**

(1)．线程创建函数格式说明

typedef int (far \*codeptr)(void); /\*定义了一个函数指针类型\*/

Int create(char \*name,codeptr code,int stck) ；

功能描述：在main()函数中调用，创建一个新线程，让其执行code开始的代码。

输入：

name：新创建线程的外部标识符；

code：新创建线程要执行的代码的入口地址，此处用函数名作为传入地址；

stck： 新创建线程的私有堆栈的长度。

输出：新创建线程的内部标识符，若创建失败，返回-1

(2)初始化线程

为了方便堆栈的初始化工作，我们可以按照堆栈中的内容设计一个以下的数据结构：

struct int\_regs {

unsigned bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags,off,seg;

};

然后用一个指向该数据结构的指针给堆栈赋值。

**2.线程间利用消息队列实现通信**

（1）消息缓冲区

struct buffer

{

int sender; /\* 消息发送者的内部标识 \*/

int size; /\* 消息长度 <=NTEXT 个字节 \*/

char text[NTEXT]; /\* 消息正文 \*/

struct buffer \*next; /\* 指向下一个消息缓冲区的指针 \*/

};

为了方便管理，创建freebuf //空闲缓冲队列

为空闲消息缓冲队列设置一互斥信号量mutexfb 初值{1,NULL}//空闲缓冲队列互斥信号量

为空闲缓冲队列设置一个计数信号量sfbsfb 初值为{NBUF,NULL}//计数信号量

（2）TCB中与通信有关的数据

struct buffer \*mq; /\* 接收线程的消息队列队首指针 \*/

semaphore mutex; /\* 接收线程的消息队列的互斥信号量 \*/

semaphore sm; /\* 接收线程的消息队列的计数信号量，用于实现同步 \*/

mq的初值为NULL

mutex的初值为{1,NULL}

sm的初值为{0,NUL｝

（3）主要函数

获取空闲缓冲区函数 struct buffer \*getbuf(void)

功能：从空闲消息缓冲队列头上取下一空闲消息缓冲区。

输入：无

输出：指向所获得的空闲消息缓冲区的指针

return(buff);

插入缓冲区到缓冲队列函数 void insert(struct buffer \*\*mq,struct buffer \*buff)

功能：将buff所指的缓冲区插到\*mq所指的缓冲队列末尾。

输入：

mq：将要插入的缓冲队列的队首指针的指针；

buff：消息缓冲区指针。

输出：无

发送原语 void send(char \*receiver,char \*a,int size)

功能：将地址a开始的size个字节发送给外部标识符为receiver的线程。

输入：

receiver：接收线程的外部标识符的指针；

a：要发送的消息正文的指针；

size：要发送的消息正文的长度。

输出：无

获取消息缓冲区函数 struct buffer \*remov(struct buffer \*\*mq,int sender)

功能：接收线程从它自己的消息队列中取下sender发送给它的消息缓冲区。

输入：

mq：接收线程的消息队列的队首指针的指针；

sender：发送线程的内部标识符；

输出：所获得的消息缓冲区的指针

接收原语 int receive(char \*sender,char \*b)

功能：从接收线程的消息缓冲队列中取一个从sender发送过来消息并复制到b中。

输入：

sender：发送线程的外部标识符；

b：接收线程内部的接收区，用来存放接收到的消息正文。

输出：接收的信息

**3.Linux消息队列通信机制**

（1）数据结构

消息缓冲区 struct msgbuf

struct msgbuf{ /\* 消息定义的参照格式 \*/

long mtype; /\* 消息类型（大于0的长整数） \*/

char mtext[1]; /\*消息正文\*/

};

消息结构 struct msg

struct msg {

struct msg \*msg\_next; /\*消息队列链接指针，指向队列中的下一条消息 \*/

long msg\_type; /\*消息类型，同struct msgbuf中的mtype\*/

char \*msg\_spot; /\* 消息正文的地址,指向msgbuf的消息正文 \*/

time\_t msg\_stime; /\* 消息发送的时间 \*/

short msg\_ts; /\* 消息正文的大小 \*/

};

IPC对象访问权限struct ipc\_perm

struct ipc\_perm{

key\_t key; /\* IPC对象键值 \*/

ushort uid; /\* owner euid and egid \*/

ushort gid;

ushort cuid; /\* creator euid and egid \*/

ushort cgid;

ushort mode; /\* 访问权限 \*/

ushort seq; /\* slot usage sequence number，即IPC对象使用频率信息 \*/

};

消息队列结构体struct msqid\_ds

struct msqid\_ds {

struct ipc\_perm msg\_perm; /\* 消息队列访问权限\*/

struct msg \*msg\_first; /\* 队列上第一条消息，即链表头\*/

struct msg \*msg\_last; /\* 队列中的最后一条消息，即链表尾 \*/

time\_t msg\_stime; /\* 发送给队列的最后一条消息的时间 \*/

time\_t msg\_rtime; /\* 从消息队列接收到最后一条消息的时间 \*/

time\_t msg\_ctime; /\* 最后修改队列的时间\*/

…

ushort msg\_cbytes; /\*队列上所有消息总的字节数 \*/

ushort msg\_qnum; /\*当前队列上消息的个数 \*/

ushort msg\_qbytes; /\* 队列允许的最大的字节数 \*/

ushort msg\_lspid; /\* 发送最后一条消息的进程的pid \*/

ushort msg\_lrpid; /\* 接收最后一条消息的进程的pid \*/

};

(2)消息队列相关的系统调用

int msgget(key\_t key, int msgflg)

//如果IPC\_CREAT单独使用，semget()为一个新创建的消息队列返回标识数，或者返回具有相同键值的已存在消息队列标识数。如果IPC\_EXCL与IPC\_CREAT一起使用，要么创建一个新的队列并返回它的标识数，如果队列已存在，则返回-1。

int msgsnd(int msqid, struct msgbuf \*msgp, size\_t msgsz, int msgflg);

//在标识数为msqid的消息队列中添加一个消息，即向标识数为msqid的消息队列发送一个消息。

ssize\_t msgrcv(int msqid, struct msgbuf \*msgp, size\_t msgsz, long msgtyp, int msgflg);

//如果传递给参数msgflg的值为IPC\_NOWAIT，并且没有可取的消息，那么给调用进程返回ENOMSG错误码，否则，调用进程阻塞，直到一条满足要求的消息到达消息队列。如果进程正在等待消息，而相应的消息队列被删除，则返回EIDRM。如果当进程正在等待消息时，捕获到了一个信号，则返回EINTR

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

//获取或设置消息队列的属性信息，或者删除消息队列

**纪文韬**

线程调度

1. **初始化DOS**
2. 函数原型申明：

void InitDos(void)

1. 函数功能描述：

获得INDOS标志的地址和严重错误标志的地址

3）输入(参数):

无

4）输出(返回值):

空

1. **判断DOS：**
2. 函数申明原型：

int DosBusy(void);

1. 函数功能描述：

获得Indos标志及严重错误标志的值，判断是否dos忙：

1. 输入(参数)：

空

1. 输出(返回值):

/\* 如果返回值是1，表示dos忙；\*/

/\* 如果返回值是0，表示dos不忙；\*/

/\* 如果返回值是-1，表示还没有调用InitDos() \*/

1. 调度程序**:**

1.

1).函数申明原型：

void interrupt my\_swtch(void)

2).函数功能描述：

解决两种原因引起的调度：线程执行完毕或正在执行的线程因等待某事件发生而不能继续执行。

3).输入(参数):

空

4).输出(返回值):

空

2.

1).函数申明原型：

void interrupt new\_int8(void)：

2).函数功能描述:

该函数主要解决因时间片到时引起的调度

3).输入(参数)：

空

4).输出(返回值):

空

Linux

1. **pipe系统调用：**

1).函数原型：

int pipe(int filedes[2]);

2).参数：filedes参数是一个输出参数，它返回两个文件描述符，其中filedes[0]指向管道的读端，filedes[1]指向管道的写端。

3).功能：

pipe在内存缓冲区中创建一个管道，并将读写该管道的一对文件描述符保存在filedes所指的数组中，其中filedes[0]用于读管道，filedes[1]用于写管道。

4).返回值：

成功返回0；失败返回-1，并在error中存入错误码

（5）错误代码

EMFILE：进程使用的文件描述符过多

ENFILE ：系统文件表已满

EFAULT ：非法参数filedes

1. **更改进程运行的程序**

1).有名管道！

有名管道通过mkfifo创建：

#include <sys/types.h>

#include <sys/stat.h>

int mkfifo(const char \* pathname, mode\_t mode)

该函数的第一个参数是一个普通的路径名，也就是创建后FIFO的名字。第二个参数与打开普通文件的open()函数中的mode 参数相同。 有名管道创建成功，mkfifo()返回0；否则返回-1。如果mkfifo的第一个参数是一个已经存在的路径名时，错误代码中会返回EEXIST错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开FIFO的函数就可以了。

与普通文件类似，有名管道在使用之前必须先进行open操作，具体类似于文件的打开方式：

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*pathname, int flags);

对有名管道的open操作必须遵循下列规则：（1）如果当前打开操作是为读而打开FIFO时，若已经有相应进程为写而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为写而打开该FIFO（当前打开操作设置了阻塞标志）；或者，成功返回（当前打开操作没有设置阻塞标志）。（2）如果当前打开操作是为写而打开FIFO时，如果已经有相应进程为读而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为读而打开该FIFO（当前打开操作设置了阻塞标志）；或者，返回ENXIO错误（当前打开操作没有设置阻塞标志）。

一旦打开操作成功，便可通过返回的文件描述符，利用read、write系统调用对管道进行读写操作，读写完成应使用close系统调用关闭有名管道。