华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

# 《从实践中学嵌入式 LINUX 操作系统》

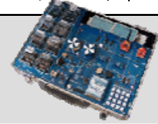
作者：华清远见

专业始于专注 卓识源于远见

# 第 6 章 进程间通信

本章简介

进程间通信就是在不同进程之间传播或交换信息。在一个操作系统中，调节各个进程对资源的共享是操作系统的重要职能之一。从原理上看，进程通信的关键技术就是在进程间建立某种共享区，利用进程都可以访问共享区的特点来建立一些通信通道。本章介绍 Linux 操作系统提供的部分进程间通信方式。包括信号量、共享内存、消息队列及管道。



# 6.1 什么是进程间通信

在单任务系统中，任务被线性执行时，不可能被抢占，所以不需要同步机制保护共享资源和临界资源，而且单任务也不存在数据交换的问题。但对于多任务操作系统来说，上述问题都是存在的。如何保护临界资源和进行数据交换，都是操作系统需要解决的问题。进程间通信（IPC）就是为了解决这些问题而提出的特有机制，它们为多任务系统提供了不同进程的通信机制，同时也提供了对于临界资源和共享资源的保护。

进程间通信的主要目的是实现同一计算机系统内部的相互协作的进程之间的数据共享与信息交换，由于这些进程处于同一软件和硬件环境下，利用操作系统提供的编程接口，用户可以方便地在程序中实现这种通信；应用程序间通信的主要目的是实现不同计算机系统中的相互协作的应用程序之间的数据共享与信息交换，由于应用程序分别运行在不同计算机系统中，它们之间要通过网络之间的协议才能实现数据共享与信息交换。

Linux 系统的进程通信方式基本上是从 UNIX 平台上的进程通信手段继承而来的。而对 UNIX 发展做出重大贡献的贝尔实验室及 BSD 在进程间通信方面的侧重点有所不同。前者对 UNIX 早期的进程间通信手段进行了系统的改进和扩充，形成了“System V IPC”，通信进程局限在单个计算机内；后者则跳过了该限制，形成了基于套接口（Socket）的进程间通信机制。Linux 则把两者都继承了下来，如图 6.1 所示。

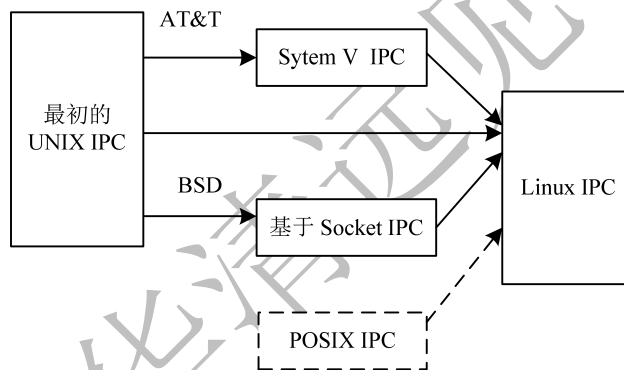



图 6.1 Linux 系统的通信方式

其中，最初 UNIX IPC 包括管道、FIFO、信号；System V IPC 包括 System V 消息队列、System V 信号灯、System V 共享内存区；Posix IPC 包括 Posix 消息队列、Posix 信号灯、Posix 共享内存区。

对于不同的嵌入式系统，进程间通信的实现方式有所不同，但是基本原理都差不多。对于进程间通信，主要有两种方式：虚拟内存系统中的进程间通信和 Falt 内存系统中的进程间通信。

μC/OS 是比较典型的 Falt 内存系统，它不支持虚拟内存机制，也没有用户空间和内核空间的区别，实际上它类似于 Linux 的内核空间，不同任务间可以相互访问，没有不同进程间内存保护机制。所以可以完全利用 Linux 系统中的同一进程中不同线程的通信机制。由于所有的任务与中断都共享同一地址空间，所以同步机制也与任务间通信在同一空间中实现，使这两种机制的相互替换成为可能。

Windows 作为一种复杂的多任务系统，也提供了多种进程间通信方式，包括文件映射、共享内存、匿名管道、命名管道、邮件槽、剪贴板、动态数据交换（DDE）、对象链接与嵌入（OLE）、动态链接库（DLL）、远程过程调用（RPC）、NetBios 函数、Sockets、 WM_COPYDATA 消息。

本章将阐述 Linux 系统是如何实现进程间通信的。

# 6.2 互斥与同步

互斥与同步是进程间通信中非常重要的一对概念，也是相交进程之间的两种主要关系。在嵌入式操作系统开发中经常会遇到同步、互斥的问题，如果处理得不好，程序就会出现很多意想不到的结果。而在多处理器之间、ISR 与 ISR 之间、ISR 与任务之间、任务与任务之间都可能需要互斥与同步。例如，不同任务优先级的抢占、中断处理等。

互斥和同步是两个紧密相关而又容易混淆的概念。所谓互斥，是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排他性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。两个互斥的进程，只能等到前一个进程运行完后，下一个进程才能运行。所谓同步，是指在互斥的基础上（大多数情况），通过其他机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

同步是一种更为复杂的互斥，而互斥是一种特殊的同步。

例如，下面的代码：

```
s=1;
 i=0;
codeostart
          p1();
          p2();
codeend
      p1(){
          while(){   //循环
              p(s);
              i++;
              printf("%d",i);
              v(s);
            }
        }
      p2(){
            while(){   //循环
                  p(s);
                  i--;
                  printf("%d",i);
                  v(s);
              }
       }
```

其中，p1()和 p2()是两个进程。i++和 i--分别是两个程序片断。如果没有执行完 p1()是不允许执行 p2()的。这里的 p1()和 p2()并不是指一个完整的进程，只是两个进程的临界资源（共有）。

这里必须掌握“原子操作”的概念。所谓原子操作，就是该操作绝不会在执行完毕前被任何其他任务或事件打断，也就是说，它是最小的执行单位，不可能有比它更小的执行单位。这里的原子实际上是使用了物理学中的物质微粒的概念。原子操作主要用于实现资源计数，很多引用计数（Refcnt）就是通过原子操作实现的。

原子操作需要硬件的支持，因此是架构相关的，其 API 和原子类型的定义都定义在内核源代码的 include/asm-generic/atomic.h 文件中，它们都使用汇编语言实现，因为 C 语言并不能实现这样的操作。以 ARM 平台为例，原子类型定义如下：

```
typedef struct { volatile int counter; } atomic_t;
```

volatile 修饰字段告诉 GCC 编译器不要对该类型的数据进行优化处理，对它的访问都是对内存的访问，而不是对寄存器的访问。

有关原子操作的更多 API，可以阅读 atomic.h 文件，这里不再赘述。

```
#define atomic_inc_not_zero(v) atomic_add_unless((v), 1, 0)

#define atomic_add(i, v)  (void) atomic_add_return(i, v)
#define atomic_inc(v)     (void) atomic_add_return(1, v)
#define atomic_sub(i, v)  (void) atomic_sub_return(i, v)
#define atomic_dec(v)     (void) atomic_sub_return(1, v)

#define atomic_inc_and_test(v) (atomic_add_return(1, v) == 0)
```

```
#define atomic_dec_and_test(v) (atomic_sub_return(1, v) == 0)
#define atomic_inc_return(v)    (atomic_add_return(1, v))
#define atomic_dec_return(v)    (atomic_sub_return(1, v))
#define atomic_sub_and_test(i, v) (atomic_sub_return(i, v) == 0)
#define atomic_add_negative(i,v) (atomic_add_return(i, v) < 0)
```

# 6.3 信号量

## 6.3.1 什么是信号量

信号量是最早出现的用来解决进程同步与互斥问题的机制，包括一个称为信号量的变量及对它进行的两个原语操作。这两个原语操作使用荷兰语命令：Prolagen（降低）和 Verhogen（升起），通常简称为 P、V 操作。

信号量可以用来保护两个或多个关键代码段，这些关键代码段不能并发调用。在进入一个关键代码段之前，线程必须获取一个信号量。如果关键代码段中没有任何线程，那么线程会立即进入该框图中的那个部分。一旦该关键代码段完成了，那么该线程必须释放信号量。其他想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量，然后将 Acquire Semaphore VI 及 Release Semaphore VI 分别放置在每个关键代码段的首末端。确认这些信号量 VI 引用的是初始创建的信号量。

下面看一下如何用信号量解决经典的生产者—消费者问题。问题描述如下：

一个仓库可以存放 *k* 件物品。生产者每生产一件产品，将产品放入仓库，仓库满了就停止生产。消费者每次从仓库中取一件物品，然后进行消费，仓库空时就停止消费。代码中，Producer 进程是生产者进程，Consumer 是消费者进程。共有的数据结构如下：

```
buffer: array [0..k-1] of integer;
in,out: 0..k-1;
```

其中，in 记录第一个空缓冲区，out 记录第一个不空的缓冲区。

```
s1,s2,mutex: semaphore;
```

其中，s1 控制缓冲区不满，s2 控制缓冲区不空，mutex 保护临界区。

初始化 s1=k，s2=0，mutex=1。

```
producer（生产者进程）:
 Item_Type item;
{
   while (true)
   {
     produce(&item);
     p(s1);
     p(mutex);
     buffer[in]:=item;
     in:=(in+1) mod k;
     v(mutex);
     v(s2);
   }
}

consumer（消费者进程）:
 Item_Type item;
{
   while (true)
   {
     p(s2);
     p(mutex);
      item:=buffer[out];
      out:=(out+1) mod k;
```

华清远见

```
            v(mutex);
            v(s1);
           consume(&item);
        }
    }
```

## 6.3.2 信号量的内核实现

Linux 内核的信号量在概念和原理上与用户态的 System V 的 IPC 机制信号量是一样的，但是它只能在内核空间使用。信号量在创建时需要设置一个初始值，表示同时可以有几个任务访问该信号量保护的共享资源，初始值为 1 就变成互斥锁（Mutex），即同时只能有一个任务可以访问信号量保护的共享资源。当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加 1 实现，如果信号量的值为非正数，表明有任务等待当前信号量，所以，它也唤醒所有等待该信号量的任务。

在 Linux 内核源代码的 kernel/printk.c 中，使用宏 DECLARE_MUTEX 声明了一个互斥锁 console_sem，它用于保护 console 驱动列表 console_drivers 及同步对整个 console 驱动系统的访问。

信号量数据结构定义在 include/linux/semaphore.h 中，代码如下：

```
struct semaphore {
    spinlock_t        lock;
    unsigned int count;
    struct list_head  wait_list;
};
```

其中，wait _list 字段存放当前等待该信号量的所有进程的链表。如果 count 大于或等于 0，该链表就为空。

与信号量相关的内核实现还包括 sema_init 函数，其中的 val 表示信号量的初始值，代码如下：

```
static inline void sema_init(struct semaphore *sem, int val)
{
    static struct lock_class_key __key;
    *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
    lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
#define init_MUTEX(sem)        sema_init(sem, 1)
#define init_MUTEX_LOCKED(sem) sema_init(sem, 0)
```

在 Linux 系统中，P 函数被称为 down，指的是该函数减小了信号量的值。它也许会将调用者置于休眠状态，然后等待信号量变得可用，之后再授予调用者对被保护资源的访问权限。

down 函数有如下几个版本，这些函数都在 semaphore.c 文件中实现。

### 1．down

```
void down(struct semaphore *sem)
{
    unsigned long flags;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);
    spin_unlock_irqrestore(&sem->lock, flags);
}
```

这个版本用来减小信号量的值，并在必要时一直等待。

### 2．down_interruptible

```
int down_interruptible(struct semaphore *sem)
{
    unsigned long flags;
    int result = 0;

    spin_lock_irqsave(&sem->lock, flags);
```

```
        if (likely(sem->count > 0))
            sem->count--;
        else
            result = __down_interruptible(sem);
        spin_unlock_irqrestore(&sem->lock, flags);

        return result;
    }
```

作为通常规则，我们不应该使用非中断版本。使用该函数时，如果操作被中断，则该函数会返回非 0 值，而调用者不会拥有该信号量。因此，对该函数的正确使用需要始终检查返回值，并做出相应的响应。

### 3．down_killable

```
    int down_killable(struct semaphore *sem)
    {
        unsigned long flags;
        int result = 0;

        spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
            sem->count--;
        else
            result = __down_killable(sem);
        spin_unlock_irqrestore(&sem->lock, flags);

        return result;
    }
```

TASK_KILLABLE 是 Linux Kernel 2.6.25 引入了一种新的进程睡眠状态。相对应的，down 函数有一个 killable 版本。它用于获取信号量 sem。如果信号量不可用，它将被置为睡眠状态；如果向它传递了一个 fatal signals，则会将它从等待列表中删除，并且需要响应此信号。

### 4．down_trylock

```
    int down_trylock(struct semaphore *sem)
    {
        unsigned long flags;
        int count;

        spin_lock_irqsave(&sem->lock, flags);
        count = sem->count - 1;
        if (likely(count >= 0))
            sem->count = count;
        spin_unlock_irqrestore(&sem->lock, flags);

        return (count < 0);
    }
```

down_trylock 函数尝试获得信号量 sem，如果能够立即获得，则获得信号量 sem 并返回 0；否则，返回非 0。它不会导致调用者睡眠，可以在中断上下文中使用。

### 5．down_ timeout

```
    int down_timeout(struct semaphore *sem, long jiffies)
    {
        unsigned long flags;
        int result = 0;

        spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
            sem->count--;
        else
            result = __down_timeout(sem, jiffies);
        spin_unlock_irqrestore(&sem->lock, flags);

        return result;
    }
```

以下几个 down 函数最终都调用了__down_common 函数：

```
static noinline void __sched __down(struct semaphore *sem)
{
    __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_interruptible(struct semaphore *sem)
{
    return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_killable(struct semaphore *sem)
{
    return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_timeout(struct semaphore *sem, long jiffies)
{
    return __down_common(sem, TASK_UNINTERRUPTIBLE, jiffies);
}

static inline int __sched __down_common(struct semaphore *sem, long state,
                                   long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;

    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = task;
    waiter.up = 0;

    for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;
        if (timeout <= 0)
            goto timed_out;
        __set_task_state(task, state);
        spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }

 timed_out:
    list_del(&waiter.list);
    return -ETIME;

 interrupted:
    list_del(&waiter.list);
    return -EINTR;
}
```

__down_common 首先进行状态检查和时间片检查；然后将当前进程的状态设置为 UNINTERRUPTIBLE；最后进行调度，并将当前进程阻塞。

Linux 的 V 函数是 up。当调用 up 之后，调用者不再拥有该信号量。函数实现代码如下：

```
void up(struct semaphore *sem)
{
    unsigned long flags;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(list_empty(&sem->wait_list)))
        sem->count++;
    else
        __up(sem);
```

```
    spin_unlock_irqrestore(&sem->lock, flags);
}

static noinline void __sched __up(struct semaphore *sem)
{
    struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
        struct semaphore_waiter, list);
    list_del(&waiter->list);
    waiter->up = 1;
    wake_up_process(waiter->task);
}
```

## 6.3.3 信号量的使用

信号量的使用通常有以下 4 个步骤：

- DECLARE_MUTEX(sem);
- down(&sem); //获得信号量
- ...[CODE]... //临界区(被保护的资源)
- up(&sem); //释放信号量

与信号量相关的内核 API 如表 6.1 所示。

**表 6.1 信号量的 API**

| 函 数 原 型 | 功 能 |
|---|---|
| DECLARE_MUTEX(name) | 声明一个信号量 name 并初始化它的值为 0，即声明一个互斥锁 |
| DECLARE_MUTEX_LOCKED(name) | 声明一个互斥锁 name，把它的初始值配置为 0，即锁在创建时处在已锁状态 |
| sema_init (struct semaphore *sem, int val) | 初始化配置信号量的初值 |
| init_MUTEX (struct semaphore *sem) | 初始化一个互斥锁，把信号量 sem 的值设置为 1 |
| init_MUTEX_LOCKED (struct semaphore *sem) | 初始化一个互斥锁，把信号量 sem 的值设置为 0 |
| down(_*)(struct semaphore * sem) | 获得信号量 sem |
| up(struct semaphore * sem) | 释放信号量 sem，即把 sem 的值加 1，假如 sem 的值为非正数，表明有任务等待该信号量，因此唤醒这些等待者 |

除了前面提到的信号量外，Linux 内核还提供了读/写信号量。读/写信号量对访问者进行了细分，或者为读者，或者为写者。读者在保持读/写信号量期间只能对该读/写信号量保护的共享资源进行读访问。如果一个任务除了需要读，可能还需要写，那么它必须被归类为写者。在对共享资源访问之前必须先获得写者身份，当写者发现自己不需要写访问时，可以降级为读者身份。读/写信号量同时拥有的读者数不受限制，也就是说可以有任意多个读者同时拥有一个读/写信号量。它适于在读多写少的情况，在 Linux 内核中对进程的内存映像描述结构的访问就使用了读/写信号量进行保护。

一般来说，当对低开销、短期、中断上下文加锁时，优先考虑自旋锁；当对长期、持有锁需要休眠的任务时，可以优先考虑信号量；其他情况建议使用读/写信号量。读/写信号量的相关 API 如表 6.2 所示。

**表 6.2 读/写信号量的 API**

| 函 数 原 型 | 功 能 |
|---|---|
| DECLARE_RWSEM(name) | 声明一个读/写信号量 name 并对其进行初始化 |
| init_rwsem(struct rw_semaphore *sem) | 对读/写信号量 sem 进行初始化 |
| down_read(struct rw_semaphore *sem) | 读者调用该函数来得到读/写信号量 sem。该函数会导致调用者睡眠，因此只能在进程上下文中使用 |

华清远见

| | |
|---|---|
| down_read_trylock(struct rw_semaphore *sem) | 该函数类似于 down_read，只是它不会导致调用者睡眠 |
| down_write(struct rw_semaphore *sem) | 写者使用该函数来得到读/写信号量 sem，它也会导致调用者睡眠，因此只能在进程上下文中使用 |
| down_write_trylock(struct rw_semaphore *sem) | 该函数类似于 down_write，只是它不会导致调用者睡眠 |
| up_read(struct rw_semaphore *sem) | 读者使用该函数释放读/写信号量 sem。它和 down_read 或 down_read_trylock 配对使用 |
| up_write(struct rw_semaphore *sem) | 写者调用该函数释放信号量 sem。它和 down_write 或 down_write_trylock 配对使用 |
| downgrade_write(struct rw_semaphore *sem) | 该函数用于把写者降级为读者 |

通过学习前面的内容，可以掌握信号量的原理及 API，下面的代码是内核模块中对信号量机制的使用。

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/sched.h>
#include<linux/sem.h>
MODULE_LICENSE("Dual BSD/GPL");

int num[2][5]={
{0,2,4,6,8},
    {1,3,5,7,9}
};
struct semaphore sem_one;
struct semaphore sem_two;
int thread_show_one(void *);
int thread_show_two(void *);

int thread_show_one(void *p)
{
    int i;
    int *num=(int *)p;
    for(i=0;i<5;i++) {
        down(&sem_one);
        printk(KERN_INFO" Semaphore:%d\n",num[i]);
        up(&sem_two);
    }
    return 0;
}
int thread_show_two(void *p)
{
    int i;
    int *num=(int *)p;
    for(i=0;i<5;i++) {
        down(&sem_two);
        printk(KERN_INFO" Semaphore:%d\n",num[i]);
        up(&sem_one);
    }
    return 0;
}
static int semdemo _init(void)
{
    init_MUTEX(&sem_one);
    init_MUTEX(&sem_two);
    kernel_thread(thread_show_one,num[0],CLONE_KERNEL);
    kernel_thread(thread_show_two,num[1],CLONE_KERNEL);
    return 0;
}
```

```
static void semdemo _exit(void)
{
    printk(KERN_ALERT" semdemo module quit\n");
}
module_init(semdemo_init);
module_exit(semdemo _exit);
```

# 6.4 共享内存

## 6.4.1 什么是共享内存

共享内存是最快的 IPC 形式。两个不同进程 A、B 共享内存的意思是，同一块物理内存被映射到进程 A、B 各自的进程地址空间。进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读/写内存，而不需要任何数据的复制。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行 4 次复制数据，而共享内存则只复制两次数据：一次是从输入文件到共享内存区，另一次是从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读/写少量数据后就解除映射，有新的通信时，会重新建立共享内存区域。而且是保持共享区域，直到通信完毕为止。这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

在 Linux 操作系统中，内核做两件事：一件是在内存中规划出一块区域来作为共享区；另一件就是把这个区域映射到参与通信的各个进程空间。具体做法是把一个已打开的文件所占用的内存空间作为共享区，然后通过调用 mmap()把这块区域映射到各个进程地址空间，从而使用户进程都可以看到这个共享区域。

mmap 原型如下：

```
void *mmap(void * start, size_t len, int prot, int flags, int fd, off_t offset);
```

其中，参数 fd 用来指定被映射的文件；offset 指定映射的起始位置偏移量；len 指定文件被映射部分的长度；start 用来指定映射到虚拟地址空间的起始地址。

华清远见

## 6.4.2 共享内存的内核实现

通常，一个共享内存区由多个共享段组成，用来描述一个共享内存段的内核数据结构是 shmid_kernel，这个结构在 include\linux\shm.h 中定义：

```
struct shmid_kernel /* private to the kernel */
{
     struct kern_ipc_perm  shm_perm;
     struct file *     shm_file;
     unsigned long     shm_nattch;
     unsigned long     shm_segsz;
     time_t                shm_atim;
     time_t                shm_dtim;
     time_t                shm_ctim;
     pid_t                 shm_cprid;
     pid_t                 shm_lprid;
     struct user_struct    *mlock_user;
};
```

SHMID_KERNEL 数据结构成员说明如表 6.3 所示。

表 6.3 SHMID_KERNEL 数据结构成员说明

| 成　员 | 描　述 |
|---|---|
| shm_perm | 描述进程间通信许可的结构 |
| shm_file | 指向共享内存页交换文件对象指针 |
| shm_nattch | 挂接到本段共享内存的进程数 |
| shm_segsz | 段大小 |
| shm_atim | 最后挂接时间 |
| shm_dtim | 最后解除挂接时间 |
| shm_ctim | 最后变化时间 |
| shm_cprid | 创建进程的 PID |
| shm_lprid | 最后使用进程的 PID |
| *mlock_user | 指向上锁的用户 |

### 6.4.3 共享内存的使用

内核为共享内存机制提供了 4 种操作：SHMGET、SHMAT、SHMDT 和 SHMCTL，它们各自对应库函数 shmget()、shmat()、shmdt()和 shmctl()，其系统调用分别由 sys_shmget()、sys_shmat()、sys_shmdt()、sys_shmctl()实现。

进行应用程序编程时，需要包含下面几个头文件：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

#### 1．共享内存的打开和创建

shmget()用来打开或创建一个共享内存区。这个函数的内部是通过系统调用 SYSCALL_DEFINE3 实现的。代码如下（见 linux 源代码目录下 ipc\shm.c）：

```
SYSCALL_DEFINE3(shmget, key_t, key, size_t, size, int, shmflg)
{
    struct ipc_namespace *ns;
    struct ipc_ops shm_ops;
    struct ipc_params shm_params;

    ns = current->nsproxy->ipc_ns;

    shm_ops.getnew = newseg;
    shm_ops.associate = shm_security;
    shm_ops.more_checks = shm_more_checks;

    shm_params.key = key;
    shm_params.flg = shmflg;
    shm_params.u.size = size;

    return ipcget(ns, &shm_ids(ns), &shm_ops, &shm_params);
}
```

参数 key 是用户给定的键值。参数 shmflg 是该函数的功能标志。如果 shmflg 的 IPC_CREATE=1，则这个系统调用会为用户创建或打开一个共享内存区，并返回其标识号；如果 IPC_CREATE=0，则会在系统已有的共享内存区中寻找与键值相同的共享内存区，找到后返回它的标识号并打开它。

这里用到了两个结构：ipc_params 和 ipc_ops，其实现都在 ipc\util.h 中。代码如下：

```
struct ipc_params {
    key_t key;
    int flg;
    union {
```

```
            size_t size;  /* for shared memories */
            int nsems;    /* for semaphores */
        } u;              /* holds the getnew() specific param */
};

struct ipc_ops {
        int (*getnew) (struct ipc_namespace *, struct ipc_params *);
        int (*associate) (struct kern_ipc_perm *, int);
        int (*more_checks) (struct kern_ipc_perm *, struct ipc_params *);
};
```

2. 共享内存与进程的链接

如果一个进程已创建或打开了一个共享内存，则在需要使用它时，要调用函数 shmat()把该共享内存链接到进程上，即要把待使用的共享内存映射到进程空间。函数 shmat()通过系统调用 SYSCALL_DEFINE3()实现，代码如下（见 ipc\shm.c）：

```
SYSCALL_DEFINE3(shmat, int, shmid, char __user *, shmaddr, int, shmflg)
{
        unsigned long ret;
        long err;

        err = do_shmat(shmid, shmaddr, shmflg, &ret);
        if (err)
                return err;
        force_successful_syscall_return();
        return (long)ret;
}
```

其中，shmid 是共享内存的标识；参数 shmaddr 是映射地址，如果该地址为 0，则由内核决定；参数 shmflg 为共享内存的标识，如果 shmflg 的值为 SHM_RDONLY，则进程以只读的方式访问共享内存，否则，以读/写的方式访问共享内存。

3. 断开共享内存与进程的链接

调用函数 shmdt()可以断开共享内存与进程的链接。该函数是由系统调用 SYSCALL_DEFINE1 实现的，代码如下：

```
SYSCALL_DEFINE1(shmdt, char __user *, shmaddr){
        struct mm_struct *mm = current->mm;
        struct vm_area_struct *vma, *next;
        unsigned long addr = (unsigned long)shmaddr;
        loff_t size = 0;
        int retval = -EINVAL;
        if (addr & ~PAGE_MASK)
                return retval;
        down_write(&mm->mmap_sem);
        vma = find_vma(mm, addr);
        while (vma) {
                next = vma->vm_next;
                if ((vma->vm_ops == &shm_vm_ops) &&
                        (vma->vm_start - addr)/PAGE_SIZE == vma->vm_pgoff) {


                        size = vma->vm_file->f_path.dentry->d_inode->i_size;
                        do_munmap(mm, vma->vm_start, vma->vm_end - vma->vm_start);
                        retval = 0;
                        vma = next;
                        break;
                }
                vma = next;
        }
        size = PAGE_ALIGN(size);
        while (vma && (loff_t)(vma->vm_end - addr) <= size) {
                next = vma->vm_next;
                /* finding a matching vma now does not alter retval */
                if ((vma->vm_ops == &shm_vm_ops) &&
```

```
            (vma->vm_start - addr)/PAGE_SIZE == vma->vm_pgoff)
            do_munmap(mm, vma->vm_start, vma->vm_end - vma->vm_start);
        vma = next;
    }

    up_write(&mm->mmap_sem);
    return retval;
}
```

唯一的一个参数 shmaddr 是链接地址。

# 6.5 消息队列

## 6.5.1 什么是消息队列

消息队列是系统定义的内存块，用于临时存储消息。消息队列就是一个消息的链表。可以把消息看做一个记录，具有特定的格式及特定的优先级。对消息队列有写权限的进程可以按照一定的规则为其添加新消息；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。目前，主要有两种类型的消息队列：POSIX 消息队列和系统 V 消息队列，系统 V 消息队列目前被大量使用。考虑到程序的可移植性，新开发的应用程序应尽量使用 POSIX 消息队列。

消息队列的工作机制如图 6.2 所示。

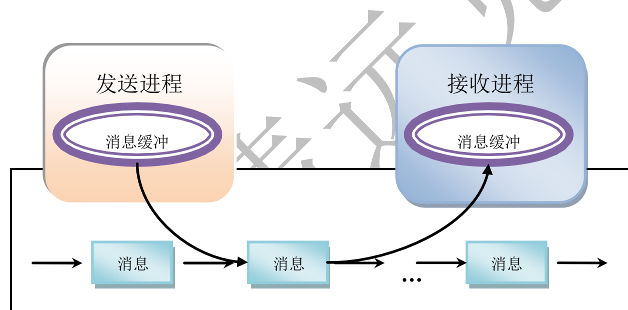



图 6.2 消息队列的工作机制

## 6.5.2 消息队列的内核实现

消息队列主要用来实现消息传递，因此我们需要掌握消息及消息队列的结构。内核分别用 msgbuf、msg_msgseg、msg_queue 描述用户空间的消息缓冲区、内核空间的消息结构和消息队列结构。

对于开发人员来说，用户空间的每个消息都类似如下数据结构：

```
struct msgbuf
{
    long mtype;
    char mtext[1];
};
```

mtype 成员代表消息类型，从消息队列中读取消息的一个重要依据就是消息的类型；mtext 是消息内容，当然长度不一定为 1。因此，对于发送消息来说，首先预置一个 msgbuf 缓冲区并写入消息类型和内容，调用相应的发送函数即可；对于读取消息来说，首先分配这样一个 msgbuf 缓冲区，然后把消息读入该缓冲区即可。

msg_msgseg 结构的定义在 ipc\msgutil.c 文件中，代码如下：

```
struct msg_msgseg {
    struct msg_msgseg* next;
    /* the next part of the message follows immediately */
```

```
};
```

一个大型消息的结构图如图 6.3 所示。

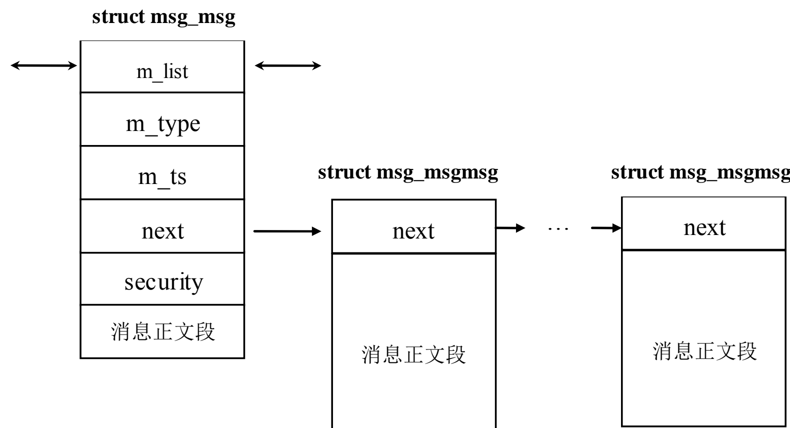



图 6.3 大型消息的结构图

其中，msg_msg 是一个稍微复杂的结构。为什么会设计这样一个结构呢？这是出于对消息的维护和管理的考虑。msg_msg 称为内核空间的首页结构，而 msg_msgmsg 称为一般页结构。msg_msg 的定义如下：

```
/* one msg_msg structure for each message */
struct msg_msg {
    struct list_head m_list;
    long  m_type;
    int m_ts;          /* message text size */
    struct msg_msgseg* next;
    void *security;
    /* the actual message follows immediately */
};
```

其中，m_list 表示链表结构；m_type 是消息类型；m_ts 是消息文本大小；next 表示下一个消息片段页。

另一个重要的结构是消息队列结构 msg_queue。每个消息队列都有一个 msg_queue 结构类型的队列头，msg_queue 的定义如下：

```
/* one msq_queue structure for each present queue on the system */
struct msg_queue {
    struct kern_ipc_perm q_perm;
    time_t q_stime;           /* last msgsnd time */
    time_t q_rtime;           /* last msgrcv time */
    time_t q_ctime;           /* last change time */
    unsigned long q_cbytes;   /* current number of bytes on queue */
    unsigned long q_qnum; /* number of messages in queue */
    unsigned long q_qbytes;   /* max number of bytes on queue */
    pid_t q_lspid;            /* pid of last msgsnd */
    pid_t q_lrpid;            /* last receive pid */

    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
};
```

其中，q_messages 是消息队列；q_receivers 是接收信号进程等待队列；q_senders 是发送信号进程等待队列。这些队列都是链表结构。

## 6.5.3 消息队列的使用

消息队列的内核持续性要求每个消息队列都在系统范围内对应唯一的键值，所以，要获得一个消息队列的描述符，只需提供该消息队列的键值即可。消息队列的主要调用有以下 4 个。

- msgget：调用者提供一个消息队列的键值，当这个消息队列存在时，这个消息调用负责返回这个队列的标识号；如果这个队列不存在，就创建一个消息队列，然后返回这个消息队列的标识号，

由 sys_msgget 执行。

- msgsnd：向一个消息队列发送一个消息，由 sys_msgsnd 执行。
- msgrcv：从一个消息队列中收到一个消息，由 sys_msgrcv 执行。
- msgctl：在消息队列上执行指定的操作。根据参数的不同和权限的不同，可以执行检索、删除等操作，由 sys_msgctl 执行。

这几个函数在使用时需要包括以下 3 个头文件：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

进程创建一个消息队列的函数是 msgget()，其对应的系统调用是 sys_msgget()，实现代码如下（见 ipc\msg.c）：

```
SYSCALL_DEFINE2(ms gget, key_t, key, int, msgflg){
    struct ipc_namespace *ns;
    struct ipc_ops msg_ops;
    struct ipc_params msg_params;
    ns = current->nsproxy->ipc_ns;
    msg_ops.getnew = newque;
    msg_ops.associate = msg_security;
    msg_ops.more_checks = NULL;
    msg_params.key = key;
    msg_params.flg = msgflg;
    return ipcget(ns, &msg_ids(ns), &msg_ops, &msg_params);
}
```

其中，参数 key 是用户给定的键值。如果 key=0，系统会为进程创建一个进程自用的消息队列；否则，创建或打开一个消息队列，msgflg 是该函数的功能标识。

类似的，消息的发送和接收分别是 msgsnd 和 msgrcv，所对应的系统调用是 sys_msgsnd 和 sys_msgrcv，其实现代码如下（见 ipc\msg.c）：

```
asmlinkage long
sys_msgsnd(int msqid, struct msgbuf __user *msgp, size_t msgsz, int msgflg)
{
    long mtype;

    if (get_user(mtype, &msgp->mtype))
        return -EFAULT;
    return do_msgsnd(msqid, mtype, msgp->mtext, msgsz, msgflg);
}

asmlinkage long sys_msgrcv(int msqid, struct msgbuf __user *msgp, size_t msgsz,long msgtyp, int
msgflg)
{
    long err, mtype;

    err =  do_msgrcv(msqid, &mtype, msgp->mtext, msgsz, msgtyp, msgflg);
    if (err < 0)
        goto out;

    if (put_user(mtype, &msgp->mtype))
        err = -EFAULT;
out:
    return err;
}
```

这两个函数分别调用了 do_msgsnd 和 do_msgrcv，程序代码留给读者自行研究。消息队列应用相对较简单，下面的实例基本上覆盖了对消息队列的所有操作，包括创建、发送、读取、改变权限及删除消息队列等。

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

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_SEND_SIZE 256
struct mymsgbuf {
long mtype;
char mtext[MAX_SEND_SIZE];
};
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
int main(int argc, char *argv[])
{
    key_t key;
    int msgqueue_id;
    struct mymsgbuf qbuf;
    if(argc == 1)
    usage();
    key = ftok(".", 'm'); /*创建 IPC 标识符，需要先获得一个 key */
    if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1)
/*如果 msg_id 不存在，则创建；否则，返回已经存在的队列标识符*/
    {
        perror("msgget");
        exit(1);
    }
    switch(tolower(argv[1][0]))
    {
    case 's':
        send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,atol(argv[2]),
        argv[3]);
        break;
    case 'r':
        read_message(msgqueue_id, &qbuf, atol(argv[2]));
        break;
    case 'd':
        remove_queue(msgqueue_id);
        break;
    case 'm':
        change_queue_mode(msgqueue_id, argv[2]);
        break;
    default: usage();
    }
    return(0);
}
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
    printf("正在发送消息 …n");
    qbuf->mtype = type;
    strcpy(qbuf->mtext, text);
    if((msgsnd(qid, (struct msgbuf *)qbuf,strlen(qbuf->mtext)+1, 0)) ==-1)
/* 向队列发生消息 */
    {
        perror("msgsnd");
        exit(1);
    }
}
void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
    printf("读取消息中…n");
    qbuf->mtype = type;
    /* 读取来自队列的消息*/
    msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
    printf("Type: %ld Text: %sn", qbuf->mtype, qbuf->mtext);
}
void remove_queue(int qid)
```

```
{
    msgctl(qid, IPC_RMID, 0); /* Remove the queue */
}
void change_queue_mode(int qid, char *mode)
{
    struct msqid_ds myqueue_ds;
    msgctl(qid, IPC_STAT, &myqueue_ds); /*获得消息队列信息*/
    /* Convert and load the mode */
    sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);
    msgctl(qid, IPC_SET, &myqueue_ds); /* Update the mode */
}
void usage(void)
{
    fprintf(stderr, "msgtool - A utility for tinkering with msg queuesn");
    fprintf(stderr, "nUSAGE: msgtool (s)end n");
    fprintf(stderr, " (r)ecv n");
    fprintf(stderr, " (d)eleten");
    fprintf(stderr, " (m)ode n");
    exit(1);
}
```

# 6.6 管道

## 6.6.1 什么是管道

管道是 Linux 从 UNIX 系统继承过来的 IPC 机制，也是 UNIX 早期的一个重要通信机制。其思想是在内存中创建一个共享文件，从而使通信双方利用这个文件进行信息传递。因为这种方式具有单向传送的特点，所以这个作为传递信息的共享文件就称为管道。管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道。管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是单独构成一种文件系统，并且只存在于内存中。

如图 6.4 所示为两个进程通过管道进程通信的示意图。显然，如果两个或多个进程同时对一个进程进行读/写，那么这些进程必须使用锁机制或信号量机制对其进行同步。当管道空闲时，在有数据写入之前，读进程一直被阻塞；同样，当管道满时，在其中数据被读出之前，写进程将发生阻塞。

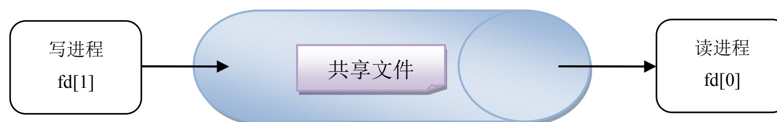



图 6.4 管道原理示意图

在管道的实现中，根据通信使用的文件是否具有名称，可分为匿名管道和命名管道两种。匿名管道没有名字，所以只能提供给进程家族中的父子进程间通信使用。命名管道又称 FIFO（先进先出），是一个能在互不相关进程之间传送数据的特殊文件。它是在实际文件系统的基础上实现的一种通信机制。

## 6.6.2 管道的内核实现

管道是内核为需要进行通信的进程之间铺设的一个共享物理页，用来描述这个物理页属性的数据结构是 pipe_inode_info（见 include\linux\pipe_fs_i.h），代码如下：

```
struct pipe_inode_info {
    wait_queue_head_t wait;
    unsigned int nrbufs, curbuf;
    struct page *tmp_page;
```

```
        unsigned int readers;
        unsigned int writers;
        unsigned int waiting_writers;
        unsigned int r_counter;
        unsigned int w_counter;
        struct fasync_struct *fasync_readers;
        struct fasync_struct *fasync_writers;
        struct inode *inode;
        struct pipe_buffer bufs[PIPE_BUFFERS];
    };
```

pipe_inode_info 数据结构成员说明如表 6.4 所示。

**表 6.4　pipe_inode_info 数据结构成员说明**

| 成　　员 | 描　　述 |
|---|---|
| wait | 描述管道的等待队列 |
| nrbufs, curbuf | 包含待读数据的缓冲区数和包含待读数据的第一个缓冲区的索引 |
| tmp_page | 高速缓存页框指针 |
| readers | 读进程的编号 |
| writers | 写进程的编号 |
| waiting_writers | 被阻塞的写进程计数器 |
| r_counter | 以只读方式访问管道的进程计数器 |
| w_counter | 以只写方式访问管道的进程计数器 |
| fasync_readers | 用于通过信号进行读的异步 I/O 通知 |
| fasync_writers | 用于通过信号进行写的异步 I/O 通知 |
| Inode | 管道文件的节点 |
| bufs[PIPE_BUFFERS] | 缓冲区数组 |

其中，bufs[PIPE_BUFFERS]是构成管道的内存缓冲区，该缓冲区通过 pipe_buffer 结构进行描述，代码如下：

```
    struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
    };
```

可以看出，管道实质上是一个被当做文件来管理的内存缓冲区。结构中的 page 是管道缓冲区页框的描述符地址，offset 是页框内有效数据的当前位置，ops 是缓冲区的操作函数指针，结构如下：

```
    struct pipe_buf_operations {
        int can_merge;
        void * (*map)(struct pipe_inode_info *, struct pipe_buffer *, int);
        void (*unmap)(struct pipe_inode_info *, struct pipe_buffer *, void *);
        int (*confirm)(struct pipe_inode_info *, struct pipe_buffer *);
        void (*release)(struct pipe_inode_info *, struct pipe_buffer *);
        int (*steal)(struct pipe_inode_info *, struct pipe_buffer *);
        void (*get)(struct pipe_inode_info *, struct pipe_buffer *);
    };
```

### 6.6.3　管道的读/写规则

管道两端可分别用描述字 fd[0]和 fd[1]来描述，需要注意的是，管道的两端是固定了任务的。即一端只能用于读，由描述字 fd[0]表示，称为管道读端；另一端则只能用于写，由描述字 fd[1]来表示，称为管

华清远见

道写端。如果试图从管道写端读取数据或者向管道读端写入数据，都将导致错误发生。一般文件的 I/O 函数都可以用于管道，如 close、read、write 等。

### 1．从管道中读取数据

如果管道的写端不存在，则认为已经读到了数据的末尾，读函数返回的读出字节数为 0。

当管道的写端存在时，如果请求的字节数大于 PIPE_BUF，则返回管道中现有的数据字节数；如果请求的字节数不大于 PIPE_BUF，则返回管道中现有的数据量小于请求的数据量字节数；或者返回请求的字节数。

### 2．向管道中写入数据

向管道中写入数据时，Linux 将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读出管道缓冲区中的数据，那么写操作将一直阻塞。

## 6.7 本章习题

1．和进程管理相关的内核文件都有哪些？
2．什么是进程和线程？
3．什么是进程描述符？怎样得到当前进程的进程描述符？
4．进程的内核栈有多大？
5．进程的状态都有哪些？在什么情况下发生转化？
6．Linux 中所有进程之间的关系是怎么样的？
7．什么是命名管道和匿名管道？它们的区别是什么？

华清远见



## 联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路银海大厦 A 座 8 层，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218