Operating Systems

Chapter 5 Mutual Exclusion(互斥) and Synchronization(同步)

Agenda

- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

5.5Message Passing(1/9)

- Enforce mutual exclusion
- Exchange information

```
send (destination, message)
receive (source, message)
```

5.5Message Passing(2/9)

- Synchronization
  - Sender and receiver may or may not be blocking
  - Blocking send, blocking receive
    - Both sender and receiver are blocked until message is delivered

5.5Message Passing(3/9)

- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither part is required to wait

5.5Message Passing(4/9)

- Direct addressing
  - Send primitive includes a specific **identifier** of the destination process
  - Receive primitive could know ahead of time which process a message is expected or receive primitive could use source parameter to return a value when the receive operation has been performed

5.5Message Passing(5/9)

- Indirect addressing
  - Messages are sent to a shared data structure consisting of queues
  - Queues are called mailboxes
  - One process sends a message to the mailbox and the other process picks up the message from the mailbox

5.5Message Passing(6/9)

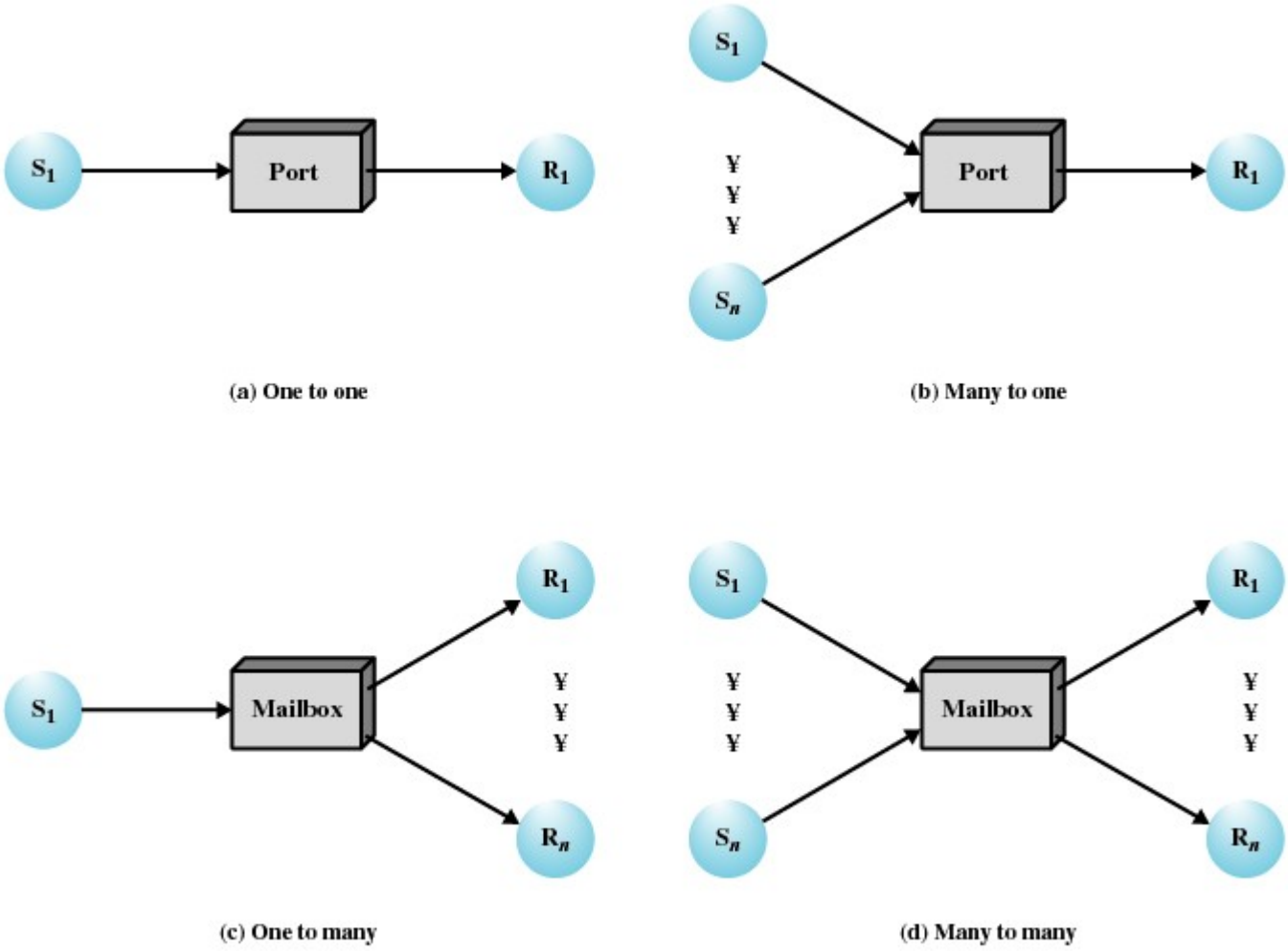


Figure 5.18 Indirect Process Communication

5.5Message Passing(7/9)

Message Format

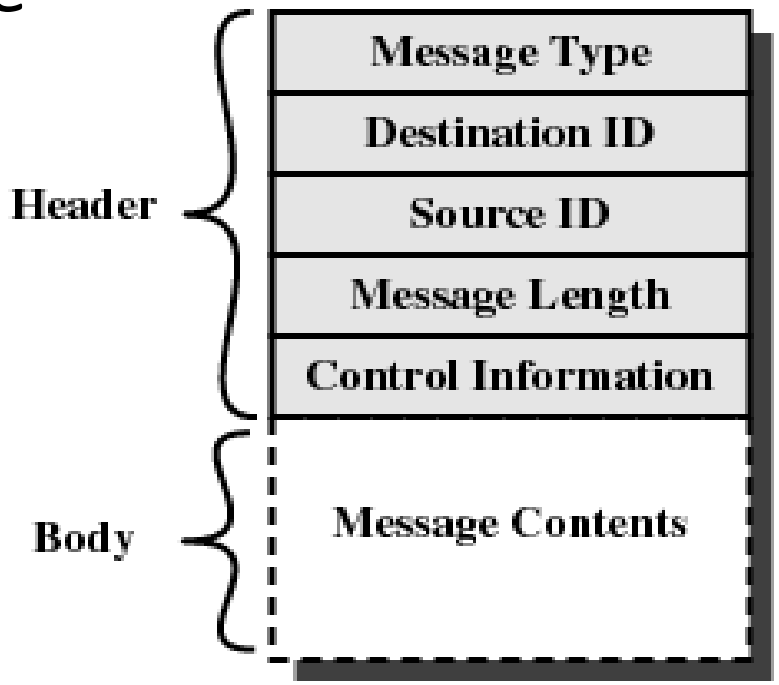


Figure 5.19 General Message Format

5.5Message Passing(8/9)

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true)
    {
     receive (mutex, msg);
     /* critical section   */;
     send (mutex, msg);
     /* remainder   */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

5.5Message Passing(9/9)

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)
    {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Figure 5.21 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Linux message queue demo Lab07 4.1

Linux-message 案例 .pptx

Agenda

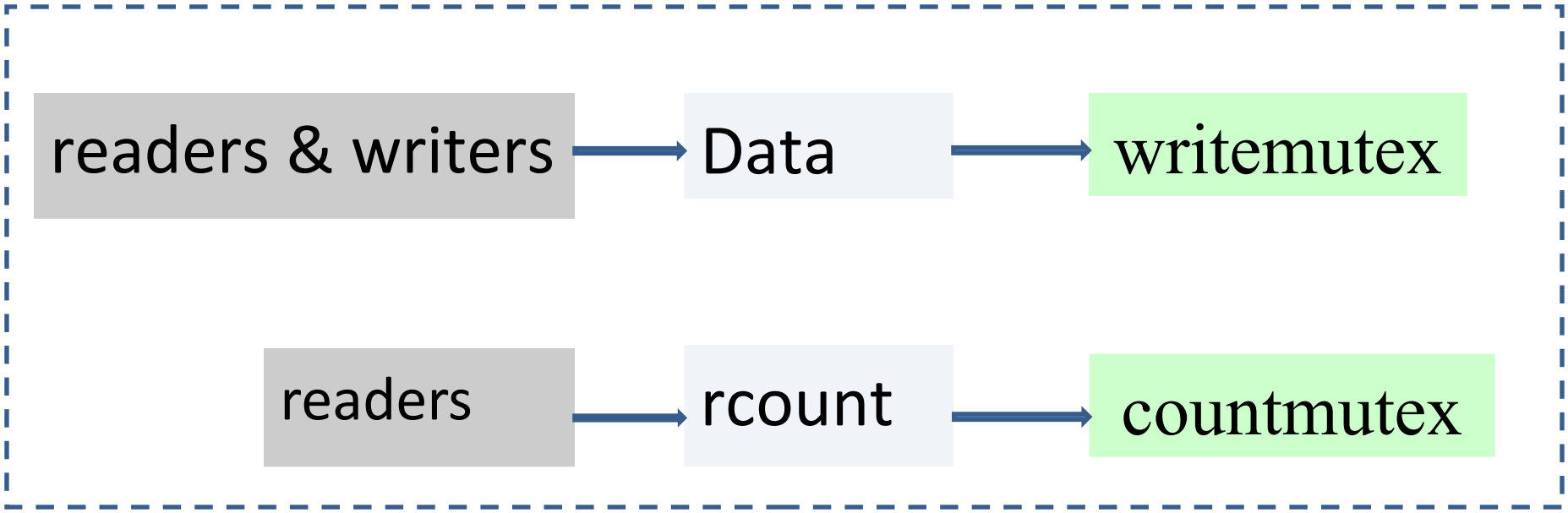
- 5.1 Principles of Concurrency
- 5.2 Mutual Exclusion: Hardware Support
- 5.3 Semaphores
- 5.4 Monitors
- 5.5 Message Passing
- 5.6 Readers/Writers Problem
- 5.7 Summary

5.6 Readers/Writers Problem(1/9)

- Any number of readers may simultaneously read the file(no writer)
  - 如何知道有其它读者存在？
- Only one writer at a time may write to the file(no other writers)
  - 如何知道有写者存在？
- If a writer is writing to the file, no reader may read it

- Summary
  - Read&write, write&write need mutual exclusion
  - Read&read doesn’t need mutual exclusion
  - the number of reader: need mutual exclusion

5.6 Readers/Writers Problem(2/9)



sem writemutex= ? , countmutex= ? ;

Semaphore → mutual exclusion Set to 1

5.6 Readers/Writers Problem(3/9)

- Reader First
  - Once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader in the act of reading. Therefore, writers are subject to starvation. 目前正在读，新的读会持续进入，写会饥饿

- Writer First
  - no new readers are allowed access to the data area once at least one writer has declared a desire to write. 当有新的写存在，新的读不能进行读

5.6 Readers/Writers Problem(4/9)

- Reader First
  - Reader:
    - 当前在写：等待写结束 (不能进入) semWait(writemutex)
    - 当前正在读：新的读进入， if(rcount>0) , rcount++
  - Writer
    - 当前在写：等待写结束 (不能进入) semWait(writemutex)
    - 当前正在读：等待读结束 (不能进入) semWait(writemutex)

5.6 Readers/Writers Problem (5/9)

Reader & Writer race to enter

sem writemutex=1, countmutex=1;

```
void* writer(void *arg){
  while(1){

      semWait(writemutex)
      writedata();
      semSignal(writemutex)

  }}
```

```
void* reader(void *arg){
  while(1){

      semWait(writemutex)

      readdata();

      semSignal(writemutex)
  }}
```

5.6 Readers/Writers Problem (6/9)

Reader First

sem writemutex=1, countmutex=1;

```
void* writer(void *arg){
  while(1){

      semWait(writemutex)
      writedata();
      semSignal(writemutex)

  }}
```

```
void* reader(void *arg){
  while(1){
   semWait(countmutex)
   rcount++
   if(rcount==1)
        semWait(writemutex)
   semSignal(countmutex)
   readdata();
   semWait(countmutex)
   rcount--
   if(rcount == 0)
        semSignal(writemutex)
   semSignal(countmutex)
  }}
```

sem writemutex=1, mutex1=1,mutex2=1;

Writer First

```
void* writer(void *arg){
  while(1){

    semWait(mutex2)
     wcount++
    semSignal(mutex2)

     semWait(writemutex)
     writedata();
     semSignal(writemutex)

     semWait(mutex2)
       wcount--
     semSignal(mutex2)

  }}
```

```
void* reader(void *arg){
   while(1){

     semWait(mutex1)
    rcount++
    if(rcount==1)
          semWait(writemutex)
     semSignal(mutex1)
     readdata();
     semWait(mutex1)
     rcount--
     if(rcount == 0)
          semSignal(writemutex)
      semSignal(mutex1)

   }}
```

sem writemutex=1,readmutex=1**, mutex1=1,mutex2=1**;

```
void* writer(void *arg){
  while(1){
      semWait(mutex2)
      wcount++
      if(wcount==1)
            semWait(readmutex)
      semSignal(mutex2)
      semWait(writemutex)
      writedata();
      semSignal(writemutex)
      semWait(mutex2)
      wcount--
      if(wcount==0)
            semSignal(readmutex)
      semSignal(mutex2)
  }}
```

```
void* reader(void *arg){
  while(1){
      semWait(readmutex)
      semWait(mutex1)
      rcount++
      if(rcount==1)
            semWait(writemutex)
      semSignal(mutex1)
      semSignal(readmutex)
      readdata();
      semWait(mutex1)
      rcount--
      if(rcount == 0)
            semSignal(writemutex)
      semSignal(mutex1)
  }}
```

Writer First

sem writemutex=1,readmutex=1**, z=1,mutex1=1,mutex2=1**;

```
void* writer(void *arg){
  while(1){
      semWait(mutex2)
      wcount++
      if(wcount==1)
            semWait(readmutex)
      semSignal(mutex2)

      semWait(writemutex)
      writedata();
      semSignal(writemutex)
      semWait(mutex2)
      wcount--
      if(wcount==0)
            semSignal(readmutex)
      semSignal(mutex2)
  }}
```

```
void* reader(void *arg){
  while(1){
      semWait(z)
      semWait(readmutex)
      semWait(mutex1)
      rcount++
      if(rcount==1)
            semWait(writemutex)
      semSignal(mutex1)
      semSignal(readmutex)
      semSignal(z)
      readdata();
      semWait(mutex1)
      rcount--
      if(rcount == 0)
            semSignal(writemutex)
      semSignal(mutex1)
  }}
```

Writer First

sem writemutex=1,readmutex=1**, z=1,mutex1=1,mutex2=1**;

```
void* writer(void *arg){
  while(1){
      semWait(mutex2)
      wcount++
      if(wcount==1)
            semWait(readmutex)
      semSignal(mutex2)

      semWait(writemutex)
      writedata();
      semSignal(writemutex)
      semWait(mutex2)
      wcount--
      if(wcount==0)
            semSignal(readmutex)
      semSignal(mutex2)
  }}
```

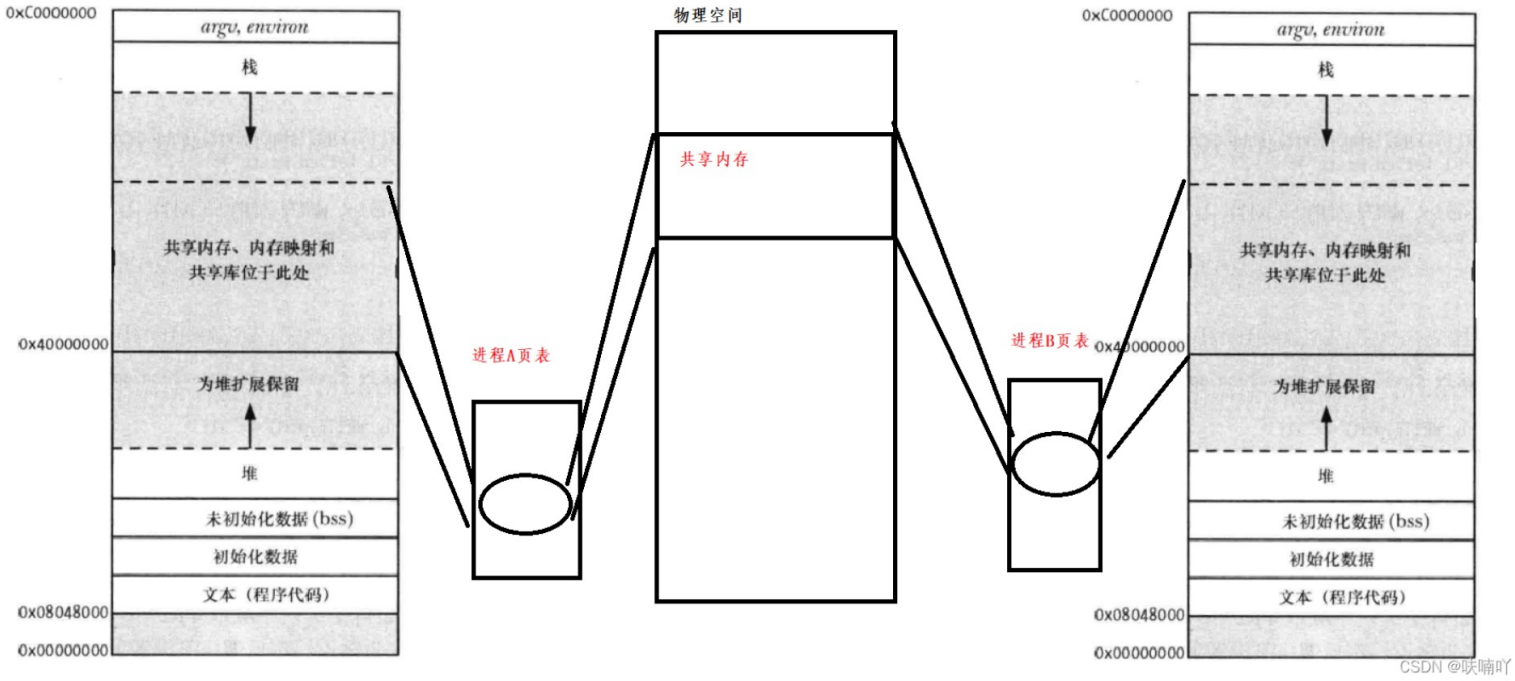
```
void* reader(void *arg){
  while(1){
      semWait(mutex1)
      semWait(readmutex)
      rcount++
      if(rcount==1)
            semWait(writemutex)
      semSignal(readmutex)
      semSignal(mutex1)

      readdata();
      semWait(mutex1)
      rcount--
      if(rcount == 0)
            semSignal(writemutex)
      semSignal(mutex1)
  }}
```

Writer First

扩展 1 ：共享内存 (1/3)

- 进程通信
  - 消息
  - 共享内存



扩展 1 ：共享内存 (2/3)

- OpenEuler 支持 POSIX 和 System V 共享内存
  - 4 个 API:
    - shmget() 创建
    - shmat() 映射入进程地址
    - shmdt() 解除映射
    - shmctl() 销毁
  - 与消息机制不同，需要注意同步与互斥

扩展 1 ：共享内存 (3/3)

- OpenEuler 支持 POSIX 和 System V 共享内存
  - 示范代码： clientlock.cpp serverlock.cpp
    - 仅互斥，未同步
    - 互斥锁需要写入共享内存

```
root@zh-VirtualBox:/home/zh/lab/lab06/sharedmem# ./serverlock
attaches shared_memory success!
client sent to server:What are you doing?
client sent to server:Hello
client sent to server:Where are you?
client sent to server:What are you doing?
client sent to server:Where are you?
client sent to server:Hello
client sent to server:Where are you?
client sent to server:Hello
client sent to server:Hello
deatach shared_memory success!
root@zh-VirtualBox:/home/zh/lab/lab06/sharedmem#
```

```
server answers client:In a server.
root@zh-VirtualBox:/home/zh/lab/lab06/sharedmem# ./clientlock
server answers client:Bye.
server answers client:I'm answering your question.
server answers client:Hi,client.
server answers client:In a server.
server answers client:I'm answering your question.
server answers client:In a server.
server answers client:Hi,client.
server answers client:In a server.
server answers client:Hi,client.
server answers client:Hi,client.
root@zh-VirtualBox:/home/zh/lab/lab06/sharedmem#
```

扩展 2 ：内存屏障 (1/4)

- 鲲鹏 CPU ， OpenEuler 中的内存屏障实现 OpenEuler

  – 问题提出：现代 CPU 支持乱序执行，且具有多核

```
a=1;
b=2;
c=3;
flag=true;
while(flag){
        do_something(a,b,c);
        return;
}
```

扩展 2 ：内存屏障 (2/4)

- 鲲鹏 CPU ， OpenEuler 中的内存屏障实现

```
CPU0
a=1;
b=2;
c=3;
flag=true;
```

```
CPU1
while(flag){
        do_something(a,b,c);
        return;
}
```

```
CPU0
a=1;
b=2;
flag=true;
c=3;
```

```
CPU1
while(flag){
        do_something(a,b,c);
        return;
}
```

扩展 2 ：内存屏障 (3/4)

- 鲲鹏 CPU ， OpenEuler 中的内存屏障实现

```
CPU0
a=1;
b=2;
c=3;
memory_barrier()
flag=true;
```

```
CPU1
ready=flag
memory_barrier()
while(ready){
        do_something(a,b,c);
        return;
}
```

扩展 2 ：内存屏障 (4/4)

- 鲲鹏 CPU ， OpenEuler 中的内存屏障实现

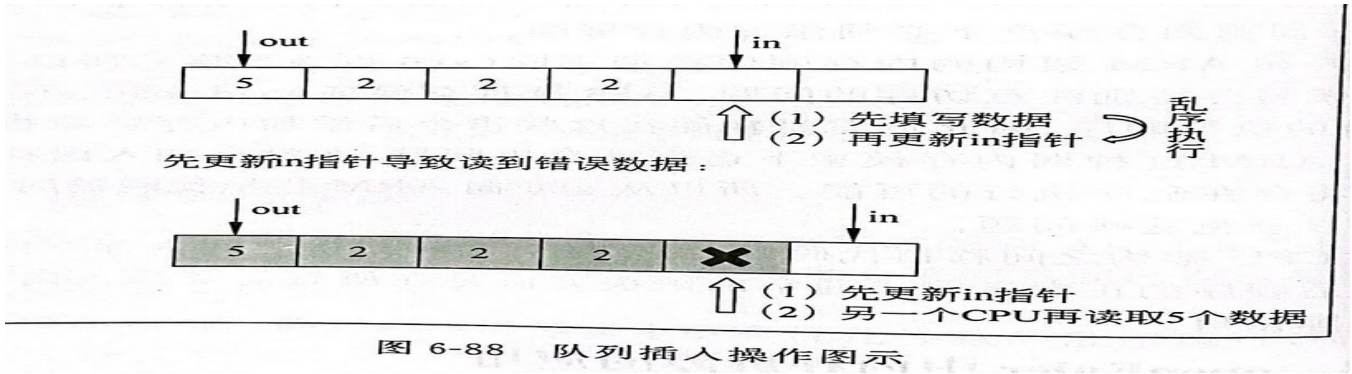


图 6-88　队列插入操作图示

```
1.    //源代码：kernel/lib/kfifo.c
2.    static void kfifo_copy_in(struct __kfifo * fifo, const void * src,
3.                                        unsigned int len, unsinged int off) {
4.      ...
5.      memcpy(fifo -> data + off, src, l);          //向队列写入数据
6.      memcpy(fifo -> data, src + l, len - l);
7.      smp_wmb();     //插入写内存屏障,防止数据完全写入前更新 in 指针
8.    }
9.    //源代码：kernel/lib/kfifo.c
10.   //入队
11.   unsigned int __kfifo_in(struct __kfifo * fifo,
12.                                   const void * buf, unsigned int len) {
13.     ...
14.     kfifo_copy_in(fifo, buf, len, fifo -> in);   //向队列中写入数据
15.     fifo -> in += len;                           //更新入队位置
16.     ...
17.   }
```

图 6-89　kfifo 部分源码