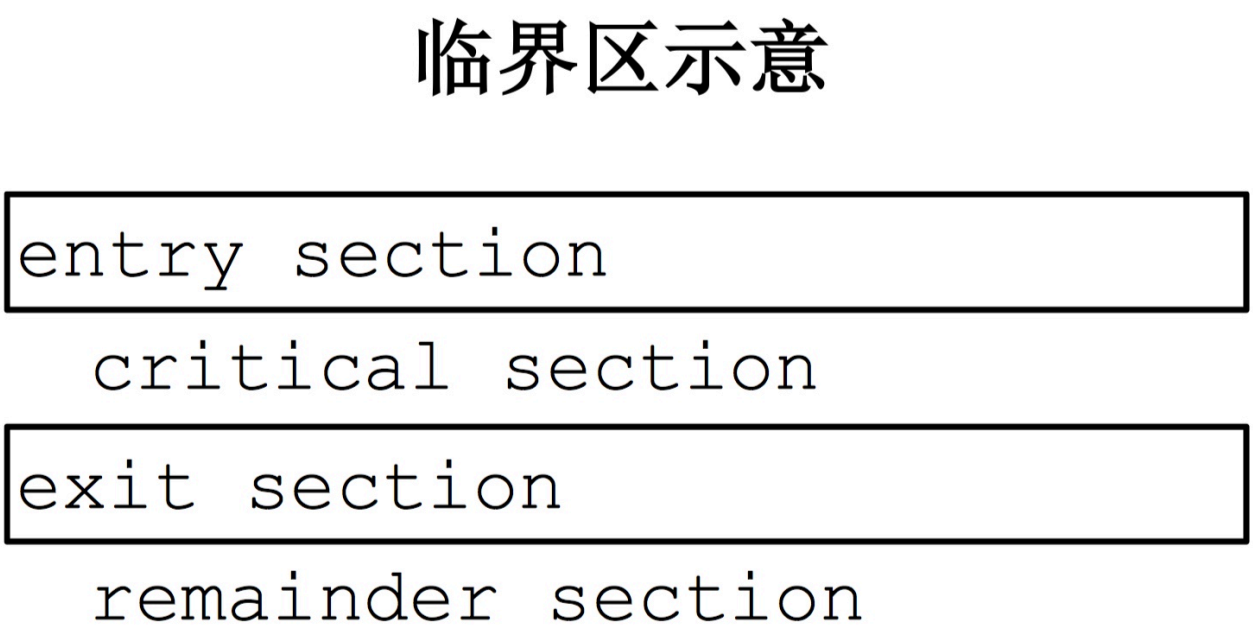
第五章 并发性：互斥和同步

1. **并发的难点**
    - 全局资源共享
    - 操作系统很难对资源分配进行最优管理
    - 很难定位程序设计错误
2. **操作系统的任务**
    - 记住所有的进程
    - 为进程分配和回收资源
        - **处理器时间**
        - **存储器**
        - **文件**
        - **I/O 设备**
    - 保护进程的数据和资源
    - 进程的运行结果必须与其它的进程无关
3. **进程的交互关系：可以按照相互感知的程度来分类**

相互感知的程度	交互关系	一个进程对其他进程的影响	潜在的控制问题
相互不感知(完全不了解其它进程的存在)	竞争(competition)	一个进程的操作对其他进程的结果无影响	互斥，死锁（可释放的资源），饥饿
间接感知(双方都与第三方交互，如共享资源)	通过共享进行协作	一个进程的结果依赖于从其他进程获得的信息	互斥，死锁（可释放的资源），饥饿，数据一致性
直接感知(双方直接交互，如通信)	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息	死锁，饥饿

1. **临界区**
    - 临界区 (critical section)： 进程中访问临界资源的一段代码。
    - 进入区 (entry section)： 在进入临界区之前，检查可否进入临界区的一段代码。如果可以进入临界区，通常设置相应 " 正在访问临界区 " 标志
    - 退出区 (exit section)： 用于将 " 正在访问临界区 " 标志清除。

- 剩余区 (remainder section)： 代码中的其余部分。



2. **应遵循的准则** 重点

   - 空闲则入：没有其他进程处于临界区；
   - 忙则等待：已有进程处于其临界区；
   - 有限等待：等待进入临界区的进程不能“死等”；
   - 有限使用：一个进程不能永远驻留在临界区内， 必须在有限时间内离开
   - 让权等待：不能进入临界区的进程，应释放 CPU （如转换到阻塞状态）

3. **进程互斥的软件方法** 重点

   - **算法1：单标志**

     - 有两个进程Pi, Pj，其中的Pi

     ```
     while (turn!= i);
     <!--turn 描述允许进入临界区的进程标识-->
         CRITICAL SECTION
     turn = j;
     <!--修改标记-->
         REMAINDER SECTION
     ```

     - 设立一个公用整型变量 turn：**描述允许进入临界区的进程标识**

       - 在进入区循环检查是否允许本进程进入：turn为i时，进程Pi可进入；
       - 在退出区修改允许进入进程标识：进程Pi退出时，改 turn为进程Pj的标识j；

- *缺点*:
  - 强制轮流进入临界区，没有考虑进程的实际需要。
  - 容易造成资源利用不充分：**在 Pi 出让临界区之后， Pj使用临界区之前， Pi 不可能再次使用临界区；**
  - 使用了忙等待，浪费处理机资源

- **算法2：双标志、先检查**
  - 有两个进程Pi, Pj，其中的Pi

```
while (flag[j]);
<!--flag判断某进程是否在占用资源-->
flag[i] = TRUE;
<!--置flag[i]为1，表示pi在占用资源-->
    CRITICAL SECTION
flag[i] = FALSE;
<!--删除标志-->
    REMAINDER SECTION
```

  - 优点：不用交替进入，可连续使用；
  - 缺点：
    - Pi 和 Pj 可能同时进入临界区。
    - 例如：按下面序列执行时，会同时进入：`Pi<a> Pj<a> Pi<b> Pj<b>` 。即在检查对方 flag 之后和切换自己 flag 之前有一段时间，结果都检查通过。
    - **这里的问题出在检查和修改操作不能连续进行。**

- **算法 3： 双标志、后检查**
  - 有两个进程Pi, Pj，其中的Pi

```
flag[i] = TRUE;
while (flag[j]);
    CRITICAL SECTION
flag[i] = FALSE;
    REMAINDER SECTION
```

  - 类似于算法 2 ，与互斥算法 2 的区别在于先修改后检查。可防止两个进程同时进入临界区。
  - 缺点：

Pi 和 Pj 可能都进入不了临界区。按下面序列执行时，会都进不了临界区：`Pi<a> Pj<a> Pi<b> Pj<b>` 。即在切换自己 flag 之后和检查对方 flag 之前有一段时间，结果都切换 flag ， 都检查不通过。

- **算法 4(Peterson's Algorithm)： 先修改、后检查、后修改者等待**
  - 有两个进程Pi, Pj，其中的Pi

```
flag[i] = TRUE; turn = j;
while  (flag[j] && turn == j);
     CRITICAL SECTION
 flag[i] = FALSE;
     REMAINDER SECTION
```

  - 结合算法 1 和算法 3 ， 是正确的算法
  - turn=j; 描述可进入的进程（同时修改标志时）
  - 在进入区先修改后检查，并检查并发修改的先后：
    - 检查对方 flag ， 如果不在临界区则自己进入－－空闲则入
    - **否则再检查 turn** ： 保存的是较晚的一次赋值，则较晚的进程等待，较早的进程进入－－先到先入，后到等待。有逻辑顺序

4. **进程互斥的硬件方法**

- 完全利用软件方法，有很大局限性，现在已很少采用。
- 可以利用某些硬件指令－－其读写操作由一条指令完成，因而保证读操作与写操作不被打断；

- *Test-and-Set指令*
  该指令读出标志后设置为TRUE

```
boolean TS(boolean *lock)
{
    boolean old;
    old = *lock; *lock = TRUE;
    <!--old为True，无法进入，old为False可以进入-->
    return old;

}
```

> lock表示资源的两种状态：**TRUE表示正被占用， FALSE表示空闲**

- **互斥算法（TS指令）**

```
while TS(&lock);
    CRITICAL SECTION
lock = FALSE;
    REMAINDER SECTION
```

- 利用 TS 实现进程互斥：每个临界资源设置一个公共布尔变量 lock ，初值为 FALSE
- 在进入区利用 TS 进行检查：有进程在临界区时，重复检查；直到其它进程退出时，检查通过；
- Swap 指令（或 Exchange 指令）
  交换两个字（字节）的内容

```
void SWAP(int *a, int *b)
 {
    int temp;
    temp = *a;
    *a = *b;
    *b =temp;
  }
```

- **互斥算法（ Swap 指令）**
  利用 Swap 实现进程互斥：每个临界资源设置一个**公共布尔变量 lock** ，初值为 FALSE 。每个进程设置一个**私有布尔变量 key**

> *key为False时候可进入*

```
key = TRUE;
do {
 SWAP(&lock, &key);
  }
while (key);
    CRITICAL SECTION
lock = FALSE;
    REMAINDER SECTION
```

- 硬件方法的优点
  - 适用于任意数目的进程，在单处理器或多处理器上
  - 简单，容易验证其正确性

    - 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量

- 硬件方法的缺点

    - 等待要耗费 CPU 时间，不能实现 " 让权等待 "
    - 可能 " 饥饿 " ： 从等待进程中随机选择一个进入临界区，有的进程可能一直选不上
    - 可能死锁

5. **信号量(semaphore)** 重点

- 前面的平等进程间的一种协商机制
- 需要一个地位高于进程的管理者来解决公有资源的使用问题。
- OS 可从进程管理者的角度来处理互斥的问题，信号量就是 OS 提供的管理公有资源的有效手段。
- 每个信号量 s 除一个整数值 **s.count**（计数）外，还有一个进程等待队列 **s.queue** ，其中是阻塞在该信号量的各个进程的标识
    - 信号量只能通过初始化和两个标准的原语来访问－－作为 OS 核心代码执行，不受进程调度的打断
    - 初始化指定一个非负整数值，表示**空闲资源总数**（又称为 " 资源信号量 " ）－－若为非负值表示**当前的空闲资源数**，若为负值其绝对值表示**当前等待临界区的进程数**

```
struct semaphore {
      int count;
      queueType queue;
}

void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0)
      {
            place this process in s.queue;
            block this process
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0)
      {
            remove a process P from s.queue;
            place process P on ready list;
      }
}
```

Figure 5.3 A Definition of Semaphore Primitives

- **P原语wait(s)**

```
s.count --; // 表示申请一个资源 ;
```

```
if (s.count < 0) // 表示没有空闲资源 ;
{
  调用进程进入等待队列 s.queue;
  阻塞调用进程 ;
}
```

- **v原语signal(s)**

```
s.count ++; // 表示释放一个资源 ;
if (s.count <= 0) // 表示有进程处于阻塞状态;
{
  从等待队列 s.queue 中取出一个进程 P;
  进程 P 进入就绪队列 ;
}
```

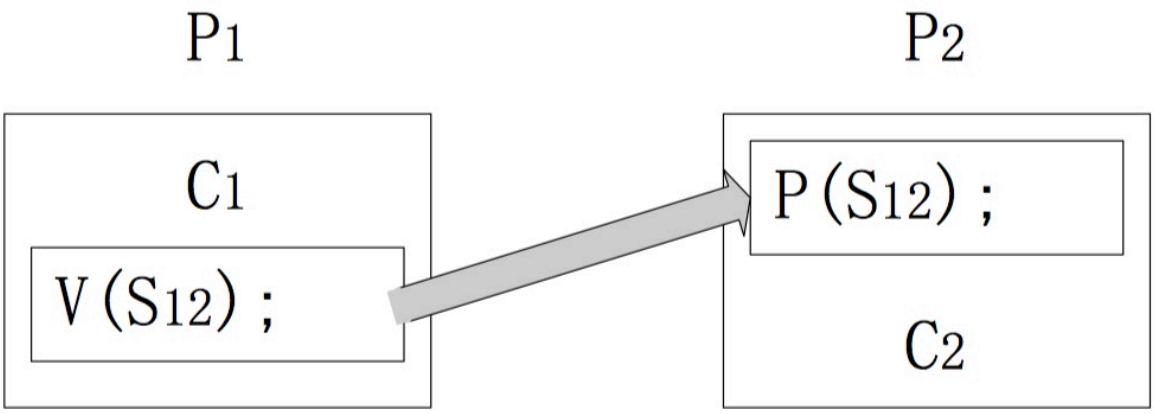
- **利用信号量实现互斥**

```
P(mutex);
    CRITICAL SECTION
V(mutex);
    REMAINDER SECTION
```

- 为临界资源设置一个**互斥信号量 mutex(Mutual Exclusion)**，其初值为1；在每个进程中将临界区代码置于 P(mutex) 和 V(mutex) 原语之间
- 必须**成对使用** P 和 V 原语：遗漏 P 原语则不能保证互斥访问，遗漏 V 原语则不能在使用临界资源之后将其释放（给其他等待的进程）； P 、V原语不能次序错误、重复或遗漏

- **利用信号量来处理前趋关系**

> 先释放S12，后才能申请S12，S12初值为0，以此完成前趋关系：C1 -> C2



  - 前趋关系：并发执行的进程 P1 和 P2 中，分别有代码 C1 和 C2 ，要求 C1 在 C2

开始前完成；

- **为每个前趋关系设置一个互斥信号量 S12， 其初值为 0**

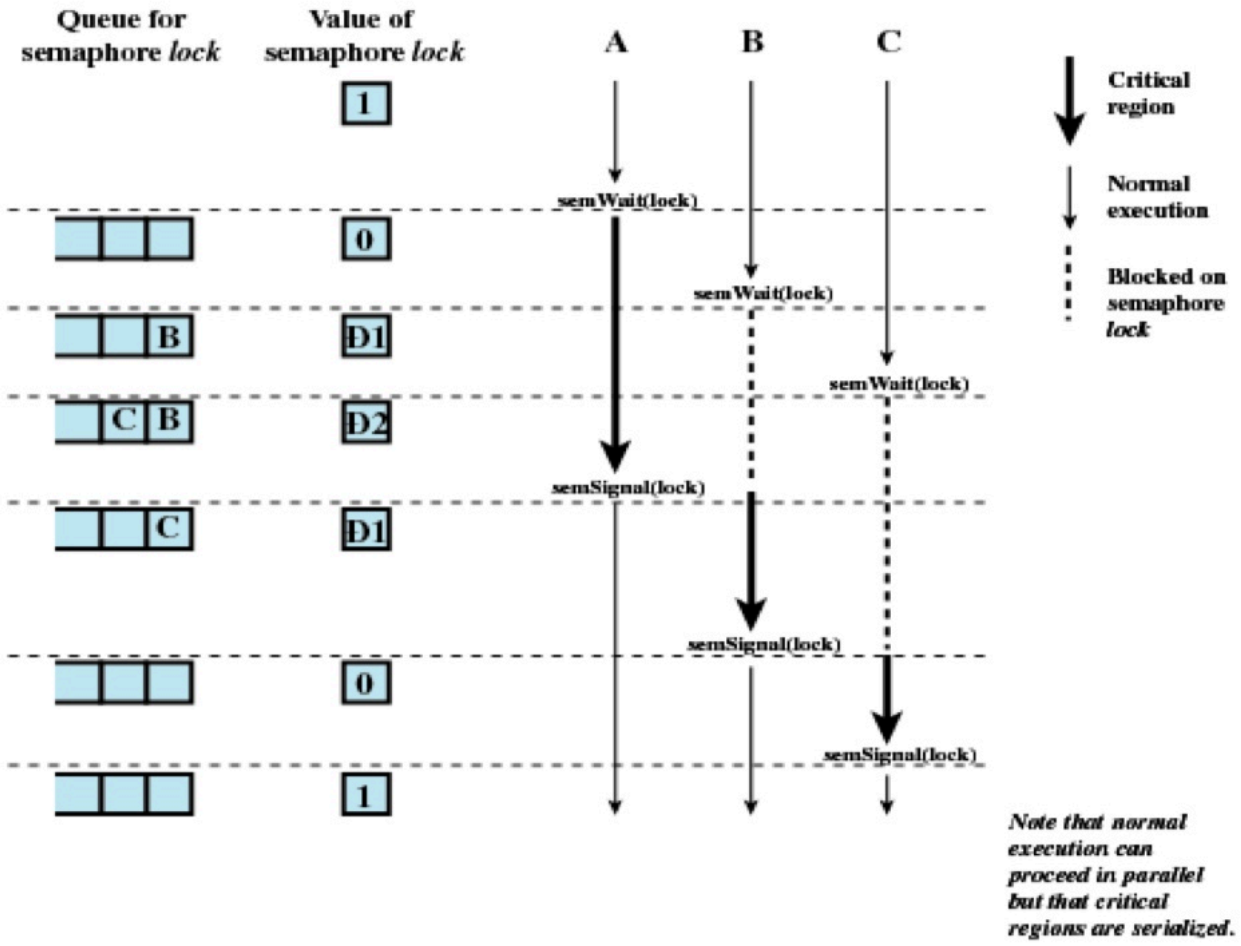
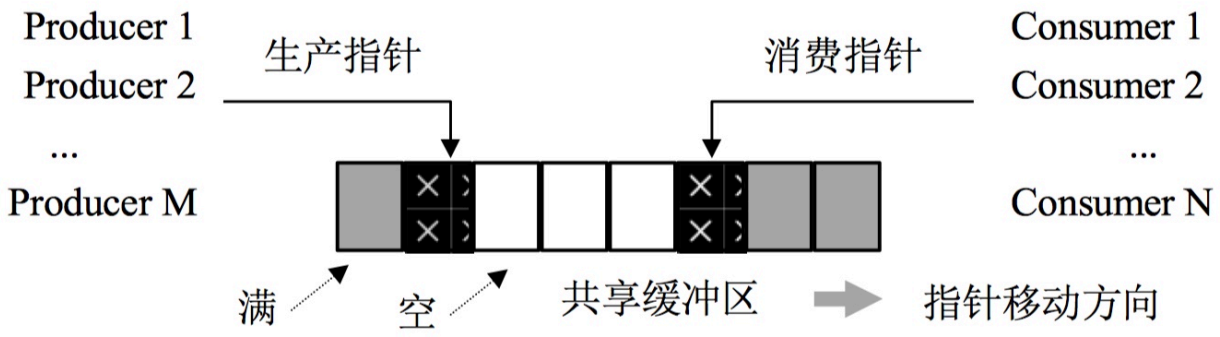


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

6. **经典进程同步问题** 重点

   - **生产者－消费者问题(the producer-consumer problem)**
   - 问题描述：若干进程通过有限的共享缓冲区交换数据。 其中，"生产者"进程不断写入，而"消费者"进程不断读出；
   - **任何时刻只能有一个进程可对共享缓冲区进行操作。**



- 采用信号量机制：
  - full 是 " 满 " 数目，初值为 0， empty 是 " 空 " 数目， 初值为 N 。实际上， full 和 empty 是同一个含义： full + empty == N

    > full empty 实现对资源的互斥

  - **mutex 用于访问缓冲区时的互斥，初值是 1**
- 每个进程中各个 P 操作的次序是重要的：先检查资源数目，再检查是否互斥 —— 否则可能死锁 (为什么？)

Producer

```
P(empty);
P(mutex);    //进入区
  one unit --> buffer;
V(mutex);
V(full);     //退出区
```

Consumer

```
P(full);
P(mutex);    //进入区
  one unit <-- buffer;
V(mutex);
V(empty);    //退出区
```

- **读者－写者问题 (the readers-writers problem)** 重点重点敲黑板
- 问题描述：对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者” 则允许多个
  - “读－写”互斥，
  - “写－写”互斥，
  - "读－读"允许

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
     semWait (x);
     readcount++;
     if (readcount == 1)
          semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0)
          semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true)
    {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
    }
```

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

> x对readcount实现互斥， wsem对资源实现互斥。

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0)
               semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true)
    {
     semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0)
               semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Figure 5. 23　A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority

- **信号量同步的缺点**
  - **同步操作分散**：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如 P 、 V 操作的次序错误、重复或遗漏）
  - **易读性差**：要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序；

- **不利于修改和维护**：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局；
- **正确性难以保证**：操作系统或并发程序通常很大， 很难保证这样一个复杂的系统没有逻辑错误；

7. **管程（ Monitor ）**

   - 1973 年， Hoare 和 Hanson 所提出；**其基本思想是把信号量及其操作原语封装在一个对象内部**。即：将共享变量以及对共享变量能够进行的所有操作集中在一个模块中。
   - 管程的定义：**管程是关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块。**
   - 管程可**增强模块的独立性**：系统按资源管理的观点分解成若干模块，用数据表示抽象系统资源，同时分析了共享资源和专用资源在管理上的差别，按不同的管理方式定义模块的类型和结构，使同步操作相对集中，从而增加了模块的相对独立性
   - **引入管程可提高代码的可读性**，便于修改和维护，正确性易于保证：采用集中式同步机制。一个操作系统或并发程序由若干个这样的模块所构成，一个模块通常较短，模块之间关系清晰。

8. **管程的主要特性**

   - 模块化：一个管程是一个基本程序单位， 可以单独编译；
     - 局部数据变量只能由管程的函数访问，外部函数不能访问
     - 进程通过调用管程提供的函数进入管程
     - 任何时刻最多只能有一个进程在管程中执行，而其他调用管程的进程均被阻塞，直到管程可用

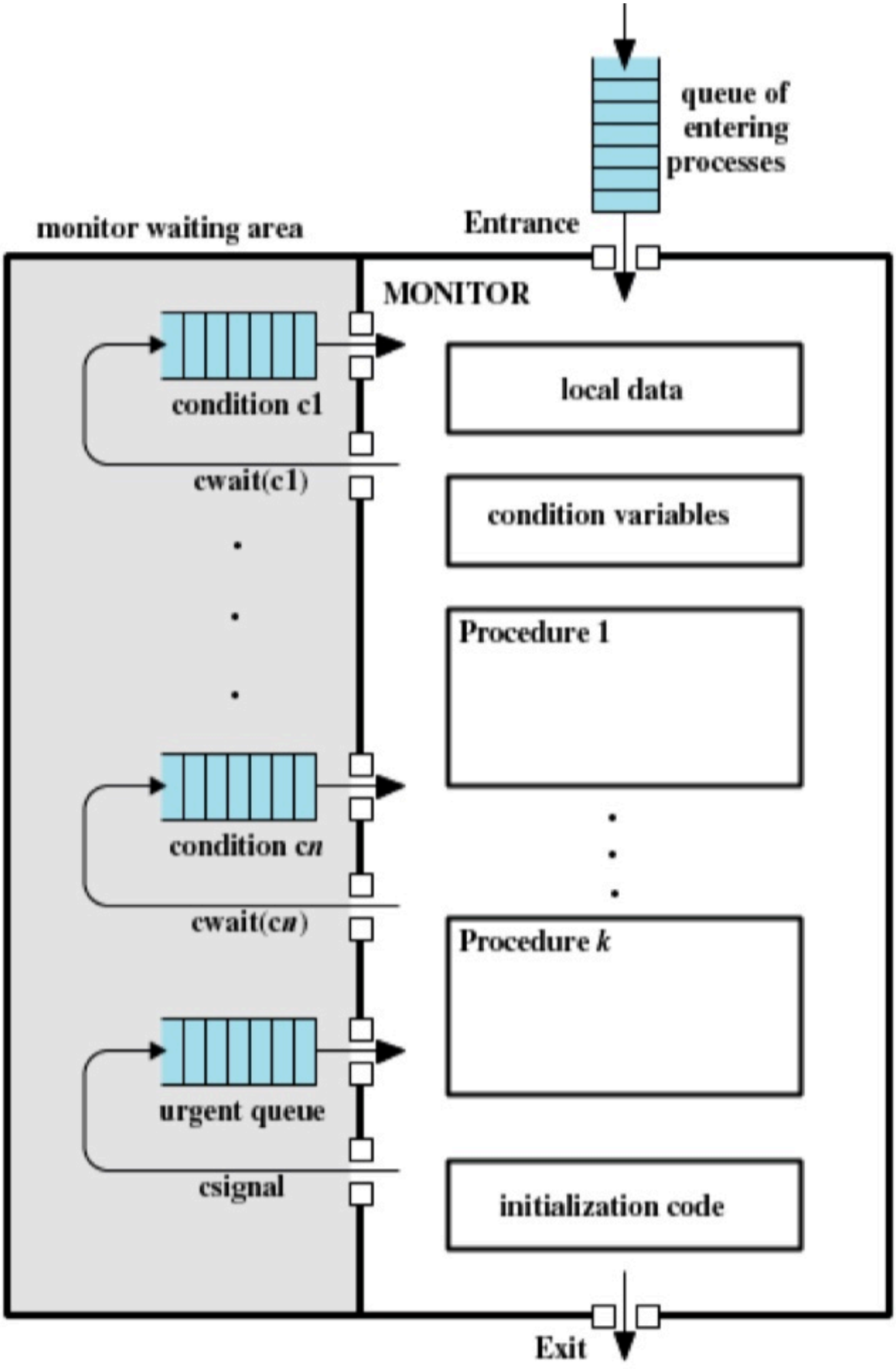


Figure 5.15 Structure of a Monitor

9. **管程中的多个进程进入**

   - 当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权；当一个进入管程的进程执行唤醒操作时（如P唤醒Q），管程中便存在两个同时处于活动状态的进程。
   - 管程中的唤醒切换方法：
     - P 等待 Q 继续，直到 Q 等待或退出；
     - Q 等待 P 继续，直到 P 等待或退出；
     - 规定唤醒为管程中最后一个可执行的操作；

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                  /* buffer pointers */
int count;                                  /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
     if (count == N)
        cwait(notfull);                    /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                        /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0)
        cwait(notempty);                /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                    /* one fewer item in buffer */
     csignal(notfull);                         /* resume any waiting producer */
}
{                                                            /* monitor body */
     nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

```
void producer()
char x;
{
     while (true)
     {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true)
     {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

10. **管程和进程的异同点**

- 设置进程和管程的目的不同
- 系统管理数据结构
  - 进程： PCB
  - 管程：等待队列
- 管程被进程调用
- 管程是操作系统的固有成分，无创建和撤消

11. **消息传递**

- 既要实施互斥，又要交换信息。
- 通过消息传递实现
- 与窗口系统中的“消息”不同。通常是不定长数据块。消息的发送不需要接收方准备好，随时可发送。
- send (destination, message)
- receive (source, message)

12. **同步（Synchronization）**

- 发送端和接收端是否因等待消息而被阻塞
  - Sender and receiver may or may not be blocking (waiting for message)
- **Blocking send, blocking receive**
  - Both sender and receiver are blocked until message is delivered
  - Called a rendezvous （会合）
- **Nonblocking send, blocking receive**
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- **Nonblocking send, nonblocking receive**
  - Neither party is required to wait

13. **寻址（Addressing）**

- 直接寻址
  - 发送原语包含目标进程的标识
  - 接收原语需要知道等待那个进程发送的消息
  - 接收原语接收消息以后能够通过源参数返回值
- 间接寻址
  - 消息发送到一个包含队列的共享数据结构
  - 队列称为邮箱
  - **One process sends a message to the mailbox and the other process picks up the message from the mailbox**

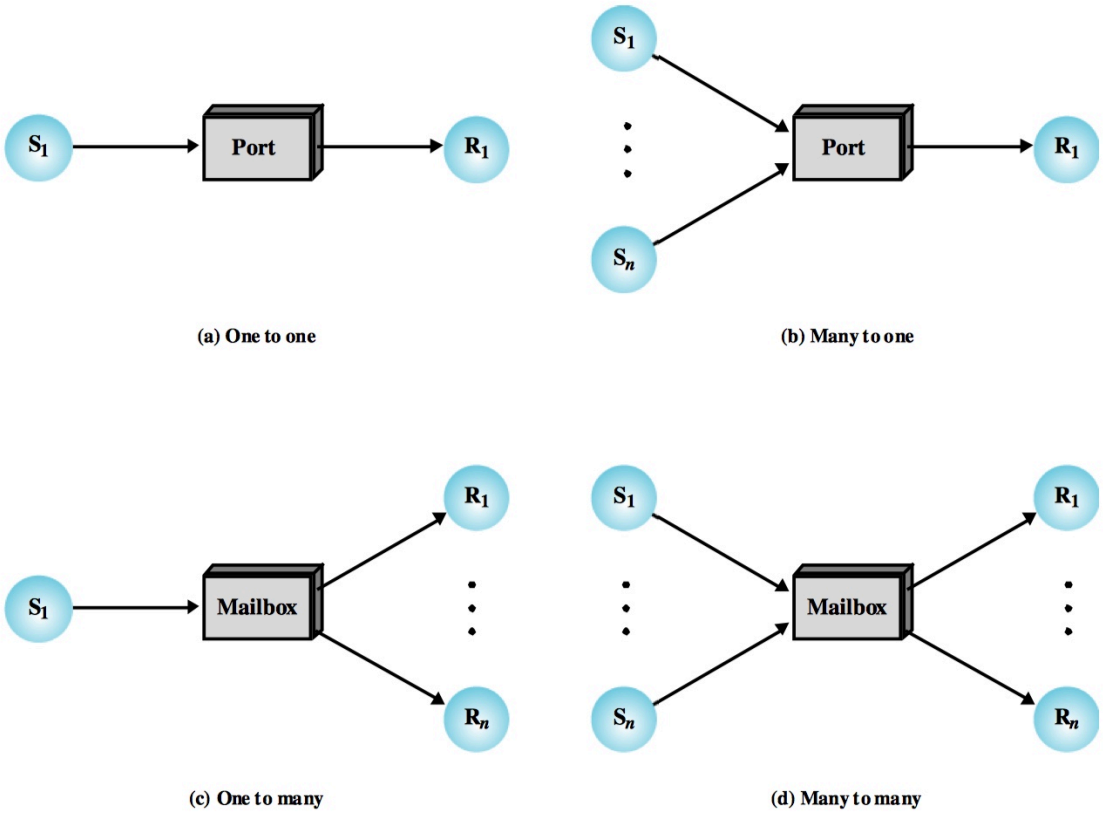


Figure 5.18 Indirect Process Communication

Message Format

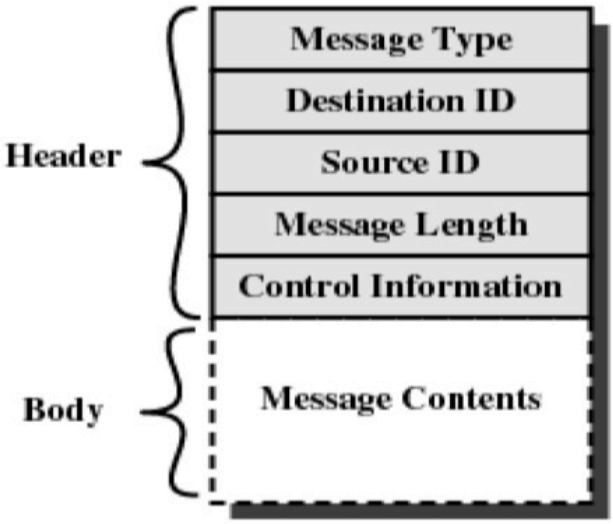


Figure 5.19 General Message Format

```
/* 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

```
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