

第5章 并行性：互斥和同步 ★★☆☆☆

- 主要内容

- 5.1 并发的原理 ★★☆☆☆
- 5.2 互斥：硬件的支持
- 5.3 信号量 ★★★★★
- 5.4 管程 ★★☆☆☆
- 5.5 消息传递 ★★☆☆☆
- 5.6 读者/写者问题 ★★☆☆☆

第5章 并行性：互斥和同步 ★★★★★

```
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"
```

```
static volatile int counter = 0;
```

```
// mythread()
//
```

```
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
```

```
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

```
mov 0x1c2, %eax
add $0x1, %eax
mov %eax, 0x1c2
```

```
// main()
```

```
//
```

```
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
```

```
//
```

```
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
```

```
// join waits for the threads to finish
```

```
Pthread_join(p1, NULL);
```

```
Pthread_join(p2, NULL);
```

```
printf("main: done with both (counter = %d)\n",
        counter);
```

```
return 0;
```

gcc thread.c -o thread -Wall -pthread

与并发相关的关键术语

- 原子操作

- 保证指令序列要么作为一个组来执行，要么都不执行；

- 临界区

- 一段代码，在这段代码中进程将访问共享资源，当一个进程已经在这段代码中运行时，另外一个进程就不能在这段代码中执行；

- 死锁

- 两个或两个以上的进程因其中的每个进程都在等待其他进程做完某些事情而不能继续执行；

- 活锁

- 两个或两个以上进程为了响应其他进程中的变化而持续改变自己的状态但不做有用的工作；

与并发相关的关键术语

- 互斥

- 当一个进程在临界区访问共享资源时，其他进程不能进入该临界区访问任何共享资源；

- 竞争条件

- 多个线程或进程在读写一个共享数据时，结果依赖于它们执行的相对时间；

- 饥饿

- 一个可运行的进程被调度程序无限期地忽略，不能被调度执行的情形。

5.1 并发的原理 ★★☆☆☆

1、基本概念

- 并发

- 单处理器多道程序设计系统中，进程交替执行；

- 并行

- 多处理器系统中，不仅可以交替执行进程，还可以重叠执行进程。

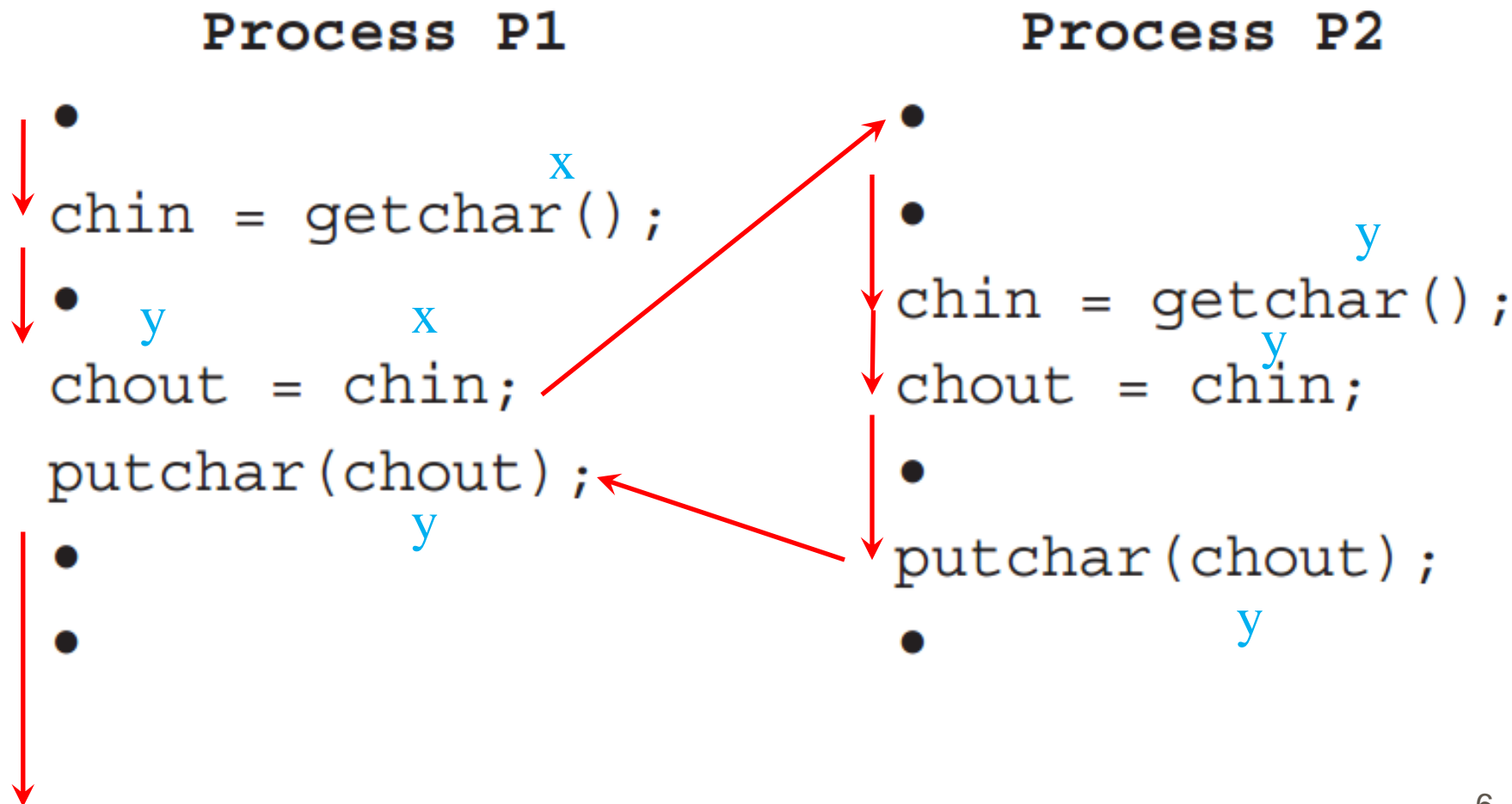
- 并发问题

- 并发进程的相对执行速度是**不可预测**的，取决于其他进程的活动、操作系统处理中断的方式以及操作系统的调度策略。
 - 可能发生各种**与时间有关**的错误。

5.1 并发的原理 ★★★★★

单处理器

```
void echo()  
{  
    chin = getchar();  
    chout = chin;  
    putchar(chout);  
}
```



5.1 并发的原理 ★★☆☆☆

多处理器

```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

Process P1

```
•  
    chin = getchar(x) ;  
•  
  
chout = ychin;  
putchar(chout) ;  
•  
  
•
```

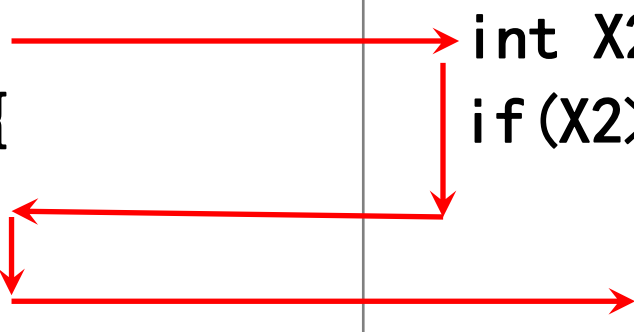
Process P2

-
-
- `chin = getchary() ;`
- `chout = chiny;`
-
- `putchar(chout) ;`
- `y`

(结果不唯一) 机票问题

//飞机票售票问题

<ul style="list-style-type: none">• void T1() {• {按旅客订票要求找到Aj};• int X1=Aj;• if (X1>=1) {• X1--;• Aj=X1;• {输出一张票};• }• else• {输出信息"票已售完"};• "};• }	<ul style="list-style-type: none">• void T2() {• {按旅客订票要求找到Aj};• int X2=Aj;• if (X2>=1) {• X2--;• Aj=X2;• {输出一张票};• }• else• {输出信息"票已售完"• }
---	---



T1、T2并发执行，可能出现如下交叉情况：

T1: $X1 = A_j$; // $X1 = m$

T2: $X2 = A_j$; // $X2 = m$

T2: $X2--$; $A_j = X2$; {输出一张票}; // $A_j = m-1$

T1: $X1--$; $A_j = X1$; {输出一张票}; // $A_j = m-1$

同一张票卖给两位旅客（若只有一张余票）或者余票数不正确（若有多张余票）。

2、操作系统关注的问题 ★★☆☆☆

- 并发带来的设计和管理问题：
 - 操作系统必须能跟踪不同的进程；
 - 操作系统必须为每个活跃进程分配和释放各种资源；
 - 操作系统必须保护每个进程的数据和物理资源；
 - 一个进程的功能和执行结果必须与执行速度无关。

3、进程的交互

- 进程间的资源竞争
 - 互斥、死锁、饥饿
- 进程间通过共享合作
 - 互斥、死锁、饥饿、数据一致性
- 进程间通过通信合作
 - 死锁、饥饿

4、互斥的要求

- 必须强制实施互斥；
- 一个在非临界区停止的进程不能干涉其他进程；
- 不允许出现需要访问临界区的进程被无限延迟的情况；
- 当没有进程在临界区时，任何需要进入临界区的进程必须能够立即进入；
- 对相关进程的执行速度和处理器的数目没有任何要求和限制；
- 一个进程驻留在临界区中的时间必须是有限的。

解决互斥问题的方法

- 软件方法
 - 由并发执行的进程担负解决问题的责任；
- 硬件方法
 - 中断禁用
 - 专用机器指令
- 操作系统或程序设计语言中提供某种级别的支持
 - 信号量
 - 管程
 - 消息传递

提问

1. 下列对临界区的描述正确的是（ ）。
 - A. 一个缓冲区
 - B. 一个共享数据区
 - C. 一段程序
 - D. 一个互斥资源
2. 判断题：一个进程一直无法得到临界资源被迫暂停执行，从而形成死锁。
 - A. 正确
 - B. 错误
3. 实施互斥机制可能会产生死锁问题。
 - A. 正确
 - B. 错误

5.2 互斥：硬件的支持 ★★☆☆☆

1、中断禁用

- while (true) {
 /* 禁用中断 */;
 /* 临界区 */;
 /* 启用中断 */;
 /* 其余部分 */;
}
- 问题：
 - 代价非常高；
 - 不能用于多处理器结构中。

2、专用机器指令

比较和交换指令

```
int compare_and_swap (int *word, int
    testval, int newval)
{
    int oldval;
    oldval = *word;
    if (oldval == testval) *word = newval;
    return oldval;
}
```


交换指令

```
void exchange (int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

```
/* program mutual exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare_and_swap(bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . ,P(n));
}
}
```

```
/* program mutual exclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (keyi, bolt)
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . , P(n));
}
```

机器指令方法的优缺点

- 优点

- 适用于在单处理器或共享内存的多处理器上的任何数目的进程；
- 非常简单且易于证明；
- 可用于支持多个临界区，每个临界区可以用它自己的变量定义。

- 缺点

- 忙等待；
- 可能饥饿；
- 可能死锁。

提问

4. 采用中断禁用来实现互斥的方法虽然会使CPU的执行效率降低，但却可以非常有效地完成互斥的任务。

A. 正确

B. 错误

5. exchange指令的优点之一是不会发生饥饿，缺点是不适用于多处理器系统。

A. 正确

B. 错误

6. 用于实现互斥机制的“比较和交换指令”属于一种软件实现方法。

A. 正确

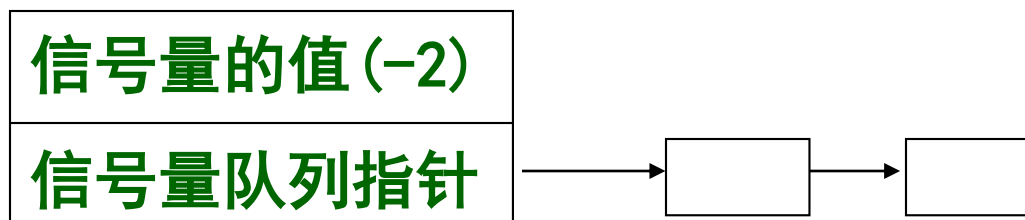
B. 错误

5.3 信号量 ★★★★★

- 1965年E. W. Dijkstra提出了新的同步工具--信号量。
- 基本原理
 - 两个或多个进程通过简单的信号进行合作，一个进程被迫在某一位置停止，直到它接收到一个特定的信号。
 - 任何复杂的合作需求都可以通过适当的信号结构得到满足。

信号量

- 信号量是一个**与队列有关**的整型变量。
 - 可以初始化成非负数；
 - `semWait` 操作使信号量减1。若值为负数，则执行 `semWait` 的进程阻塞，否则继续执行；
 - `semSignal` 操作使信号量加1。若值小于或等于0，则被 `semWait` 操作阻塞的进程被解除阻塞。



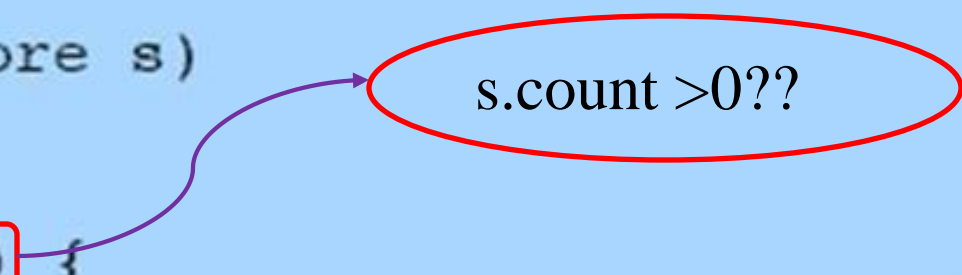
`semSignal(s)`:通过信号量`s`传送信号进程执行的原语
`semWait(s)` :通过信号量`s`接收信号进程执行的原语

信号量



- $S.\text{count} \geq 0$: 可以执行 `semWait(s)` 而不被阻塞的进程数。
- $S.\text{count} < 0$: 阻塞在 `s.queue` 中的进程数。


```
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 */;  
    }  
}
```



s.count > 0??

等待队列的移出

最公平策略：先进先出 FIFO

采用该策略定义的信号量称为强信号量。

弱信号量

等待队列的移出

例子（强信号量）：
进程A、B、C依赖于
进程D的结果。

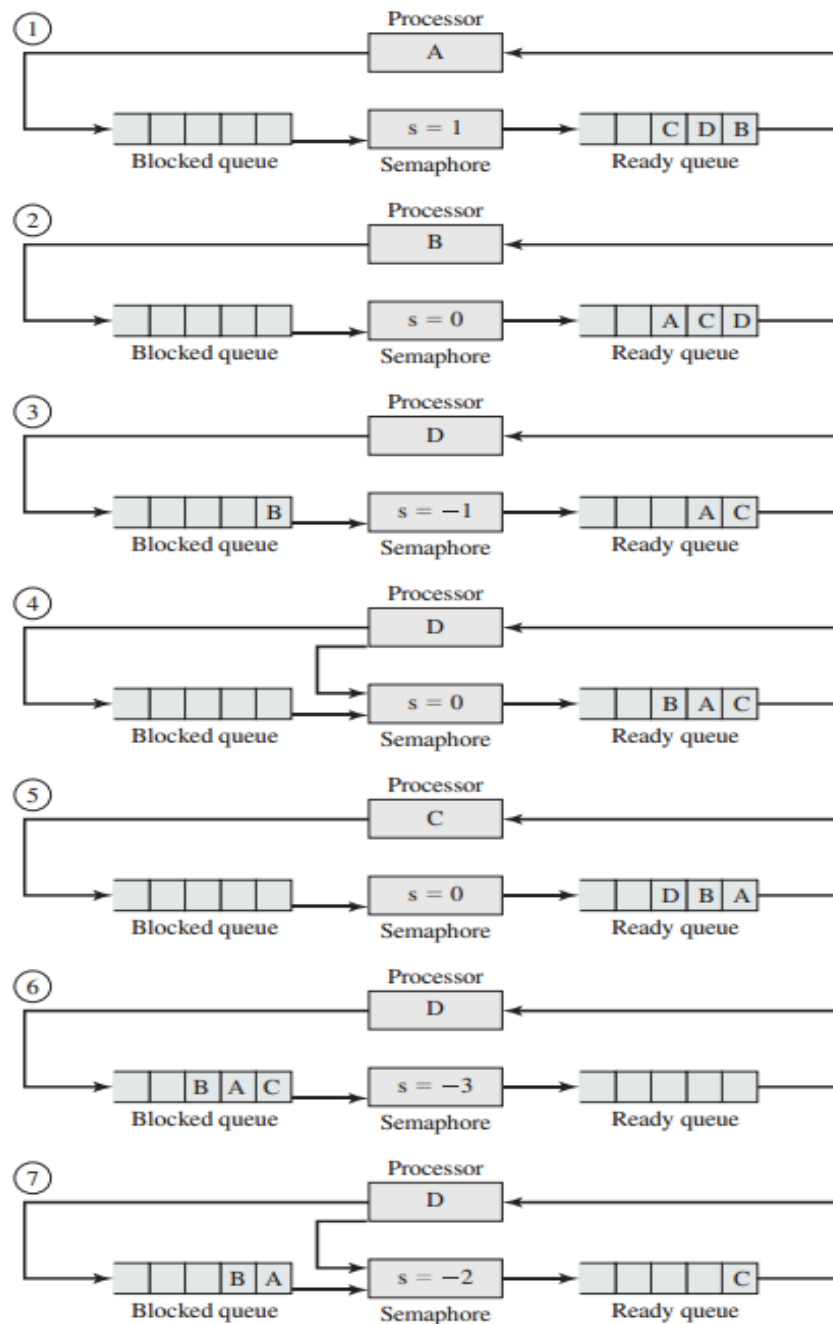
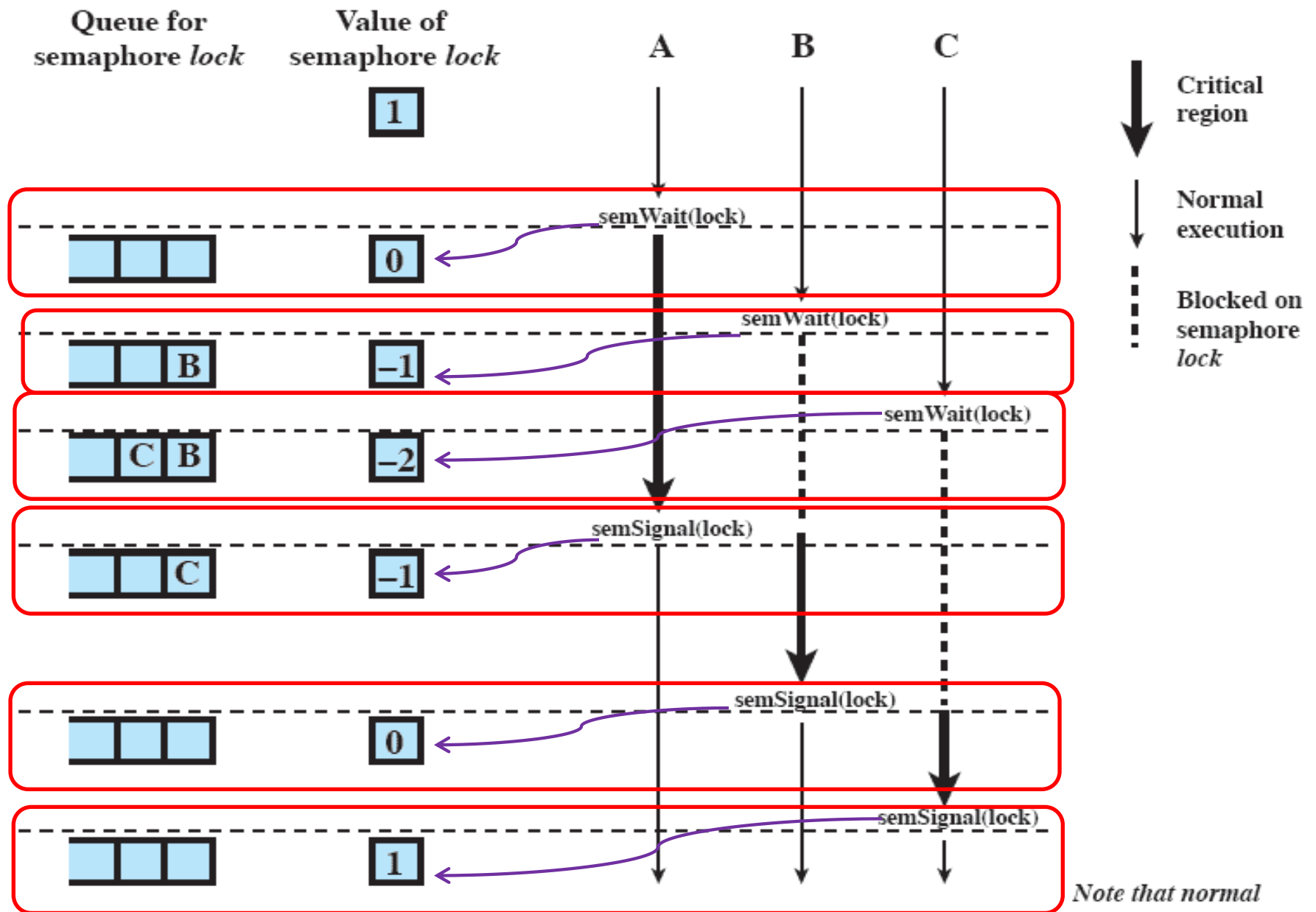


Figure 5.5 Example of Semaphore Mechanism

5.3.1 互斥 ★★★★★

- 信号量一般初始化为1。
- $S.count \geq 0$: 可以执行 `semWait(s)` 而不被阻塞的进程数。
- $S.count < 0$: 阻塞在 `s.queue` 中的进程数。

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```



5.3.2 生产者/消费者问题 ★★★★★

- 问题描述：

- 有一个或多个生产者生产某种类型的数据，并放置在缓冲区中；
- 有一个消费者从缓冲区中取数据，每次取一项；
- 系统保证避免对缓冲区的重复操作，即任何时候只有一个主体（生产者或消费者）可以访问缓冲区。
- 缓存已满时，生产者不能继续添加数据；
- 缓存已空时，消费者不能继续移走数据。

1、无限缓冲区的生产者/消费者问题

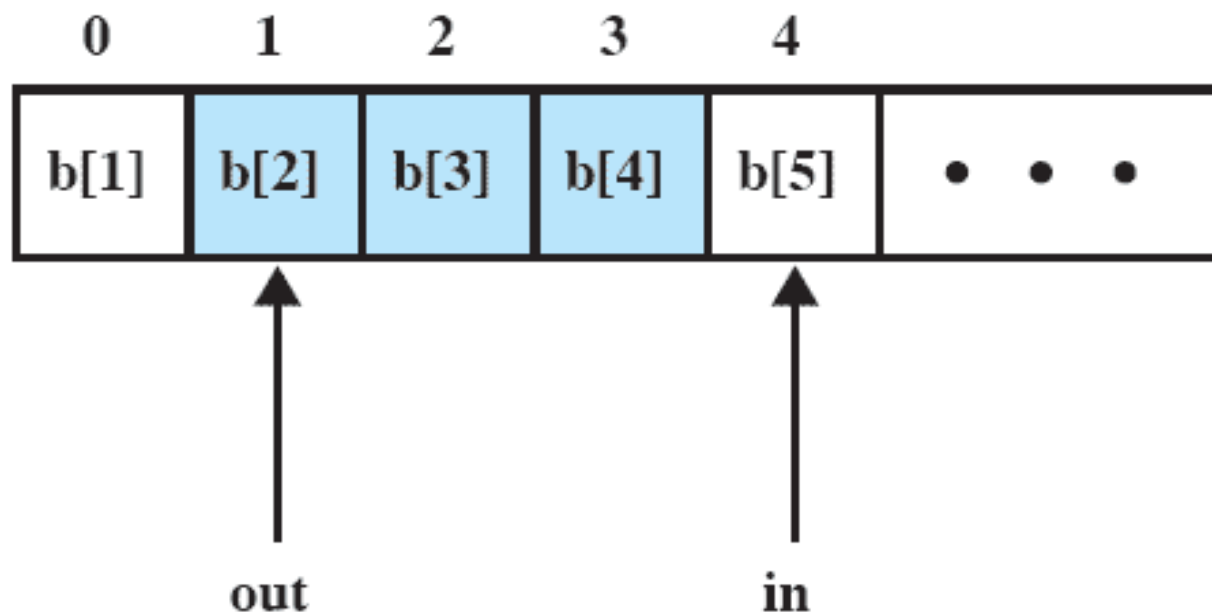
producer:

```
while (true) {  
    /* produce item v */  
    b[in] = v;  
    in++;  
}
```

consumer:

```
while (true) {  
    while (in <= out)  
        /*do nothing */;  
    w = b[out];  
    out++;  
    /* consume item w */  
}
```


无限缓冲区



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};

void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

**注意：并没有在else分支设置
“s.value=one”，为什么？**




Figure 5.4 A Definition of Binary Semaphore Primitives

```

/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}

```

Figure 5.9 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

Table 5.4 Possible Scenario for the Program of Figure 5.9

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		if (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		if (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignalB(s)	1	-1	0

Note: White areas represent the critical section controlled by semaphore s.

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Figure 5.9 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

	Producer	Consumer	s	n	delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		if (n==0) (semWaitB(delay))	0	0	0
10	semWaitB(s)		0	0	0
11	n++				
12	if (n==1) (semSignalB(delay))				
13	semSignalB(s)				
14		semSignalB(s)			

死锁！！

数据须由Producer产生

Consumer:
因无法得到数据而阻塞

Producer:
因无法得到缓冲区而阻塞

缓冲区须由Consumer释放

使用信号量解决无限缓冲区生产者/消费者问题

```
Semaphore s=1, n=0 ;
```

```
void producer () {
```

```
    while (true) {
```

```
        produce() ;
```

```
        semWait(s) ;
```

```
        append() ;
```

```
        semSignal(s) ;
```

```
        semSignal(n) ;
```

```
    }
```

```
}
```

```
void consumer () {
```

```
    while (true) {
```

```
        semWait(n) ;
```

```
        semWait(s) ;
```

```
        take() ;
```

```
        semSignal(s) ;
```

```
        consume() ;
```

```
    }
```

```
}
```

使用信号量解决无限缓冲区生产者/消费者问题

```
Semaphore s=1, n=0 ;
```

```
void producer () {  
    while (true) {  
        produce () ;  
        semWait (s) ;  
        append () ;  
        semSignal (s) ;  
        semSignal (n) ;  
    }  
}
```

```
void consumer () {  
    while (true) {  
        semWait (s) ;  
        semWait (n) ;  
        take () ;  
        semSignal (s) ;  
        consume () ;  
    }  
}
```


	Producer	Consumer	s	n
1			1	0
2		semWait(s)	0	0
3		semWait(n)	0	-1
4	produce()		0	-1
5	semWait(s)		-1	-1
6	append()			
7	semSignal(s)			
8	semSignal(n)			
10		take()		
11		semSignal(s)		
13		consume		
14				

对调了
semWait(n)与
semWait(n)
引起了死锁!

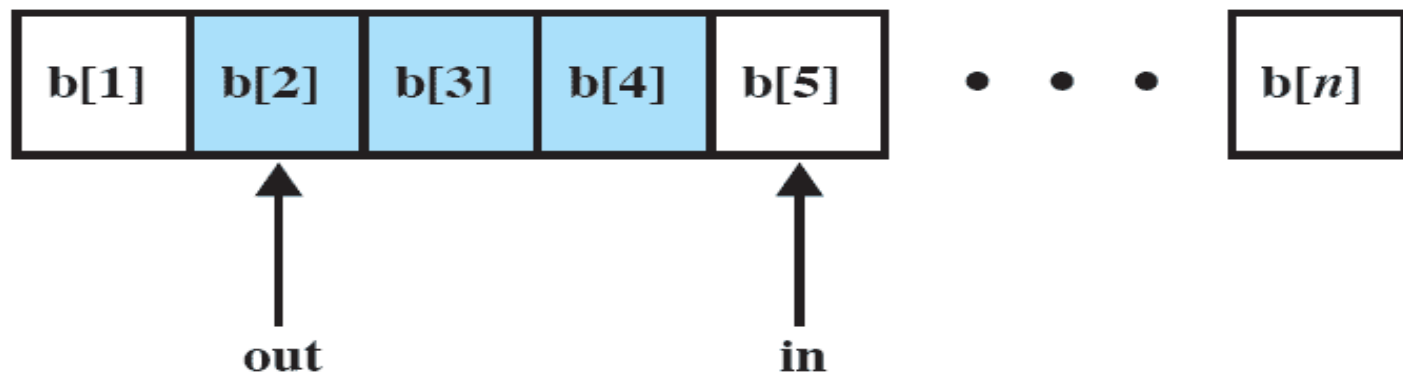
2、有限缓冲缓冲区的生产者/消费者问题

producer:

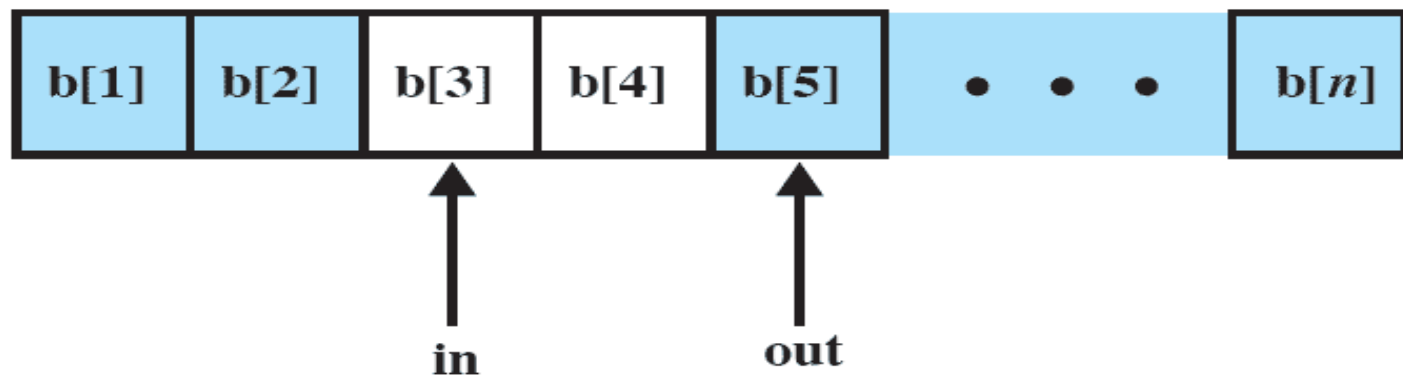
```
while (true) {  
    /* produce item v */  
    while ((in + 1) % n == out)  
        /* do nothing */;  
    b[in] = v;  
    in = (in + 1) % n  
}
```

```
consumer:
while (true) {
    while (in == out)
        /* do nothing */;
    w = b[out];
    out = (out + 1) % n;
    /* consume item w */
}
```

有限缓冲区



(a)



使用信号量解决有限缓冲区生产者/消费者问题

```
semaphore n=0, s=1,  
e=buf-size;  
void producer () {  
    while (true) {  
        produce();  
        semWait(e);  
        semWait(s);  
        append();  
        semSignal(s);  
        semSignal(n);  
    }  
}
```

```
void consumer () {  
    while (true) {  
        semWait(n);  
        semWait(s);  
        take();  
        semSignal(s);  
        semSignal(e);  
        consume();  
    }  
}
```

5.6 读者/写者问题 ★★☆☆☆

问题定义

- 有一个由多个进程共享的数据区，一些进程只读取这个数据区中的数据，一些进程只往数据区中写数据。并须满足以下条件：
 - 任意多的读进程可以同时读文件；
 - 一次只有一个写进程可以写文件；
 - 如果一个写进程正在写文件，那么禁止任何读进程读文件。

读者优先—信号量-读者进程

```
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);
}}
```

写者进程

```
void writer() {  
    while (true) {  
        semWait(wsem);  
        WRITEUNIT();  
        semSignal(wsem);  
    }  
}
```


总结

- 信号量小结
- `semWait`和`semSignal`操作小结
- 针对信号量问题的补充练习

1、信号量小结

❖ 一个信号量可用于 n 个进程的同步互斥；且只能由`semWait`、`semSignal`操作修改。

➤ 用于互斥时， S 初值为1，取值为 $1 \sim (n-1)$
(相当于临界区的通行证，实际上也是资源个数)

$S=1$ ：临界区可用

$S=0$ ：已有一进程进入临界区

$S<0$ ：临界区已被占用， $|S|$ 个进程正等待进入

➤ 用于同步时， S 初值 ≥ 0

$S \geq 0$ ：表示可用资源个数

$S<0$ ：表示该资源的等待队列长度

2、 semWait、semSignal操作小结

- semWait(S)：请求分配一个资源。
- semSignal(S)：释放一个资源。
- semWait、semSignal操作必须成对出现。
 - 用于互斥时，位于同一进程内；
 - 用于同步时，交错出现于两个合作进程内。
- 多个semWait操作的次序不能颠倒，否则可能导致死锁。
- 多个semSignal操作的次序可任意。

3、针对信号量问题的补充练习

1) 桌子上有一个盘子，可以存放一个水果。父亲总是放苹果到盘子中，而母亲总是放香蕉到盘子中；儿子专等吃盘中的香蕉，而女儿专等吃盘中的苹果。

- 分析：生产者—消费者问题的一种变形，生产者、消费者以及放入缓冲区的产品都有两类，但每类消费者只消费其中固定的一种产品。
- 数据结构：semaphore dish, apple, banana;
 - dish：表示盘子是否为空，初值为Max
 - apple：表示盘中是否有苹果，初值为0
 - banana：表示盘中是否有香蕉，初值为0

➤ 数据结构: semaphore dish, apple, banana;

➤ dish: 表示盘子是否为空, 初值为N

➤ apple: 表示盘中是否有苹果, 初值为0

➤ banana: 表示盘中是否有香蕉, 初值为0

```
process father() {  
    semWait(dish);  
    将苹果放到盘中;  
    semSignal(apple);  
}
```

```
process mother() {  
    semWait(dish);  
    将香蕉放到盘中;  
    semSignal(banana);  
}
```

```
process son() {  
    semWait(banana);  
    从盘中取出香蕉;  
    semSignal(dish);  
}
```

```
process daughter() {  
    semWait(apple);  
    从盘中取出苹果;  
    semSignal(dish);  
}
```

2) 在一个盒子里，混装了数量相等的黑白围棋子。现在用自动分拣系统把黑子、白子分开，设分拣系统有两个进程P1和P2，其中P1拣白子，P2拣黑子。规定每个进程每次拣一子，当一个进程在拣时，不允许另一个进程去拣；当一个进程拣了一子时，必须让另一个进程去拣。试用信号量协调两个进程的并发执行。

- 分析：实际上就是两个进程的同步问题。
- 数据结构：semaphore S1, S2 ;
- S1 和S2 分别表示可拣白子和黑子，不失一般性，若令先拣白子。初值， S1=1; S2=0;

```
process P1 () {  
    while(true) {  
        semWait(S1);  
        拣白子;  
        semSignal(S2);  
    }  
}
```

```
process P2() {  
    while(true){  
        semWait(S2);  
        拣黑子;  
        semSignal(S1);  
    }  
}
```

5.4 管程 ★★☆☆☆

- 管程是一个或多个过程、一个初始化序列和局部数据组成的软件模块，主要特点如下：
 - 局部数据变量只能被管程的过程访问，任何外部过程都不能访问；
 - 一个进程通过调用管程的一个过程进入管程；
 - 在任何时候，只能有一个进程在管程中执行，调用管程的任何其他进程都被阻塞，以等待管程可用。
- 管程通过条件变量提供对同步的支持。条件变量只有在管程中才能被访问。
 - `cwait(c)`：调用管程的进程在条件c上阻塞。
 - `csignal(c)`：恢复执行在`cwait`之后因为某些条件而阻塞的进程。

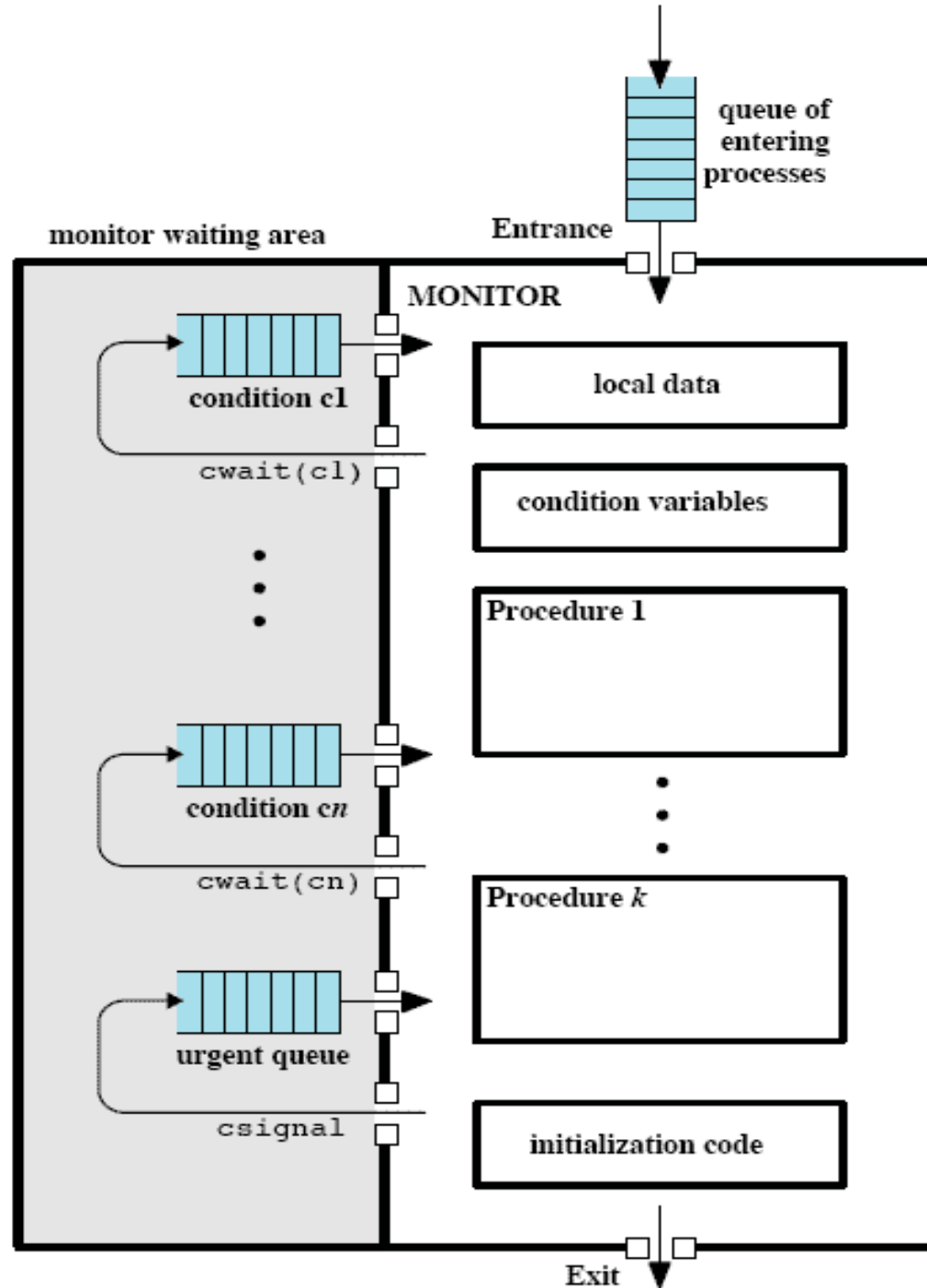
5.4 管程 ★★☆☆☆

- 通过给进程强加规定，管程可以提供一种**互斥机制**：管程中的数据变量每次只能被一个进程访问。可以把一共享数据结构放在管程中，从而提供对它的保护。
- 管程通过**条件变量**提供对**同步**的支持。条件变量只有在管程中才能被访问。
 - `cwait(c)`：调用管程的进程在条件c上阻塞。
 - `csignal(c)`：恢复执行在`cwait`之后因为某些条件而阻塞的进程。

5.4 管程 ★★☆☆☆

- Hoare管程
- Mesa管理

管程的结构



```

/* 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);  
}
```

5.5 消息传递 ★★☆☆☆

- 消息传递：合作进程之间进行信息交换。
- 消息传递原语
 - send (destination, message)
 - receive (source, message)

5.5.1 同步

- 阻塞send, 阻塞receive
 - 发送者和接收者都被阻塞，直到完成信息的投递。
- 无阻塞send, 阻塞receive
 - 接收者阻塞，直到请求的信息到达。
- 无阻塞send, 无阻塞receive
 - 不要求任何一方等待。

5.5.2 寻址

- 直接寻址

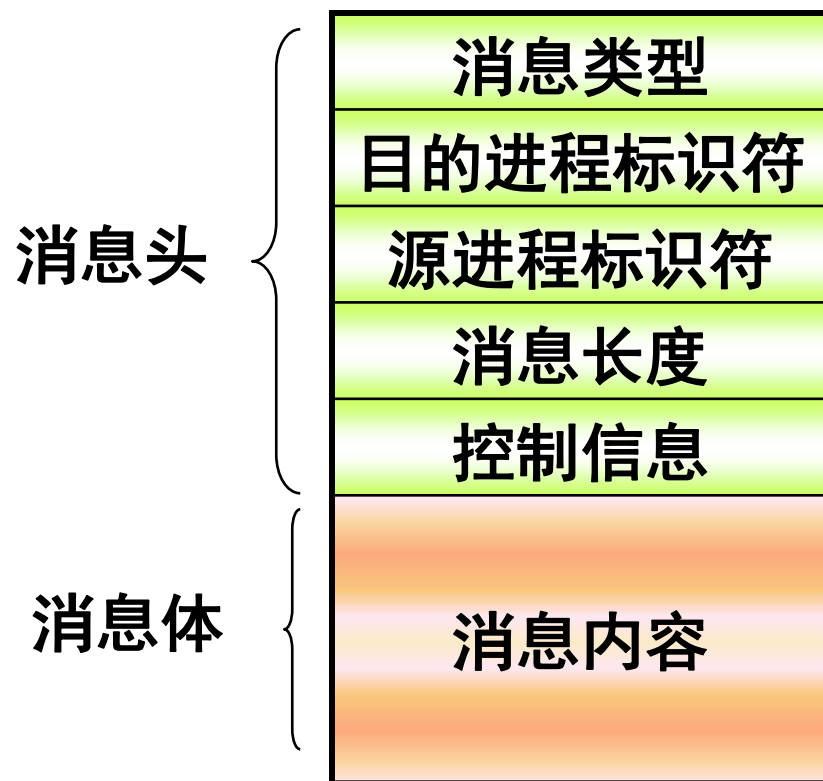
- send原语包含目标进程的标识号；
- receive原语可显式地指定源进程，也可不指定。

- 间接寻址

- 发送者将消息发送到合适的信箱；
- 接收者从信箱中获得消息。

5.5.3 消息格式

- 固定长度消息
- 可变长度消息



5.5.4 排队原则

- 先进先出
- 优先级

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

```

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);
}

```

作业

- 1、写出信号量定义，semWait和semSignal原语，以及用信号量实现互斥的伪代码。
- 2、假设一个阅览室有100个座位，没有座位时读者在阅览室外等待；每个读者进入阅览室时都必须在阅览室门口的一个登记本上登记座位号和姓名，然后阅览，离开阅览室时要去掉登记项。每次只允许一个人登记或去掉登记。用信号量操作描述读者的行为。

作业

- 3、设公共汽车上，司机和售票员活动如下：
- 1) 司机：启动汽车，正常行车，到站停车；
 - 2) 售票员：关车门，售票，开门上下客。
- 用信号量操作描述司机和售票员的同步。
- 4、（选做）独木桥问题：东、西向汽车过独木桥。桥上无车时允许一方汽车过桥，待全部过完后才允许另一方汽车过桥。用信号量操作写出同步算法。（提示：参考读者优先的解法）