Synchronization

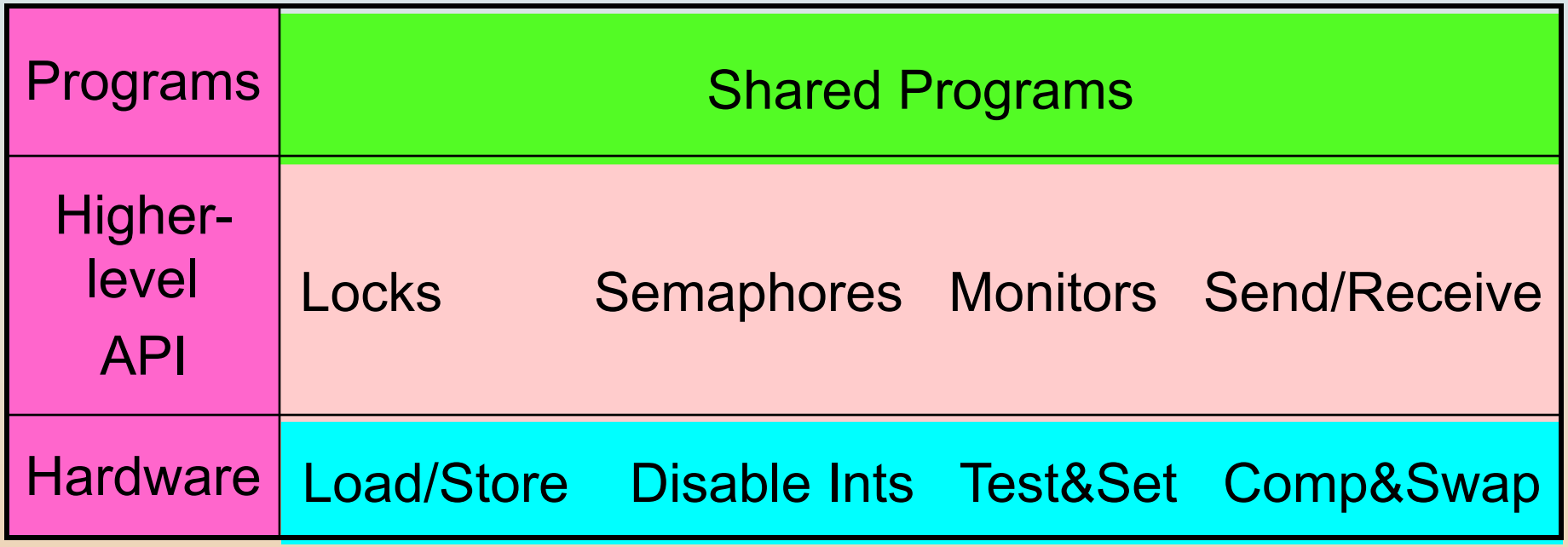
- Background
- Critical Section Problem (临界区问题)
- 经典同步问题

Concurrent

- Concurrent threads introduce problems when accessing shared data
  - 乱序：Programs must be insensitive to it
  - 一致性：Careful design to guarantee consistent shared variables

- 对共享数据的并发访问会导致数据的不一致性

- 为维护数据的一致性和执行结果的正确性，进程间必须同步，协作按一定次序执行


Programs
Shared Programs
Higher-level API
Locks
Semaphores
Monitors
Send/Receive
Hardware
Load/Store
Disable Ints
Test&Set
Comp&Swap

Concurrency? How to solve?

- Concurrent threads introduce problems when accessing shared data
  - 乱序：Programs must be insensitive to it
  - 一致性：Careful design to guarantee consistent shared variables

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives

- How to protect a critical section with only atomic load and store
  ⇒ pretty complex!

- 竞争（race condition)
  - 当两个进程竞相访问同一数据时，就会发生竞争。由于时间片的原因，执行结果可能会被破坏或者被错误地解释。

“counter:=counter+1”

机器语言实现:

register1 = counter

register1 = register1 + 1

counter = register1

“counter:=counter-1”

机器语言实现:

register2 = counter
register2 = register2 – 1
counter = register2

虽然上面的两个进程在单独运行时，分别看都是正确的，而且两者在顺序执行时其结果也会是正确的，但若并发执行时，就会出现差错

问题就在于这两个进程共享变量counter。生产者对它做加1操作，消费者对它做减1操作，

假设：counter的当前值是5。如果A进程先执行，然后B进程再执行，则最后共享变量counter的值仍为5；反之，如果让B进程先执行，然后再让 A 进程执行，counter值也还是5，但是，如果按下述顺序执行：

register 1 ： =counter; (register 1=5)

register 1 ： =register 1+1; (register 1=6)

register 2 ： =counter; (register 2=5)

register 2 ： =register 2-1; (register 2=4)

counter: =register 1; (counter=6)

counter: =register 2; (counter=4)

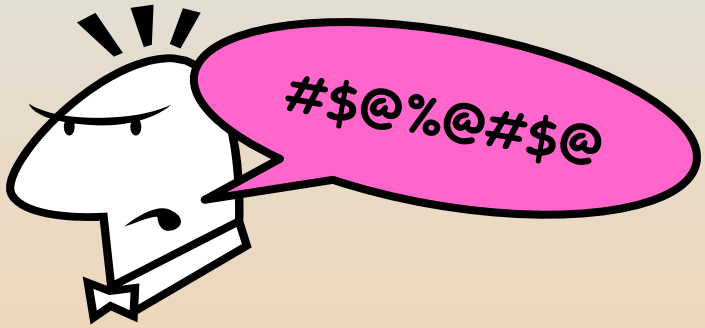
- 竞争：若干进程并发地访问并且操纵共享数据的情况。
  - 共享数据的值取决于哪个进程最后完成
- 为防止竞争，并发进程必须同步

Too Much Milk?

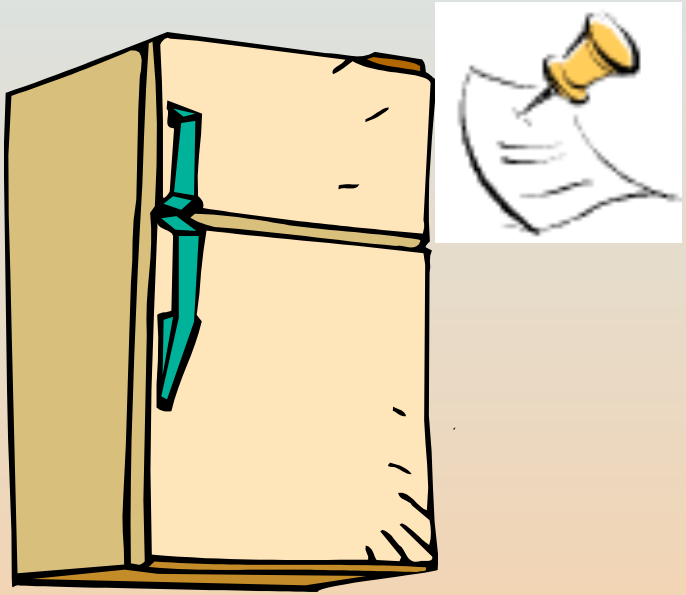
- Milk? Need or not?
- Who buy?
  - Never more than one person buys

	A	B
3:00	查看冰箱，没有牛奶	
3:05	去商店	查看冰箱，没有牛奶
3:10	到商店	
3:15	买牛奶	去商店
3:20	到家, 放牛奶，离开	到商店
3:25		买牛奶
3:30		到家, 放牛奶，离开

Lock it !!!



Leave a note?



```
if (noMilk) {
                    if (noNote) {
                        leave Note;
                        buy milk;
                        remove note;
                    }
            }
```

```
leave Note;
                if (noMilk) {
                    if (noNote) {
                        leave Note;
                        buy milk;
                    }
          }
                remove note;
```

No one buys!

Thread A

```
leave note A;
if (noNote B) {
    if (noMilk) {
        buy Milk;
    }
}
remove note A;
```

Thread B

```
leave note B;
if (noNoteA) {
    if (noMilk) {
        buy Milk;
    }
}
remove note B;
```

The other will buy!

Thread A

```
leave note A;
while (note B) {
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
   if (noNote A) { //Y
            if (noMilk) {
               buy milk;
         }
      }
        remove note B;
```

- A's code is different from B's – what if lots of threads?
- While A is waiting, it is consuming CPU time (Busy-waiting!)
- Really complex

临界资源

硬件或软件（如外设、共享代码段、共享数据结构），多个进程在对其进行访问时（关键是进行写入或修改），必须互斥地进行- - 有些共享资源可以同时访问，如只读数据

- 进程间资源访问冲突
  - 共享变量的修改冲突
  - 操作顺序冲突
- 进程间的制约关系
  - 间接制约：进行竞争－－独占分配到的部分或全部共享资源，“互斥”
  - 直接制约：进行协作－－等待来自其他进程的信息，“同步”

- 临界资源:一次只允许一个进程使用(访问)的资源。如：硬件打印机、磁带机等，软件的消息缓冲队列、变量、数组、缓冲区等。

临界区问题

- n 个进程竞争使用一些共享的数据
- 每个进程有一个代码段, 称为临界区, 通过其访问共享数据
- 问题- 保证当一个进程正在临界区执行时，没有另外的进程进入临界区执行

临界区(critical section)

多个进程共享临界资源时必须互斥使用

一个访问临界资源的循环进程描述如下:

```
repeat
        entry section
       critical section;
        exit section
       remainder section;
until false;
```

- 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。

有界缓冲区（生产者-消费者)问题

生产者-消费者(producer-consumer)问题是一个著名的进程同步问题。

1. 生产者进程与消费者进程能并发执行，所以需要在两者之间设置一个具有n个缓冲区的缓冲池，

2. 生产者进程将它所生产的产品放入一个缓冲区中；

3. 消费者进程可从一个缓冲区中取走产品去消费。

4. 所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步：即不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

有界缓冲区（生产者-消费者)问题

- 一个有限缓冲池
  - 一个或多个缓冲区

- 两类线程
  - 生产者：生产者把产品放入缓冲区
  - 消费者：消费者把产品取出缓冲区

有界缓冲区（生产者-消费者)问题

- 缓冲区满
  - 生产者在缓冲区满时必须等待，直到缓冲区有空间才继续生产
- 缓冲区空
  - 消费者在缓冲区空时必须等待，直到缓冲区中有产品才能继续读取

- 可能的办法?
  - 引入一个整型变量counter, 其初始值为0
  - 每当生产者进程向缓冲池中投放一个产品后，使counter加1；反之，每当消费者进程从中取走一个产品时， 使counter减1。

有界缓冲区

- Shared data

```
#define BUFFER_SIZE 10
    typedef struct {
            . . .
    } item;
    item buffer[BUFFER_SIZE];
    int in = 0;
    int out = 0;
    int counter = 0;
```

- 生产者进程

```
while (1) {
        while (counter == BUFFER_SIZE)
            ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
    }
```

- 消费者进程

```
while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
    }
```

有界缓冲区（生产者-消费者)问题

- 竞争（race condition)
  - 当两个进程竞相访问同一数据时，就会发生竞争。由于时间片的原因，执行结果可能会被破坏或者被错误地解释。

根据之前的例子，假设：counter的当前值是5。如果生产者进程先执行，然后消费者进程再执行，则最后共享变量counter的值仍为5；反之，如果让消费者进程先执行，然后再让生产者进程执行，counter值也还是5，但是，如果按下述顺序执行：

register 1 : =counter; (register 1=5)

register 1 : =register 1+1; (register 1=6)

register 2 : =counter; (register 2=5)

register 2 : =register 2-1; (register 2=4)

counter: =register 1; (counter=6)

counter: =register 2; (counter=4)

虽然生产者程序和消费者程序，在分别看时都是正确的，而且两者在顺序执行时其结果也会是正确的，但若并发执行时，就会出现差错，问题就在于这两个进程共享变量counter。生产者对它做加1操作，消费者对它做减1操作，

- 竞争：若干进程并发地访问并且操纵共享数据的情况。
  - 共享数据的值取决于哪个进程最后完成
- 为防止竞争，并发进程必须同步

有界缓冲区（生产者-消费者)问题

- 缓冲区满

- 缓冲区空
- 竞争

解决临界区问题遵循的原则

- 空闲让进

  当无进程进入临界区时，相应的临界资源处于空闲状态，因而允许一个请求进入临界区的进程立即进入自己的临界区。

- 忙则等待(互斥)

  当已有进程进入自己的临界区时，即相应的临界资源正被访问，因而其它试图进入临界区的进程必须等待，以保证进程互斥地访问临界资源。

- 有限等待

  对要求访问临界资源的进程，应保证进程能在有限时间进入临界区，以免陷入“饥饿”状态。

解决同步问题遵循的原则

- 空闲让进

当无进程进入临界区时，相应的临界资源处于空闲状态，因而允许一个请 求进入临界区的进程立即进入自己的临界区。

- 忙则等待(互斥)

当已有进程进入自己的临界区时，即相应的临界资源正被访问，因而其它试图进入临界区的进程必须等待，以保证进程互斥地访问临界资源。

- 有限等待

对要求访问临界资源的进程，应保证进程能在有限时间进入临界区，以免陷入“饥饿”状态。

- 让权等待

当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入忙等。

信号量机制

- 1965年，荷兰学者Dijkstra提出的信号量机制是一种卓有成效的进程同步工具，在长期广泛的应用中，信号量机制又得到了很大的发展，它从整型信号量机制发展到记录型信号量机制，进而发展为“信号集”机制。现在信号量机制已广泛应用于OS中。一种不需要忙等待(Busy-waiting)的同步工具.

- 整型信号量

- 记录型信号量

- 信号集

信号量机制

- 每个信号量s除一个整数值s.value（计数）外，还有一个进程等待队列s.L，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
- "二进制信号量(binary semaphore)"：只允许信号量取0或1值

1. 整型信号量

最初由Dijkstra把整型信号量定义为一个整型量，除初始化外，仅能通过两个标准的原子操作(Atomic Operation) wait(S)和signal(S)来访问。这两个操作一直被分别称为P、V操作。 wait和signal操作可描述为:

wait(S): while S≤0 do no-op

S:=S-1;

signal(S): S:=S+1;

利用信号量实现进程互斥

- 为使多个进程能互斥地访问某临界资源，只需为该资源设置一个互斥信号量mutex,并设其初值为1，然后将各进程的临界区置于wait(mutex)和signal(mutex)操作之间即可。

用信号量实现互斥

- 共享数据:

semaphore mutex; //初始 mutex = 1

Process Pi:

```
do {
        wait(mutex);
        critical section
                signal(mutex);
        remainder section
} while (1);
```

2. 记录型信号量

在整型信号量机制中的wait操作，只要是信号量S≤0，就会不断地测试。因此，该机制并未遵循“让权等待”的准则， 而是使进程处于“忙等”的状态。

2. 记录型信号量

记录型信号量机制，则是一种不存在“忙等”现象的进程同步机制。

但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况。为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个进程链表L，用于链接上述的所有等待进程。

记录型信号量是由于它采用了记录型的数据结构而得名的。它所包含的上述两个数据项可描述为：

```
type semaphore=record
        value:integer;
        L:list of process;
        end
```

相应地，wait(S)和signal(S)操作可描述为:

```
procedure wait(S)
    var S: semaphore;
    begin
      S.value ： = S.value-1;
      if S.value < 0 then block(S,L)
    end
  procedure signal(S)
    var S: semaphore;
    begin
     S.value ： = S.value+1;
     if S.value≤0 then wakeup(S,L);
    end
```

在记录型信号量机制中，S.value的初值表示系统中某类资源的数目，因而又称为资源信号量，对它的每次wait操作，意味着进程请求一个单位的该类资源，因此描述为S.value: =S.value-1； 当S.value＜0时，表示该类资源已分配完毕，因此进程应调用block原语，进行自我阻塞，放弃处理机，并插入到信号量链表S.L中。

该机制遵循了“让权等待”准则。 此时S.value的绝对值表示在该信号量链表中已阻塞进程的数目。 对信号量的每次signal操作，表示执行进程释放一个单位资源，故S.value: =S.value+1操作表示资源数目加1。 若加1后仍是S.value≤0，则表示在该信号量链表中，仍有等待该资源的进程被阻塞，故还应调用wakeup原语，将S.L链表中的第一个等待进程唤醒。如果S.value的初值为1，表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号量。

3. AND型信号量

在两个进程中都要包含两个对Dmutex和Emutex的操作， 即

process A:　　process B:

wait(Dmutex);　　wait(Emutex);

wait(Emutex);　　wait(Dmutex);

3. AND型信号量

若进程A和B按下述次序交替执行wait操作:

process A: wait(Dmutex); 于是Dmutex=0

process B: wait(Emutex); 于是Emutex=0

process A: wait(Emutex); 于是Emutex=-1 A阻塞

process B: wait(Dmutex); 于是Dmutex=-1 B阻塞

AND同步机制的基本思想是：将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。

只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给他。亦即，对若干个临界资源的分配，采取原子操作方式：要么全部分配到进程，要么一个也不分配。　由死锁理论可知，这样就可避免上述死锁情况的发生。为此，在wait操作中，增加了一个“AND”条件，故称为AND同步，或称为同时wait操作，　　即

Swait(S_1, S_2, ..., S_n)

if $S_i \geq 1$ and ... and $S_n \geq 1$ then

for i : = 1 to n do

S_i := S_i-1;

endfor

else

place the process in the waiting queue associated with the first S_i found with $S_i < 1$, and set the program count of this process to the beginning of Swait operation

endif

Ssignal(S1, S2, ..., Sn)

for i: = 1 to n do

$S_i=S_i+1$;

Remove all the process waiting in the queue associated with S_i into the ready queue.

endfor;

4. 信号量集

Swait(S_1, t_1, d_1, ..., S_n, t_n, d_n)

if $S_i \geq t_1$ and ... and $S_n \geq t_n$ then

for i:=1 to n do

$S_i := S_i - d_i$;

endfor

else

Place the executing process in the waiting queue of the first S_i with $S_i < t_i$ and set its program counter to the beginning of the Swait Operation.

endif

signal(S_1, d_1, ..., S_n, d_n)

for i :=1 to n do

$S_i := S_i + d_i$;

Remove all the process waiting in the queue associated with S_i into the ready queue

endfor;

一般“信号量集”的几种特殊情况：

(1) Swait(S, d, d)。 此时在信号量集中只有一个信号量S， 但允许它每次申请d个资源，当现有资源数少于d时，不予分配。

(2) Swait(S, 1, 1)。 此时的信号量集已蜕化为一般的记录型信号量(S > 1时)或互斥信号量(S=1时)。

(3) Swait(S, 1, 0)。这是一种很特殊且很有用的信号量操作。当S≥1时，允许多个进程进入某特定区；当S变为0后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。

2.3.3 信号量的应用

1. 利用信号量实现进程互斥

利用信号量实现进程互斥的进程可描述如下：

```
Var mutex:semaphore : = 1;
   begin
   parbegin
    process 1: begin
                        repeat
                        wait(mutex);
                        critical section
                        signal(mutex);
                        remainder seetion
                        until false;
                  end
```

```
process 2: begin
                repeat
                    wait(mutex);
                    critical section
                    signal(mutex);
                     remainder section
                until false;
                end
    parend
```

wait、signal 重点讨论

- wait、signal操作必须成对出现，有一个wait操作就一定有一个signal操作。
  - 当为互斥操作时，它们同处于同一进程
  - 当为同步操作时，则不在同一进程中出现
  - 如果两个wait操作相邻，那么它们的顺序至关重要，而两个相邻的signal操作的顺序无关紧要。一个同步wait操作与一个互斥wait操作在一起时，同步wait操作在互斥wait操作前。

wait、signal 优缺点

- 优点：简单（用wait、signal操作可解决任何同步互斥问题。）
- 缺点：不够安全；wait、signal操作使用不当会出现死锁；实现复杂。

```
Semaphore fullBuffer = 0; // Initially, no milk
Semaphore emptyBuffers = numBuffers;
                         // Initially, num empty slots
Semaphore mutex = 1;     // No one using machine

Producer(item) {
    emptyBuffers.P();    // Wait until space
    wait(mutex);         // Wait until buffer free
    Enqueue(item);
    signal(mutex);
    fullBuffers.V();     // Tell consumers there is
                         // more milk
}
```

```
Semaphore fullBuffer = 0;     // Initially, no milk
Semaphore emptyBuffers = numBuffers;
                              // Initially, num empty slots
Semaphore mutex = 1;          // No one using machine


Consumer() {
    fullBuffers.P();          // Check if there's milk
    wait(mutex);              // Wait until machine free
    item = Dequeue();
    signal(mutex);
    emptyBuffers.V();         // tell producer need more
    return item;
}
```

经典进程的同步问题

1 生产者–消费者问题

前面对生产者–消费者问题(The producer-consumer problem)中，未考虑进程的互斥与同步问题，因而可能会造成数据Counter的不定性。

由于生产者–消费者问题是相互合作的进程关系的一种抽象，例如，在输入时，输入进程是生产者，计算进程是消费者；而在输出时，则计算进程是生产者，而打印进程是消费者，因此，该问题有很大的代表性及实用价值。

1. 利用记录型信号量解决生产者—消费者问题

假定在生产者和消费者之间的公用缓冲池中，具有n个缓冲区，这时可利用互斥信号量mutex实现诸进程对缓冲池的互斥使用；利用信号量empty和full分别表示缓冲池中空缓冲区和满缓冲区的数量。又假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。对生产者—消费者问题可描述如下：

```
Var mutex, empty, full:semaphore  :=   1,n,0;
    buffer:array  [0, ..., n-1]   of item;
    in, out: integer  :=   0, 0;
    begin
     parbegin
      proceducer:begin
            repeat
            ...
            producer an item nextp;
            ...
            wait(empty);
            wait(mutex);
            buffer(in)  :=   nextp;
            in  :=   (in+1) mod n;
            signal(mutex);
            signal(full);
            until false;
           end
```

```
consumer:begin
        repeat
         wait(full);
         wait(mutex);
         nextc : = buffer(out);
         out : = (out+1) mod n;
         signal(mutex);
         signal(empty);
         consumer the item in nextc;
        until false;
       end
  parend
end
```

在生产者—消费者问题中应注意:

首先，在每个程序中用于实现互斥的wait(mutex)和signal(mutex)必须成对地出现;

其次，对资源信号量empty和full的wait和signal操作，同样需要成对地出现，但它们分别处于不同的程序中。例如，wait(empty)在计算进程中，而signal(empty)则在打印进程中，计算进程若因执行wait(empty)而阻塞，则以后将由打印进程将它唤醒;

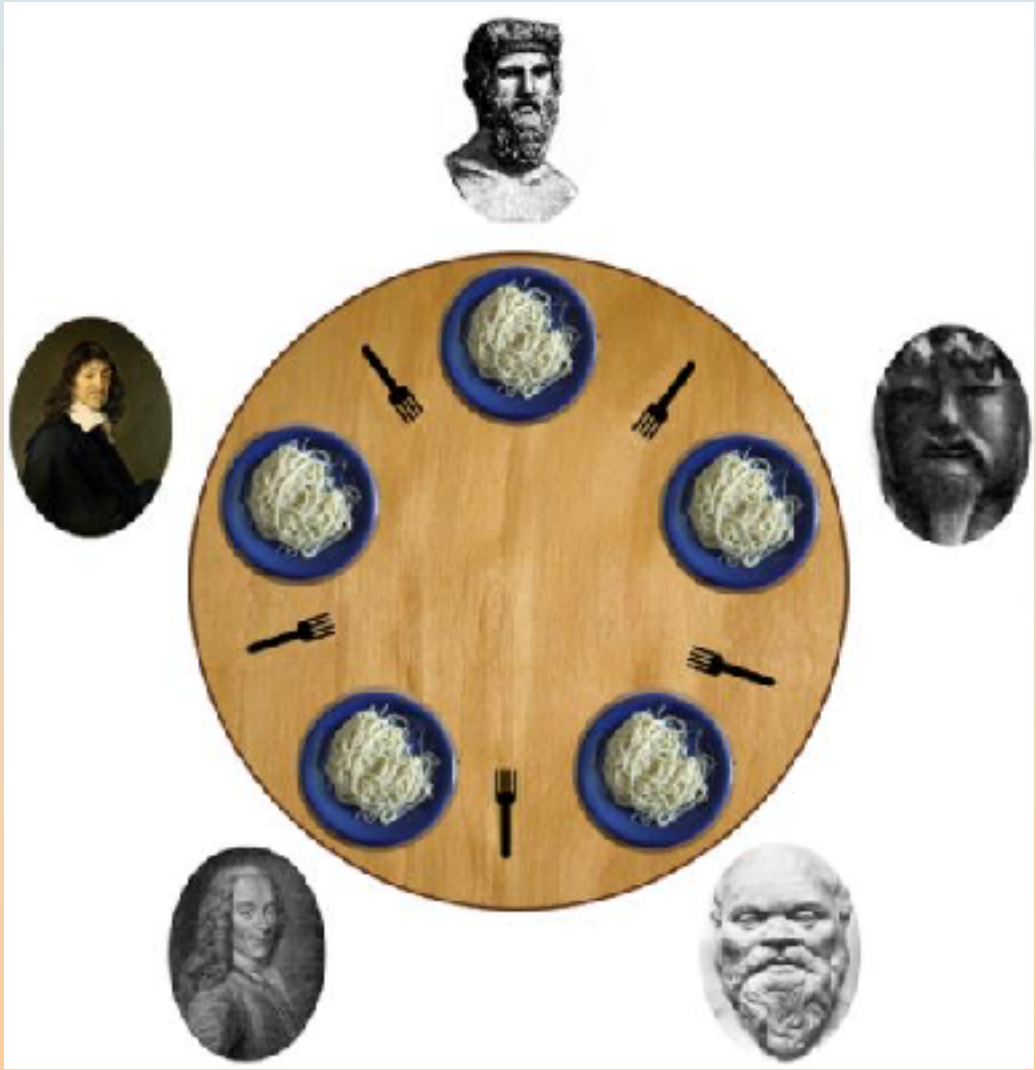
最后，在每个程序中的多个wait操作顺序不能颠倒。应先执行对资源信号量的wait操作，然后再执行对互斥信号量的wait操作，否则可能引起进程死锁。

2. 利用AND信号量解决生产者—消费者问题

```
var mutex, empty, full:semaphore : = 1, n, 0;
   buffer:array [0, ..., n-1] of item;
   in out:integer : = 0, 0;
  begin
   parbegin
     producer:begin
          repeat
          ...
          produce an item in nextp;
          ...
          Swait(empty, mutex);
          buffer(in) : = nextp;
          in : = (in+1)mod n;
          Ssignal(mutex, full);
         until false;
       end
```

```
consumer:begin
        repeat
         Swait(full, mutex);
         nextc : = buffer(out);
         out : = (out+1) mod n;
         Ssignal(mutex, empty);
         consumer the item in nextc;
        until false;
      end
   parend
end
```

2 哲学家进餐问题



2 哲学家进餐问题

1. 利用记录型信号量解决哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：

Var chopstick: array［0, ..., 4］ of semaphore;

所有信号量均被初始化为1，　　第i位哲学家的活动可描述为：

```
repeat
    wait(chopstick［i］);
    wait(chopstick［(i+1) mod 5］);
    ...
    eat;
    ...
    signal(chopstick［i］);
    signal(chopstick［(i+1) mod 5］);
    ...
    think;
```

可采取以下几种解决方法:

(1) 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。

(2) 仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。

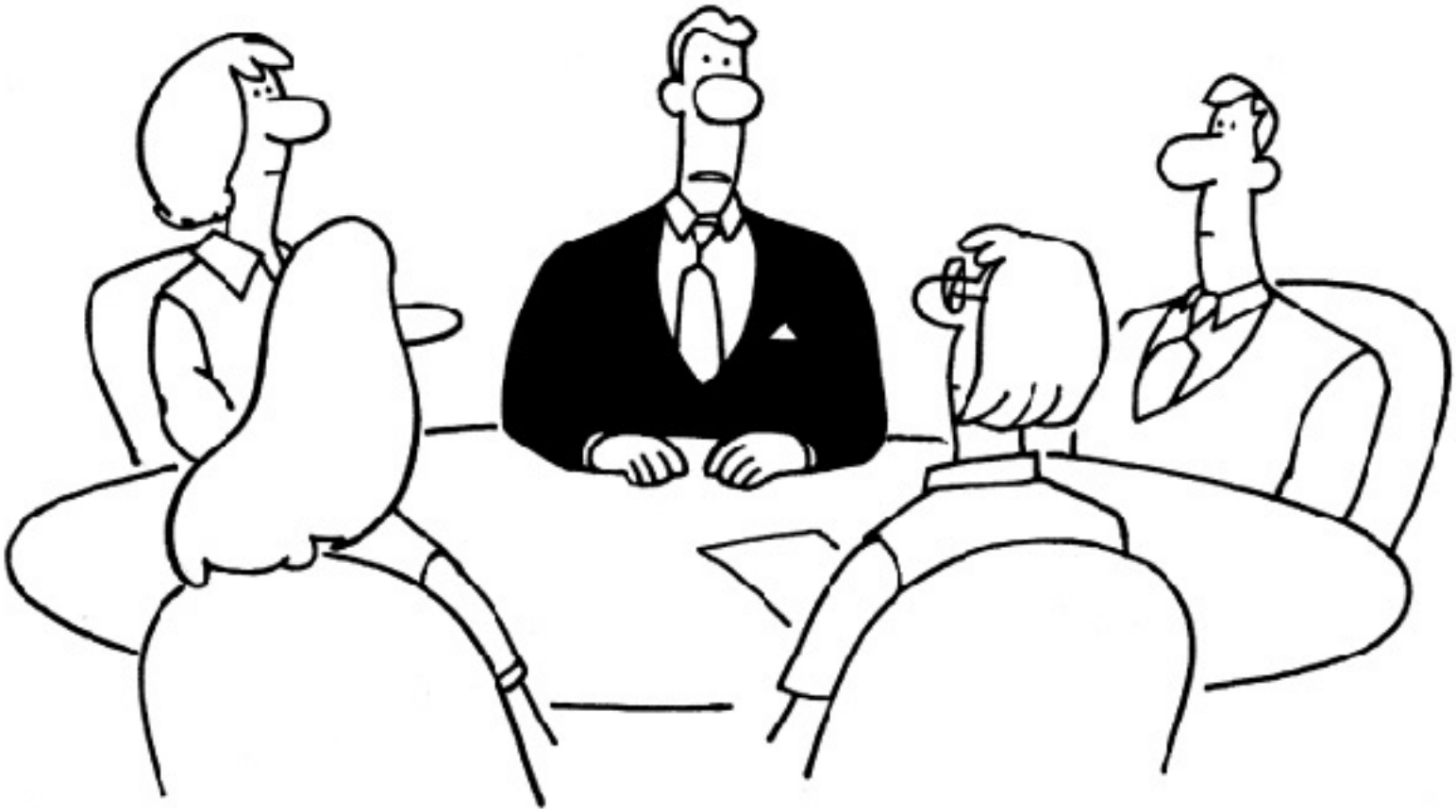
(3) 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子；而偶数号哲学家则相反。按此规定，将是1、　　2号哲学家竞争1号筷子；3、4号哲学家竞争3号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

2. 利用AND信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐，这在本质上就是前面所介绍的AND同步问题，故用AND信号量机制可获得最简洁的解法。

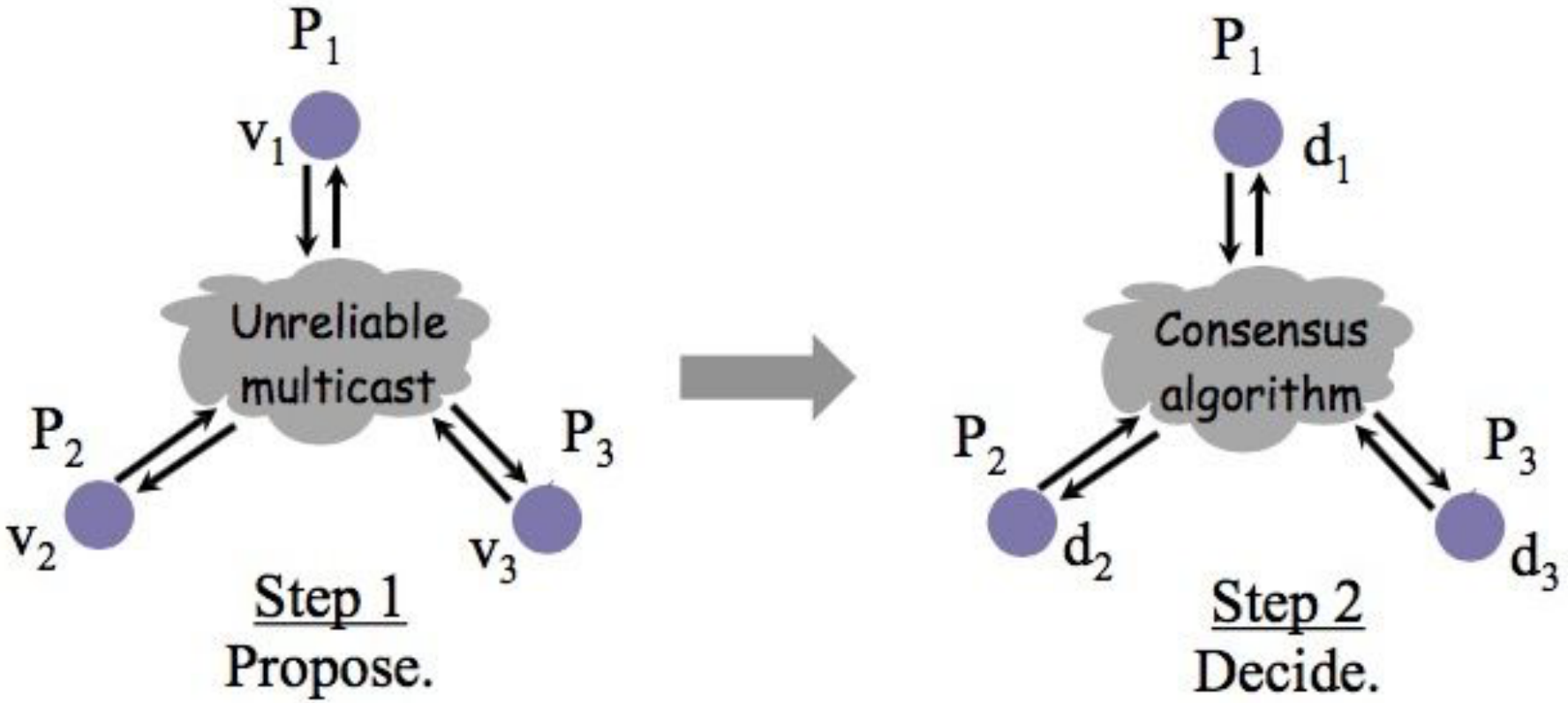
```
Var chopsiick array ［0, ..., 4］ of semaphore ： = (1,1,1,1,1);
   processi
      repeat
      think;
      Sswait(chopstick ［(i+1) mod 5］, chopstick ［i］);
      eat;
      Ssignat(chopstick ［(i+1) mod 5］, chopstick ［i］);
    until false;
```

分布式一致性问题

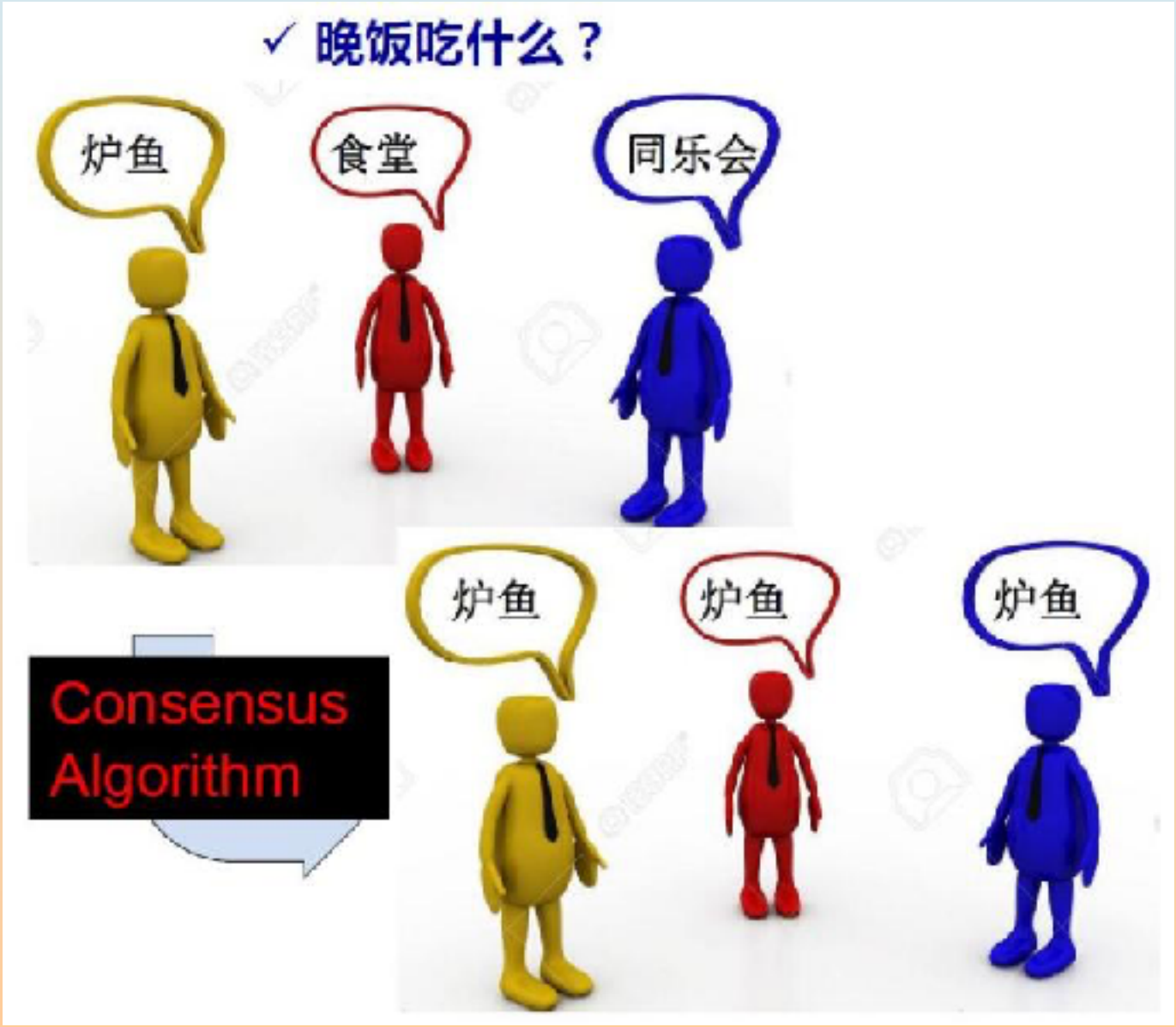


"Whew! That was close!
We almost decided something!"

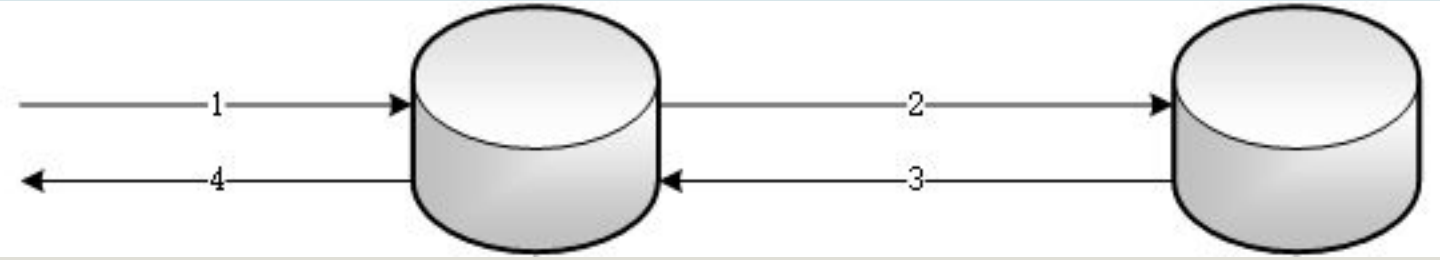
分布式一致性问题


P1
v1
Unreliable multicast
P2
P3
v2
v3
Step 1
Propose.
P1
d1
Consensus algorithm
P2
P3
d2
d3
Step 2
Decide.

分布式一致性问题

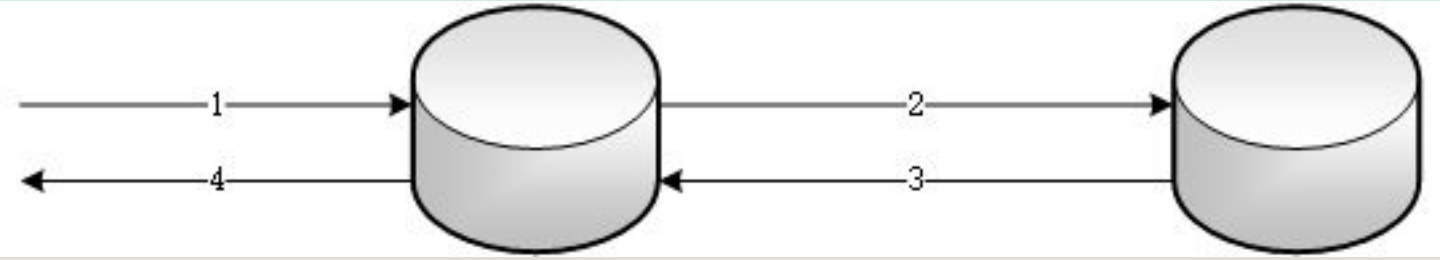


多副本一致性问题



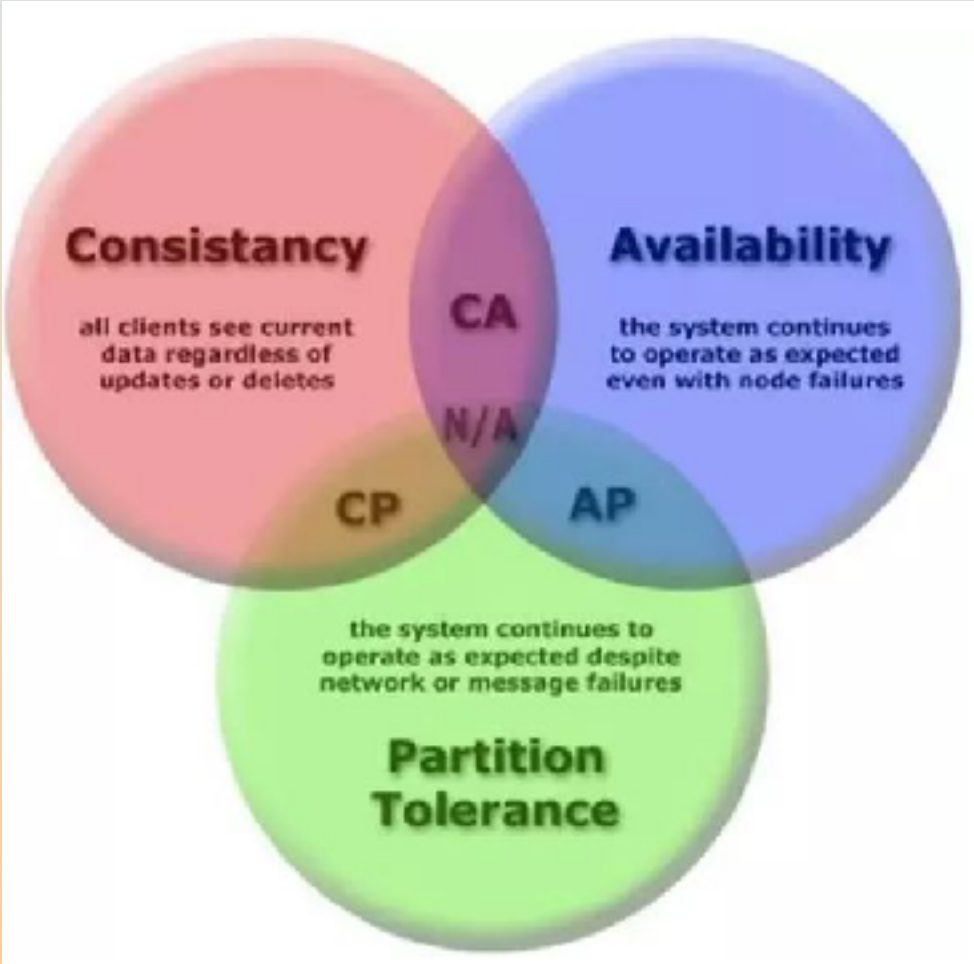
- 主从同步方式，
  - 写请求首先发送给主副本，主副本同步更新到其它副本后返回。这种方式可以保证副本之间数据的强一致性，写成功返回之后从任意副本读到的数据都是一致的。但是可用性很差，只要任意一个副本写失败，写请求将执行失败。

多副本一致性问题

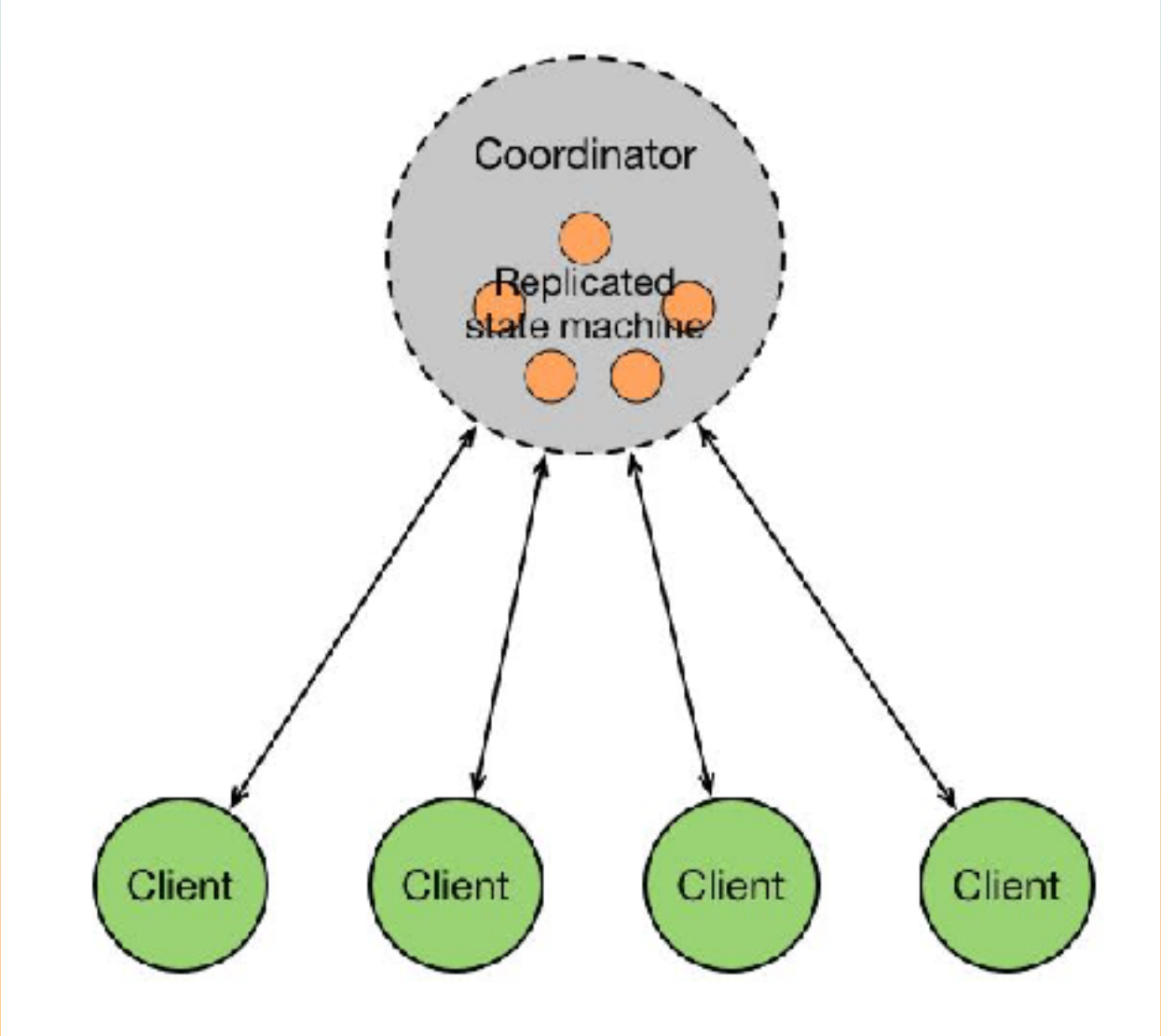


- 主从异步方式：采用异步复制的方式，主副本写成功后立即返回，然后在后台异步的更新其它副本
  - 写请求首先发送给主副本，主副本写成功后立即返回，然后异步的更新其它副本。这种方式可用性较好，只要主副本写成功，写请求就执行成功。但是不能保证副本之间数据的强一致性，写成功返回之后从各个副本读取到的数据不保证一致，只有主副本上是最新的数据，其它副本上的数据落后，只提供最终一致性

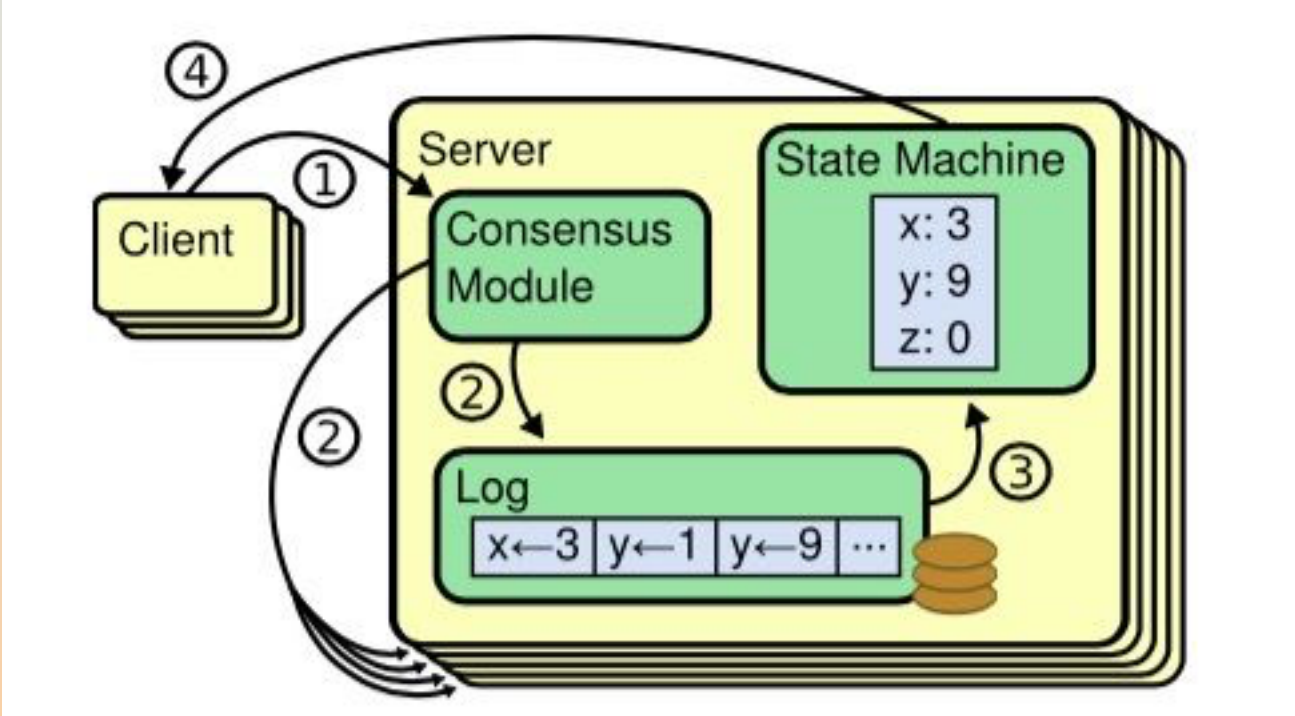
CAP问题


Consistancy
all clients see current data regardless of updates or deletes
CA
Availability
the system continues to operate as expected even with node failures
N/A
CP
AP
the system continues to operate as expected despite network or message failures
Partition Tolerance

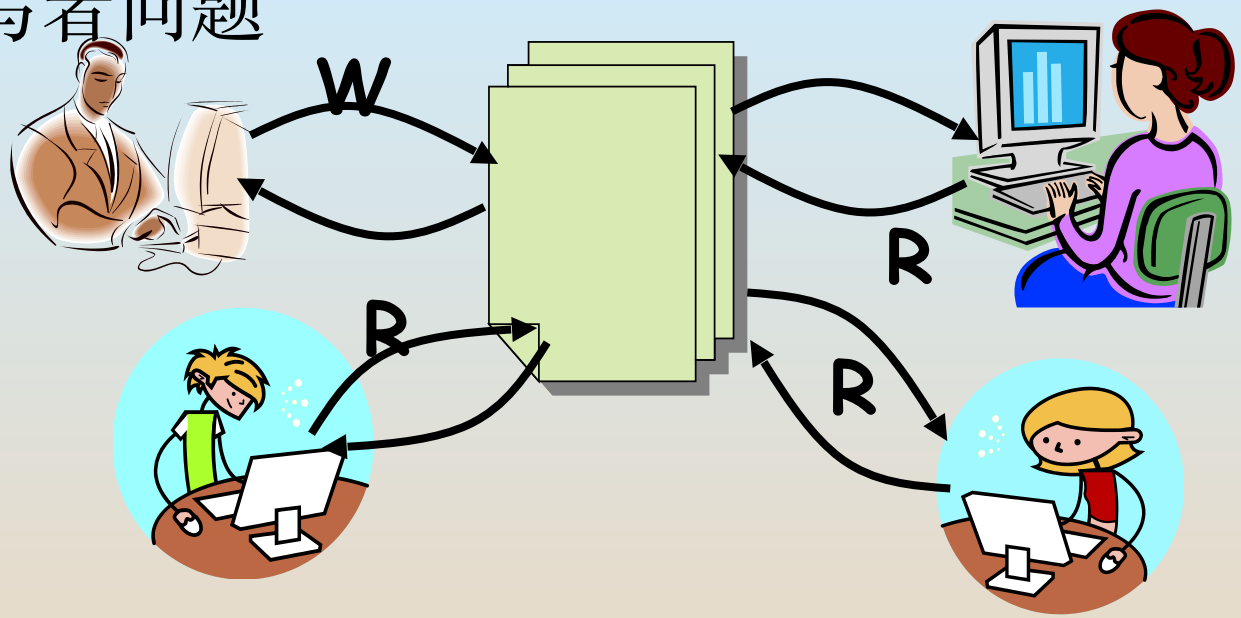
多副本状态机


Coordinator
Replicated
state machine
Client
Client
Client
Client

多副本状态机


④
Server
State Machine
①
Client
Consensus
Module
x: 3
y: 9
z: 0
②
②
③
Log
x←3
y←1
y←9
…

3 读者-写者问题



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - Like to have many readers at the same time
    - Only one writer at a time

1. 利用记录型信号量解决读者-写者问题

•为保证Reader与Writer进程在读或写时互斥，设置一个互斥信号量Wmutex。

•再设置一个整型变量Readcount表示正在读的进程数目。由于只要有一个Reader进程在读，便不允许Writer进程去写。因此，仅当Readcount=0，表示尚无Reader进程在读时，Reader进程才需要执行Wait(Wmutex)操作。

•若wait(Wmutex)操作成功，则Reader进程去读，相应地做Readcount+1操作。

•当Reader进程在执行了Readcount减1操作后其值为0时，才须执行signal(Wmutex)操作，以便让Writer进程写。

•因为Readcount是一个可被多个Reader进程访问的临界资源，因此，应该为它设置一个互斥信号量rmutex。

读者-写者问题可描述如下：

```
Var rmutex, wmutex:semaphore : = 1,1;
   Readcount:integer : = 0;
   begin
   parbegin
    Reader:begin
      repeat
       wait(rmutex);
       if readcount=0 then wait(wmutex);
        Readcount : = Readcount+1;
       signal(rmutex);
            // if I am the first reader tell all others
            // that the database is being read
        ...
       perform read operation;
        …
```

```
        wait(rmutex);
         readcount  :  =  readcount-1;
         if readcount=0 then signal(wmutex);
         signal(rmutex);
        until false;
       end

writer:begin
        repeat
         wait(wmutex);
         perform write operation;
         signal(wmutex);
        until false;
       end
    parend
  end
```

2. 利用信号量集机制解决读者-写者问题

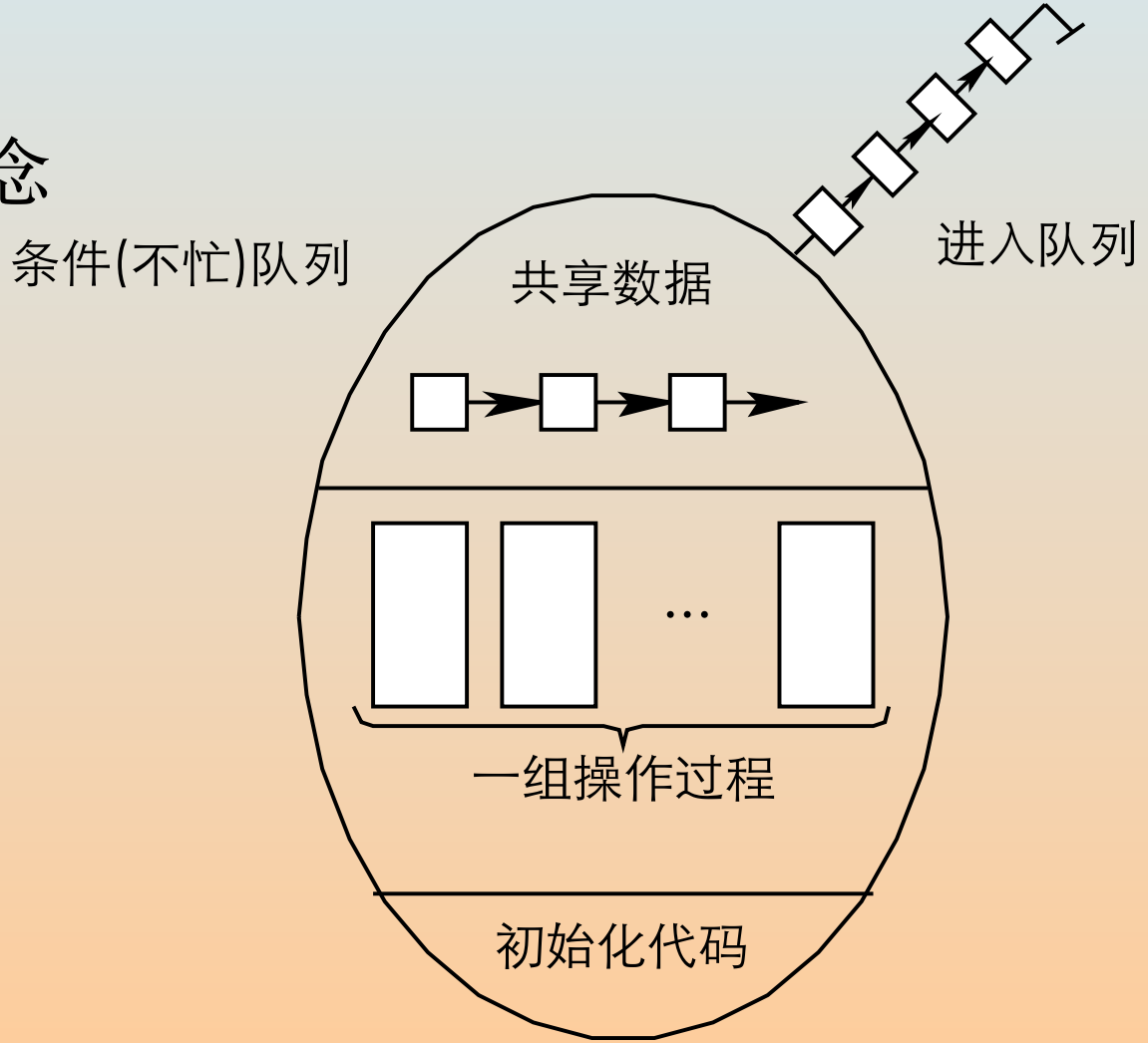
```
Var RN integer;
   L, mx:semaphore  :   =   RN,1;
  begin
   parbegin
    reader:begin
       repeat
        Swait(L,1,1);
        Swait(mx,1,0);
         ...
        perform read operation;
         ...
```

```
Ssignal(L,1);
        until false;
       end

writer:begin
         repeat
          Swait(mx,1,1; L,RN,0);
          perform write operation;
          Ssignal(mx,1);
         until false;
       end
    parend
  end
```

管 程 机 制

1 管程的基本概念



管程的提出：

（1）P.V操作的缺点：

1.易读性差：因为要了解对于一组共享变量及信号量的操作是否正确，则必须通读整个系统或者并发程序。

2.不利于修改和维护：因为程序的局部性很差，所以任一组变量或一段代码的修改都可能影响全局。

3.正确性难以保证：因为操作系统或并发程序通常很大，而P, V操作代码都是由用户编写的，系统无法有效地控制和管理这些P，V操作，要保证这样一个复杂的系统没有逻辑错误是很难的，它将导致死锁现象的产生。

（2）管程的引入

1.把分散在各进程中的临界区集中起来进行管理；

2.防止进程有意或无意的违法同步操作；

3.便于用高级语言来书写程序，也便于程序正确性验证。

- 管程的属性：
- （1）共享性：
- （2）安全性：
- （3）互斥性：

- 管程的组成部分：
- （1）名称
- （2）数据结构说明
- （3）对该数据结构进行操作的一组过程/函数
- （4）初始化语句

- 管程的形式：

TYPE monitor_name = MONITOR;

共享变量说明

define 本管程内所定义、本管程外可调用的过程（函数）名字表;

use　本管程外所定义、本管程内将调用的过程（函数）名字表;

PROCEDURE 过程名（形参表）;

过程局部变量说明;

BEGIN

语句序列;

END;

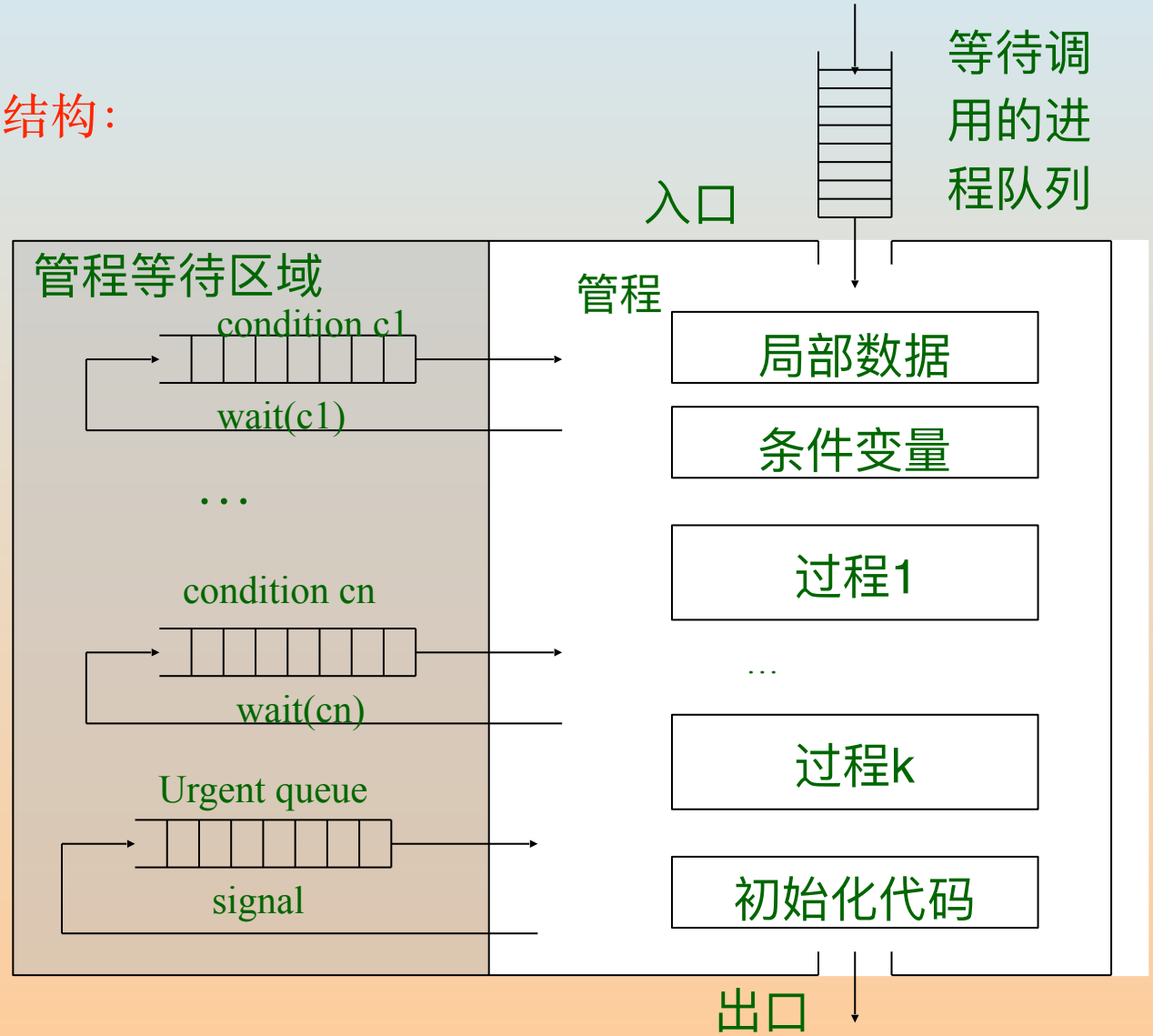
……

```
FUNCTION 函数名（形参表）：值类型;
        函数局部变量说明;
                    BEGIN
                      语句序列;
                    END;
.....
BEGIN
  共享变量初始化语句序列;
END;
```

- 管程的特征：
- （1）模块化：一个管程是一个基本程序单位，可以单独编译；
- （2）抽象数据类型：管程是一种特殊的数据类型，其中不仅有数据，而且有对数据进行操作的代码；
- （3）信息掩蔽：管程是半透明的，管程中的外部过程（函数）实现了某些功能，而这些功能是怎样实现的，在其外部则是不可见的。

- 管程的要素：
- （1）管程中的共享变量在管程外部是不可见的，外部只能通过调用管程中所说明的外部过程（函数）来间接地访问管程中的共享变量；
- （2）为了保证管程共享变量的数据完整性，规定管程互斥进入；
- （3）管程通常是用来管理资源的，因而在管程中应当设有进程等待队以及相应的等待及唤醒操作。

管程的结构：


等待调用的进程队列
入口
管程等待区域
condition c1
wait(c1)
...
condition cn
wait(cn)
Urgent queue
signal
管程
局部数据
条件变量
过程1
...
过程k
初始化代码
出口

- 管程的实例：

```
TYPE SSU = MONITOR
  var busy : boolean;
      nobusy : semaphore;
  define require, return;
  use wait, signal;
  procedure require;
    begin
      if busy then wait(nobusy); /*调用进程加入等待队列*/
      busy := true;
    end;
  procedure return;
    begin
      busy := false;
      signal(nobusy);          /*从等待队列中释放进程*/
    end;
  begin                        /*管程变量初始化*/
    busy := false;
  end;
```

管程的条件变量：

（1）条件变量：当调用管程过程的进程无法运行时，用于阻塞进程的一种信号量。

（2）同步原语wait：当一个管程过程发现无法继续时，它在某些条件变量condition上执行wait，这个动作引起调用进程阻塞。另一个进程可以通过对其伙伴在等待的同一个条件变量condition上执行同步原语signal操作来唤醒等待进程。

（3）条件变量与P、V操作中信号量的区别：使用signal释放等待进程时，可能出现两个进程同时停留在管程内。解决方法：

1.执行signal的进程等待，直到被释放进程退出管程或等待另一个条件。

2.被释放进程等待，直到执行signal的进程退出管程或等待另一个条件。

3.霍尔采用了第一种办法，汉森选择了两者的折衷，规定管程中的过程所执行的signal操作是过程体的最后一个操作。

管程与进程的异同：

（1）管程定义的是公用数据结构，而进程定义的是私有数据结构；

（2）管程把共享变量上的同步操作集中起来，而临界区却分散在每个进程中；

（3）管程是为管理共享资源而建立的，进程主要是为占有系统资源和实现系统并发性而引入的；

2 利用管程解决生产者-消费者问题

在利用管程方法来解决生产者-消费者问题时， 首先便是为它们建立一个管程，并命名为Producer-Consumer, 或简称为PC。其中包括两个过程:

(1) put(item)过程。生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量count来表示在缓冲池中已有的产品数目，当count≥n时， 表示缓冲池已满， 生产者须等待。

(1) put(item)过程。 生产者利用该过程将自己生产的产品投放到缓冲池中， 并用整型变量count来表示在缓冲池中已有的产品数目，当count≥n时，表示缓冲池已满，生产者须等待。

(2) get(item)过程。消费者利用该过程从缓冲池中取出一个产品，当count≤0时，表示缓冲池中已无可取用的产品，消费者应等待。

```
type producer-consumer=monitor
    Var in,out,count:integer;
     buffer:array [0,...,n-1] of item;
     notfull, notempty:condition;
     procedure entry put(item)
       begin
        if count≥n then notfull.wait;
         buffer(in) : = nextp;
         in : = (in+1) mod n;
         count : = count+1;
         if notempty.queue then notempty.signal;
        end
```

```
procedure entry get(item)
      begin
       if count≤0 then notempty.wait;
       nextc  :  =  buffer(out);
       out  :  =  (out+1) mod n;
       count  :  =  count-1;
       if notfull.quene then notfull.signal;
      end
  begin in  :  =  out  :  =  0; count  :  =  0 end
```

在利用管程解决生产者-消费者问题时， 其中的生产者和消费者可描述为:

```
producer:begin
        repeat
         produce an item in nextp;
         PC.put(item);
        until false;
       end
    consumer:begin
        repeat
         PC.get(item);
         consume the item in nextc;
        until false;
       end
```

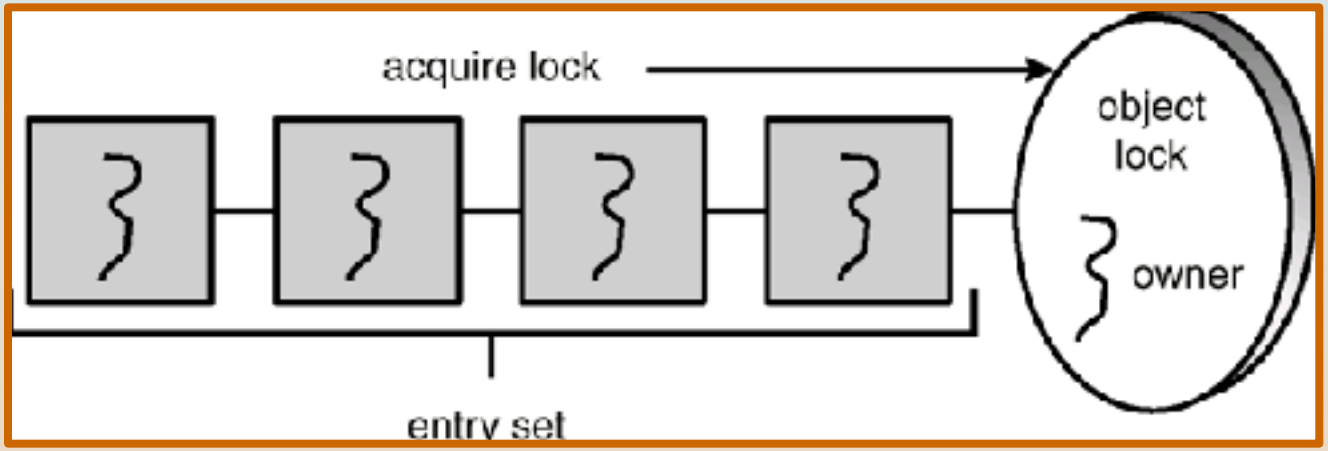
Java Synchronization

- Synchronized, wait(), notify() statements
- Multiple Notifications
- Block Synchronization
- Java Semaphores
- Java Monitors

synchronized Statement

- Every object has a lock associated with it
- Calling a synchronized method requires “owning” the lock
- If a calling thread does not own the lock (another thread already owns it), the calling thread is placed in the wait set for the object’s lock
- The lock is released when a thread exits the synchronized method

Entry Set

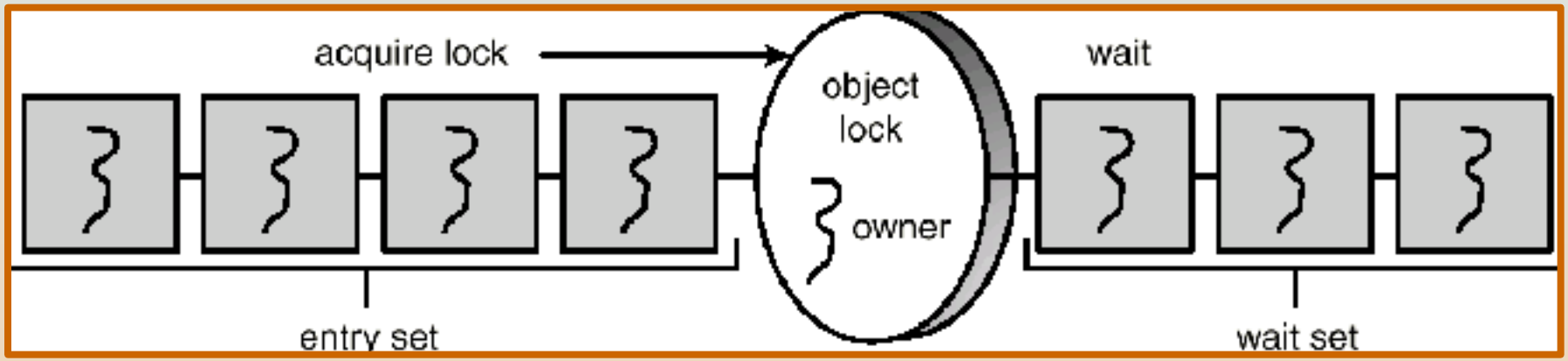


The wait() Method

When a thread calls wait(), the following occurs:

1. the thread releases the object lock
2. thread state is set to blocked
3. thread is placed in the wait set

Entry and Wait Sets



The notify() Method

When a thread calls notify(), the following occurs:

1. selects an arbitrary thread *T* from the wait set
2. moves *T* to the entry set
3. sets *T* to Runnable

T can now compete for the object's lock again

Multiple Notifications

- notify() selects an arbitrary thread from the wait set.
  *This may not be the thread that you want to be selected.
  - Java does not allow you to specify the thread to be selected

- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
  - notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set

Block Synchronization

- Scope of lock is time between lock acquire and release
- Blocks of code – rather than entire methods – may be declared as **synchronized**
- This yields a lock scope that is typically smaller than a synchronized method

Java Semaphores

Java does not provide a semaphore, but a basic semaphore can be constructed using Java synchronization mechanism

进 程 通 信

Linux进程通信的类型

- 管道（pipe）
- 命名管道（FIFO）
- 信号（signal）
- 信号量
- 共享内存
- 内存映射
- 消息队列
- 套接字（socket)

进 程 通 信

类型	无连接	可靠	流控制	记录消息类型	优先级
普通PIPE	N	Y	Y		N
流PIPE	N	Y	Y		N
命名PIPE(FIFO)	N	Y	Y		N
消息队列	N	Y	Y		Y
信号量	N	Y	Y		Y
共享存储	N	Y	Y		Y
UNIX流SOCKET	N	Y	Y		N
UNIX数据包SOCKET	Y	Y	N		N

管道通信

- 管道这种通讯方式有两种限制，一是半双工的通信，数据只能单向流动，二是只能在具有亲缘关系的进程间使用，进程的亲缘关系通常是指父子进程关系，管道允许一个进程和另一个与它有共同祖先的进程之间进行通信

- 流管道s_pipe: 去除了第一种限制,可以双向传输

- 命名管道:name_pipe克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信，可以用于任何两个进程之间的通信

信号量通信

- 信号量用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

- linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）；

消息队列通信

- 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。
- 消息队列是消息的链接表。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。
- 消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

内存映射通信

- 内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现

共享存储通信

- 共享内存就是映射一段能被其他进程所访问的内存，让多个进程可以访问同一块内存空间，这段共享内存由一个进程创建，但多个进程都可以访问。
- 共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

Socket通信

- 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信
- socket是更为一般的进程间通信机制，可用于不同机器之间的进程间通信。由Unix系统的BSD分支起源，但现在一般可以移植到其它类Unix和Linux系统上，应用十分广泛。

各种通信方式比较

- 管道：速度慢，容量有限，只有父子进程能通讯
- 命名管道（FIFO）：任何进程间都能通讯，但速度慢
- 消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题
- 信号量：不能传递复杂消息，只能用来同步
- 共享内存区：容量可控，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全。当然，共享内存区同样可以用作线程间通讯，不过一般没这个必要，线程间本来就已经共享了同一进程内的一块内存

各种通信方式比较

- 若用户传递的信息较少，需要通过信号来触发某些行为，软中断信号/信号量：
- 若进程间要求传递的信息量比较大或者进程间存在交换数据的要求：
  - 无名管道简单方便．但局限于单向通信的工作方式．并且只能在创建它的进程及其子孙进程之间实现管道的共享：
  - 有名管道虽然可以提供给任意关系的进程使用．但是由于其长期存在于系统之中，使用不当容易出错．所以普通用户一般不建议使用。
  - 消息缓冲可以不再局限于父子进程，而允许任意进程通过共享消息队列来实现进程间通信，并由系统调用函数来实现消息发送和接收之间的同步，从而使得用户在使用消息缓冲进行通信时不再需要考虑同步问题，使用方便，但是信息的复制需要额外消耗CPU的时间，不适宜于信息量大或操作频繁的场合。

各种通信方式比较

- 共享内存针对消息缓冲的缺点改而利用内存缓冲区直接交换信息，无须复制，快捷、信息量大是其优点。

- 共享内存的通信方式是通过将共享的内存缓冲区直接附加到进程的虚拟地址空间中来实现的，
  - 这些进程之间的读写操作的同步问题操作系统无法实现。必须由各进程利用其他同步工具解决。
  - 由于内存实体存在于计算机系统中，所以只能由处于同一个计算机系统中的诸进程共享。不方便网络通信。

- 共享内存块提供了在任意数量的进程之间进行高效双向通信的机制。每个使用者都可以读取写入数据，但是所有程序之间必须达成并遵守一定的协议，以防止诸如在读取信息之前覆写内存空间等竞争状态的出现。

- Linux无法严格保证提供对共享内存块的独占访问，甚至是在使用IPC_PRIVATE创建新的共享内存块的时候也不能保证访问的独占性。 同时，多个使用共享内存块的进程之间必须协调使用同一个键值。

进 程 通 信

进程通信的类型

1. 共享存储器系统(Shared-Memory System)

(1) 基于共享数据结构的通信方式。

(2) 基于共享存储区的通信方式。

2. 消息传递系统(Message passing system)

不论是单机系统、多机系统，还是计算机网络，消息传递机制都是用得最广泛的一种进程间通信的机制。在消息传递系统中，进程间的数据交换，是以格式化的消息(message)为单位的；在计算机网络中，又把message称为报文。程序员直接利用系统提供的一组通信命令(原语)进行通信。操作系统隐藏了通信的实现细节，大大减化了通信程序编制的复杂性，而获得广泛的应用。消息传递系统的通信方式属于高级通信方式。又因其实现方式的不同而进一步分成直接通信方式和间接通信方式两种。

3. 管道(Pipe)通信

所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程)， 以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于UNIX系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：① 互斥，即当一个进程正在对pipe执行读/写操作时，其它(另一)进程必须等待。 ② 同步，指当写(输入)进程把一定数量(如4 KB)的数据写入pipe，便去睡眠等待， 直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空pipe时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。③ 确定对方是否存在，只有确定了对方已存在时，才能进行通信。

消息传递通信的实现方法

1. 直接通信方式

这是指发送进程利用OS所提供的发送命令，直接把消息发送给目标进程。此时，要求发送进程和接收进程都以显式方式提供对方的标识符。通常，系统提供下述两条通信命令(原语)：

Send(Receiver, message); 发送一个消息给接收进程；

Receive(Sender, message); 接收Sender发来的消息；

例如，原语Send(P_2，m_1)表示将消息m_1发送给接收进程P_2；而原语Receive(P_1，m_1)则表示接收由P_1发来的消息m_1。

在某些情况下，接收进程可与多个发送进程通信，因此，它不可能事先指定发送进程。例如，用于提供打印服务的进程，它可以接收来自任何一个进程的“打印请求”消息。对于这样的应用，在接收进程接收消息的原语中的源进程参数，是完成通信后的返回值，接收原语可表示为：

Receive (id, message);

我们还可以利用直接通信原语，来解决生产者-消费者问题。当生产者生产出一个产品(消息)后，便用Send原语将消息发送给消费者进程；而消费者进程则利用Receive原语来得到一个消息。如果消息尚未生产出来，消费者必须等待，直至生产者进程将消息发送过来。生产者-消费者的通信过程可分别描述如下：

```
repeat
  ...
 produce an item in nextp;
  ...
 send(consumer, nextp);
until false;
repeat
 receive(producer, nextc);
  ...
 consume the item in nextc;
until false;
```

2. 间接通信方式

(1) 信箱的创建和撤消。进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性(公用、私用或共享)；对于共享信箱， 还应给出共享者的名字。当进程不再需要读信箱时，可用信箱撤消原语将之撤消。

(2) 消息的发送和接收。当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的下述通信原语进行通信。

Send(mailbox, message); 将一个消息发送到指定信箱；

Receive(mailbox, message); 从指定信箱中接收一个消息；

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。

1) 私用信箱

用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路的信箱来实现。　当拥有该信箱的进程结束时，信箱也随之消失。

2) 公用信箱

它由操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用双向通信链路的信箱来实现。通常，公用信箱在系统运行期间始终存在。

3) 共享信箱

它由某进程创建，在创建时或创建后，指明它是可共享的，同时须指出共享进程(用户)的名字。信箱的拥有者和共享者，都有权从信箱中取走发送给自己的消息。

在利用信箱通信时，在发送进程和接收进程之间，存在以下四种关系：

(1) 一对一关系。这时可为发送进程和接收进程建立一条两者专用的通信链路，使两者之间的交互不受其他进程的干扰。

(2) 多对一关系。允许提供服务的进程与多个用户进程之间进行交互，也称为客户/服务器交互(client/server interaction)。

(3) 一对多关系。允许一个发送进程与多个接收进程进行交互，使发送进程可用广播方式，向接收者(多个)发送消息。

(4) 多对多关系。允许建立一个公用信箱，让多个进程都能向信箱中投递消息；也可从信箱中取走属于自己的消息。

消息传递系统实现中的若干问题

1. 通信链路(communication link)

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条通信链路。有两种方式建立通信链路。

第一种方式是：由发送进程在通信之前，用显式的"建立连接"命令(原语)请求系统为之建立一条通信链路；在链路使用完后，也用显式方式拆除链路。

第二种方式是：发送进程无须明确提出建立链路的请求，只须利用系统提供的发送命令(原语)，系统会自动地为之建立一条链路。这种方式主要用于单机系统中。

根据通信链路的连接方法，又可把通信链路分为两类： ① 点–点连接通信链路，这时的一条链路只连接两个结点(进程)；② 多点连接链路，指用一条链路连接多个($n > 2$)结点(进程)。而根据通信方式的不同，则又可把链路分成两种： ① 单向通信链路，只允许发送进程向接收进程发送消息； ② 双向链路，既允许由进程A向进程B发送消息，也允许进程B同时向进程A发送消息。

2. 消息的格式

在某些OS中，消息是采用比较短的定长消息格式，这减少了对消息的处理和存储开销。这种方式可用于办公自动化系统中，为用户提供快速的便笺式通信；但这对要发送较长消息的用户是不方便的。

在有的OS中，采用另一种变长的消息格式，即进程所发送消息的长度是可变的。系统在处理和存储变长消息时，须付出更多的开销，但方便了用户。　这两种消息格式各有其优缺点，故在很多系统(包括计算机网络)中，是同时都用的。

消息缓冲队列通信机制

1. 消息缓冲队列通信机制中的数据结构

(1) 消息缓冲区。在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区。它可描述如下：

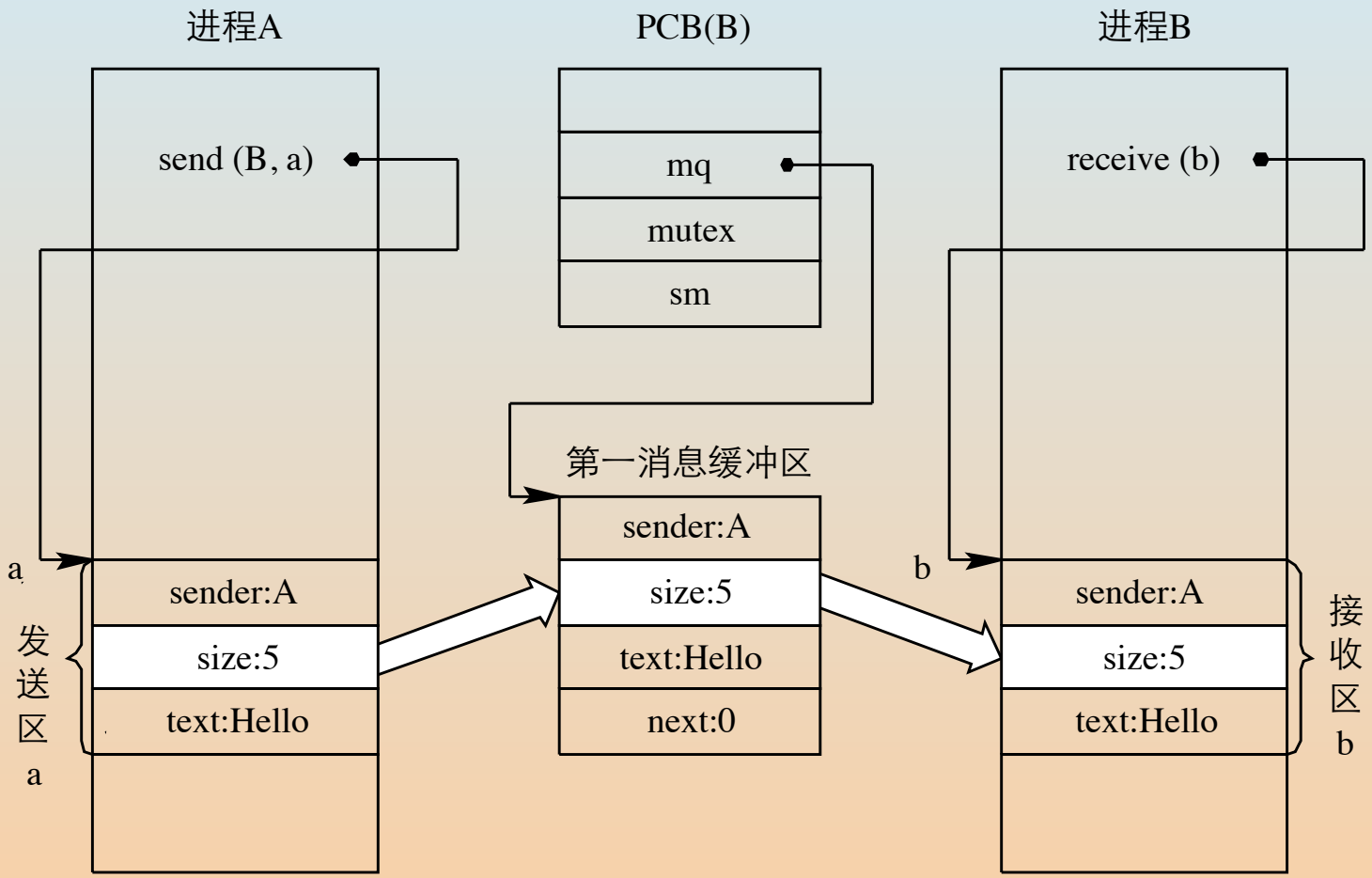
```
type message buffer=record
        sender; 发送者进程标识符
        size; 消息长度
        text; 消息正文
      next; 指向下一个消息缓冲区的指针
    end
```

(2) PCB中有关通信的数据项。在利用消息缓冲队列通信机制时，在设置消息缓冲队列的同时，还应增加用于对消息队列进行操作和实现同步的信号量，并将它们置入进程的PCB中。在PCB中应增加的数据项可描述如下：

```
type processcontrol block=record
 ...
mq; 消息队列队首指针
mutex; 消息队列互斥信号量
sm; 消息队列资源信号量
  ...
end
```

2. 发送原语

发送进程在利用发送原语发送消息之前，应先在自己的内存空间，设置一发送区a，把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。发送原语首先根据发送区a中所设置的消息长度a.size来申请一缓冲区i，接着，把发送区a中的信息复制到缓冲区i中。为了能将i挂在接收进程的消息队列mq上，应先获得接收进程的内部标识符j，然后将i挂在j.mq上。由于该队列属于临界资源， 故在执行insert操作的前后，都要执行wait和signal操作。


进程A
PCB(B)
进程B
send (B, a)
mq
mutex
sm
receive (b)
第一消息缓冲区
sender:A
size:5
text:Hello
next:0
a
b
sender:A
size:5
text:Hello
发送区a
sender:A
size:5
text:Hello
接收区b

消息缓冲通信

```
procedure send(receiver, a)
    begin
      getbuf(a.size,i);                    根据a.size申请缓冲区;
      i.sender:  =  a.sender;  将发送区a中的信息复制到消息缓冲区之中;
      i.size:  =  a.size;
      i.text:  =  a.text;
      i.next:  =  0;
    getid(PCB set, receiver.j);   获得接收进程内部标识符;
    wait(j.mutex);
    insert(j.mq, i);   将消息缓冲区插入消息队列;
    signal(j.mutex);
    signal(j.sm);
  end
```

3. 接收原语

接收原语描述如下:

```
procedure receive(b)
  begin
   j : = internal name; j为接收进程内部的标识符;
   wait(j.sm);
   wait(j.mutex);
   remove(j.mq, i); 将消息队列中第一个消息移出;
   signal(j.mutex);
   b.sender : = i.sender; 将消息缓冲区i中的信息复制到接收区b;
   b.size : = i.size;
   b.text : = i.text;
 end
```