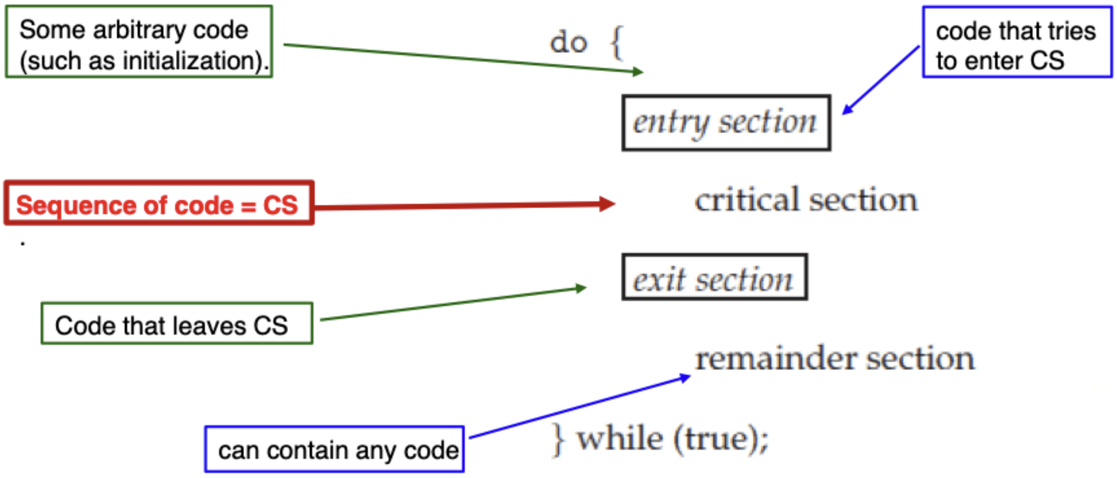
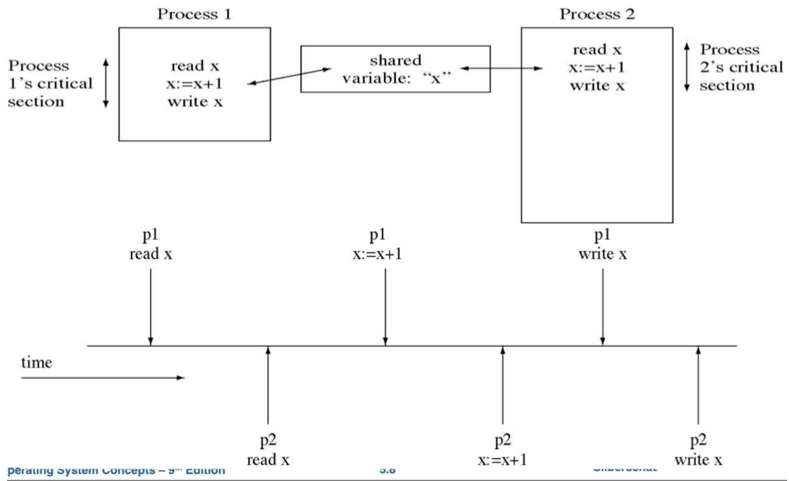
4 Process Synchronization

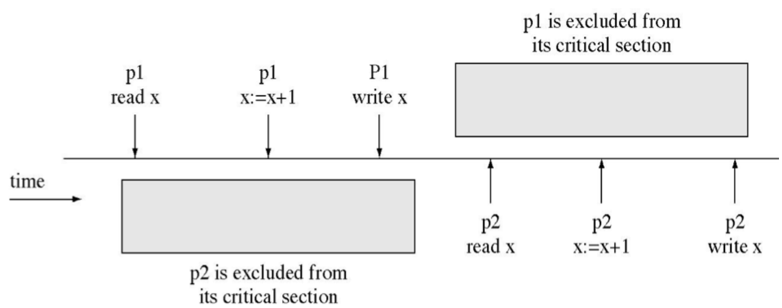
- Background
  - What is Process Synchronization (PS)?
  - PS is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources, at one time.PS是协调进程执行的任务，使两个进程不能同时访问相同的共享数据和资源。
  - n processes all competing to use some shared resource.N个进程都在竞争使用一些共享资源
  - Concurrent access to shared data may result in data inconsistency.
    - 并发访问共享数据可能导致数据不一致。
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
    - 维护数据一致性需要一些机制来确保协作流程的有序执行。
  - Race condition: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
    - 竞态条件:多个进程同时访问和操作共享数据的情况。共享数据的最终值取决于哪个进程最后完成。
  - To prevent race conditions, concurrent processes must be synchronized
    - 为了防止竞争条件，并发进程必须同步
- The Critical-Section Problem
  - Critical Sections are sequences of code that cannot be interleaved among multiple threads/processes.
    - 临界区是不能在多个线程/进程之间交错的代码序列。
  - Each (concurrent) thread/process has a code segment, called Critical Section (CS),in which the shared data is accessed.
    - 每个(并发的)线程/进程都有一个称为临界段(Critical Section, CS)的代码段，在这个代码段中访问共享数据。
  - When using critical sections, the code can be broken down into the following sections:
    - 当使用临界段时，代码可以分解为以下部分:
  - a



- Race condition updating a variable
  - CS = codes that reference one variable in a “read-update-write” fashionCS =以“读-更新-写”方式引用一个变量的代码
  - a



  - Multiprogramming allows logical parallelism (multiple programs to exist in memory at the same time) uses devices efficiently but we lose correctness when there is a race condition.
    - 多道编程允许逻辑并行(多个程序同时存在于内存中)有效地使用设备，但当存在竞争条件时，我们会失去正确性。
  - •Avoid/ forbid / deny execution in parallel inside critical section, even we lose some efficiency, but we gain correctness.
    - 避免/禁止/拒绝在临界区内并行执行，即使我们损失了一些效率，但我们获得了正确性。



- Solutions to CS problem
  - Concurrent processes come into conflict when they use thesame resource (competitively or shared)
    - 当并发进程使用相同的资源(竞争或共享)时，它们会发生冲突。
  - for example: I/O devices, memory, processor time, clock
    - 例如:I/O设备、内存、处理器时间、时钟
  - There are 3 requirements that must stand for a correctsolution:
    - 正确的解决方案必须满足3个要求:
  - ❑ Mutual exclusion❑ Progress❑ Bounded waiting
    - 相互排斥、进步3有限等待
- Mutual Exclusion: When a process/thread is executing in its critical section, no other process/threads can be executing in their critical sections.
  - 互斥:当一个进程/线程在它的临界区执行时，没有其他进程/线程可以在它们的临界区执行。
- Progress: If no process/thread is executing in its critical section, and if there are some processes/threads that wish to enter their critical sections, then one of these processes/threads will get into the critical section. No process running outside its critical region may block any process.
  - Progress:如果没有进程/线程在其临界区执行，并且如果有一些进程/线程希望进入其临界区，那么其中一个进程/线程将进入临界区。任何在临界区域外运行的进程都不能阻塞其他进程。
- Bounded Waiting: No process/thread should have to wait forever to enter into the critical section.
  - 有界等待:没有进程/线程必须永远等待才能进入临界区。
  - - the waiting time of a process/thread outside a critical section should be limited (otherwise the process/thread could suffer from starvation).
    - 临界区外的进程/线程的等待时间应该被限制(否则进程/线程可能会饿死)。
- Types of solutions to CS problem
  - Software solutions
    - 软件解决方案
    - ❑algorithms whose correctness relies only on the assumption that only one process/thread at a time can access a memory location/resource
      - 这种算法的正确性仅仅依赖于一次只有一个进程/线程可以访问一个内存位置/资源的假设
  - ❑ Hardware solutions
    - 3、硬件解决方案
    - ❑ rely on special machine instructions for “locking”

                    - 3依靠特殊的机器指令进行“锁定”
            - ❑ Operating System and Programming Language solutions (e.g., Java)
                - 3、操作系统和编程语言解决方案(如Java)
                - ❑provide specific functions and data structures for programmers to use for synchronization.
                    - 3、提供特定的功能和数据结构，供程序员用于同步。
- Software solutions
    - Peterson’s Solution
        - 基本结构

```
do {
  entry section
   critical section - CS
  exit section
   remainder section - RS
} while(TRUE)
```

```
do {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);

        CRITICAL SECTION

    flag[i] = FALSE;

        REMAINDER SECTION
 }
while (TRUE);
```

Initialization: flag[0]:=flag[1]:=false
turn:= 0 or 1

Process P_0

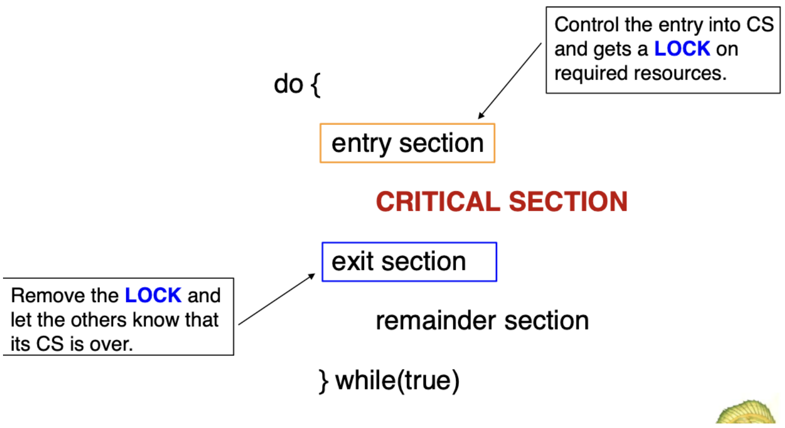
```
do {
// Critical Section
   flag[0] = true; // It means P0 is ready to
enter its critical section
   turn = 1; // It means that if P1 wants to
enter than allow it to enter and P0 will wait
 // Condition to check if the flag of P1 is true
and turn == 1, this will only break when one
of the conditions gets false.
   while (flag[1] && turn == 1); // do
nothing
/critical section/
// It sets the flag of P0 to false because it
has completed its critical section.
   flag[0] = false;
// Remainder Section
} while (true);
```

Process P_1

```
do {
 // Critical Section
 // It means process P0 is ready to enter its
critical section
   flag[1] = true;
   turn = 0; // It means that if P0 wants to
enter than allow it to enter and P1 will wait
// Condition to check if the flag of P0 is true
and turn == 0, this will only break when one
of the conditions gets false.
   while (flag[0] && turn == 0); // do nothing
/critical section/
//it sets the flag of P1 to false because it has
completed its critical section.
   flag[1] = false;
// Remainder Section
} while (true);
```

- is used for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.用于互斥，允许两个或多个进程共享单个使用的资源而不发生冲突，仅使用共享内存进行通信。
- The central problem is to design the entry and exit sections核心问题是入口和出口部分的设计
- Only 2 processes, P0 and P1
- Processes may share some common variables to synchronize their actions.
  - 进程可以共享一些公共变量来同步它们的操作。
- int turn;
- // indicates whose turn it is to enter the critical section.
  - //该轮到谁进入临界区。
- boolean flag[2];
  - // initialized FALSE,
    - //初始化FALSE;
  - // indicates when a process wants to enter into their CS.
    - //指示进程何时想要进入其CS。
  - // flag[i] = true implies that process Pi is ready (i = 0,1)
    - // flag[i] = true表示进程Pi已经就绪(i = 0,1)
- NEED BOTH the turn and flag[2] to guarantee Mutual Exclusion, Bounded waiting, and Progress.
  - 需要turn和flag[2]来保证互斥、有限等待和进度。

- This solution is correct:
- The three CS requirements are met:
  - Mutual Exclusion is assured as only one process can access the critical section at any time.
    - 互斥保证了在任何时候只有一个进程可以访问临界区。
    - each Pi enters its critical section only if either:flag[j] = false or turn = i
      - 只有当flag[j] = false或turn = i时，每个Pi才会进入临界区
  - ❑ Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
    - 这种进步也是有保证的，因为一个在临界区外的进程不会阻止其他进程进入临界区外。
  - ❑ Bounded Waiting is preserved as every process gets a fair chance.
    - ●有限度的等待，因为每个进程都有公平的机会
- Hardware Solutions
  - using LOCKS
    - Remove the LOCK and let the others know that its CS is over.
      - 移除锁，让其他人知道它的CS结束了。
    - 控制进入CS的入口，并获得所需资源的锁。
    - Control the entry into CS and gets a LOCK on required resources.



    - Single-processor environment
      - could disable interrupts Effectively stops scheduling other processes可以禁用中断有效地停止调度其他进程
      - Initially: lock value is set to 0
      - Lock value = 0 means the critical section is currently vacant and no process is present inside it.
        - 锁值= 0表示临界区当前是空的，并且其中没有进程存在。
      - Lock value = 1 means the critical section is currently occupied and a process is present inside it.
        - 锁值= 1表示当前临界区被占用，并且其中存在进程。

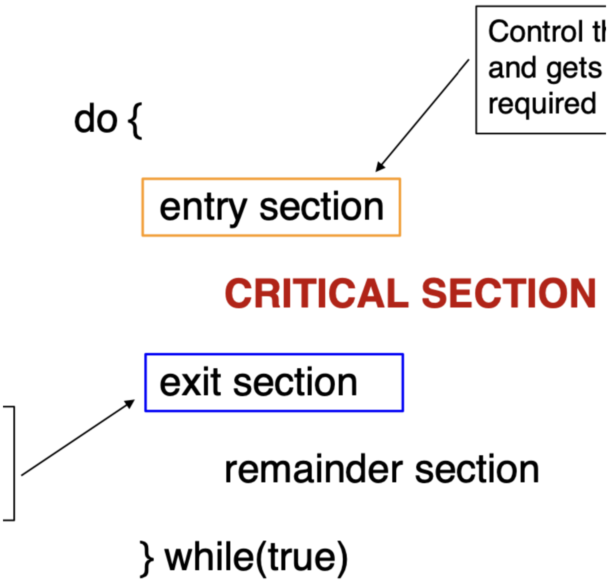
```
do {
  while (test_and_set(&lock))
    ; /* do nothing */

    /* critical section */

  lock = false;

    /* remainder section */
} while (true);
```

- satisfy the mutual exclusion requirement, but unfortunately do not guarantee bounded waiting.满足互斥要求，但不能保证有界等待。
- Multi-processor environment
  - - provides special atomic hardware instructions. Atomic means non-interruptable (i.e., the instruction executes as one unit)
    - -提供特殊的原子硬件指令。原子意味着不可中断(即，指令作为一个单元执行)
  - - a global variable lock is initialized to 0.- the only Pi that can enter CS is the one which finds lock = 0
    - —全局变量锁初始化为0。-唯一可以进入CS的Pi是发现lock = 0的Pi
  - - this Pi excludes all other Pj by setting lock to 1.
    - -该Pi通过将lock设置为1来排除所有其他Pj。

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */

    /* critical section */

  lock = 0;

    /* remainder section */
} while (true);
```

- 评估
  - Advantages
    - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
      - 适用于单个处理器或共享主存的多个处理器上的任意数量的进程
    - Simple and easy to verify
      - 简单，易于验证
    - It can be used to support multiple critical sections; each critical section can be defined by its own variable
      - 可支持多个临界截面;每个临界区都可以用它自己的变量来定义
  - Disadvantages
    - Busy-waiting is when a process is waiting for access to a critical section it continues to consume processor time.

        - 忙碌等待是指进程在等待访问临界区时，它会继续消耗处理器时间。
      - Starvation is possible when a process executes its critical section, and more than one process is waiting for a long time.
        - 当一个进程执行它的临界区，并且有多个进程等待很长时间时，就有可能出现饥饿。
      - Deadlock is the permanent blocking of a set of processes waiting an event (the freeing up of CS) that can only be triggered by another blocked process in the set
        - 死锁是一组进程的永久阻塞，等待一个事件(释放CS)，该事件只能由该组中另一个被阻塞的进程触发
  -
- Synchronization Hardware
  - Solution to CS Problem using LOCKS
    - 控制进入CS的入口，并获得所需资源的锁。
    - Control the entry into CS and gets a LOCK on required resources.



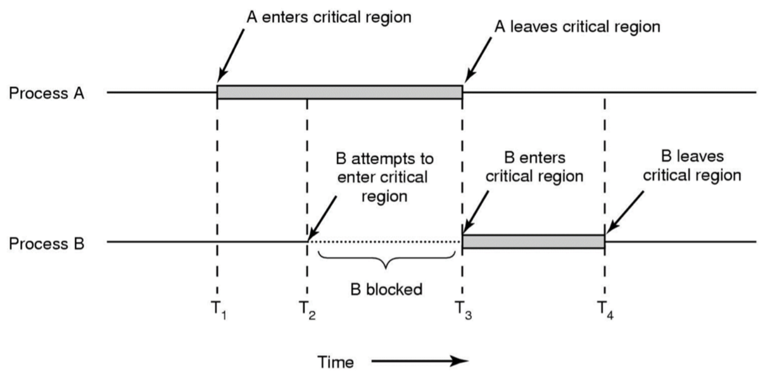
    - Remove the LOCK and let the others know that its CS is over.
      - 移除锁，让其他人知道它的CS结束了。
  - Advantages
    - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
      - 适用于单个处理器或共享主存的多个处理器上的任意数量的进程
    - Simple and easy to verify
      - 简单，易于验证

        - It can be used to support multiple critical sections; each critical section can be defined by its own variable
            - 可支持多个临界截面;每个临界区都可以用它自己的变量来定义
    - Disadvantages
        - Busy-waiting is when a process is waiting for access to a critical section it continues to consume processor time.
            - 忙碌等待是指进程在等待访问临界区时，它会继续消耗处理器时间。
        - Starvation is possible when a process executes its critical section ,and more than one process is waiting for a long time.
            - 当一个进程执行它的临界区，并且有多个进程等待很长时间时，就有可能出现饥饿。
        - Deadlock is the permanent blocking of a set of processes waiting an event (the freeing up of CS) that can only be triggered by another blocked process in the set.
            - 死锁是一组进程的永久阻塞，等待一个事件(释放CS)，该事件只能由该组中另一个被阻塞的进程触发。
- Operating Systems andProgramming Language Solutions Mutex- Semaphore
    - Mutex Locks / Mutual exclusion
        - A mutex is a programming flag used to grab and release an object.
            - 互斥锁是用于获取和释放对象的编程标志。
        - ▪ When data processing is started that cannot be performed simultaneously elsewhere in the system, the mutex is set to lock which blocks other attempts to use it.
            - 当数据处理开始时，不能在系统的其他地方同时执行，互斥锁被设置为锁定，阻止其他尝试使用它。
        - ▪ The mutex is set to unlock when the data are no longer needed, or the routine is finished.
            - 互斥锁被设置为在不再需要数据或例程结束时解锁。
        - –To enforce mutex at the kernel level and prevent the corruption of shared data structures - disable interrupts for the smallest number of instructions is the best way.
            - 为了在内核级别强制互斥并防止共享数据结构的损坏，对最少数量的指令禁用中断是最好的方法。
        - –To enforce mutex in the software areas – use the busy-wait mechanism
            - 为了在软件领域强制互斥——使用忙等待机制
            - busy-waiting mechanism is a mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.
                - 忙碌等待机制是一种机制，在这种机制中，进程在无限循环中执行，等待锁变量的值来指示可用性。

- using mutex is to acquire a lock prior to entering a critical section, and to release it when exiting使用互斥锁是为了在进入临界区之前获得锁，并在退出临界区时释放锁

  ```
  do {
      Acquire Lock
        CRITICAL SECTION
      Release Lock
        REMAINDER SECTION
  } while (TRUE);
  ```

- Mutex object is locked or unlocked by the process requesting or releasing the resource互斥对象由请求或释放资源的进程锁定或解锁

- This type of mutex lock is called a spinlock because the process“spins” while waiting for the lock to become available
  - 这种类型的互斥锁被称为自旋锁，因为进程在等待锁可用时“自旋”

- Semaphores
  - 简介
    - A semaphore S may be initialized to a non-negative integer value.
      - 信号量S可以初始化为非负整数值。
    - - is accessed only through two standard atomic operations: wait() and signal().
      - -只能通过两个标准原子操作:wait()和signal()来访问。
      - wait() operation decrements the semaphore value
        - Wait()操作减少信号量的值
        - If the S<0, then the process executing the wait() is blocked. Otherwise, the process continues execution.
          - 如果S<0，则执行wait()的进程被阻塞。否则，流程继续执行。
      - signal() operation increments the semaphore value.
        - Signal()操作增加信号量的值。

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

ore value.

```
signal(S) {
    S++;
}
```

- Using semaphores for solving CS Problem
  - For n processes
    - 对于n个过程
  - Initialize semaphore S to 1
    - 将信号量S初始化为1
  - Then only one process is allowedi nto CS (mutual exclusion)
    - 然后只允许一个进程进入CS(互斥)
  - To allow k processes into CS at a time, simply initialize mutex to k
    - 要一次允许k个进程进入CS，只需将互斥锁初始化为k

      ```
      Process Pi:

      do {
        wait(S);
         CRITICAL SECTION
        signal(S);
         RS
      } while(true)
      ```
- There are two main types of semaphores:
  - COUNTING SEMAPHORE – allow an arbitrary resource count.
    - 计数信号量-允许任意资源计数。
    - Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.
      - 它的取值范围可以是不受限制的域。它用于控制对具有多个实例的资源的访问。

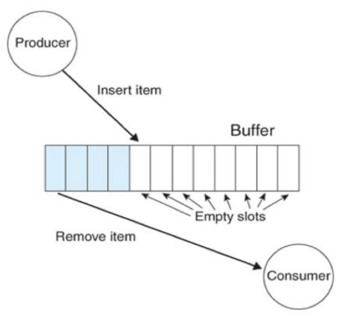
- The semaphore S is initialized to the number of available resources.
  - 信号量S初始化为可用资源的数量。
- Each process that uses a resource, it performs a WAIT()operation on the semaphore (thereby decrementing the number of available resources).
  - 每个使用资源的进程都会对信号量执行WAIT()操作(从而减少可用资源的数量)。
- When a process releases a resource, it performs a SIGNAL()operation (incrementing the number of available resources).
  - 当进程释放资源时，它执行SIGNAL()操作(增加可用资源的数量)。
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will be block until the count becomes greater than 0.
  - 当信号量的计数变为0时，表示正在使用所有资源。在此之后，希望使用资源的进程将被阻塞，直到计数大于0。

- ❑ BINARY SEMAPHORE – similar to mutex lock. It can haveonly two values: 0 and 1.
  - 二进制信号量——类似于互斥锁。它只能有两个值:0和1。
  - Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
    - 它的值初始化为1。它用于实现多工序临界截面问题的求解。
  - A binary semaphore may only take on the values 0 and 1.
    - 二进制信号量只能取值0和1。
  - 1. A binary semaphore may be initialized to 1.
    - 1. 二进制信号量可以初始化为1。
  - 2. The WAIT() operation (decrementing) checks the semaphore value.
    - 2. WAIT()操作(递减)检查信号量的值。
    - ▪ If the value is 0, then the process executing the wait() is blocked.
      - ▪如果该值为0，则执行wait()的进程被阻塞。
    - ▪ If the value is 1, then the value is changed to 0 and the processcontinues execution.
      - ▪如果该值为1，则将该值更改为0，进程继续执行。
  - 3. The SIGNAL() operation (incrementing) checks to see if anyprocesses are blocked on this semaphore (semaphore value equals 0).
    - 3.SIGNAL()操作(递增)检查是否有进程在这个信号量上被阻塞(信号量的值等于0)。
    - ▪ If so, then a process blocked by a signal() operation is unblocked.
      - 如果是，那么被signal()操作阻塞的进程被解除阻塞。

- ▪ If no processes are blocked, then the value of the semaphore is set to 1.
    - ▪如果没有进程被阻塞，那么信号量的值被设置为1。
- Mutex vs. Binary semaphore
    - A key difference between the a mutex and a binary semaphore is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
        - 互斥量和二进制信号量之间的一个关键区别是，锁定互斥量(将其值设置为0)的进程必须是解锁互斥量(将其值设置为1)的进程。
    - In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it. (example in the tutorial)
        - 相反，一个进程可以锁定一个二进制信号量，而另一个进程可以解锁它。(教程中的例子)
    - Same issues of semaphore
        - Starvation - when the processes that require a resource are delayed for a long time. Process with high priorities continuously uses the resources preventing low priority process to acquire the resources.
            - 饥饿——需要资源的进程被延迟了很长时间。高优先级进程持续使用资源，阻止低优先级进程获取资源。
        - Deadlock is a condition where no process proceeds for execution, and each waits for resources that have been acquired by the other processes.
            - 死锁是一种没有进程继续执行的情况，每个进程都等待其他进程获得的资源。

- Classical Problems of Synchronization

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```



- The Bounded-Buffer / Producer-Consumer Problem
    - The mutex binary semaphore provides mutual exclusion for accesses tothe buffer pool and is initialized to the value 1.
    - The empty and full semaphores count the number of empty and full buffers.
    - the semaphore empty is initialized to the value n;
    - the semaphore full is initialized to the value 0.

- 互斥二进制信号量为访问缓冲池提供互斥，并初始化为值1。
- 空信号量和满信号量计算空缓冲区和满缓冲区的数量。
- •信号量空初始化为值n;
- •信号量full初始化为值0。

- The Readers–Writers Problem
  - A data set is shared among a number of concurrent processes.
  - • Only one single writer can access the shared data at the same time, any other writers or readers must be blocked.
  - • Allow multiple readers to read at the same time, any writers must be blocked.
  - Solution: Acquiring a reader–writer lock requires specifying the mode of the lock: either read or write access.
  - 数据集在多个并发进程之间共享。
  - •同一时间只有一个读写器可以访问共享数据，任何其他读写器都必须被阻止。
  - •允许多个读者同时阅读，任何作家必须被阻止。
  - 解决方案:获取读写锁需要指定锁的模式:读访问或写访问。
- The Dining-Philosophers Problem
  - Allow only 4 philosophers to be hungry ata time.
  - Allow pickup only if both chopsticks areavailable. (Done in critical section)
  - • Odd # philosopher always picks up leftchopstick 1st,
  - • Even # philosopher always picks up rightchopstick 1st.
    - 一次只允许4个哲学家饿着肚子。
    - •只有当两根筷子都可用时才允许取。(关键区完成)
    - •奇的哲学家总是先拿起左边的筷子,
    - •偶哲学家也总是先拿起右边的筷子。
      -