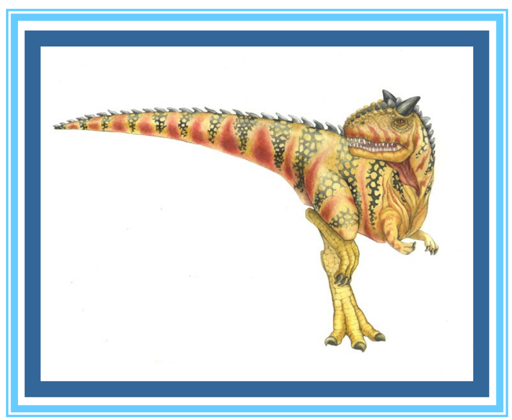
Chapter 7: Synchronization Examples

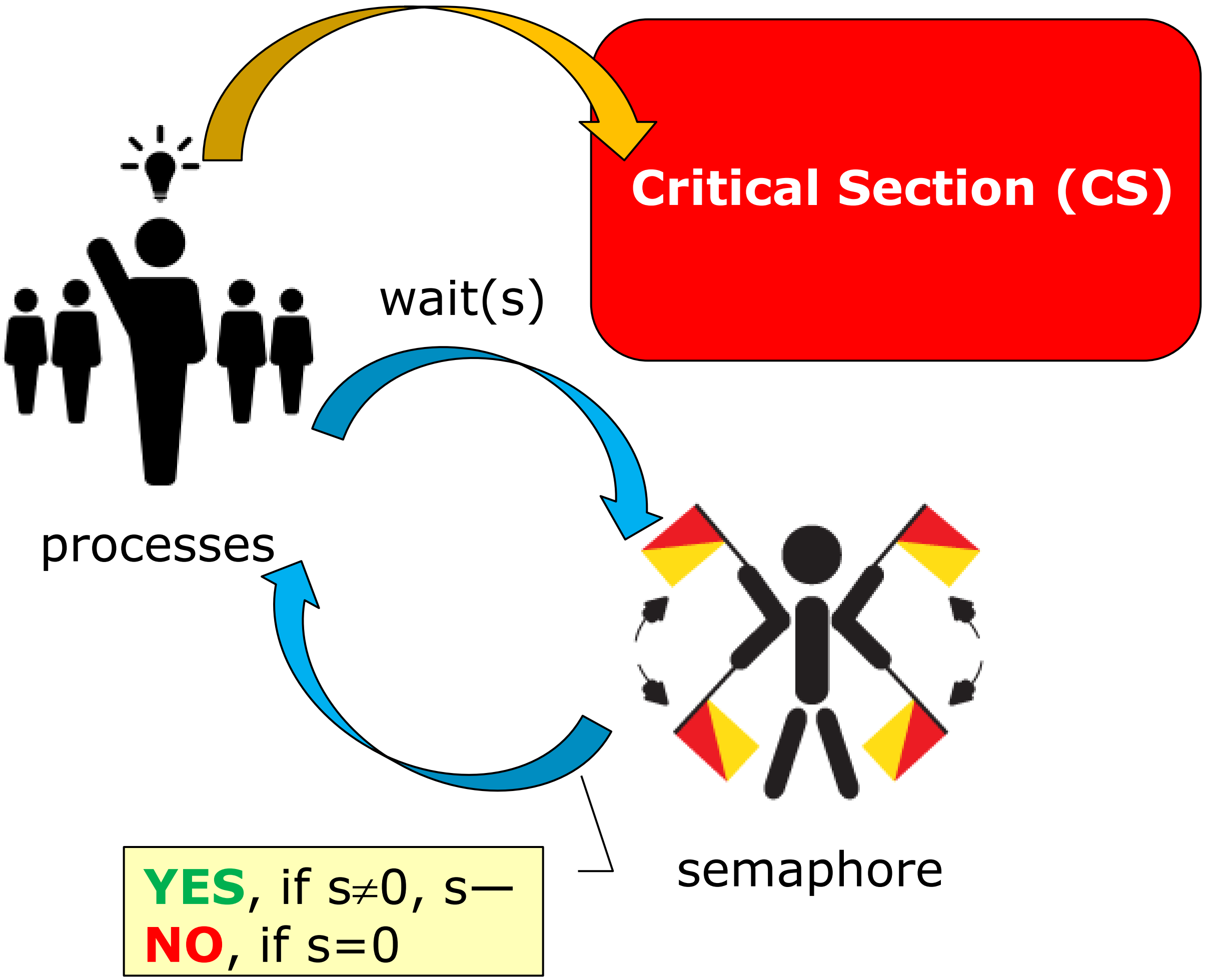


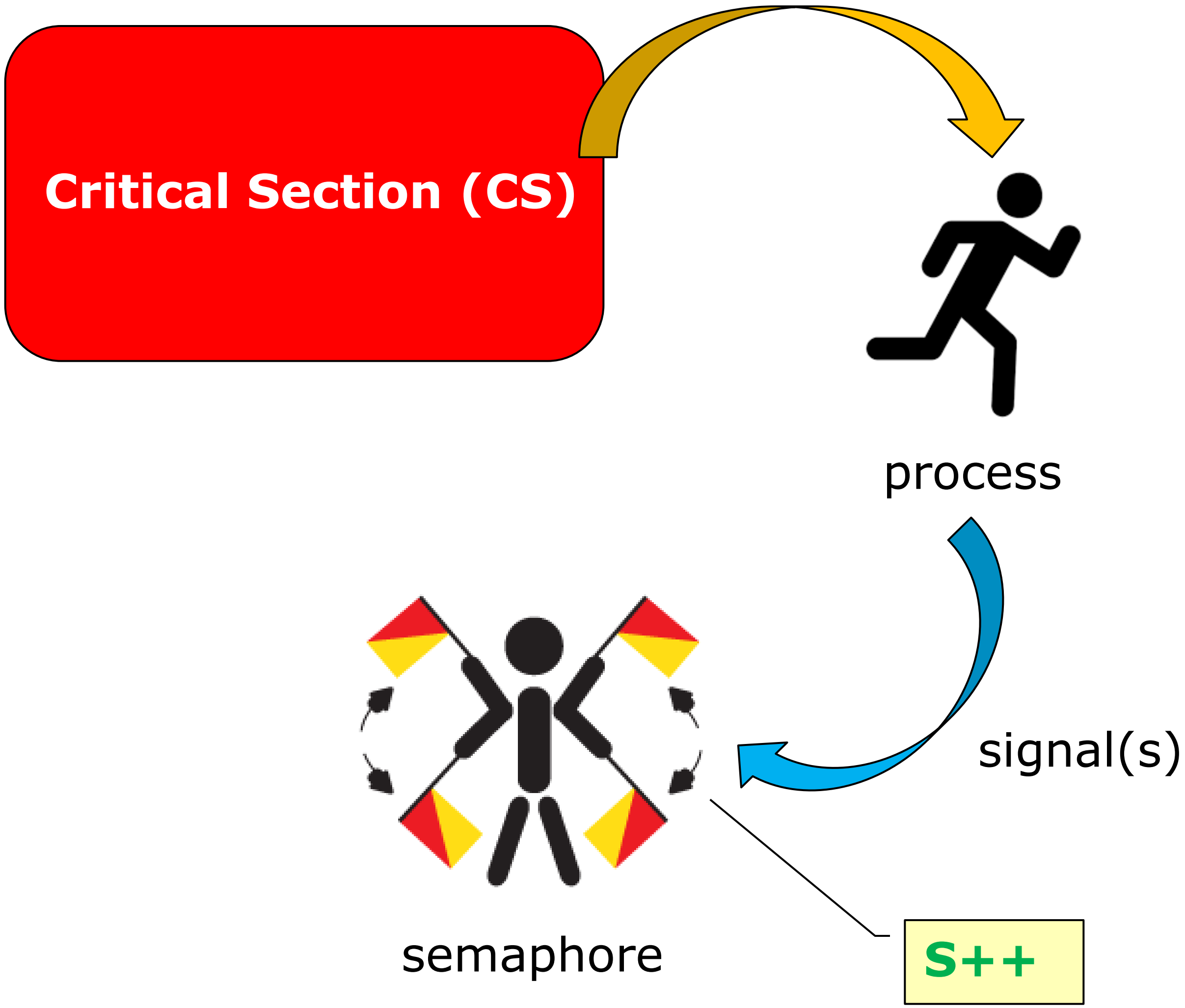
Chapter 7: Synchronization Examples

- Classic Problems of Synchronization
- Synchronization within the Kernel
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

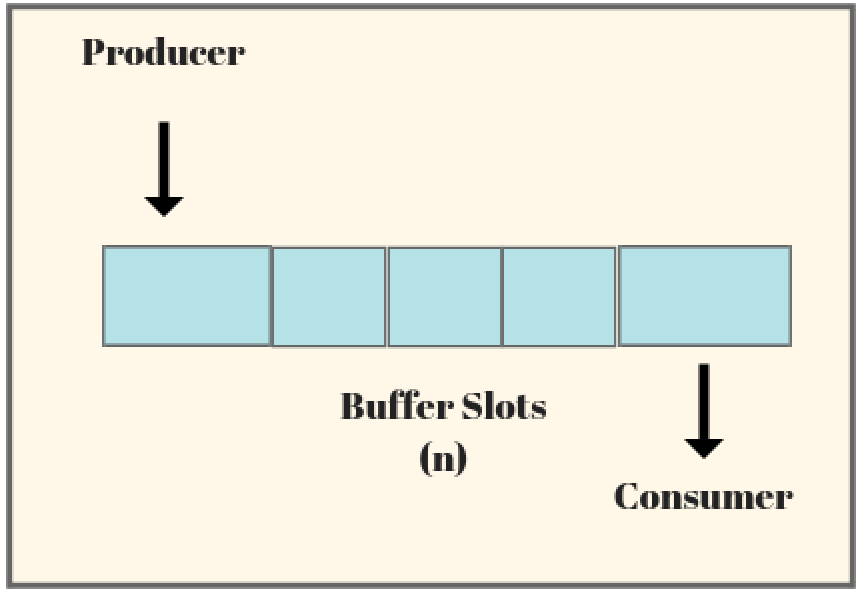
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem


Critical Section (CS)
wait(s)
processes
semaphore
YES, if s≠0, s—
NO, if s=0


Critical Section (CS)
process
signal(s)
semaphore
S++

Bounded-Buffer Problem



1. CS 最多只能只有一個 process進入，producer或consumer
2. 當 buffer是 empty 時，consumer不能進入CS取資料，必須被 block
   當 buffer是 full 時，producer不能進入CS放資料，必須被 block

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - 最多一個 process (producer or consumer) 可以進入存取 buffer
  - mutex = 0 → 必須 waiting，不能進入
- Semaphore **`full`** initialized to the value 0
  - full = 0 → 表示buffer全都是empty，consumer 被 block，不能從buffer取到東西
- Semaphore **`empty`** initialized to the value n
  - empty = 0 → 表示沒有空的buffer，producer 被 block不能放東西到buff

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
      ...
      /* produce an item in next_produced */
      ...
   wait(empty);
   wait(mutex);
      ...
      /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(full);
} while (true);
```

empty = 0,
表示沒有空的buffer
empty ≠ 0
表示有空的buffer, empty減1

mutex = 0,
表示有其他process使用buffer
mutex ≠ 0
表示可以存取buffer, mutex減1

離開CS, mutex加1, 讓其他
process可以進入CS

full加1
表示buffer有資料可以讓
consumer取得

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
     wait(full);
     wait(mutex);
         ...
     /* remove an item from buffe
         ...
     signal(mutex);
     signal(empty);
         ...
     /* consume the item in next consumed */
         ...
 } while (true);
```

full = 0,
表示buffer全都 empty
full $\neq$ 0
表示有buffer可以取得，full減1

mutex = 0,
表示有其他process使用buffer
mutex $\neq$ 0
表示可以存取buffer，mutex減1

離開CS，mutex加1，讓其他
process可以進入CS

empty加1
表示有 empty的 buffer
可以讓producer放入資料

Bounded Buffer Problem - Thinking

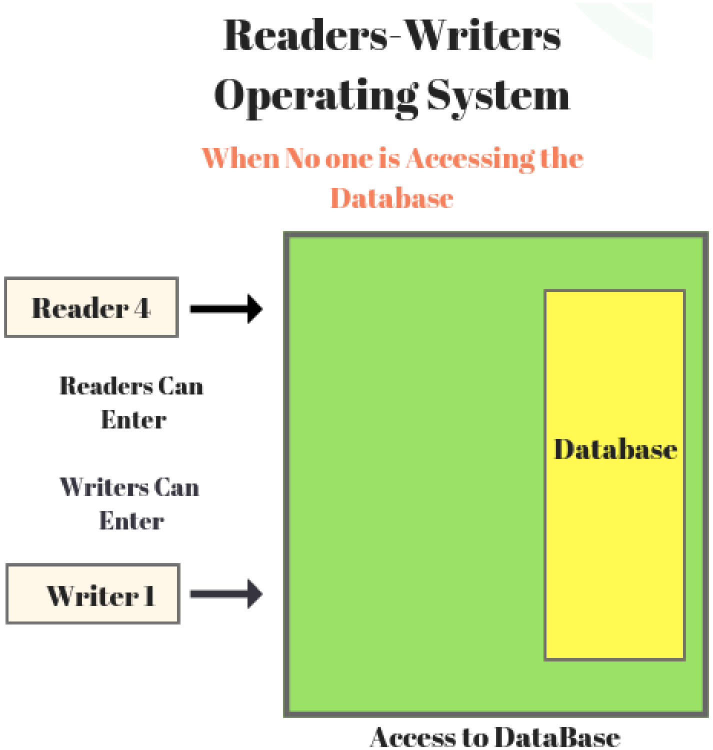
- The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(mutex);
    wait(empty);
                ...
        /* add next produced to the buffer */
         ...
    signal(full);
    signal(mutex);
 } while (true);
```

Readers-Writers Problem Variations

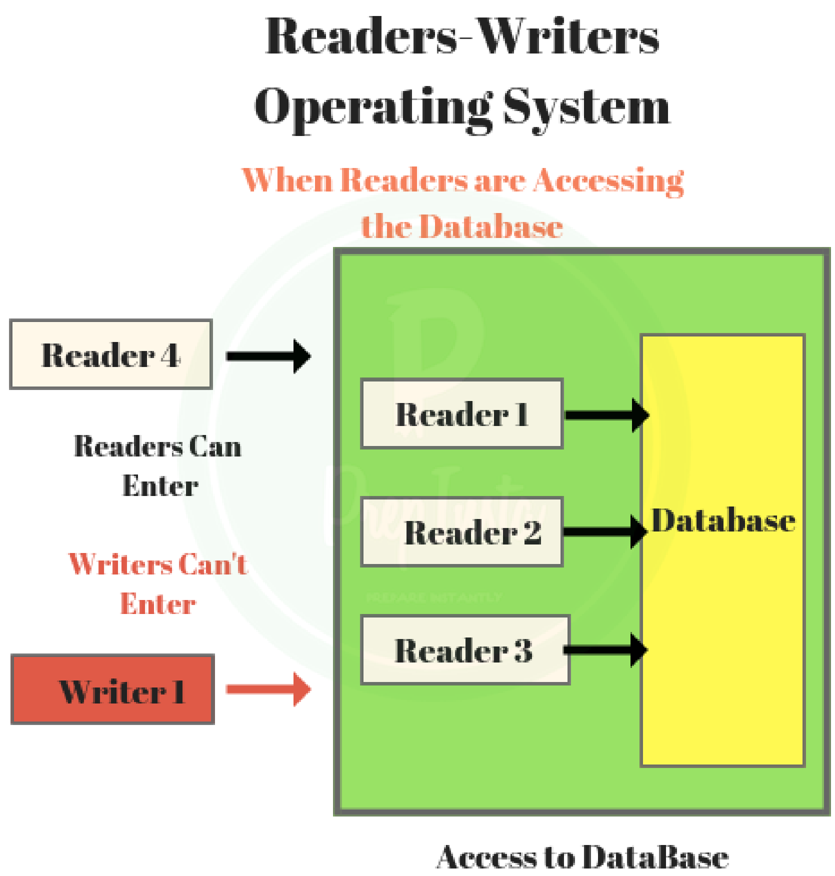
- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP (As Soon As Possible)
- Both may have **starvation** leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

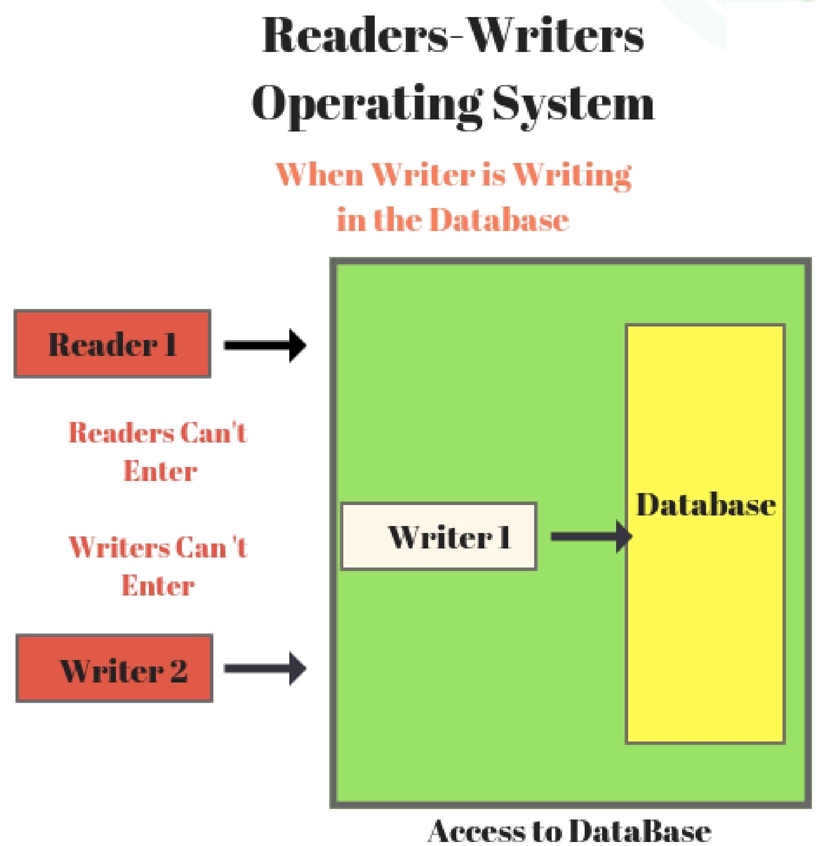
Readers-Writers Problem



- *a. first readers-writers problem*
  - 當目前沒有任何reader的時候，writer才可以進行寫入。
  - 目前沒有writer在進行的話，reader彼此間不需要等待。
  - 當writer獲得修改權限並開始進行寫入的時候，reader要等待寫入完成。
  - Quiz：甚麼情況會導致writer的starvation
- *b. second readers-writers problem*
  - 當一個writer在等待，想寫入共享資源的時候，不能有新的reader開始讀取共享資源。
  - Quiz：甚麼情況會導致reader的starvation

Readers-Writers Problem





Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – **allow multiple readers to read** at the same time
  - **Only one single writer can access** the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 (檢查 CS 內有沒有 reader or writer)
    - 如果 reader在CS，writer不能進入
    - 如果writer在CS，reader不能進入
    - 簡單思考，只要有人在CS，不管是 reader or writer，就要限制住 writer 不能進入
  - Semaphore **`mutex`** initialized to 1
    - mutex用來保護 read_count, read_count視為CS，一次只有一個reader可以存取
  - Integer **`read_count`** initialized to 0 (用來統計目前有多少 reader)

Readers-Writers Problem (Cont.)

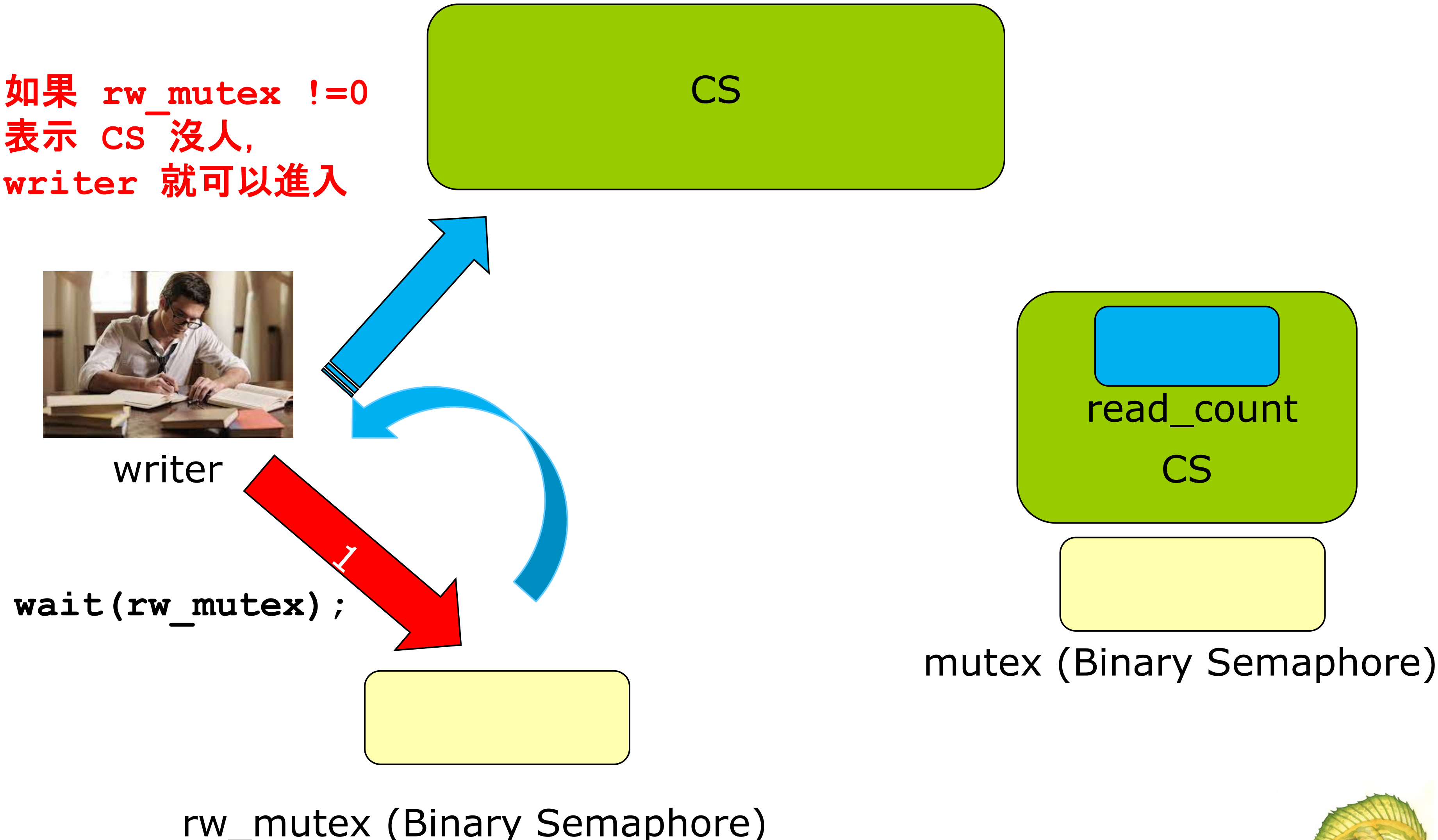
- The structure of a writer process

```
do {
      wait(rw_mutex);
            ...
         /* writing is performed */
            ...
      signal(rw_mutex);
} while (true);
```

rw_mutex = 0,
表示有reader，writer不能進入CS
rw_mutex ≠ 0
表示沒有reader，writer可以進入CS

rw_mutex加1
表示writer離開，
排隊等的readers或writers可以進入CS

Readers-Writers Problem (Cont.)



Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
            wait(rw_mutex);
        signal(mutex);
          ...
        /* reading is performed */
          ...
        wait(mutex);
        read count--;
        if (read_count == 0)
            signal(rw_mutex);
        signal(mutex);
} while (true);
```

對 read_count 進行CS管制

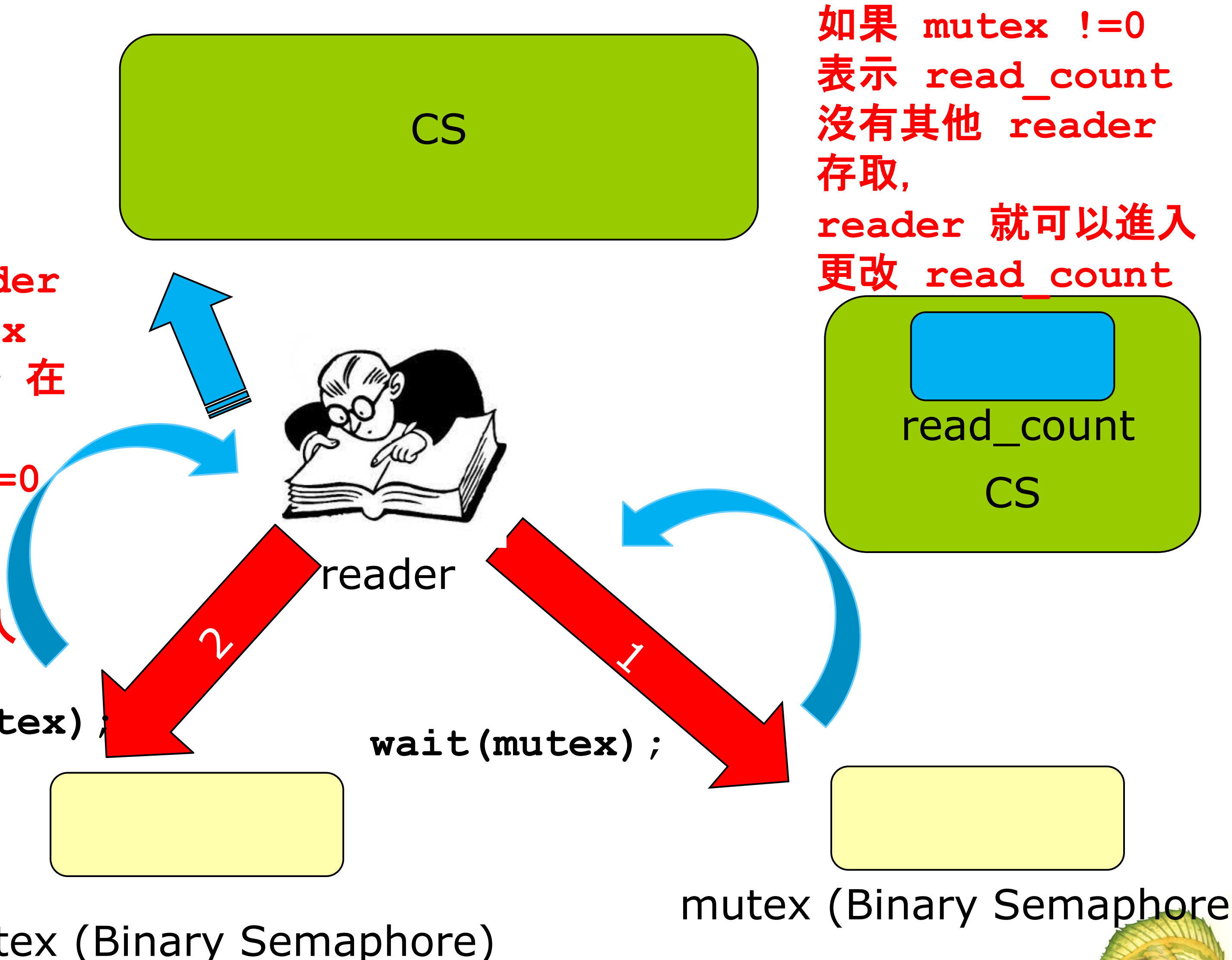
如果是第一個 reader
rw_mutex = 0,
表示有writer, reader不能進入CS
rw_mutex ≠ 0
表示沒有writer, reader可以進入CS,
rw_mutex會減1設為0, 阻擋writer進入

其他reader可以存取read_count

如果是最後一個reader離開CS
rw_mutex加1,
可以讓排隊的writer進入CS

Readers-Writers Problem (Cont.)

如果是第一個 reader
要去檢查 rw_mutex
看是否有 writer 在
裡面
如果 rw_mutex !=0
表示 CS 沒有
writer
reader 就可以進入



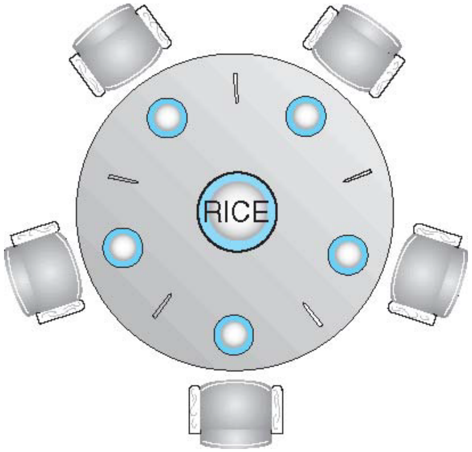
Readers-Writers Problem (Cont.)

為什麼只有第一個 reader 要去檢查 rw_mutex

```
wait(rw_mutex);
```

之後的 readers 都不用去檢查 rw_mutex 直接進入 CS ？

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - **Need both to eat, then release both when done**
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
      wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );

            //  eat

      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

            //  think

} while (TRUE);
```

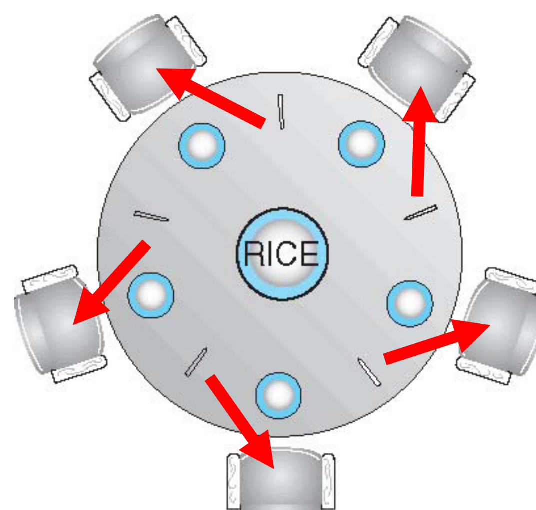
檢查左邊筷子是否可以拿

檢查右邊筷子是否可以拿

放下左邊筷子是否可以拿

放下右邊筷子是否可以拿

- What is the problem with this algorithm?



Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
   }

   void putdown (int i) {
          state[i] = THINKING;
                 // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
   }
```

設定哲學家 i 為飢餓狀態

檢查哲學家 i 左右鄰居是否處於吃飯狀態

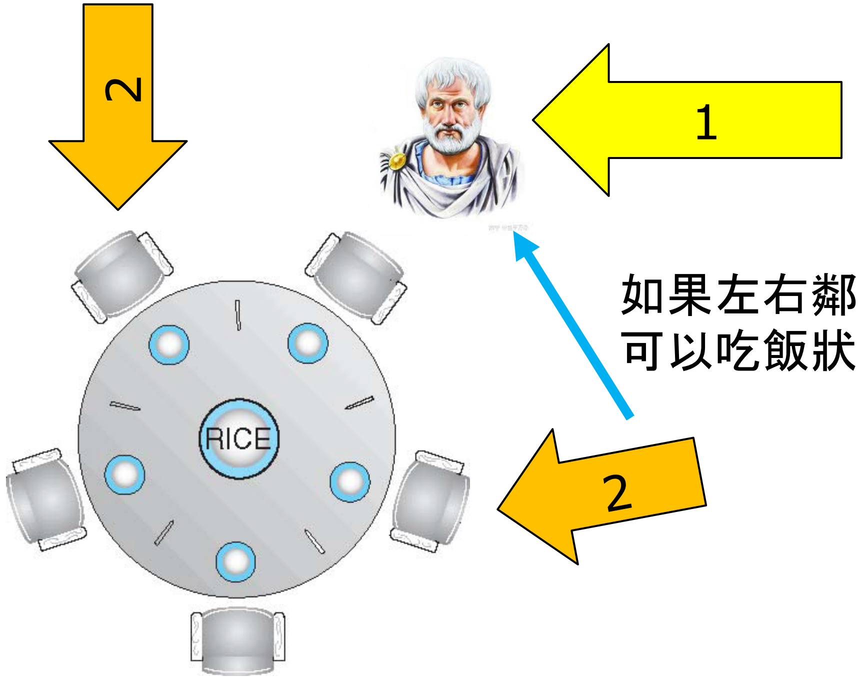
如果哲學家 i 不能吃飯，則等待

設定哲學家 i 為思考狀態

哲學家 i 的左邊鄰居L如果是飢餓狀態，且L的左右都不是處於吃飯狀態，喚醒L吃飯，並設定L的狀態為吃飯，

哲學家 i 的右邊鄰居R如果是飢餓狀態，且R的左右都不是處於吃飯狀態，喚醒R吃飯，並設定R的狀態為吃飯，

Dining-Philosophers Problem



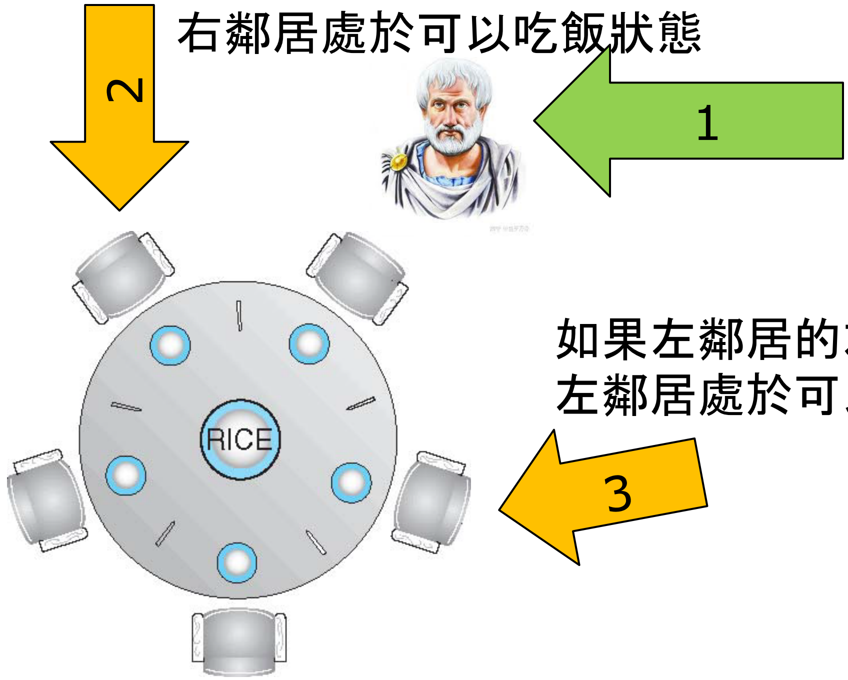
如果左右鄰居都沒有人吃飯，設定自己處於可以吃飯狀態

```
void pickup (int i) {
    state[i] = HUNGRY; //1 設定自己為飢餓
    test(i); //2 檢查左右鄰居是否有人在吃飯
             //  如果都沒有人吃飯，設定自己處於可以吃飯狀態
    if (state[i] != EATING) self[i].wait; //3.自己不是處於可以吃
                                          //  飯狀態，則等左右鄰居
                                          //  吃完

}
```

Dining-Philosophers Problem

如果右鄰居的左右鄰居都沒有人吃飯，設定
右鄰居處於可以吃飯狀態



如果左鄰居的左右鄰居都沒有人吃飯，設定
左鄰居處於可以吃飯狀態

```
void putdown (int i) {
   state[i] = THINKING; //1. 吃完飯，將自己狀態設為思考，放下筷子
   // test left and right neighbors
   test((i + 4) % 5); //2. 幫右鄰居檢查他可不可以吃飯
   test((i + 1) % 5); //3. 幫左鄰居檢查他可不可以吃飯
}
```

Solution to Dining Philosophers (Cont.)

如果哲學家 i 處於飢餓狀態
而且
左右鄰居沒有處於吃飯狀態
設定哲學家 i的狀態是吃飯

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                 state[i] = EATING ;
             self[i].signal () ;
            }
    }

    initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
    }
}
```

喚醒哲學家 i 吃飯

Solution to Dining Philosophers (Cont.)

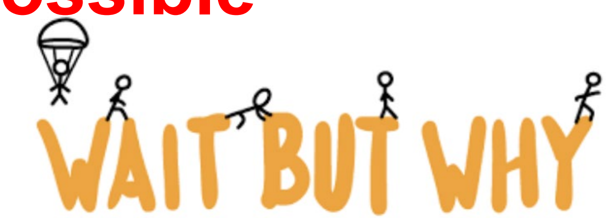
- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

     EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but **starvation is possible**



End of Chapter 7