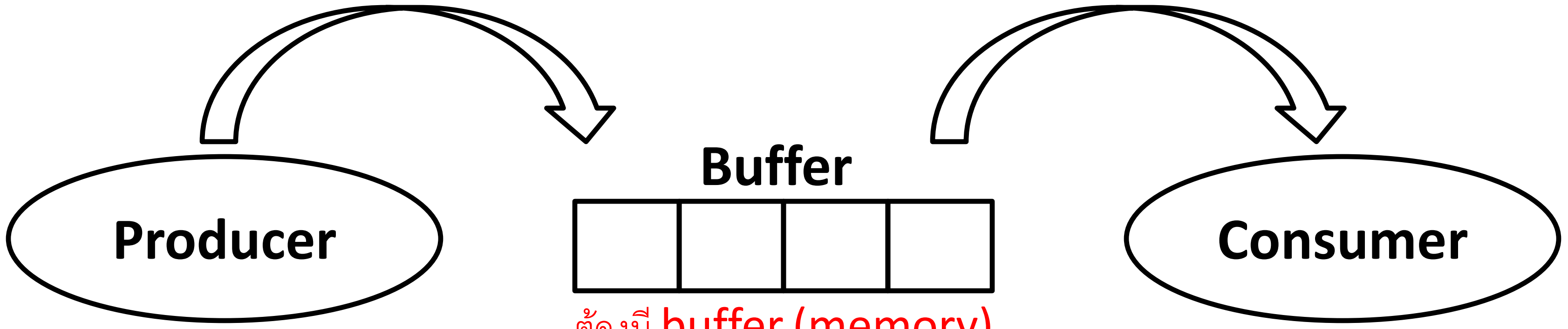
แก้ไขครั้งล่าสุดเมื่อวันที่ 21 กันยายน 2567

Process Synchronization



ต้องมี buffer (memory)

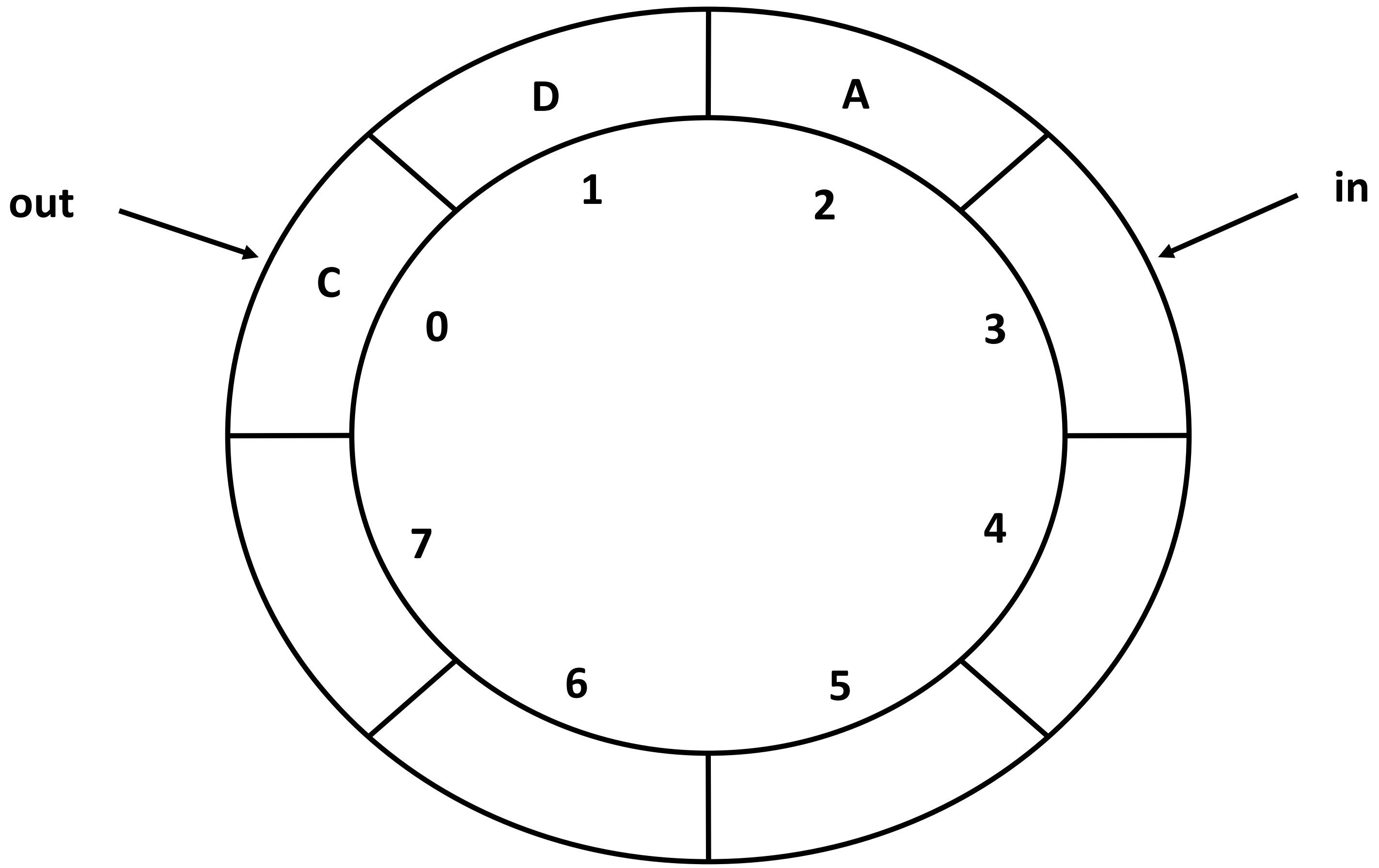
ถ้า อัตราการผลิต/อัตราการบริโภค ไม่เท่ากัน

Example	Producer	Consumer	Linux Command
1	tar	gzip	tar cvf - . | gzip > target.tar.gz
2	cat	grep	cat dictionary | grep board

ไม่ต้องเขียนลงฮาร์ดดิสก์ เขียนลง memory แทน

และไม่เปลือง memory มาก

Circular Queue



count = 3
BUFFER_SIZE = 8

Producer & Consumer problem

Producer:

```
while (ยังต้อง produce อยู่) {
        while (count == BUFFER_SIZE) { }
        buffer[in] = nextProduced
        in = (in + 1) % BUFFER_SIZE
        count++
}
```

Consumer:

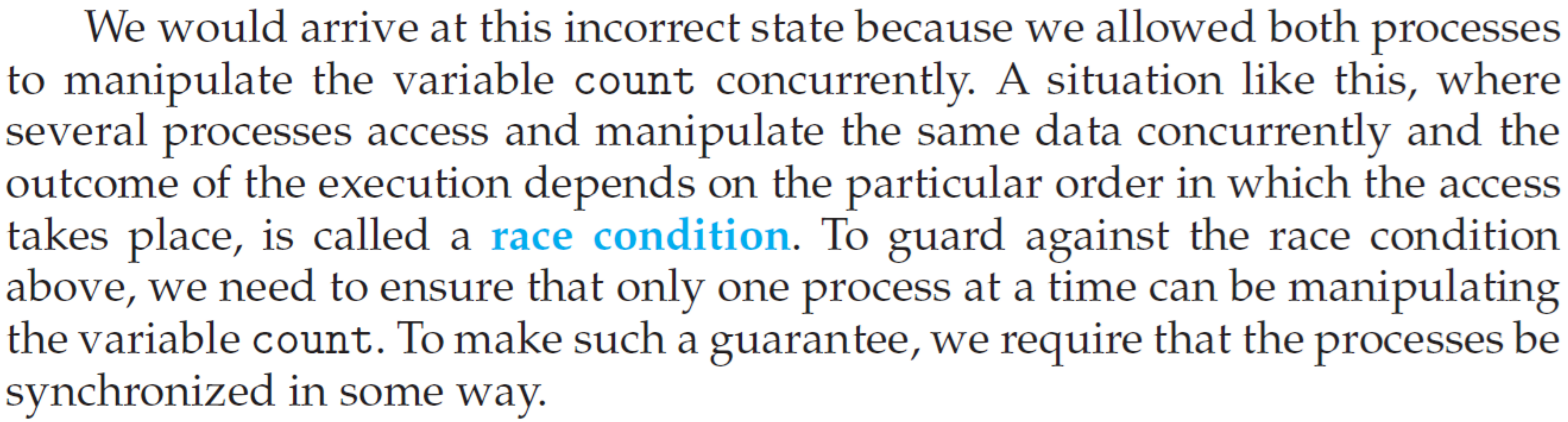
```
while (ยังต้อง consume อยู่) {
        while (count == 0)
        nextConsumed = buffer[out]
        out = (out + 1) % BUFFER_SIZE
        count --
}
```

Race Condition

count++	**instr 1: register1 = count**
	instr 2: register1 = register1 + 1
	instr 3: count = register1
count--	**instr 1: register2 = count**
	instr 2: register2 = register2 – 1
	instr 3: count = register2

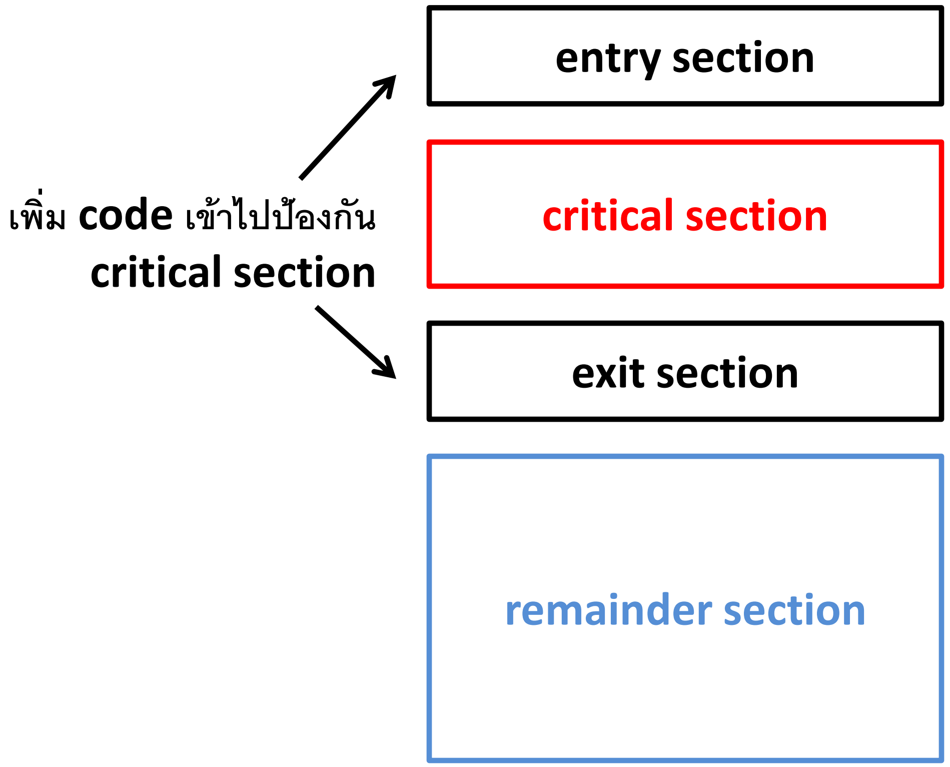
Let count = 5

Executing (single core) 1 2 1 2 3 3 results in "count = 4"



We would arrive at this incorrect state because we allowed both processes to manipulate the variable count concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable count. To make such a guarantee, we require that the processes be synchronized in some way.

Critical Section

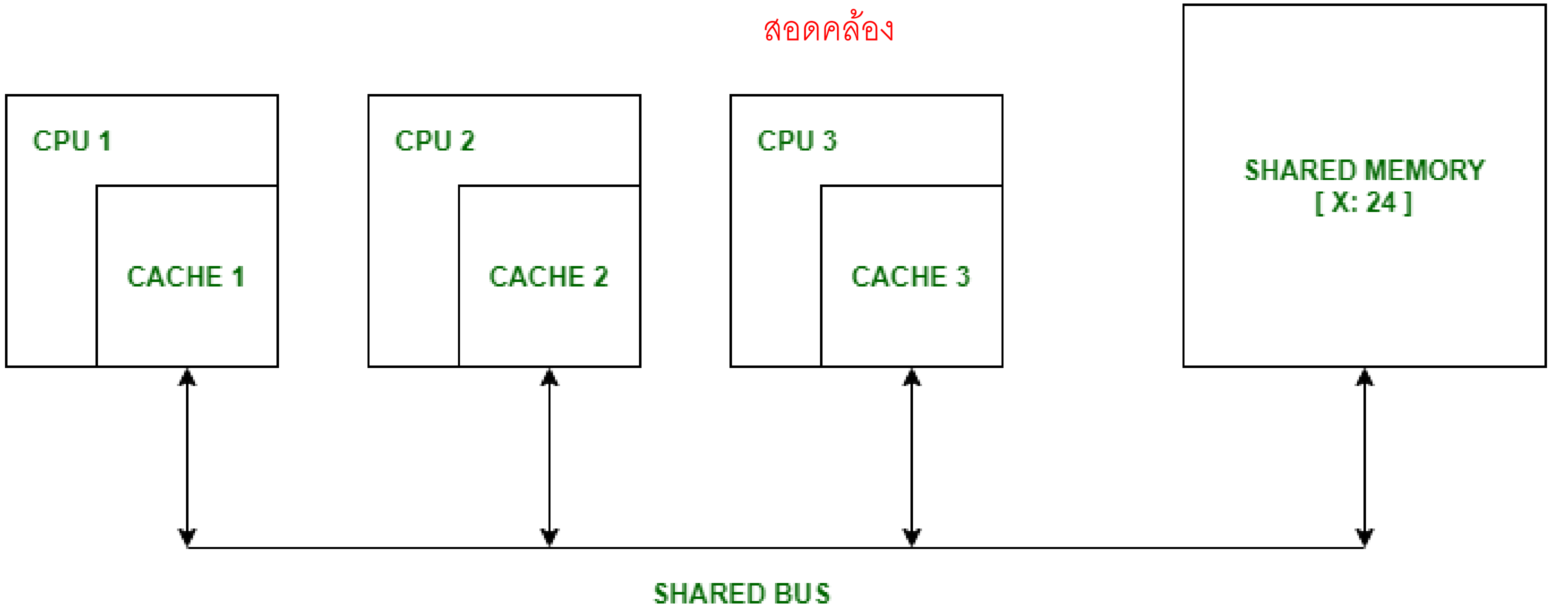


A solution to the critical-section problem must satisfy:

1. **Mutual exclusion**
   process/thread ได้เข้าใช้ critical section ทีละ 1 เท่านั้น
2. **Progress**
   เลือก process/thread เข้ามาใน critical section ได้เสมอ
   ไม่ติดตาย deadlock
3. **Bounded waiting**
   รับประกันว่ารอนานไม่เกินเวลาที่กำหนด
   จะต้องได้เข้า critical section

Cache Coherence

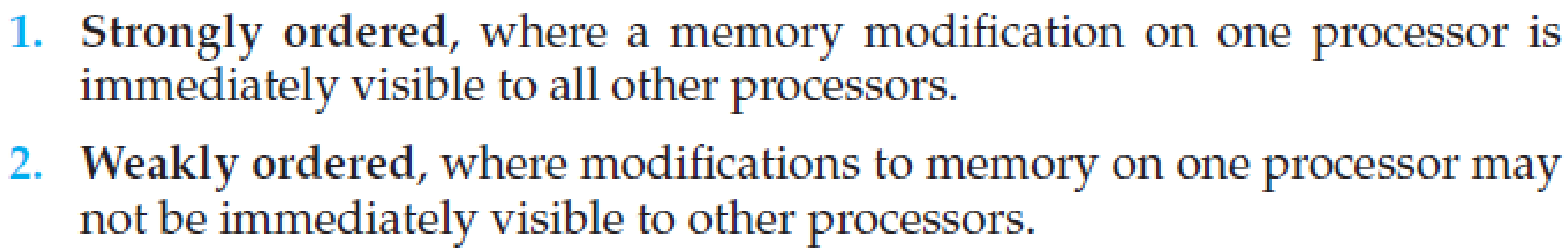
สอดคล้อง



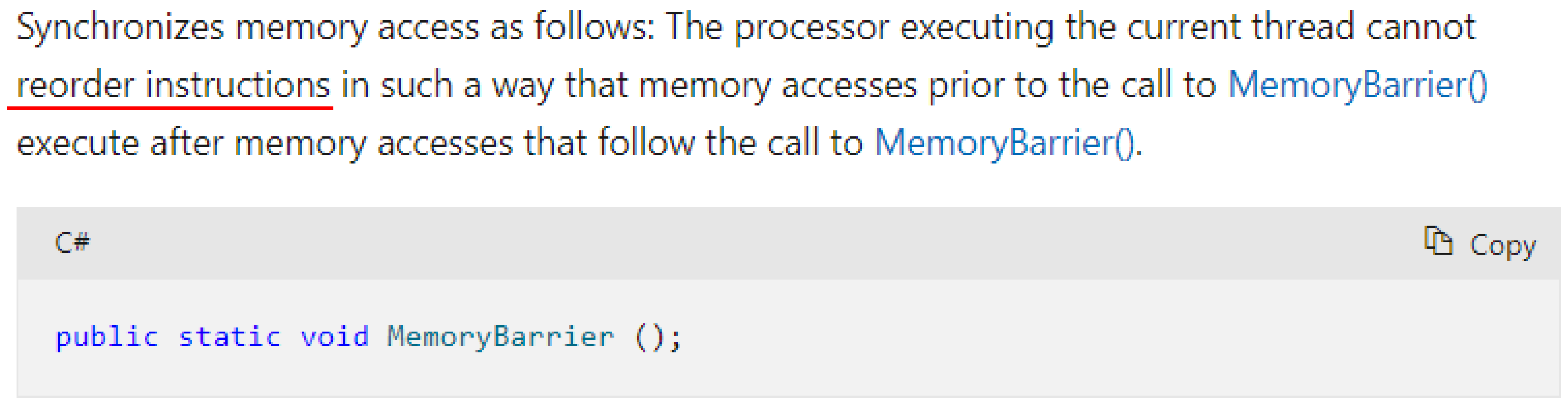
- **Processor 1 read X :** obtains 24 from the memory and caches it.
- **Processor 2 read X :** obtains 24 from memory and caches it.
- **Again, processor 1 writes as X :** 64, Its locally cached copy is updated. Now, processor 3 reads X, what value should it get?
- Memory and processor 2 thinks it is 24 and processor 1 thinks it is 64.

https://www.geeksforgeeks.org/cache-coherence/

Memory Barriers



1. **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.
2. **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.

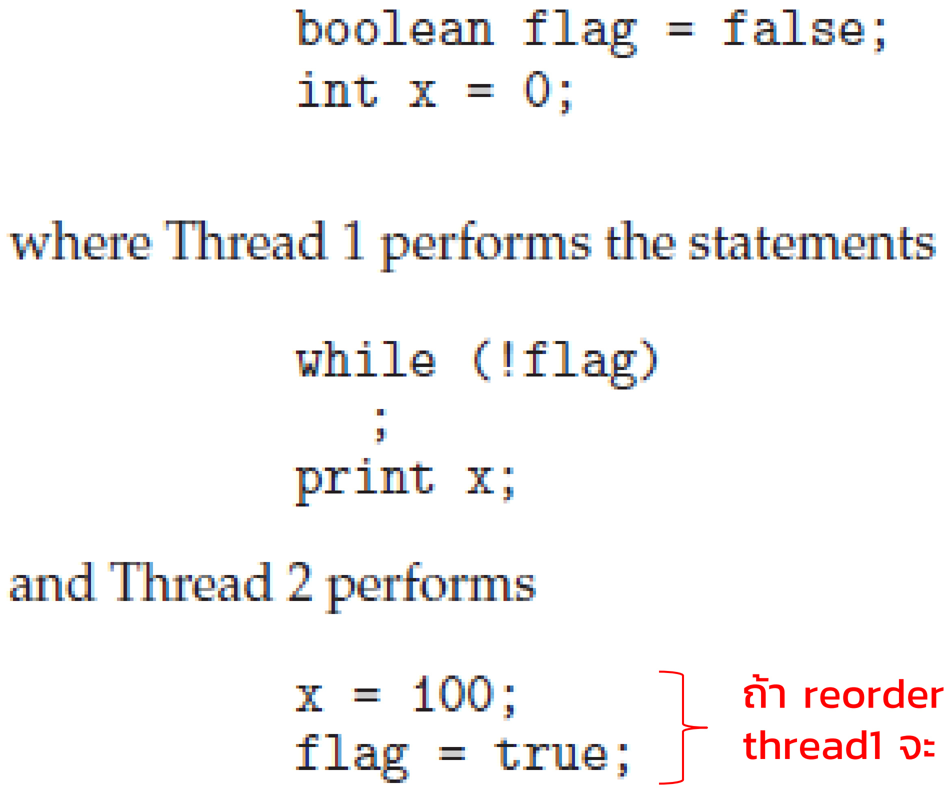


Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier() execute after memory accesses that follow the call to MemoryBarrier().

```
public static void MemoryBarrier ();
```

For most purposes, the C# lock statement, the Visual Basic SyncLock statement, or the Monitor class provide easier ways to synchronize data.

กำหนดค่าเริ่มต้นให้ตัวแปร



```
boolean flag = false;
int x = 0;
```

where Thread 1 performs the statements

```
while (!flag)
   ;
print x;
```

and Thread 2 performs

```
x = 100;
flag = true;
```

ถ้า reorder 2 คำสั่งนี้
thread1 จะ print 0 แทนที่จะ print 100

จะต้องใส่ MemoryBarrier() เข้าไปตรงไหน?
เพื่อให้ Thread 1 พิมพ์ (print) 100 ออกมาเสมอ

Peterson's Solution

```
int     turn;      // turn = 0, 1 process that is allowed to execute in its CS.
bool    flag[2];   // flag[i] = true, process i is ready to enter its CS.
```

Process 0

```
do {   ใครเขียนตัวแปร turn ก่อน จะได้ใช้ cs ก่อน
    flag[0] = TRUE;   P0 พร้อมเข้า CS
    turn = 1;   เป็นตาของ P1
    while (flag[1] && turn == 1);
        critical section
    flag[0] = FALSE
        remainder section
} while (TRUE);
```

Process 1

```
do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn == 0);
        critical section
    flag[1] = FALSE
        remainder section
} while (TRUE);
```

Mutual exclusion, progress, bounded waiting are satisfied. แต่ทำได้แค่ 2 process เท่านั้น

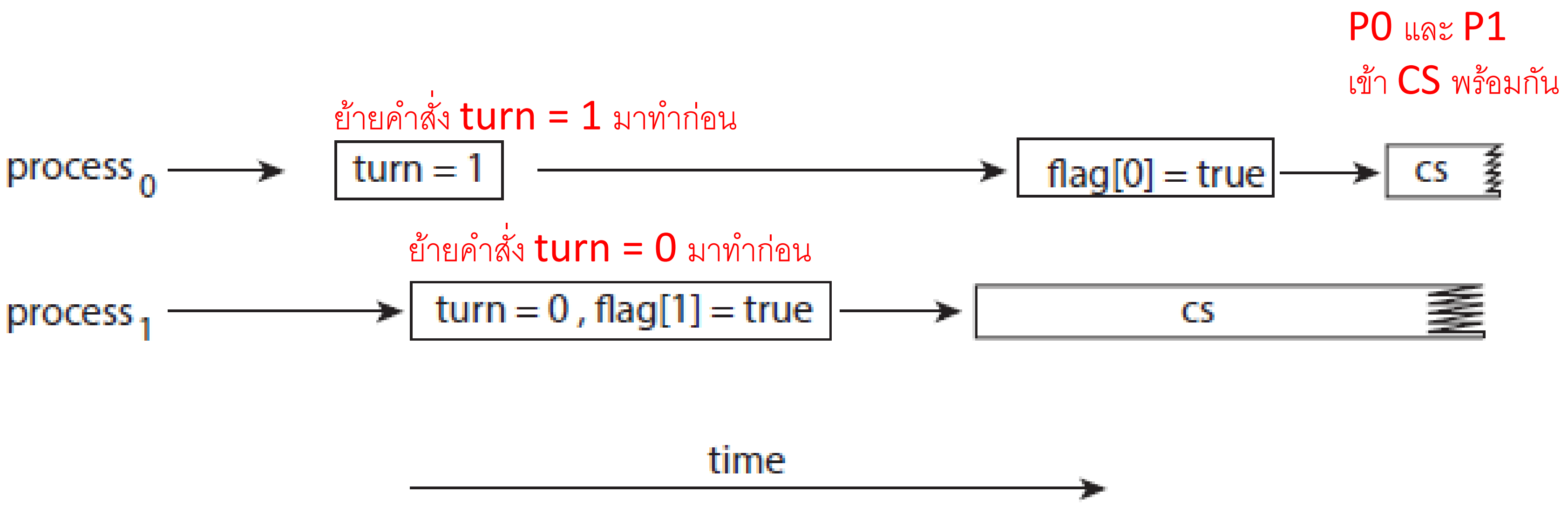

P0 และ P1
เข้า CS พร้อมกัน
ย้ายคำสั่ง turn = 1 มาทำก่อน
process 0
turn = 1
flag[0] = true
cs
ย้ายคำสั่ง turn = 0 มาทำก่อน
process 1
turn = 0 , flag[1] = true
cs
time

Figure 6.4 The effects of instruction reordering in Peterson's solution.

Sync. HW: TestAndSet

```
bool TestAndSet(bool *target) {     // atomic instruction
    bool rv = *target;              // rv น่าจะหมายถึง return value
    *target = TRUE;
    return rv;
}
```

(object)
pass by reference

Shared variable

เริ่มต้น lock = FALSE;

```
do {
    while (TestAndSet(&lock));

        // critical section

    lock = FALSE;

        // remainder section

} while (TRUE);
```

```
do {
    while (TestAndSet(&lock));

        // critical section

    lock = FALSE;

        // remainder section

} while (TRUE);
```

Not satisfy bounded-waiting requirement!

อยู่ใน textbook เล่มเก่า

Bounded-waiting mutual exclusion

ต้องจัดคิวให้ process ที่รออยู่

เริ่มต้น waiting[i] = false และ lock = false

```
do {
        waiting[i] = TRUE;   process i รอใช้ CS
        key = TRUE;  มี process ใช้ CS อยู่
        while (waiting[i] && key) key = TestAndSet(&lock);
        waiting[i] = FALSE;  ได้เข้าใช้ CS แล้ว

                // critical section

        j = (i + 1) % n;  ค้นหา process ที่รอใช้ CS (หาไปทางขวา)
        while ((j != i) && !waiting[j]) j = (j + 1) % n;

        if (j == i)   ไม่มี process อื่นที่รอใช้ CS
                lock = FALSE;  unlock ปล่อย CS
        else  มี process j รอใช้ CS
                waiting[j] = FALSE;  ส่งมอบ CS ให้ process ตัวแรกที่อยู่ถัดไปทางขวา

                // remainder section
} while (TRUE);
```

อยู่ใน textbook เล่มเก่า

Sync. HW: Swap

```
void Swap(bool *a, bool *b) {          // atomic instruction
        bool tmp = *a;
        *a = *b;
        *b = tmp;
}
          lock = FALSE;
```

Shared variable

```
do {
    key = TRUE;
    while (key == TRUE) Swap(&lock, &key);

    // critical section

    lock = FALSE;

    // remainder section

} while (TRUE);
```

```
do {
    key = TRUE;
    while (key == TRUE) Swap(&lock, &key);

    // critical section

    lock = FALSE;

    // remainder section

} while (TRUE);
```

Not satisfy bounded-waiting requirement

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 6.7 The definition of the atomic compare_and_swap() instruction.

เริ่มต้น lock = 0

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

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Figure 6.8 Mutual exclusion with the compare_and_swap() instruction.

```
boolean waiting[n];
int lock;
```

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = 0;
   else
      waiting[j] = false;

      /* remainder section */
}
```

ใช้ TestAndSet() ก็ได้

Intel ใช้ compare_and_swap

Figure 6.9 Bounded-waiting mutual exclusion with compare_and_swap().

MAKING COMPARE-AND-SWAP ATOMIC

On Intel x86 architectures, the assembly language statement `cmpxchg` is used to implement the `compare_and_swap()` instruction. To enforce atomic execution, the `lock` prefix is used to lock the bus while the destination operand is being updated. The general form of this instruction appears as:

```
lock cmpxchg <destination operand>, <source operand>
```

```
increment(&sequence);
```

where the `increment()` function is implemented using the CAS instruction:

```
void increment(atomic_int *v)
{
   int temp;

   do {
      temp = *v;
   }
   while (temp != compare_and_swap(v, temp, temp+1));
}
```

library มักจะมีฟังก์ชัน
- acquire()
- release()

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

<u>ภาษา **C#** ใช้</u>

```
lock (object) {
    // critical section
}
```

Figure 6.10 Solution to the critical-section problem using mutex locks.

The definition of `acquire()` is as follows:

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
```

The definition of `release()` is as follows:

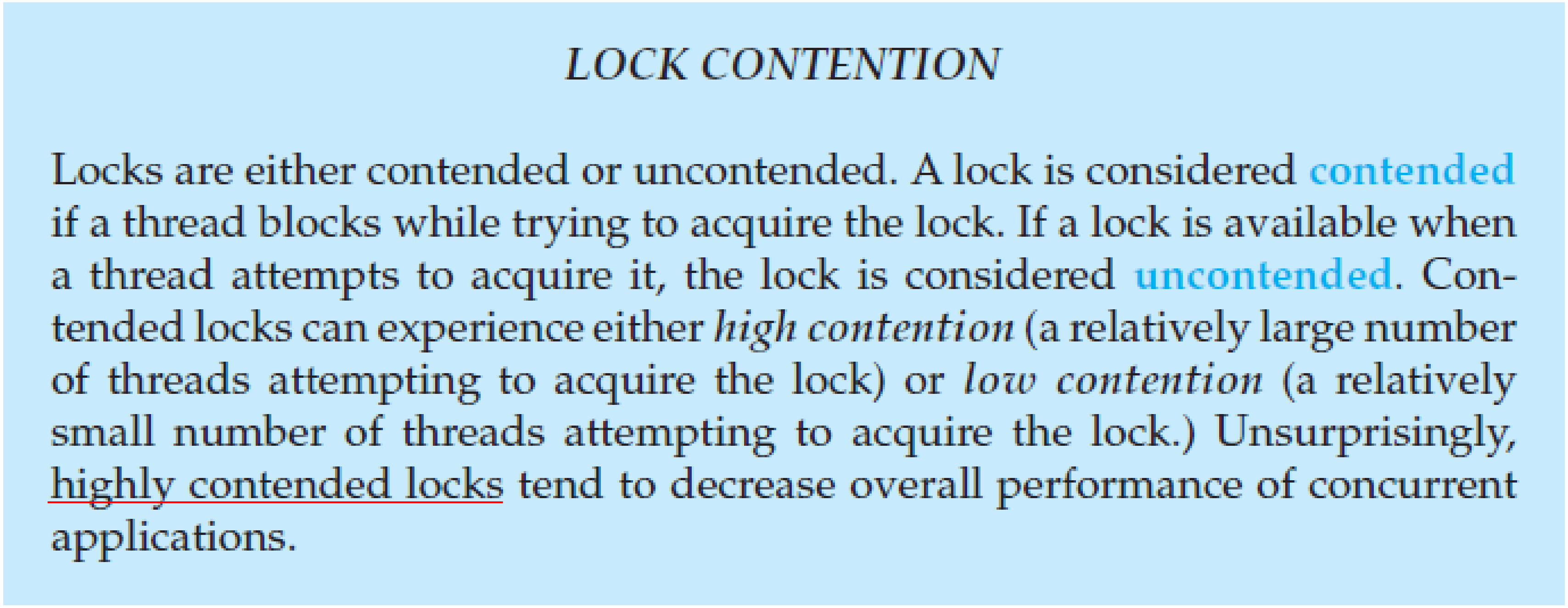
```
release() {
   available = true;
}
```

Calls to either `acquire()` or `release()` must be performed atomically.

On multiprocessor systems, ensuring atomicity exists is a little harder. It is still possible to use a lock (e.g. a spinlock) the same as on single processor systems, but merely using a single instruction or disabling interrupts will not guarantee atomic access. **You must also ensure that no other processor or core in the system attempts to access the data you are working with. The easiest way to achieve this is to ensure that the instructions you are using assert the 'LOCK' signal on the bus, which prevents any other processor in the system from accessing the memory at the same time.** On x86 processors, some instructions automatically lock the bus (e.g. 'XCHG') while others require you to specify a 'LOCK' prefix to the instruction to achieve this (e.g. 'CMPXCHG', which you should write as 'LOCK CMPXCHG op1, op2').

OS สมัยก่อนเป็นแบบ non-preemptive เพราะ CPU ยังไม่มี atomic instruction ดังนั้นต้องทำ critical section ให้เสร็จก่อนปล่อย CPU เช่น iPad, iOS < 4 (อาจจะเพราะพัฒนา kernel ไม่ทันด้วย)

https://stackoverflow.com/questions/49346612/atomic-operation-definition-and-multiprocessor


LOCK CONTENTION

Locks are either contended or uncontended. A lock is considered contended if a thread blocks while trying to acquire the lock. If a lock is available when a thread attempts to acquire it, the lock is considered uncontended. Contended locks can experience either *high contention* (a relatively large number of threads attempting to acquire the lock) or *low contention* (a relatively small number of threads attempting to acquire the lock.) Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications.

ตัวอย่างการลด Lock Contention เช่น

- แยก lock ของแต่ละบัญชีธนาคาร (account) ไม่ใช้ global lock อันเดียว
- แยก lock ของ reader(s) และ writer ในปัญหา readers-writers problem

Semaphores

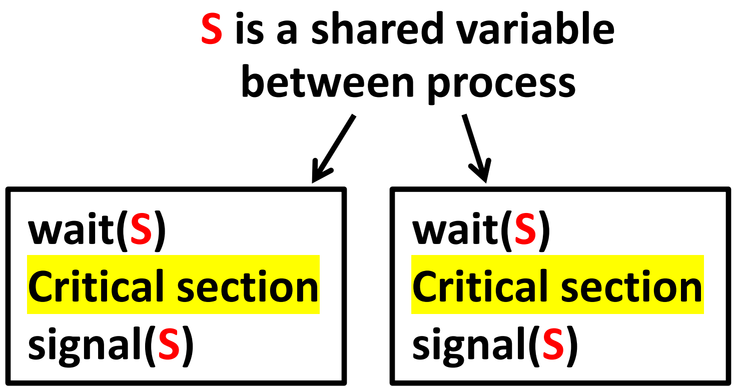
ใน critical section มี resource อยู่ S ชิ้น เป็น mutual exclusive

```
wait(S) {
        while (S <= 0); // ไม่มีทรัพยากรเหลือ ต้องรอ
        S--; // ยืมทรัพยากรไปใช้ 1 ชิ้น
}
```

```
signal(S) {
        S++; // คืนทรัพยากร 1 ชิ้น
}
```

S คือจำนวนทรัพยากรที่มี

ถ้า 1 process ใช้ทรัพยากร 1 ชิ้น ก็จะเข้าใช้ CS ได้พร้อมกันไม่เกิน S process(es)



Semaphore เป็น system call มีวิธีป้องกัน concurrent threads เช่น การทำ S++ ด้วยฟังก์ชัน increment() ที่ใช้คำสั่ง compare-and-swap การทำ S-- ก็คล้าย ๆ กัน

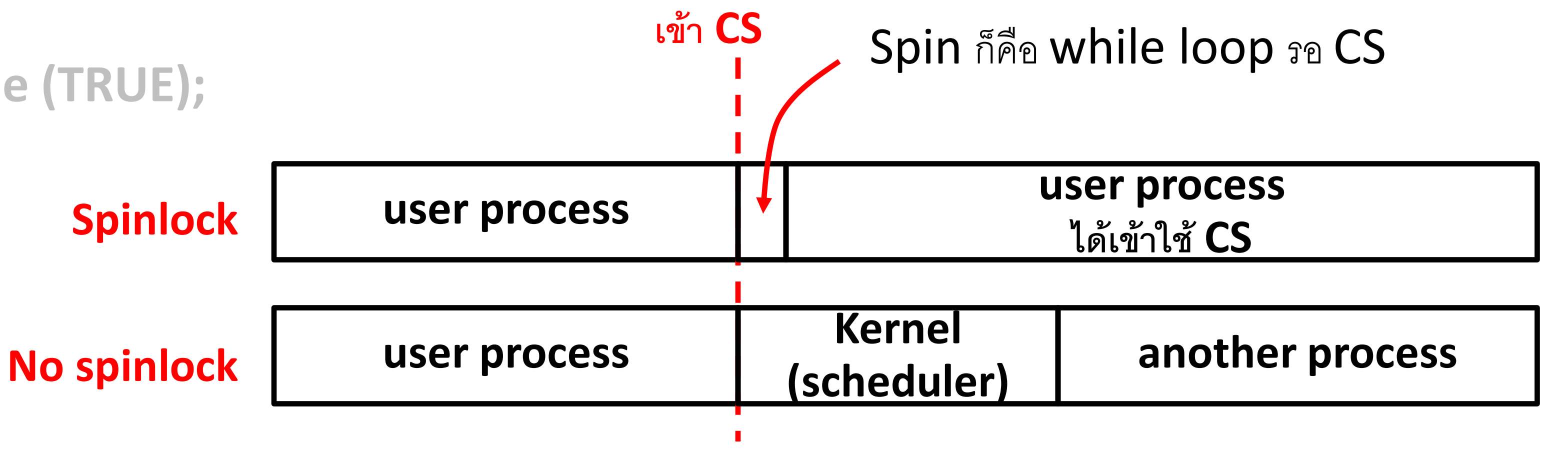
Semaphores: spinlock

```
do {          // n = #resources

    wait(n);

        // critical section

    signal(n);

        // remainder section

} while (TRUE);
```

หยุดรอจนกว่าจะเข้า CS ได้ หรือ หมด time slice / time quantum

การหยุดรอ CS ทำได้ 2 แบบคือ

1. Spinlock ใช้ time slice ต่อไป
2. No spinlock เรียก context-switch



Semaphore (spinlock / no spinlock) ก็ยังไม่รับประกัน **bounded waiting !!!**

Semaphores: no-spinlock

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

$- \longleftarrow$ value $\longrightarrow +$

ค่าลบคือ **#process** ที่รอเข้าใช้ **CS**

ค่าบวกคือ **#process** ที่ยังสามารถเข้า **CS** ได้

ใช้ <= 0 เพราะ value++ ไปแล้ว
เช่น -1, -2, -3 จะกลายเป็น 0, -1, -2

```
wait(semaphore *S) {
      S -> value--;
      if (S -> value < 0) {
            append this process to S -> list;
            sleep(); // context switch
      }
}
```

No spinlock

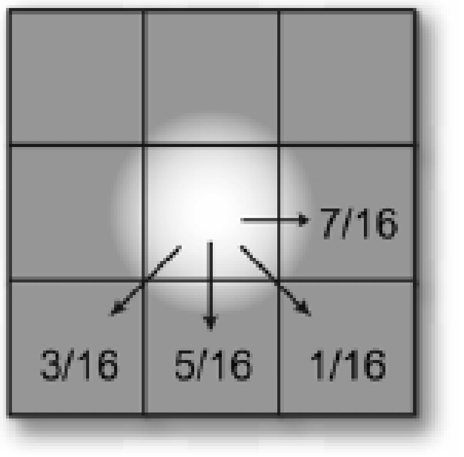
```
signal(semaphore *S) {
      S -> value++;
      if (S -> value <= 0) {
            remove a process P from S -> list;
            wakeup(P);
      }
}
```

ใช้หลัก FIFO

Ensure bounded waiting by FIFO queue

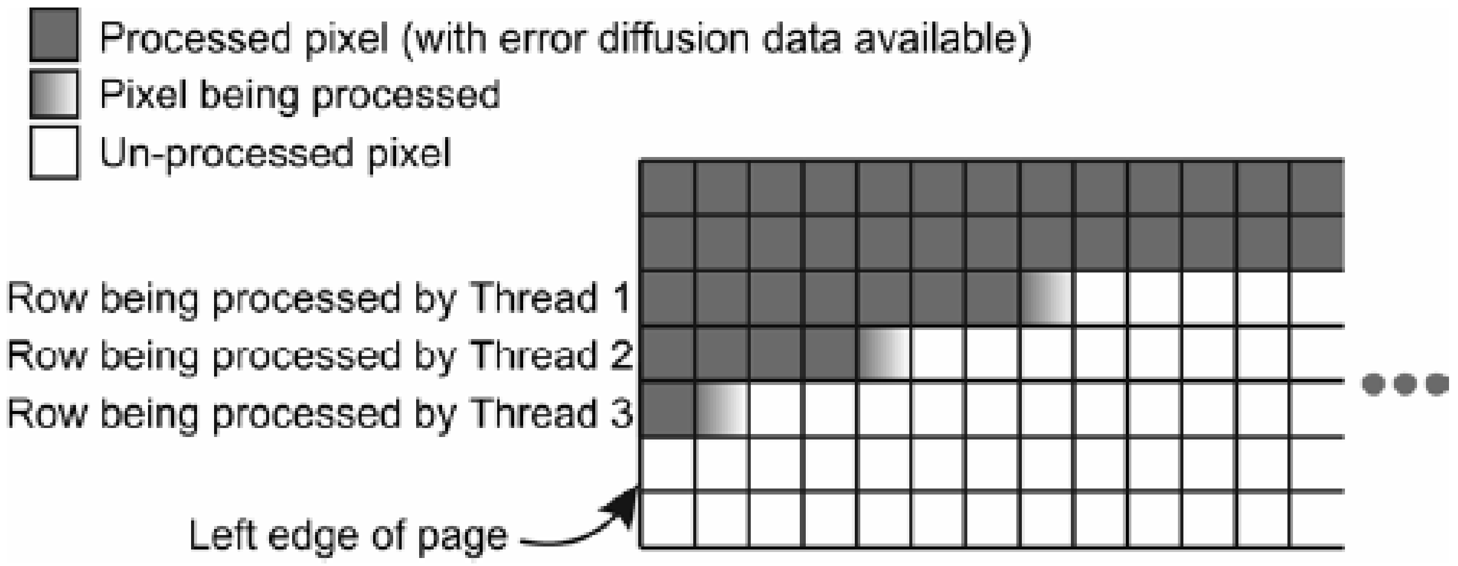
Spinlock: A case study

Error diffusion on multi-core processors



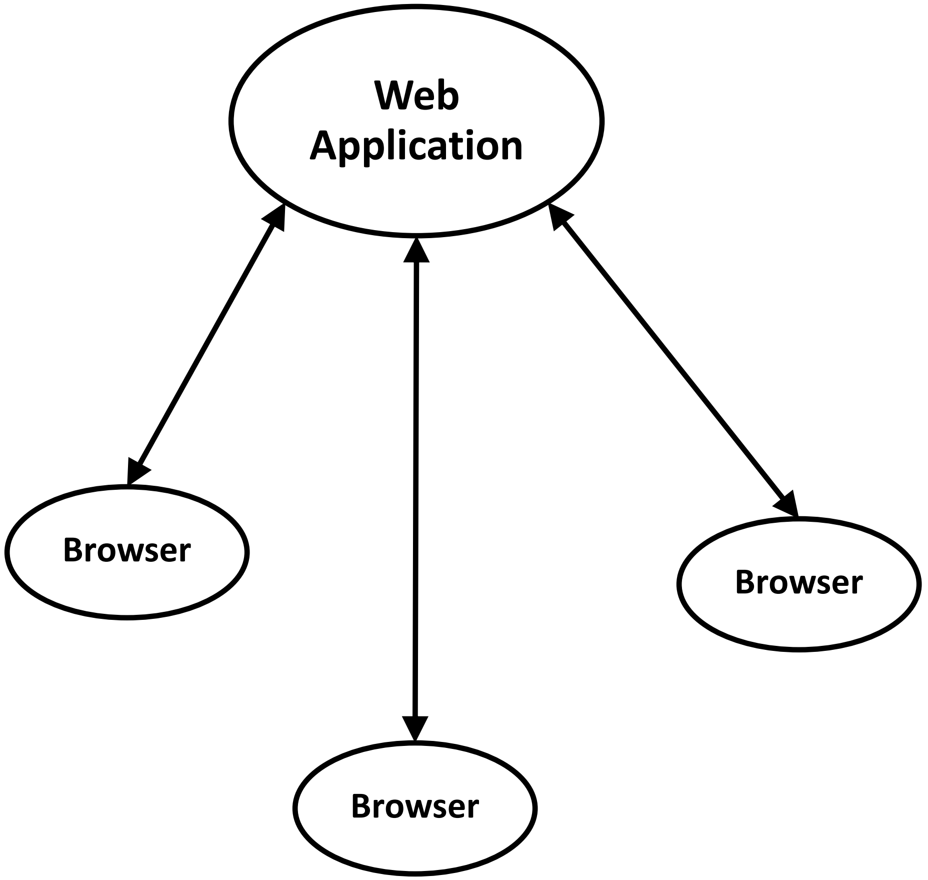
ทำ **spinlock** เพื่อไม่ให้เกิด **context switch**

ใช้ **loop** รอแถวบนทำเสร็จ อย่าเรียก **semaphore**


Processed pixel (with error diffusion data available)
Pixel being processed
Un-processed pixel
Row being processed by Thread 1
Row being processed by Thread 2
Row being processed by Thread 3
Left edge of page

Multiple threads are able to process multiple rows simultaneously.

Race Condition in Web Application



<u>กรณีศึกษา</u> เว็บรับสมัครสอบ

เริ่มต้น n = 0

เมื่อมีผู้สมัครสอบได้สำเร็จ n = n + 1

จะสร้าง id ให้ผู้สมัครสอบ เช่น year + n

ถ้าไม่ป้องกัน CS จะมีคนได้ id สอบซ้ำกัน

และถ้าได้ผู้สมัครเต็มจำนวนที่นั่งสอบแล้ว

จะหยุดรับสมัคร เช่น n ต้องไม่เกิน 1,000

ถ้าไม่ป้องกัน CS จะรับสมัครเกินจำนวนที่นั่ง

<u>**History's Worst Software Bugs**</u>

July 28, 1962 -- Mariner I space probe
1982 -- Soviet gas pipeline
1985-1987 -- Therac-25 medical accelerator (race condition)
At least five patients die; others are seriously injured.
1988 -- Buffer overflow in Berkeley Unix finger daemon

1985-1987 -- Therac-25 medical accelerator. A radiation therapy device malfunctions and delivers lethal radiation doses at several medical facilities. Based upon a previous design, the Therac-25 was an "improved" therapy system that could deliver two different kinds of radiation: either a low-power electron beam (beta particles) or X-rays. The Therac-25's X-rays were generated by smashing high-power electrons into a metal target positioned between the electron gun and the patient. A second "improvement" was the replacement of the older Therac-20's electromechanical safety interlocks with software control, a decision made because software was perceived to be more reliable.
What engineers didn't know was that both the 20 and the 25 were built upon an operating system that had been kludged together by a programmer with no formal training. Because of a subtle bug called a "race condition," a quick-fingered typist could accidentally configure the Therac-25 so the electron beam would fire in high-power mode but with the metal X-ray target out of position. At least five patients die; others are seriously injured.

lock Statement (C# Reference)

```
class Account
{
    decimal balance;
    private Object thisLock = new Object();
```

ถ้าเติม static จะเป็น global lock (มี lock เพียงอันเดียว สำหรับทุก ๆ account)

```
    public void Withdraw(decimal amount)
    {
        lock (thisLock)
        {
            if (amount > balance)
            {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

เกิด lock contention สูง

เรียนมาถึงตรงนี้แล้ว ถ้าไม่มีคำสั่ง lock ดูออกหรือยังว่าโปรแกรมนี้ bug ยังไง ?

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement

Synchronization Example

7.1.1 The Bounded-Buffer Problem

The *bounded-buffer problem* was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter.

In our problem, the producer and consumer processes share the following data structures:

```
int n;                    ขนาดของ buffer (buffer size)
semaphore mutex = 1;      จำนวน thread ที่จะเข้าไปใช้ buffer
semaphore empty = n;      จำนวนที่เก็บ สำหรับ producer
semaphore full = 0        จำนวนผลผลิต สำหรับ consumer
```

We assume that the pool consists of `n` buffers, each capable of holding one item. The `mutex` binary semaphore provides mutual exclusion for accesses to the 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.

The code for the producer process is shown in Figure 7.1, and the code for the consumer process is shown in Figure 7.2. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

สำหรับ producer หลาย ๆ process/thread

```
while (true) {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);  producer ตรวจสอบก่อนว่ามีที่เก็บใน buffer หรือไม่
   wait(mutex);
      . . .
   /* add next_produced to the buffer */  การแก้ไข buffer
      . . .                                ทำได้แค่ทีละ 1 thread
   signal(mutex);
   signal(full);  บอก consumer ว่าผลิตเพิ่มได้ 1 ชิ้น
}
```

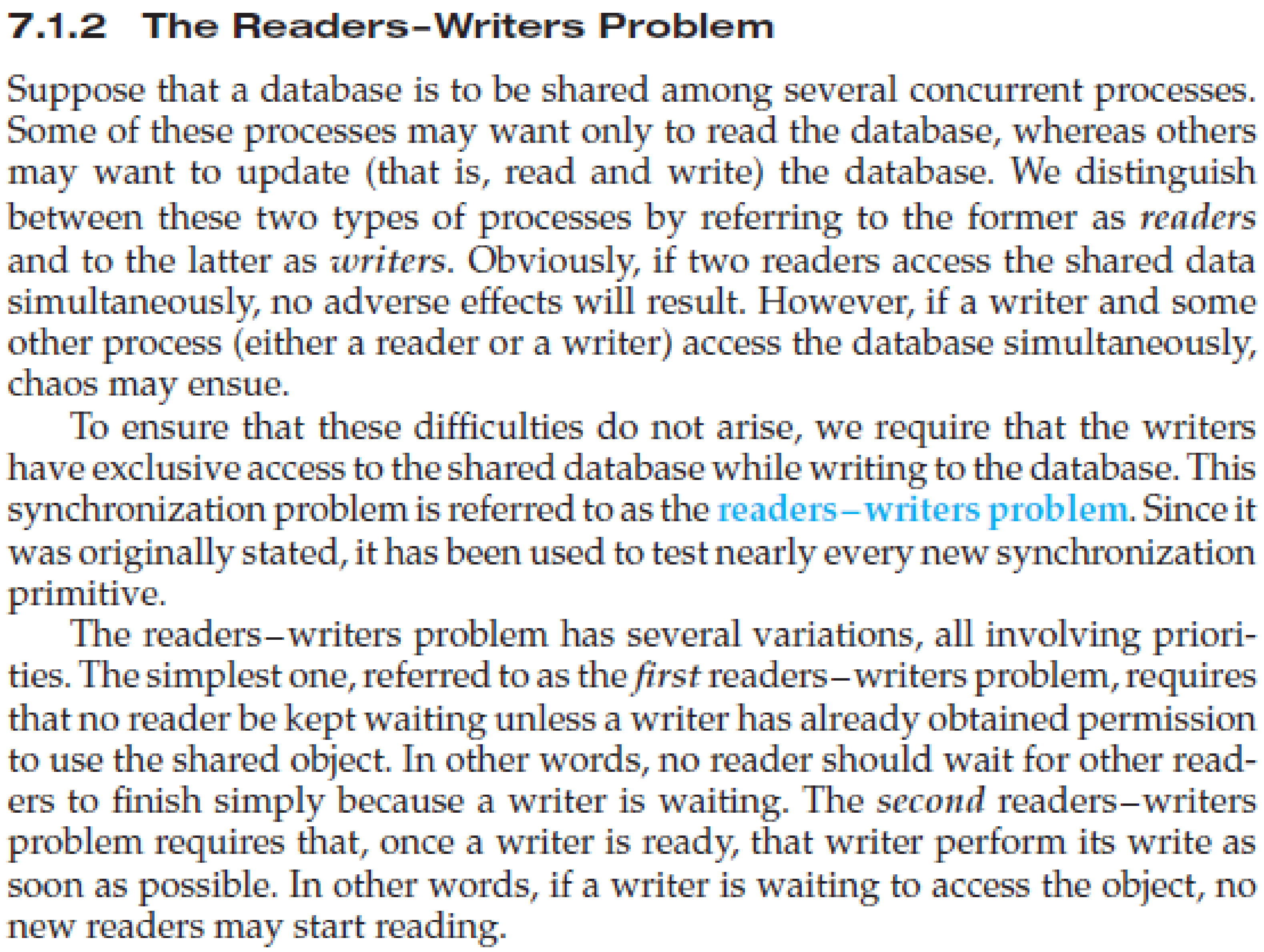
Figure 7.1 The structure of the producer process.

สำหรับ consumer หลาย ๆ process/thread

```
while (true) {
   wait(full);  consumer ตรวจสอบก่อนว่ามีผลผลิตใน buffer หรือไม่
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */  แก้ไข data structure
      . . .                                            ทำได้แค่ทีละ 1 thread
   signal(mutex);
   signal(empty);  บอก producer ว่าเอาผลผลิตไปแล้ว 1 ชิ้น
      . . .
   /* consume the item in next_consumed */
      . . .
}
```

Figure 7.2 The structure of the consumer process.

บาง library ให้ data structure ที่เป็น thread-safe มาเลย เราไม่ต้องป้องกัน CS เอง



7.1.2 The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, read and write) the database. We distinguish between these two types of processes by referring to the former as *readers* and to the latter as *writers*. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers–writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive.

The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

การบ้านให้เขียนโค้ดเพื่อแก้ปัญหา the first readers-writers problem

```
semaphore rw_mutex = 1;     จำนวน reader(s) หรือ writer ที่จะใช้ db ได้ (readers หลายตัวนับเป็น 1)
semaphore mutex = 1;        จำนวน thread ที่จะแก้ไข read_count และ rw_mutex ได้
int read_count = 0;         จำนวน reader process ที่รอ read หรือกำลัง read
```

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

Figure 7.3 The structure of a writer process.

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);     รอ writer (ถ้ามี)
   signal(mutex);         ให้ออกไปก่อน
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);   reader ตัวสุดท้าย
   signal(mutex);         signal ให้ writer
}                         (ถ้ามี) เข้ามาได้
```

Figure 7.4 The structure of a reader process.

อันนี้คือแบบ first problem เพราะ reader ที่เข้ามาทีหลัง แซง writer ไปก่อนได้ ถ้ามี reader ก่อนหน้ากำลัง read อยู่ ถ้ามี reader เข้ามาเรื่อย ๆ ไม่หยุด จะทำให้ writer เกิด starvation ได้ (จึงมี second problem เพื่อแก้ starvation)

ให้เขียนโปรแกรมดังนี้

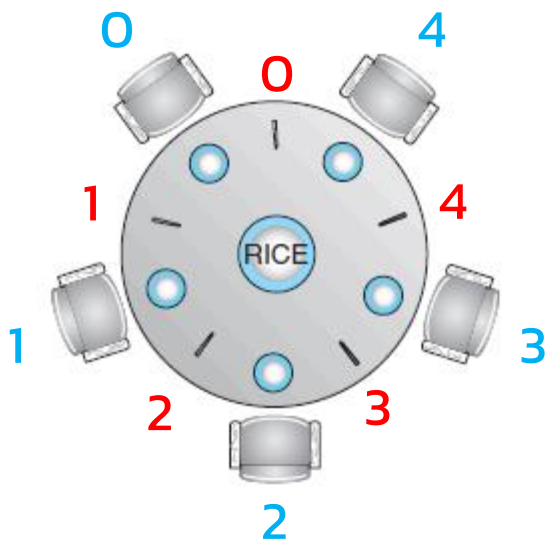
- สร้าง thread มา 1,000 thread สุ่ม 1% ให้เป็น writer อีก 99% เป็น reader
- ทั้ง reader และ writer ใช้เวลา 1 วินาที (ใส่ delay ช่วย)
- ถ้า synchronization ถูก โปรแกรมควรจะใช้เวลารันแค่ 1,000 x 1% x 1 = 10 วินาที เศษ เพราะต้องทำ writer ทีละตัว แต่ reader มันทำพร้อม ๆ กันได้ (ถ้าทำทีละ thread ก็ 1,000 วินาทีเสร็จ)
- thread ที่เป็น reader ให้ print "reading" 1 ครั้ง แล้ว sleep 1 วินาที
- thread ที่เป็น writer ให้ print "writing" 1 ครั้ง แล้ว sleep 1 วินาที

ภาษา C# มี semaphore

https://docs.microsoft.com/en-us/dotnet/api/system.threading.semaphore

7.1.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 7.5). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.



```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

หยิบตะเกียบด้านซ้ายมือก่อน แล้วค่อยหยิบตะเกียบทางขวา
อาจจะเกิด deadlock ได้

Figure 7.6 The structure of philosopher i.

Several possible remedies to the deadlock problem are the following:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of **starvation**.